

JClass LiveTable™

Programmer's Guide

Version 6.3
for Java 2 (JDK 1.3.1 and higher)

The Essential Java Grid/Table Component



8001 Irvine Center Drive
Irvine, CA 92618
949-754-8000
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
e-mail: info@quest.com
U.S. and Canada: 949.754.8000

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

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

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

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, all files included with the source code, and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,

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

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

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

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

Table of Contents

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

Part I: Using JClass LiveTable

1 'Hello Table' –	
JClass LiveTable Tutorial	9
1.1 The Basic Table	10
1.2 Overview of Table Changes	12
1.3 Improving the Table's Appearance	13
Adding and Formatting Labels	13
Introduction to Cell Styles	15
Changing Foreground and Background Colors	15
Changing Alignment	16
Changing the Fonts	17
Adding Color to an Individual Cell	18
Changing the Cell Borders and Spacing	19
Displaying More of the Cells	20
1.4 Adding Interactivity	21
Making the Cells Editable	21
Enabling Cell Selection	22
Resizing Using Labels Only	23
Enabling Column Sorting	24
1.5 Proceeding from Here	25

1.6	Internationalization	26
2	Building a Table.	27
2.1	Table Anatomy 101	27
2.2	Setting and Getting Properties	28
	Table Contexts	29
	Setting Table Properties with Java Code	31
	Setting Properties with a Java IDE at Design-Time	32
2.3	Preset Table Styles	32
2.4	Global Table Properties	32
	Focus Rectangle Appearance	33
	Screen Cursor Type	33
	Scrollbars	33
	Cell Selection Colors	34
	Row and Column Labels	35
	Cell and Label Border Width	37
	Cell and Label Margins	37
	Component Borders	38
	Frame Border Attributes	38
	Row and Column Definition	40
	Controlling Cell Editor Size	42
2.5	Column Width and Row Height Properties	43
	Character Height and Width	43
	Absolute Pixel Height and Width	44
	Variable Pixel Height and Width	45
	Maximum and Minimum Pixel Height and Width	46
	Displaying and Editing Multiple Lines in Cells	46
	Using Row Height and Width to Hide Rows and Columns	46
2.6	Cell Styles	47
	Cell Style Properties and Implementation	47
	Defining Your Own or Changing Built-In Cell Styles	48
	Using and Modifying JClass LiveTable's Built-In Styles	50
	Working with Colors	52
	Text and Image Alignment	53
	Cell and Label Fonts	54
	Border Types	54
	Cell and Label Border Sides	57
	Text and Image Clipping	57
	Displaying Images in Table Cells	58

2.7	Cell and Label Spanning	58
	Using Spanning to Create Multiline Headers	60
3	Working with Table Data	61
3.1	Overview: Data Handling in JClass LiveTable	61
	How the Table and Data Source Communicate	61
3.2	Getting Data into your Table	62
	Making the Data Source Editable	63
3.3	Using Stock Data Sources	63
	JCVectorDataSource: the Data Source Workhorse	64
	Getting Data from an Input Stream	64
	Getting Data from a Database	65
	Caching Data with JCCachedDataSource	65
	Using Swing TableModel Data Objects	66
3.4	Setting Stock Data Source Properties	66
	Working with Rows and Columns	66
	Working with Other Properties	69
3.5	Loading Data from an XML Source	69
	XML Primer	69
	Using XML in JClass	70
	Example XML Files for JClass LiveTable	71
	Tags	71
	Creating a Swing TableModel class	71
3.6	Creating your own Data Sources	72
3.7	Dynamically Updating Data	74
	Adding and Removing Columns and Rows	78
4	Displaying and Editing Cells	79
4.1	Overview	79
4.2	Default Cell Rendering and Editing	80
4.3	Rendering Cells	81
	JClass Cell Renderers	81
	Setting a Cell Renderer for a Series	82
	Mapping a Data Type to a Cell Renderer	83
	Creating your own Cell Renderers	84

4.4	Editing Cells	89
	Default Cell Editors	90
	Setting a Cell Editor for a Series	91
	Mapping a Data Type to a Cell Editor	91
	Creating Your Own Cell Editors	92
4.5	The JCCellInfo Interface	100
5	Adding Formulas to JClass LiveTable	103
5.1	Introduction	103
5.2	com.klg.jclass.util.formulae's Hierarchy	103
5.3	Expressions and Results	105
5.4	Math Values	105
	MathScalar	106
	MathVector	106
	MathMatrix	107
5.5	Operations	108
	The Defined Mathematical Operations	109
	Reducing Operations to Values	111
5.6	Expression Lists	112
5.7	Events and Listeners	112
5.8	Exceptions	113
5.9	Using Formulae in JClass LiveTable	113
	Registering a Cell Editor and a Cell Renderer with the JClass Central Registry	113
	Performing a Mathematical Operation on a Range of Cells	114
6	Programming User Interactivity	115
6.1	Cell Traversal	115
	Default Cell Traversal	115
	Customizing Cell Traversal	115
	Minimum Cell Visibility	116
	Forcing Traversal	116
	Controlling Interactive Traversal	117
6.2	Resizing Rows and Columns	118
	Default Resizing Behavior	118
	Disallowing Cell Resizing	118
	Controlling Resizing	118

6.3	Table Scrolling	120
	Default Scrolling Behavior	120
	Managing Table Scrolling	120
	Scroll Listener Methods	122
6.4	Cell Selection	123
	Default Cell Selection	123
	Selection Colors	124
	Customizing Cell Selection	124
	Selected Cell List	125
	Working with Selected Ranges	125
	Removing Selections	126
	Runtime Selection Control	126
6.5	Dragging Rows and Columns	126
6.6	Sorting Columns	127
	Sort by Clicking on a Column Label	129
	Resetting the Table after Sorting	129
6.7	Custom Mouse Pointers	129
7	Events and Listeners	131
7.1	Displaying Cells	131
7.2	Editing Cells	133
7.3	Painting Tables	136
7.4	Printing Tables	137
7.5	Resizing Cells	138
7.6	Scrolling in Tables	141
7.7	Selecting Cells	144
7.8	Sorting Table Data	147
7.9	Table Data Changes	149
7.10	Traversing Cells	151
8	Table Printing	155
8.1	Printing	155
	Setting Page Layout Properties	155
	Page Resolution	156
	Printing Headers and Footers	156
8.2	Print Preview	157

9	JClass LiveTable Beans and IDEs	159
9.1	An Introduction to JavaBeans	159
	Properties	159
	Setting Properties in a Java IDE at Design-Time	160
	Setting Properties using Methods in the API	160
9.2	JClass LiveTable and JavaBeans	160
9.3	Setting Properties for the LiveTable Bean	161
	JClass LiveTable Property Editors	161
	LiveTable Properties	164
9.4	Tutorial: Building a Table in an IDE	174
	The Basic Table	175
	Improving the Table's Appearance	176
	Adding Interactivity	181
	The Final Program	183
9.5	Data Binding with IDEs	183
	Data Binding LiveTable with a JBuilder Data Source	184
	Data Binding Using JClass DataSource	189
9.6	Interacting with Data Bound Tables	193
9.7	Property Differences Between the JClass LiveTable Beans	194

Part II: Reference Appendices

A	Event Summary	199
B	JClass LiveTable Property Listing	203
B.1	Properties of com.klg.jclass.table.JCTable	203
B.2	Properties of com.klg.jclass.table.CellStyleModel	212
B.3	Properties of com.klg.jclass.table.beans.LiveTable	214
B.4	Properties of com.klg.jclass.table.db.jbuilder.JBdbTable	215
B.5	Properties of com.klg.jclass.table.db.datasource.DSdbTable	217
C	Porting JClass 3.6.x Applications	219
C.1	Overview of Changes	219
C.2	Porting Strategies	220
C.3	Highlights of Main Changes	220
D	Colors and Fonts	223
D.1	Colorname Values	223

D.2	RGB Color Values	223
D.3	Fonts	228
E	JClass LiveTable Inheritance Hierarchy	229
F	Distributing Applets and Applications	231
F.1	Using JarMaster to Customize the Deployment Archive . . .	231
G	Overview of Examples and Demos	233
G.1	JClass LiveTable Examples	233
G.2	JClass LiveTable Demos	238
	Index	241

Preface

[Introducing JClass LiveTable](#) ■ [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 LiveTable

JClass LiveTable is a Java GUI component that displays rows and columns of user-interactive text, images, hypertext links, and other Java components in a scrollable window.

JClass LiveTable may be used in conjunction with Quest Software's JClass Field, in that a Field component may be added to a JClass LiveTable cell.

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

You can freely distribute Java applets and applications containing JClass components according to the terms of the License Agreement.

Feature Overview

You can set the properties of JClass LiveTable components to determine how the table will look and behave. You can control:

- The data source for the table.
- Preset and custom cell editing and display behavior for all types of data.
- Labels for columns and rows.
- Colors, fonts, borders (including custom borders), alignment, and spacing for cells and labels.
- Row and column dragging.
- Column sorting.
- Adding, deleting, moving, and dragging rows and columns.
- Scrolling and attaching default or custom scrollbars.
- Cell selection and traversal.

Assumptions

This manual assumes that you have some experience with the Java programming language. You should have a basic understanding of object-oriented programming and

Java programming concepts such as classes, methods, and packages before proceeding with this manual. See [Related Documents](#) later in this section of the manual for additional sources of Java-related information.

Typographical Conventions in this Manual

Typewriter Font

- Java language source code and examples of file contents.
- JClass LiveTable and Java classes, objects, methods, properties, constants, and events.
- HTML documents, tags, and attributes.
- Commands that you enter on the screen.

Italic Text

- Pathnames, filenames, URLs, programs, and method parameters.
- New terms as they are introduced, and to emphasize important words.
- Figure and table titles.
- The names of other documents referenced in this manual, such as *Java in a Nutshell*.

Bold

- Keyboard key names and menu references.

Overview of the Manual

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

Chapter 1, [‘Hello Table’ – JClass LiveTable Tutorial](#), provides a tutorial exercise to familiarize new users with the basics of writing a JClass LiveTable program.

Chapter 2, [Building a Table](#), explains how to set most JClass LiveTable properties to customize the appearance and display of JClass LiveTable applications.

Chapter 3, [Working with Table Data](#), gives details on getting data into and out of tables using the Model View Controller data handling in JClass LiveTable.

Chapter 4, [Displaying and Editing Cells](#), describes how to configure JClass LiveTable so users can edit cells of any data type.

Chapter 5, [Adding Formulas to JClass LiveTable](#), outlines the *com.klg.jclass.util* package in *com.klg.jclass.util*, which has special capabilities for working with mathematical objects.

Chapter 6, [Programming User Interactivity](#), explains how to control how users interact with your table application, including cell traversal, selection, sorting, and more.

Chapter 7, [Events and Listeners](#), explains how to send events and register event listeners in your JClass LiveTable programs.

Chapter 8, [Table Printing](#), describes the enhanced printing features of JClass LiveTable.

Chapter 9, [JClass LiveTable Beans and IDEs](#), describes the JClass LiveTable JavaBeans and how to use them within a Java Development Environment.

Part II – Reference Appendices – provides quick access to detailed information on JClass LiveTable features and implementation.

Appendix A, [Event Summary](#), lists events and corresponding event listeners.

Appendix B, [JClass LiveTable Property Listing](#), is a quick reference to properties, their functions, and settable values.

Appendix C, [Porting JClass 3.6.x Applications](#), explains how to properly migrate existing LiveTable 3.x applications to LiveTable 4.x.

Appendix D, [Colors and Fonts](#), lists all of the color names and RGB values available to JClass LiveTable applications. It also lists all of the available fonts and font style constants.

Appendix E, [JClass LiveTable Inheritance Hierarchy](#), summarizes the `com.klg.jclass.table` package.

Appendix F, [Distributing Applets and Applications](#), is a quick tutorial that demonstrates how to take a completed Java applet and deploy it on a Web page and Web server.

Appendix G, [Overview of Examples and Demos](#), summarizes all JClass LiveTable examples and demos, and refers you to the chapter that covers the predominant feature(s) used in a particular example or demo.

API Reference

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

Licensing

In order to use JClass LiveTable, you need a valid license. Complete details about licensing are outlined in the [Installation Guide](#), which is automatically installed when you install JClass LiveTable.

Related Documents

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

- “*Java Tutorial*” at <http://java.sun.com/docs/books/tutorial/index.html> from Sun Microsystems.
- *Java in a Nutshell, 2nd Edition* from O’Reilly & Associates Inc.
- Resources for using JavaBeans are at <http://java.sun.com/beans/resources.html>.

Please note that these documents are not required to develop applications using JClass LiveTable and Java.

About Quest

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

Contacting Quest Software

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

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

Customer Support

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

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

You can use SupportLink to do the following:

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

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

To Contact JClass Support

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

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

JClass Direct Technical Support	
JClass Support Email	support@quest.com
Telephone	949-754-8000
Fax	949-754-8999

**European Customers
Contact Information**

Telephone: +31 (0)20 510-6700

Fax: +31 (0)20 470-0326

Product Feedback and Announcements

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

Please send your comments to:

Quest Software

8001 Irvine Center Drive

Irvine, CA 92618

Telephone: 949-754-8000

Fax: 949-754-8999

Part ***I***

*Using
JClass
LiveTable*

‘Hello Table’ – JClass LiveTable Tutorial

[The Basic Table](#) ■ [Overview of Table Changes](#) ■ [Improving the Table's Appearance](#)
[Adding Interactivity](#) ■ [Proceeding from Here](#) ■ [Internationalization](#)

You can immediately learn about some fundamental JClass LiveTable programming concepts by compiling and running an example program¹. This program displays information about orders for “The Musical Fruit”, a fictional wholesale coffee distributor, based on the following data:

Customer Name	Order Date	Item	Quantity (lbs.)	Price/lb.
The Cuppa	11/11/97	French Mocha	60	\$7.01
The Underground Cafe	11/14/97	Brazilian Medium	112	\$6.80
RocketFuel and Cake Cafe	10/30/97	Espresso Dark	300	\$8.02
WideEyes Coffee House	11/12/97	Colombian/Ir ish Cream Flavored	120	\$5.30
Jitters Caffeine Cavern	10/01/97	Ethiopian Medium	80	\$7.50
Twitchie’s on the Mall	12/06/97	French Roast Kona	160	\$14.50
Quest Software Inc.	12/12/97	Colombian	22,000	\$5.28

1. This exercise assumes that you are familiar with Java programming concepts and have previously written and compiled Java programs. It also begs forgiveness for yet another play on the coffee theme of Java.

1.1 The Basic Table

The following code is from *ExampleTable1.java*, found in the *examples/table/intro* directory of your JClass LiveTable installation directory. The code creates a very plain looking table, without column labels or any other JClass LiveTable features to improve usability and appearance.

```
package examples.table.intro;

// import the necessary java classes, including the Table package
import java.awt.Component;
import javax.swing.JPanel;
import com.klg.jclass.util.swing.JCExitFrame;
import com.klg.jclass.table.JCTable;
import com.klg.jclass.table.data.JCVectorDataSource;

// initiate the class declaration
public class ExampleTable1 extends JPanel {

// set the cell values as a matrix of strings
String cells[][] = {
    {"The Cuppa", "11/11/97", "French Mocha", "60", "$7.01"},
    {"The Underground Cafe", "11/14/97", "Brazilian Medium", "112", "$6.80"},
    {"RocketFuel and Cake", "10/30/97", "Espresso Dark", "300", "$8.02"},
    {"WideEyes Coffee House", "11/12/97", "Colombian/Irish Cream
Flavored", "120", "$5.30"},
    {"Jitters Caffeine Cavern", "10/01/97", "Ethiopian Medium
Roast", "80", "$7.50"},
    {"Twitchy's on the Mall", "12/06/97", "French Roast Kona", "160", "$14.50"},
    {"Quest Software Inc.", "12/12/97", "Colombian", "22,000", "$5.28"}
};

// initialize the Table object
protected JCTable table;

// Build the table, point to the data source and define the table
properties.
public ExampleTable1() {
    setLayout(new java.awt.GridLayout());

    // Create a default table object
    table = new JCTable();

    // Create a vector data source to contain our data
    JCVectorDataSource ds = new JCVectorDataSource();

    // Turn off column labels
    table.setColumnLabelDisplay(false);

    // Turn off row labels
    table.setRowLabelDisplay(false);

    // Set the data source to the vector data source from earlier
    table.setDataSource(ds);
}
```

```

        // Set the number of rows in the data source.
        ds.setNumRows(7);

        // Set the number of columns in the data source.
        ds.setNumColumns(5);
        // Set the cell data in the data source.
        ds.setCells(cells);

        this.add(table);
    }

    public static void main(String args[]) {
        JCExitFrame f = new JCExitFrame("ExampleTable1");
        ExampleTable1 et = new ExampleTable1();
        f.getContentPane().add(et);
        f.setSize(600, 200);
        f.setVisible(true);
    }
}

```

Note: As you change the *ExampleTable1.java* file throughout this tutorial, it may be necessary to resize the frame to fit the content.

How the Table Handles Data

The table uses a Model-View-Controller (MVC) data mechanism; the table data is stored in a separate object. For this table example, we have used `JCVectorDataSource`, a class provided with `JClass LiveTable` that retrieves data from the data source and stores it in memory (see [Using Stock Data Sources](#), in Chapter 3, for more information).

The data source is set using the `table.setDataSource()` method:

```

table.setDataSource(ds);
ds.setNumRows(7);
ds.setNumColumns(5);
ds.setCells(cells);

```

Once the data source is set to the `JCVectorDataSource` (`ds`) object, that object handles the data, including setting the number of rows and columns, and accessing the cell values. The data in the cells is of type `String`.

What the Table Looks Like

If you compile and run the modified *ExampleTable1.java* program,¹ the following table is displayed:



Item	Date	Type	Quantity	Price
The Cuppa	11/11/97	French Mocha	60	\$7.01
The Underground C...	11/14/97	Brazilian Medium	117	\$6.80
RocketFuel and Cak...	10/30/97	Espresso Dark	300	\$8.02
WideEyes Coffee H...	11/12/97	Colombian/Irish Cr...	120	\$5.30
Jitters Caffeine Cav...	10/01/97	Ethiopian Medium R...	80	\$7.50
Twitchy's on the Mall	12/06/97	French Roast Kona	160	\$14.50
Quest Software Inc.	12/12/97	Colombian	22,000	\$5.28

The clip arrows indicate that the cells are not large enough to display their entire contents. By default, users can resize rows and columns to view the contents of the cell. Notice that if you click a cell, a *focus rectangle* appears, showing the current cell.

1.2 Overview of Table Changes

The following sections walk you through the modification of the example table. It is assumed that you are changing the code in the *ExampleTable1.java* file, compiling and then running it after each step to view the results. Of course, it is recommended that you make a copy of the original file.

For each of the table's modifications, all the code that needs to be added is provided. Since some code segments rely on the presence of code from previous steps, it is recommended that you perform all modifications in the order in which they appear in this chapter.

Additionally, all changes made in this tutorial are reflected in the other example files found in the *examples/table/intro* directory. You can also compile and run those files to compare and verify the changes you make to *ExampleTable1.java*. Throughout the chapter, you will be alerted when the cumulative changes can be seen in another example file.

Change to ExampleTable1.java	Example File that Encompasses Changes Made
defining and adding labels	<i>examples/table/intro/ExampleTable2.java</i>
label colors	<i>examples/table/intro/ExampleTable3.java</i>
label text alignment	<i>examples/table/intro/ExampleTable4.java</i>
label font	

1. Note that the example programs in your JClass LiveTable distribution contain a package name. To run the compiled class, you must type the full package name, for example: `java examples.table.intro.ExampleTable1`

Change to ExampleTable1.java	Example File that Encompasses Changes Made
color of an individual non-label cell cell and frame borders and spacing	<i>examples/table/intro/ExampleTable5.java</i>
cell height and width enabling cell editing	<i>examples/table/intro/ExampleTable6.java</i>
enabling cell selection resize only with labels	<i>examples/table/intro/ExampleTable7.java</i>
enable column sorting	<i>examples/table/intro/ExampleTable8.java</i>

1.3 Improving the Table's Appearance

Using some of the properties for modifying a table's appearance, you can easily move from the basic, drab table in *ExampleTable1.java*, to a table that is easier to understand, easier to use, and more visually appealing.

All properties for a table can be specified when you create the table, or they may be changed at any time as the program runs by using event listeners. Each property has two *accessor methods*: *set* and *get*. An example of a *set* method for a property is `setBackground()`, which sets the background color of a cell or label. You can retrieve the current value of any property using the property's *get* method, as in `getBackground()`.

1.3.1 Adding and Formatting Labels

Background

The table displayed by the *ExampleTable1.java* program is not very useful to an end-user. Not only is it uninteresting to look at, but you cannot tell what kinds of information the cells contain because there are no column labels. In the original data outline for the table (at the beginning of the chapter), we specified the following column headers or labels:

- Customer Name
- Order Date
- Item
- Quantity (lbs.)
- Price/lb.

Labels are cells that can never be edited and can contain any Object, (for example, Strings, images, integers). You can apply labels to rows and columns. The label values, like cell values, are set in the data source object.

Procedure

In *ExampleTable1.java*, set the labels as a String by inserting this line immediately after the cell values String statement:

```
String labels[] = {"Customer Name","Order Date","Item", "Quantity  
(lbs.)","Price/lb."};
```

Once you have defined the values for the column labels, you have to instruct the Table object to display labels. The program currently contains the line:

```
table.setColumnLabelDisplay(false);
```

By default, column labels are set to true. Change the label setting back to this default by entering this code:

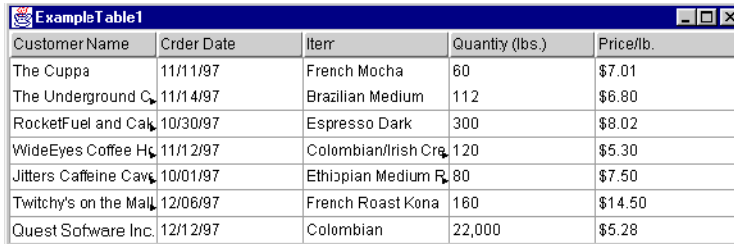
```
table.setColumnLabelDisplay(true);
```

Once the ColumnLabelDisplay property is set to true, you can set the column labels in the data source. After `ds.setCells(cells);`, add the line:

```
ds.setColumnLabels(labels);
```

This uses the data source to set the values of the column labels from the data specified in the String cells.

Compile and run the modified *ExampleTable1.java* file. The table now looks like this:



Customer Name	Order Date	Item	Quantity (lbs.)	Price/lb.
The Cuppa	11/11/97	French Mocha	60	\$7.01
The Underground C...	11/14/97	Brazilian Medium	112	\$6.80
RocketFuel and Cal...	10/30/97	Espresso Dark	300	\$8.02
WideEyes Coffee H...	11/12/97	Colombian/Irish Cr...	120	\$5.30
Jitters Caffeine Cav...	10/01/97	Ethiopian Medium R...	80	\$7.50
Twitchy's on the Mal...	12/06/97	French Roast Kona	160	\$14.50
Quest Software Inc.	12/12/97	Colombian	22,000	\$5.28

Note: You can also run *ExampleTable2.java*, which already contains these changes.

Notice that the column labels are now part of the table. Also note that if you click a label, you do not get the focus rectangle that would appear on a selected cell, as labels cannot be edited and are not included in cell traversal. In certain situations, clicking a label performs an action (this will be discussed in Section 1.4, [Adding Interactivity](#)). However, in this case, the labels do not perform any interactive function.

1.3.2 Introduction to Cell Styles

Cell Styles provide a very flexible model for changing the appearance (and some behavior) of a table's cells or labels. A style contains attributes that can be applied to cells and labels, including color, text properties, and text/image alignment.

JClass LiveTable comes with several constructs that are part of Cell Styles:

- `CellStyleModel`: An interface that defines the methods required by an object to specify the attributes of a cell.
- `JCCellStyle`: The default implementation of the `CellStyleModel` interface.
- The default cell and label styles: These are preset styles (one for labels, one for cells) whose look and feel change with any changes to the pluggable look and feel (PLAF) implementation of a table.

The visual table changes found in the next four sections are defined using these Cell Style constructs.

For in-depth coverage of cells styles, please refer to [Building a Table](#), in Chapter 2.

1.3.3 Changing Foreground and Background Colors

Background

There are thirteen AWT color constants that can be used in Java. The color constant values are:

- `Color.black`
- `Color.magenta`
- `Color.blue`
- `Color.orange`
- `Color.cyan`
- `Color.pink`
- `Color.darkGray`
- `Color.red`
- `Color.gray`
- `Color.white`
- `Color.green`
- `Color.yellow`
- `Color.lightGray`

Procedure

In order to make changes with AWT colors, you need to include the `java.awt.Color` package to the *ExampleTable1.java* file that you are modifying. Add this to your list of import statements:

```
import java.awt.Color;
```

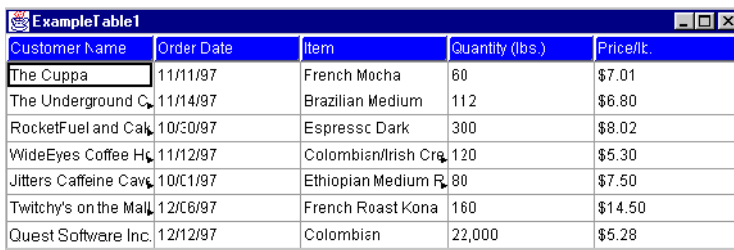
Since you are using default styles in the examples, you need to import the interface with which you implement the style. To do this, import the `CellStyleModel` class by adding this line to your list of import statements:

```
import com.klg.jclass.table.CellStyleModel;
```

Now that the required AWT color and Cell Style classes are accessible, set the background color of the labels to blue, and the foreground color (text) to white. Do this by inserting the following lines into the file's `ExampleTable1` class:

```
CellStyleModel labelStyle = table.getDefaultLabelStyle();
labelStyle.setBackground(Color.blue);
labelStyle.setForeground(Color.white);
```

Here, you acquire the default label style. You then tweak the default label color attributes by changing the default colors to blue and white. Recompile and run the modified *ExampleTable1.java* file. The table now looks like this:



Customer Name	Order Date	Item	Quantity (lbs.)	Price/lb.
The Cuppa	11/11/97	French Mocha	60	\$7.01
The Underground C...	11/14/97	Brazilian Medium	112	\$6.80
RocketFuel and Cak...	10/30/97	Espresso Dark	300	\$8.02
WideEyes Coffee Ho...	11/12/97	Colombian/Irish Cra...	120	\$5.30
Jitters Caffeine Cav...	10/1/97	Ethiopian Medium F...	80	\$7.50
Twitthy's on the Mall	12/6/97	French Roast Kona	160	\$14.50
Quest Software Inc.	12/12/97	Colombian	22,000	\$5.28

Note: You can also run *ExampleTable3.java*, which already contains these changes.

1.3.4 Changing Alignment

Background

Another way to visually differentiate the text that appears within a table is to change its alignment within a cell relative to the text alignment in other cells. By default, text (or anything else you insert into specific cells in a table) is shifted to the top and left margins of the cell. With Cell Styles, the horizontal and vertical positioning of a cell's contents can be defined.

If you want to set the labels in the sample program to appear horizontally centered and at the top of the label, continue to modify the default label style that was set in the previous step.

Procedure

Add this line to your set of import statements:

```
import com.klg.jclass.table.JCTableEnum;
```

Then, append these lines to the `labelStyle` statements that were added in the previous step:

```
labelStyle.setHorizontalAlignment(JCTableEnum.CENTER);
labelStyle.setVerticalAlignment(JCTableEnum.TOP);
```

1.3.5 Changing the Fonts

Background

It is also possible to change fonts and their appearance. This is another way to visually distinguish one part of a table from another, or to change the overall appearance of the table.

Java defines five different platform-independent font names that are found (or have close equivalents) on most computer platforms. Valid Java AWT font names are:

- Courier
- Dialog
- DialogInput
- Helvetica
- TimesRoman

Note: Font names are case-sensitive.

There are also four standard font style constants that can be used. Valid Java AWT font style constants are:

- Font.BOLD
- Font.PLAIN
- Font.ITALIC
- Font.BOLD + Font.ITALIC

Procedure

We want to change the text column labels in the modified *ExampleTable1.java*, from their default to a 14 point, bold-italic, Times Roman text. In order to make changes with AWT fonts, you need to include the `java.awt.Font` package to the program. Add this to your list of import statements:

```
import java.awt.Font;
```

Append this line to the `labelStyle` statements added in the last two steps:

```
labelStyle.setFont(new Font("TimesRoman", Font.BOLD +
                             Font.ITALIC, 14));
```

Recompile and run your modified *ExampleTable1.java* file. Having changed the text font and alignment, your table now looks like this:



Customer Name	Order Date	Item	Quantity (lbs.)	Price/lb.
The Cuppa	11/1/97	French Roast	60	\$7.0
The Underground Caf	11/1/97	Brazilian Medium	1.2	\$6.80
RrrrckErieland Cake	10/30/97	Espresso Dark	300	\$8.0
WideEyes Coffee Hou	11/1/97	Colombian/Irish Cream	120	\$5.30
Jitters Caffeine Cover	10/01/97	Ethiopian Medium Roast	60	\$7.50
Twitchy's on the Mall	12/08/97	French Roast Kona	180	\$14.50
Quest Software Inc.	12/1/97	Colombian	22.00	\$5.28

Note: You can also run *ExampleTable4.java*, which already contains these changes.

The type of font displayed on a user's system depends entirely on the fonts that are local to that user's computer. If a font name specified in a Java program is not found on a user's system, the closest possible match is used as determined by the Java AWT.

1.3.6 Adding Color to an Individual Cell

Background

In some cases, you will want the information in a certain cell or range of cells to stand out from the rest. As previously mentioned, Cell Styles can be used with individual or ranges of cells.

In our modified *ExampleTable1.java* file, we want to highlight the premium coffee order using different foreground and background colors – in this case, Twitchy's on the Mall (row 6, column 1).

When we originally made changes to the labels using Cell Styles (the first change made was to the label colors), we retrieved the default label style and implemented them into the `CellStyleModel` class. This made a change to *all* labels. Now that you are working with a *single* cell, using the default Cell Style for non-label cells requires a similar action, but with an added step.

Procedure

First, import `JCCellStyle` by adding this line to your set of import statements:

```
import com.klg.jclass.table.JCCellStyle;
```

Then, similarly to what was done with the labels' style, retrieve the default style for non-label cells by adding this new line to the `ExampleTable1` class:

```
CellStyleModel cellStyle = table.getDefaultCellStyle();
```

Next, add this line to create a Cell Style for the single "Twitchy's on the Mall" cell, which creates a new unique Cell Style that inherits all the style settings from the default Cell Style:

```
CellStyleModel specialStyle = new JCCellStyle((JCCellStyle)cellStyle);
```

Since we want to change the specific cell's color, but do not want these changes applied to all the cells, the `specialStyle` object was created. Now, change the colors of the cell by adding these lines:

```
specialStyle.setForeground(Color.red);
specialStyle.setBackground(Color.yellow);
table.setCellStyle(5, 0, specialStyle);
```

Note: The cell found at row 6, column 1 in the displayed table is designated as row 5, column 0 in the code. This is because row and column indexes begin at zero. The top left cell in the table is at location (0, 0).

Recompile and run the modified `ExampleTable1.java` file. The colors in cell (5, 0) have changed, and the table now looks like this:



Customer Name	Order Date	Item	Quantity (lbs.)	Price/lb.
The Cuppa	11/1/97	French Mocha	60	\$7.00
The Underground Cafe	11/14/97	Brazilian Medium	172	\$6.80
RockerFuel and Cake	10/30/97	Espresso Dark	300	\$8.00
WideEye Coffee House	11/12/97	Colombian/Irish Cream	120	\$5.30
Jitters Caffeine Cavern	10/01/97	Ethiopian Medium Roast	100	\$7.50
Wilutz's on the Mall	12/08/97	French Roast Kona	180	\$14.50
Quest Software Inc.	12/12/97	Colombian	22,000	\$5.28

1.3.7 Changing the Cell Borders and Spacing

Background

There are a number of properties that can be used to define cell/frame borders and cell spacing. These are outlined in [Building a Table](#), in Chapter 2. For the example program, we are going to thicken the cell borders, as well as the table's frame border. Also, the border style for the cells (not labels) and frame will be changed.

Procedure

First, in order to work with borders, import the `JCCellBorder` class by adding this line to your list of import statements in your modified `ExampleTable1.java`:

```
import com.klg.jclass.table.JCCellBorder;
```

Next, add these lines to the `ExampleTable1` class:

```
cellStyle.setCellBorder(new JCCellBorder(JCTableEnum.BORDER_OUT));
table.setCellBorderWidth(5);

table.setFrameBorderWidth(3);
table.setFrameBorder(new JCCellBorder(JCTableEnum.BORDER_OUT));
```

Now that you have made these additions to the code, recompile and run the modified *ExampleTable1.java* file. Having changed the cell and frame border properties, the table now looks like this:

Customer Name	Order Date	Item	Quantity (lbs.)	Price/D.
The Cuppa	11/1/37	French Mocha	60	\$7.01
The Underground Cafe	11/4/37	Brazilian Medium	112	\$6.80
Hockey Hat and Cake	10/30/37	Espresso Dark	300	\$8.02
Wide Eyes Coffee House	11/2/37	Colombian French Roast	170	\$5.30
Jitters Coffee Company	10/01/37	Ethiopian Medium Roast	80	\$7.50
The Griffs on the Move	12/00/37	French Roast, Kona	100	\$14.50
Guest Software Inc.	12/2/37	Colombian	22.00L	\$5.28

Note: You can also run *ExampleTable5.java*, which already contains these changes.

1.3.8 Displaying More of the Cells

Background

The example table has come a long way after setting only a few properties, but there is still a small problem: the table may clip the cell's contents. This means that the user has to resize the rows or columns in order to read the contents of some cells. By default, JClass LiveTable sets all of the cells to a width of 10 characters and a height of one character. You could specify the height and width of the cells in rows and columns in terms of lines and characters using the `CharHeight` and `CharWidth` properties. However, in this program we want the cells to size themselves to display the entire contents (if possible).

Procedure

Add the following lines of code to the modified *ExampleTable1.java*:

```
table.setPixelHeight(JCTableEnum.ALLCELLS, JCTableEnum.VARIABLE);
table.setPixelWidth(JCTableEnum.ALLCELLS, JCTableEnum.VARIABLE);
```


These lines set the `PixelHeight` and `PixelWidth` properties to a variable size for all rows and all columns, ensuring that the table will attempt to display the entire contents of each cell. Recompile and run the modified *ExampleTable1.java*. The table looks like this:



Customer Name	Order Date	Item	Quantity (lbs.)	Price/lb.
The Cuppa	11/11/97	French Mocha	60	\$7.01
The Underground Cafe	11/14/97	Brazilian Medium	112	\$6.80
RocketFuel and Cake	10/30/97	Espresso Dark	300	\$8.02
WideEyes Coffee House	11/12/97	Colombian/Irish Cream Flavored	120	\$5.30
Jitters Caffeine Cavern	10/01/97	Ethiopian Medium Roast	80	\$7.50
Twitchy's on the Mall	12/06/97	French Roast Kona	160	\$14.50
Quest Software Inc.	12/12/97	Colombian	22,000	\$5.28

You can also set these properties to specific pixel values for rows and columns; see the section on how to set [Column Width and Row Height Properties](#), in Chapter 2, for more details.

So far, all the changes that have been made to *ExampleTable1.java* have centered around a table's set of visual properties. Keeping the changes made thus far, we will continue by making changes to some of *ExampleTable1.java*'s interactive properties.

1.4 Adding Interactivity

In a hypothetical scenario, our example table could be used to track orders and accounts with a large number of customers. Your users will likely want to update the data, sort the information displayed in the table, and select sections of the table to perform operations on them.

We will add some basic user-interactivity to our example table to give you a sense of some of the things you can do with JClass LiveTable. You can explore user-interactivity further in [Programming User Interactivity](#), in Chapter 6.

1.4.1 Making the Cells Editable

Background

As far as user interaction goes, one of the problems with this example table is that it is not editable. If a user clicks a cell, the focus changes, but nothing else happens. To make the cell editable, we have to change the data source object to an editable data source. The `JCVectorDataSource` class we used as our data source has an editable counterpart called `JCEditableVectorDataSource`.

Procedure

The modified *ExampleTable1.java* currently contains the lines:

```
import com.klg.jclass.table.data.JCVectorDataSource;  
JCVectorDataSource ds = new JCVectorDataSource();
```

Change them to:

```
import com.klg.jclass.table.data.JCEditableVectorDataSource;  
JCEditableVectorDataSource ds = new JCEditableVectorDataSource();
```

Once you change these lines, recompile and run the modified *ExampleTable1.java* file. The table now looks like this:



Customer Name	Order Date	Item	Quantity (lbs.)	Price/lb.
The Cuppa	11/11/97	French Mocha	60	\$7.01
The Underground Cafe	11/14/97	Brazilian Medium	112	\$6.80
RocketFuel and Cake	10/30/97	Espresso Dark	300	\$8.02
WideEyes Coffee House	11/12/97	Colombian/Irish Cream Flavored	120	\$5.30
Jitters Coffee Co.	10/01/97	Ethiopian Medium Roast	80	\$7.50
Twitchy's on the Mall	12/06/97	French Roast Kona	160	\$14.50
Quest Software Inc.	12/12/97	Colombian	22,000	\$5.28

Figure 1 The table with editable cells. Note cell (3, 2) is being edited.

Note: You can also run *ExampleTable6.java*, which already contains these changes.

Clicking a cell will bring up the editing component for the type of data in the cell. Since all of the cells contain Strings, the editing component is a text editor. For more information, see [Displaying and Editing Cells](#), in Chapter 4.

1.4.2 Enabling Cell Selection

Background

JClass LiveTable provides methods that set how users can select cells, ranges of cells, and entire rows and columns. Selection is enabled by setting the `SelectionPolicy` property. By default, cell selection reverses the foreground and background colors of the cells to highlight the selection.

Procedure

You can enable selection by adding the following code to the example program:

```
table.setSelectionPolicy(JCTableEnum.SELECT_RANGE);
```

This allows users to select one or more cells in rows or columns by clicking and dragging the mouse, or using keyboard combinations.

By default, setting the `SelectionPolicy` property enables selection of entire rows or columns by clicking on the row or column label. When the user clicks on the column label, the column display (including the label) is reversed to highlight the selection. Similarly, when the user clicks on the row label, the row display (including the label) is reversed and the selection is highlighted.

You can configure the table not to highlight the label by using the following line of code:

```
table.setSelectIncludeLabels(false);
```

You can also change the default highlighting colors by setting the `SelectedForeground` and `SelectedBackground` properties. See [Customizing Cell Selection](#), in Chapter 6, for more information.

1.4.3 Resizing Using Labels Only

Background

By default, users can resize rows, columns, and labels by clicking their borders and dragging. You can change this functionality so that the resize capability is available only from the label. To resize a column, the user resizes its label instead of its cells. `JClass LiveTable` provides the `AllowResizeBy` property to enable this feature.

Procedure

In the modified *ExampleTable1.java*, add this line to the `ExampleTable1` class:

```
table.setAllowResizeBy(JCTableEnum.RESIZE_BY_LABELS);
```

Recompile and run the modified *ExampleTable1.java* file. The mouse cursor becomes a “resize” cursor only when it is located over the borders of the column labels.



Customer Name	Order Date	Item	Quantity (lbs.)	Price/lb.
The Cuppa	11/11/97	French Mocha	60	\$7.01
The Underground Cafe	11/14/97	Brazilian Medium	112	\$6.80
RocketFuel and Cake	10/30/97	Espresso Dark	300	\$8.02
WideEyes Coffee House	11/12/97	Colombian/Irish Cream Flavored	120	\$5.30
Jitters Caffeeino Cavem	10/01/97	Ethiopian Medium Roast	80	\$7.50
Twitchy's on the Mall	12/06/97	French Roast Kona	180	\$14.50
Quest Software Inc.	12/12/97	Colombian	22,000	\$5.28

Figure 2 A table with cell selection and exclusive label resizing. Note that the cell range of (2, 0) through (2, 2) has been selected.

Note: You can also run *ExampleTable7.java*, which already contains these changes.

1.4.4 Enabling Column Sorting

Background

It might be easier for your users to find certain information if they can sort the table based on cell values in a column. For example, that way they can find a customer name alphabetically or find large orders by sorting the “Quantity (lbs.)” column.

A simple way to allow your users to sort a row or column is to add a *trigger* that maps a column or row event onto a label. Since the program currently selects a column when you click its corresponding label, you need a way to differentiate between a selection and a call to sort the column.

Procedure

You can allow users to sort the column by using a **Shift-click** combination. Add these lines to your list of import statements in the modified *ExampleTable1.java*:

```
import com.klg.jclass.table.MouseActionInitiator;
import java.awt.event.InputEvent;
```

These will allow your program to work with different mouse events. Now, to add the action, add this line to the `ExampleTable1` class:

```
table.addAction(new MouseActionInitiator(
    MouseActionInitiator.ANY_BUTTON_MASK, InputEvent.
    SHIFT_MASK), JCTableEnum.COLUMN_SORT_ACTION);
```

When you recompile and run the program, you will see that holding down the **Shift** key and clicking a column label sorts the rows in ascending alphabetical/numerical order, based on the contents of the column.

Use SHIFT+mouse click to sort a column

Customer Name	Order Date	Item	Quantity (lbs.)	Price/lb.
The Cuppa	11/11/97	French Mocha	60	\$7.01
The Underground Cafe	11/14/97	Brazilian Medium	112	\$6.80
RocketFuel and Cake	10/30/97	Espresso Dark	300	\$8.02
WideEyes Coffee House	11/12/97	Colombian/Irish Cream Flavored	120	\$5.30
Jitters Caffeine Cavern	10/01/97	Ethiopian Medium Roast	80	\$7.50
Twitchy's on the Mail	12/06/97	French Roast Kona	160	\$14.50
Quest Software Inc.	12/12/97	Colombian	22,000	\$5.28

Customer Name	Order Date	Item	Quantity (lbs.)	Price/lb.
The Underground Cafe	11/14/97	Brazilian Medium	112	\$6.80
Quest Software Inc.	12/12/97	Colombian	22,000	\$5.28
WideEyes Coffee House	11/12/97	Colombian/Irish Cream Flavored	120	\$5.30
RocketFuel and Cake	10/30/97	Espresso Dark	300	\$8.02
Jitters Caffeine Cavern	10/01/97	Ethiopian Medium Roast	80	\$7.50
The Cuppa	11/11/97	French Mocha	60	\$7.01
Twitchy's on the Mail	12/06/97	French Roast Kona	160	\$14.50

Use SHIFT+mouse click to sort a column

Figure 3 Before enabling column sorting, when the third column's label was clicked, all column cells were selected (left). After inserting the code, Shift-clicking the column's label resulted in an alphabetical sort (right).

Note: You can also run *ExampleTable8.java*, which already contains these changes.

1.5 Proceeding from Here

This exercise has given you a simple overview of some of the types of things you can do with JClass LiveTable.

- For detailed information on the design elements of JClass LiveTable, see [Building a Table](#), in Chapter 2. [Appendix B, JClass LiveTable Property Listing](#), contains the JClass LiveTable Properties in table format.
- To learn about using the new JClass LiveTable data model, see [Working with Table Data](#), in Chapter 3, and [Displaying and Editing Cells](#), in Chapter 4.

- To learn about the `com.klg.jclass.util` package in *com.klg.jclass.util*, which has special capabilities for working with mathematical objects, see [Adding Formulas to JClass LiveTable](#), in Chapter 5.
- To learn about user-interaction with JClass LiveTable, see [Programming User Interactivity](#), in Chapter 6.
- To try this same tutorial in a JavaBeans development environment, see [JClass LiveTable Beans and IDEs](#), in Chapter 9.

You can find many more examples of ways to customize and enhance applications and applets in the *demos* directory of your JClass LiveTable distribution.

1.6 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>.

Building a Table

Table Anatomy 101 ■ *Setting and Getting Properties* ■ *Preset Table Styles* ■ *Global Table Properties*
Column Width and Row Height Properties ■ *Cell Styles* ■ *Cell and Label Spanning*

Using the JClass LiveTable API, you can customize the appearance of your tables with colors, borders, custom scrollbars, and other display properties. This chapter describes the properties you can set to define the structure and appearance of your tables. The properties are set for rows, columns, and cells. See Appendix B, [JClass LiveTable Property Listing](#), for a reference summary of the properties.

Many of the table's properties are set using methods of the `JCTable` class. However, some properties are set in the data source. For more information on setting properties using methods in the data source, see [Working with Table Data](#), in Chapter 3, and [Displaying and Editing Cells](#), in Chapter 4. The following descriptions note whether setting the property from the data source is applicable.

JClass LiveTable property accessor methods are also exposed to JavaBeans-compatible IDEs through the LiveTable Bean.

2.1 Table Anatomy 101

JClass LiveTable provides a scrollable viewing area for its cells and labels.

Row Label	Column Label	Col	Current Cell		
1	Customer Name	Order Date	Item	Quantity (lbs)	Price/lb
1	The Cuppa	11/11/97	French Mocha	80	\$7.12
2	The Underground C...	11/14/97	Brazilian Medium	112	\$3.30
3	Roelof Jaol and Cal...	10/30/97	Espresso Dark	300	\$3.22
4	White Eyes Coffee H...	11/12/97	Columbian First Class	120	\$5.30
5	Jitters Catherine Cars	11/01/97	Ethiopian Medium Roast	80	\$7.50
6	Twitch's on the Mat...	12/06/97	French Roast/Kona	160	\$17.50
	Quest Software Inc...				

Figure 4 The components of a Table.

The following list defines the terminology used with JClass LiveTable:

Label

A label is a non-editable cell appearing in a row at the top or bottom of the table, or in a column at the left or right side of a table. Like cells, labels can contain text or components, or can display an image. Please refer to sections later in this chapter, starting with Section 2.4.5, [Row and Column Labels](#), for more information about labels.

Scrollbar Components

These components are created and displayed if the number of rows or columns in the table is greater than the number of rows or columns visible on the screen. They provide end-users with the ability to scroll through the entire table. You can learn more about scrollbars in [Programming User Interactivity](#), in Chapter 6.

Cell

A cell is an individual container of table data. A cell is *visible* if it is currently scrolled into view. The entire collection of displayed cells is called the *cell area*. You can find more information about defining cell appearance later in this chapter.

Current Cell

This is the cell that currently has the user input focus. End-users can enter and edit the value of this cell, unless this ability is disabled.

Cell Rendering, Editing and Management

Cell drawing and editing is handled by the `com.klg.jclass.cell` package. Specifically, cell rendering and editing are handled by the `JCLightCellRenderer`, `JCComponentCellRenderer`, and `JCCellEditor` interfaces.

Cells are drawn into the cell area by either a `JCLightCellRenderer` object that understands how to draw that specific type of data, or a `JCComponentCellRenderer` object that uses a lightweight component such as `JLabel` to render data.

If the user types or clicks a cell, and there is a `JCCellEditor` for the data type of the cell, the editor component is displayed over the cell. See [Displaying and Editing Cells](#), in Chapter 4, for more information on cell editors and renderers.

2.2 Setting and Getting Properties

There are two ways to set and retrieve `JClass LiveTable` properties:

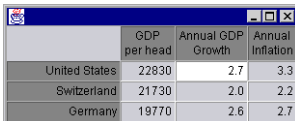
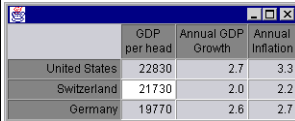
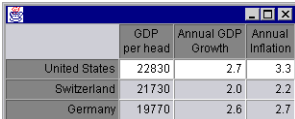
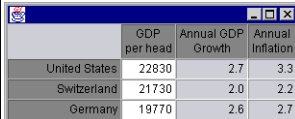
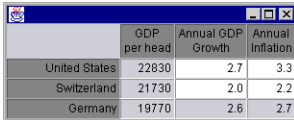
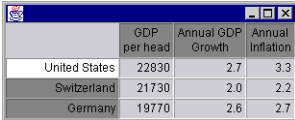
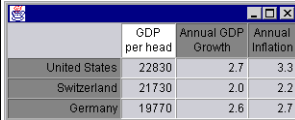
1. By calling property `set` and `get` methods in a Java program
2. By using a Java IDE at design-time (`JavaBeans`)

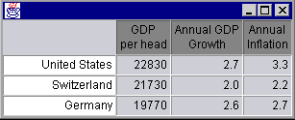
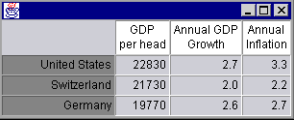
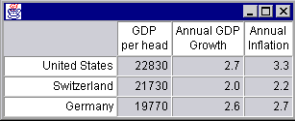
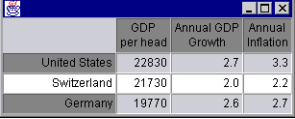
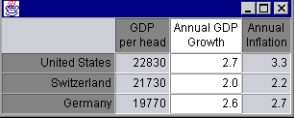
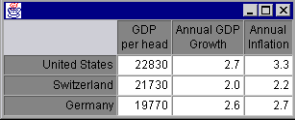
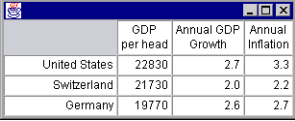
Each method changes the same table property. This manual therefore uses *properties* to discuss how features work, rather than using the method or Property Editor you might use to set that property.

2.2.1 Table Contexts

A *context* is composed of a row and column index, both zero-based. The *current context* specifies the portion of a table's cells and labels for which an application sets and retrieves properties. **Specifying a table context is part of any method that sets table properties.**

The following table outlines all table contexts. The example set the background color to white for any cells encompassed by the defined context.

Context selection	Examples	
<p>a cell</p> <p>Referenced by a row index and a column index.</p>	<p>(0,1)</p> 	<p>(1,0)</p> 
<p>all row or column cells</p> <p>Referenced by the constant <code>JCTableEnum.ALLCELLS</code> in conjunction with a row or column index.</p> <p>This does not include labels.</p>	<p>(0,JCTableEnum.ALLCELLS)</p> 	<p>(JCTableEnum.ALLCELLS,0)</p> 
<p>a range of cells</p> <p>Referenced by the location of the top-left cell/label and the location of the bottom-right cell/label in the range.</p> <p>A range be referenced as one context when defining <code>JCCellRange</code>.</p>	<p>(range); /* range defined as <code>JCCellRange(0,1,1,2) */</code></p> 	
<p>a row or column label</p> <p>Referenced by the constant <code>JCTableEnum.LABEL</code> in conjunction with a row or column index.</p>	<p>(0,JCTableEnum.LABEL)</p> 	<p>(JCTableEnum.LABEL,0)</p> 

Context selection	Examples	
<p>all row or column labels</p> <p>Referenced by both <code>JTableEnum.ALL</code> and <code>JTableEnum.LABEL</code>, the order dependent on which set of labels is being referenced.</p>	<p><code>(JTableEnum.ALL, JTableEnum.LABEL)</code></p> 	<p><code>(JTableEnum.LABEL, JTableEnum.ALL)</code></p> 
<p>all labels</p> <p>Referenced using <code>(JTableEnum.LABEL, JTableEnum.LABEL)</code>.</p>	<p><code>(JTableEnum.LABEL, JTableEnum.LABEL)</code></p> 	
<p>an entire row or column</p> <p>Referenced by the constant <code>JTableEnum.ALL</code> in conjunction with a row or column index.</p> <p>The context includes labels.</p>	<p><code>(1, JTableEnum.ALL)</code></p> 	<p><code>(JTableEnum.ALL, 1)</code></p> 
<p>all table cells</p> <p>Referenced by <code>(JTableEnum.ALLCELLS, JTableEnum.ALLCELLS)</code>.</p> <p>The context does not include labels.</p>	<p><code>(JTableEnum.ALLCELLS, JTableEnum.ALLCELLS)</code></p> 	
<p>an entire table</p> <p>Referenced by <code>(JTableEnum.ALL, JTableEnum.ALL)</code>.</p> <p>The context includes labels.</p>	<p><code>(JTableEnum.ALL, JTableEnum.ALL)</code></p> 	

2.2.2 Setting Table Properties with Java Code

When setting table properties, you work with either general table properties or Cell Styles. Cell Styles affect specific cell appearance and behavior settings for elements such as color, typefaces, border types, editability and cell text/image alignment. All other non-cell properties are handled as table-wide settings. To learn what can be defined with Cell Styles, please refer to Section 2.6, [Cell Styles](#), later in this chapter.

Setting Regular Cell and Label Properties

Setting cell and label properties that are not handled with Cell Styles involve the straightforward use of set and get methods. Every JClass LiveTable property has a set and get method associated with it.

For example, to set the value of the `PixelHeight` property to a value of 60 for all labels and the first non-label row, the `setPixelHeight()` method is called:

```
table.setPixelHeight(JCTableEnum.LABEL,60);
table.setPixelHeight(0,60);
```

As another example, to set the value of the `PixelWidth` property to a value of 90 for all labels and the first non-label row, the `setPixelWidth()` method is called:

```
table.setPixelWidth(JCTableEnum.LABEL,90);
table.setPixelWidth(0,90);
```

You can also set properties for the entire table. For example, use the `MarginHeight()` and `MarginWidth()` properties to set the distance between cell borders and cell contents:

```
table.setMarginHeight(10);
table.setMarginWidth(10);
```

Setting Cell Style Properties

Setting Cell Style properties involves the implementation of the `CellStyleModel` interface. This interface provides all information that the cell editors/renderers use.

JClass LiveTable includes the `JCCellStyle` class, which is the default implementation of `CellStyleModel`. Also included are default look and feel settings for labels and cells.

For example, the following sample code adopts the default values set in `JCCellStyle`, but changes the cell colors to black (background) and yellow (text), and applies this change to cell(2, 2):

```
JCCellStyle cellcolors = new JCCellStyle();
cellcolors.setBackground(Color.black);
cellcolors.setForeground(Color.yellow);
table.setCellStyle(1, 1, cellcolors);
```

In this example, the code acquires the default label look and feel for the particular operating system you are using. Then, the foreground and background colors are changed for all labels displayed in the table:

```
CellStyleModel labelStyle = table.getDefaultLabelStyle();
labelStyle.setBackground(Color.blue);
labelStyle.setForeground(Color.white);
```

Most properties can be applied to individual cells as well as ranges. You can also set properties for a range of cells defined by a `JCCellRange`.

The following example sets a property to a range of cells using `JCCellRange`:

```
JCCellRange range = new JCCellRange(0,3,2,4);
JCCellStyle cell = new JCCellStyle();
cell.setBackground(Color.red);
table.setCellStyle(range, cell);
```

For more information about Cell Styles, please see Section 2.6, [Cell Styles](#), later in this chapter.

2.2.3 Setting Properties with a Java IDE at Design-Time

`JClass LiveTable` can be used with a Java Integrated Development Environment (IDE), and its properties can be manipulated at design time. Consult the IDE documentation for details on how to load third-party Bean components into the IDE.

See [JClass LiveTable Beans and IDEs](#), in Chapter 9, for complete details on using `JClass LiveTable`'s `JavaBeans` in IDEs.

2.3 Preset Table Styles

You can quickly build a standard table with a number of default settings by using the `JCListTable` class. The preset features of this class affect:

- Cell selection: when users click a single cell, the entire row is selected.
- Label selection: labels are not included in selections.
- Resizing: the table's cell sizes can only be changed by dragging label borders.
- Traversal: individual cells are traversable.

These settings are overridden by any properties you specifically set later on in your program.

To view the `JCListTable` class in action, please look at the Cars example in the `JCLASS_HOME/examples/table/layout/` directory, and the Stocks demo in the `JCLASS_HOME/demos/table/stocks/` directory.

2.4 Global Table Properties

The following sections outline all properties that globally affect the appearance of your table. When any of these properties are set, they are set for the entire table.

2.4.1 Focus Rectangle Appearance

The focus rectangle visually informs the user which cell currently has the table's focus. You can change the color of the focus rectangle by using the `setFocusColor()` method. For example:

```
setFocusColor(Color.blue);
```

Using the `setFocusIndicator()` method lets you set the type of focus indicator used. Valid indicators are:

```
FOCUS_NONE  
FOCUS_HIGHLIGHT  
FOCUS_RECTANGLE  
FOCUS_THIN_RECTANGLE  
FOCUS_DASHED_RECTANGLE
```

2.4.2 Screen Cursor Type

Use the `setCursor()` method to determine which AWT cursor type is used in your table. If cursor tracking is set to `false`, then a constant cursor is used (cursor tracking can be used to change the cursor appearance, depending over which part of the table the cursor is). By default, `TrackCursor` is set to `true`.

2.4.3 Scrollbars

`JClass LiveTable` offers control over the appearance and behavior of scrollbars. This section outlines how to program the appearance of scrollbars. For information about programming scrollbar behavior, please refer to [Table Scrolling](#), in Chapter 6.

Positioning Scrollbars

The way scrollbars should be attached to the table depends on the style of table you need for your application. Standard-style tables attach the scrollbars to the cell/label area and move them to match changes to the size of the visible area.

The `HorizSBPosition` property sets how the horizontal scrollbar is attached to the table. Similarly, `VertSBPosition` sets how the vertical scrollbar is attached to the table.

- When set to `JCTableEnum.POSITION_BY_CELLS` (default), the scrollbar is attached to the cell/label viewport (that is, the cells that are visible).
- When set to `JCTableEnum.POSITION_BY_SIDE`, the scrollbar is attached to the side of the table (that is, the whole of the table).

`HorizSBAttachment` sets how the end of the horizontal scrollbar is attached to the table. When set to `JCTableEnum.SIZE_TO_CELLS` (default), the scrollbar ends at the edge of the visible cells. When set to `JCTableEnum.SIZE_TO_TABLE`, the scrollbar ends at the edge of the table.

To specify standard-style table scrollbars, leave the position and attachment properties at their default values.

`HorizSBOffset` and `VertSBOffset` specify the offset between the scrollbars and the table (default: 0 pixels). This offset usually applies to the space between the scrollbars and the table's cells/labels. However, when the scrollbars are attached to the side of the component, it can also apply to the space between the scrollbars and the side of the component, and only when there is space between the last row/column and the edge of the component.

Setting the Top Row and Left Column

When a table initially appears, you can set it so that a particular row and column are set as the top and left. Scrolling is set up automatically. Use `setTopRow()` and `setLeftColumn()` to define the top row and left-most column. This value is updated as a user scrolls through a table.

Setting Scrollbar Display Conditions

By default, `JClass LiveTable` displays each scrollbar only when the table is larger than the number of rows/columns visible on the screen. To display a scrollbar at all times, set `HorizSBDisplay` and/or `VertSBDisplay` to `JCTableEnum.SCROLLBAR_ALWAYS`. Set them to `JCTableEnum.SCROLLBAR_NEVER` to completely disable the scrollbar display. To display scrollbars only when the table size is greater than the viewing area, set them to `JCTableEnum.SCROLLBAR_AS_NEEDED`.

Note: Although scrollbars are removed, a user can still scroll with the keyboard. See [Managing Table Scrolling](#), in Chapter 6, for complete information on disabling interactive scrolling.

Using your own Scrollbar Component

You may want to use a scrollbar component other than the default provided with `JClass LiveTable`. To do this, use the `setVertSB()` and `setHorizSB()` methods. The scrollbar must be a `JScrollbar` instance.

2.4.4 Cell Selection Colors

When users select a cell or a range of cells, it often helps to highlight them. This section outlines how to control the appearance of selected cells. For information about programming cell selection behavior, please refer to [Cell Selection](#), in Chapter 6.

Setting Cell Selection Colors

The background and foreground colors used for selected cells are specified by `setSelectedBackground()` and `setSelectedForeground()`. By default, selected cells are displayed in reverse video (i.e., the normal background and foreground color values have been swapped). The current cell displays the selection colors in its border.

Using the previous methods requires you to select a specific foreground or background color. Instead of committing to one color, you can also use color mode methods that allow you to define selection colors by associating them with other foreground and background colors.

Use `setSelectedBackgroundMode()` to set how selected background colors are determined. Valid modes include:

- `USE_SELECTED_BACKGROUND`: the selected background color is the same as the color defined in the `SelectedBackground` property.
- `USE_CELL_BACKGROUND`: the selected background is the same as the cell background color.
- `USE_CELL_FOREGROUND`: the background and foreground colors are inverted.

Use `setSelectedForegroundMode()` to set how selected foreground colors are determined. Valid modes include:

- `USE_SELECTED_FOREGROUND`: the selected foreground color is the same as the color defined in the `SelectedForeground` property.
- `USE_CELL_FOREGROUND`: the selected background is the same as the cell foreground color.
- `USE_CELL_BACKGROUND`: the background and foreground colors are inverted.

2.4.5 Row and Column Labels

A row or column label is a non-editable cell that identifies the row or column to the user. Row and column label values are set in the data source (see [Working with Table Data](#), in Chapter 3). By default, row and column labels are displayed in your table, regardless of whether you have specified contents for the labels in the data source (they will be empty if there are no labels defined in the data source). To prevent row and column labels from displaying, you must use the methods:

```
table.setRowLabelDisplay(false);
table.setColumnLabelDisplay(false);
```

Placing Labels

You can specify the positioning of row/column labels on the screen using the `setRowLabelPlacement()` and `setColumnLabelPlacement()` methods. If you insert the placement methods in the `table.setColumnLabelPlacement(placement)` or `table.setRowLabelPlacement(placement)` statements, valid values include:

Column and Row Placement	Example																																																
<code>JCTableEnum.PLACE_TOP</code> <code>JCTableEnum.PLACE_RIGHT</code> The labels are displayed at the top and to the right of the table (default).	<table border="1"> <thead> <tr> <th>Customer Name</th> <th>Order Date</th> <th>Item</th> <th>Quantity (lb)</th> <th>Price/lb.</th> <th></th> </tr> </thead> <tbody> <tr> <td>The Cuppa</td> <td>11/11/97</td> <td>French Mocha</td> <td>60</td> <td>\$7.01</td> <td>1</td> </tr> <tr> <td>The Underg...</td> <td>11/14/97</td> <td>Brazilian Mocha</td> <td>112</td> <td>\$6.80</td> <td>2</td> </tr> <tr> <td>RocketFuel...</td> <td>10/30/97</td> <td>Espresso D...</td> <td>300</td> <td>\$8.02</td> <td>3</td> </tr> <tr> <td>WideEyes C...</td> <td>11/12/97</td> <td>Colombian M...</td> <td>120</td> <td>\$5.30</td> <td>4</td> </tr> <tr> <td>Jitters Caff...</td> <td>10/01/97</td> <td>Ethiopian M...</td> <td>80</td> <td>\$7.50</td> <td>5</td> </tr> <tr> <td>Twitchy's or...</td> <td>12/06/97</td> <td>French Roast</td> <td>160</td> <td>\$14.50</td> <td>6</td> </tr> <tr> <td>Quest Soft...</td> <td>12/12/97</td> <td>Colombian</td> <td>22,000</td> <td>\$5.28</td> <td>7</td> </tr> </tbody> </table>	Customer Name	Order Date	Item	Quantity (lb)	Price/lb.		The Cuppa	11/11/97	French Mocha	60	\$7.01	1	The Underg...	11/14/97	Brazilian Mocha	112	\$6.80	2	RocketFuel...	10/30/97	Espresso D...	300	\$8.02	3	WideEyes C...	11/12/97	Colombian M...	120	\$5.30	4	Jitters Caff...	10/01/97	Ethiopian M...	80	\$7.50	5	Twitchy's or...	12/06/97	French Roast	160	\$14.50	6	Quest Soft...	12/12/97	Colombian	22,000	\$5.28	7
Customer Name	Order Date	Item	Quantity (lb)	Price/lb.																																													
The Cuppa	11/11/97	French Mocha	60	\$7.01	1																																												
The Underg...	11/14/97	Brazilian Mocha	112	\$6.80	2																																												
RocketFuel...	10/30/97	Espresso D...	300	\$8.02	3																																												
WideEyes C...	11/12/97	Colombian M...	120	\$5.30	4																																												
Jitters Caff...	10/01/97	Ethiopian M...	80	\$7.50	5																																												
Twitchy's or...	12/06/97	French Roast	160	\$14.50	6																																												
Quest Soft...	12/12/97	Colombian	22,000	\$5.28	7																																												

Column and Row Placement	Example																																																
JTableEnum.PLACE_TOP JTableEnum.PLACE_LEFT The labels are displayed at top and to the left of the table.	<table border="1"> <thead> <tr> <th></th> <th>Customer Name</th> <th>Order Date</th> <th>Item</th> <th>Quantity (lb.)</th> <th>Price/lb.</th> </tr> </thead> <tbody> <tr><td>1</td><td>The Cuppa</td><td>11/11/97</td><td>French Moc</td><td>60</td><td>\$7.01</td></tr> <tr><td>2</td><td>The Underg</td><td>11/14/97</td><td>Brazilian Me</td><td>112</td><td>\$6.80</td></tr> <tr><td>3</td><td>RocketFuel</td><td>10/30/97</td><td>Espresso D</td><td>300</td><td>\$8.02</td></tr> <tr><td>4</td><td>WideEyes Q</td><td>11/12/97</td><td>Colombian</td><td>120</td><td>\$5.30</td></tr> <tr><td>5</td><td>Jitters Caffee</td><td>10/01/97</td><td>Ethiopian M</td><td>80</td><td>\$7.50</td></tr> <tr><td>6</td><td>Twitchy's on</td><td>12/06/97</td><td>French Roas</td><td>160</td><td>\$14.50</td></tr> <tr><td>7</td><td>Quest Softw</td><td>12/12/97</td><td>Colombian</td><td>22,000</td><td>\$5.28</td></tr> </tbody> </table>		Customer Name	Order Date	Item	Quantity (lb.)	Price/lb.	1	The Cuppa	11/11/97	French Moc	60	\$7.01	2	The Underg	11/14/97	Brazilian Me	112	\$6.80	3	RocketFuel	10/30/97	Espresso D	300	\$8.02	4	WideEyes Q	11/12/97	Colombian	120	\$5.30	5	Jitters Caffee	10/01/97	Ethiopian M	80	\$7.50	6	Twitchy's on	12/06/97	French Roas	160	\$14.50	7	Quest Softw	12/12/97	Colombian	22,000	\$5.28
	Customer Name	Order Date	Item	Quantity (lb.)	Price/lb.																																												
1	The Cuppa	11/11/97	French Moc	60	\$7.01																																												
2	The Underg	11/14/97	Brazilian Me	112	\$6.80																																												
3	RocketFuel	10/30/97	Espresso D	300	\$8.02																																												
4	WideEyes Q	11/12/97	Colombian	120	\$5.30																																												
5	Jitters Caffee	10/01/97	Ethiopian M	80	\$7.50																																												
6	Twitchy's on	12/06/97	French Roas	160	\$14.50																																												
7	Quest Softw	12/12/97	Colombian	22,000	\$5.28																																												
JTableEnum.PLACE_BOTTOM JTableEnum.PLACE_RIGHT The labels are displayed at the bottom and to the right of table.	<table border="1"> <tbody> <tr><td>The Cuppa</td><td>11/11/97</td><td>French Moc</td><td>60</td><td>\$7.01</td><td>1</td></tr> <tr><td>The Underg</td><td>11/14/97</td><td>Brazilian Me</td><td>112</td><td>\$6.80</td><td>2</td></tr> <tr><td>RocketFuel</td><td>10/30/97</td><td>Espresso D</td><td>300</td><td>\$8.02</td><td>3</td></tr> <tr><td>WideEyes Q</td><td>11/12/97</td><td>Colombian</td><td>120</td><td>\$5.30</td><td>4</td></tr> <tr><td>Jitters Caffee</td><td>10/01/97</td><td>Ethiopian M</td><td>80</td><td>\$7.50</td><td>5</td></tr> <tr><td>Twitchy's on</td><td>12/06/97</td><td>French Roas</td><td>160</td><td>\$14.50</td><td>6</td></tr> <tr><td>Quest Softw</td><td>12/12/97</td><td>Colombian</td><td>22,000</td><td>\$5.28</td><td>7</td></tr> <tr> <th>Customer Name</th> <th>Order Date</th> <th>Item</th> <th>Quantity (lb.)</th> <th>Price/lb.</th> <th></th> </tr> </tbody> </table>	The Cuppa	11/11/97	French Moc	60	\$7.01	1	The Underg	11/14/97	Brazilian Me	112	\$6.80	2	RocketFuel	10/30/97	Espresso D	300	\$8.02	3	WideEyes Q	11/12/97	Colombian	120	\$5.30	4	Jitters Caffee	10/01/97	Ethiopian M	80	\$7.50	5	Twitchy's on	12/06/97	French Roas	160	\$14.50	6	Quest Softw	12/12/97	Colombian	22,000	\$5.28	7	Customer Name	Order Date	Item	Quantity (lb.)	Price/lb.	
The Cuppa	11/11/97	French Moc	60	\$7.01	1																																												
The Underg	11/14/97	Brazilian Me	112	\$6.80	2																																												
RocketFuel	10/30/97	Espresso D	300	\$8.02	3																																												
WideEyes Q	11/12/97	Colombian	120	\$5.30	4																																												
Jitters Caffee	10/01/97	Ethiopian M	80	\$7.50	5																																												
Twitchy's on	12/06/97	French Roas	160	\$14.50	6																																												
Quest Softw	12/12/97	Colombian	22,000	\$5.28	7																																												
Customer Name	Order Date	Item	Quantity (lb.)	Price/lb.																																													
JTableEnum.PLACE_BOTTOM JTableEnum.PLACE_LEFT The labels are displayed at the bottom and to the left of the table (reversed default).	<table border="1"> <tbody> <tr><td>1</td><td>The Cuppa</td><td>11/11/97</td><td>French Moc</td><td>60</td><td>\$7.01</td></tr> <tr><td>2</td><td>The Underg</td><td>11/14/97</td><td>Brazilian Me</td><td>112</td><td>\$6.80</td></tr> <tr><td>3</td><td>RocketFuel</td><td>10/30/97</td><td>Espresso D</td><td>300</td><td>\$8.02</td></tr> <tr><td>4</td><td>WideEyes Q</td><td>11/12/97</td><td>Colombian</td><td>120</td><td>\$5.30</td></tr> <tr><td>5</td><td>Jitters Caffee</td><td>10/01/97</td><td>Ethiopian M</td><td>80</td><td>\$7.50</td></tr> <tr><td>6</td><td>Twitchy's on</td><td>12/06/97</td><td>French Roas</td><td>160</td><td>\$14.50</td></tr> <tr><td>7</td><td>Quest Softw</td><td>12/12/97</td><td>Colombian</td><td>22,000</td><td>\$5.28</td></tr> <tr> <th>Customer Name</th> <th>Order Date</th> <th>Item</th> <th>Quantity (lb.)</th> <th>Price/lb.</th> <th></th> </tr> </tbody> </table>	1	The Cuppa	11/11/97	French Moc	60	\$7.01	2	The Underg	11/14/97	Brazilian Me	112	\$6.80	3	RocketFuel	10/30/97	Espresso D	300	\$8.02	4	WideEyes Q	11/12/97	Colombian	120	\$5.30	5	Jitters Caffee	10/01/97	Ethiopian M	80	\$7.50	6	Twitchy's on	12/06/97	French Roas	160	\$14.50	7	Quest Softw	12/12/97	Colombian	22,000	\$5.28	Customer Name	Order Date	Item	Quantity (lb.)	Price/lb.	
1	The Cuppa	11/11/97	French Moc	60	\$7.01																																												
2	The Underg	11/14/97	Brazilian Me	112	\$6.80																																												
3	RocketFuel	10/30/97	Espresso D	300	\$8.02																																												
4	WideEyes Q	11/12/97	Colombian	120	\$5.30																																												
5	Jitters Caffee	10/01/97	Ethiopian M	80	\$7.50																																												
6	Twitchy's on	12/06/97	French Roas	160	\$14.50																																												
7	Quest Softw	12/12/97	Colombian	22,000	\$5.28																																												
Customer Name	Order Date	Item	Quantity (lb.)	Price/lb.																																													

Defining Label Spacing

Normally, there is no space between labels and the cell area. The `RowLabelOffset` property specifies the distance in pixels between the row labels and the cell area. Similarly, the `ColumnLabelOffset` property specifies the distance in pixels between the column labels and the cell area. If you specify a negative value, the cell area overlaps the labels.

Offset value examples

```
ColumnLabelOffset(0);
RowLabelOffset(0);
```

	Column 1	Column 2	Column 3
Row 1			
Row 2			
Row 3			

Offset value examples

```
ColumnLabelOffset(15);  
RowLabelOffset(15);
```

	Column 1	Column 2	Column 3
Row 1	I		
Row 2			
Row 3			

```
ColumnLabelOffset(-10);  
RowLabelOffset(-10);
```

	Column 1	Column 2	Column 3
Row 1	I		
Row 2			
Row 3			

2.4.6 Cell and Label Border Width

The width of the borders around the cells and labels is specified by the `setCellBorderWidth()` method. This method's actions apply to the entire table. By default, the borders are 1 pixel wide. The following table demonstrates the effect of different bordercell widths:

CellBorderWidth Examples

```
table.setCellBorderWidth(2);
```

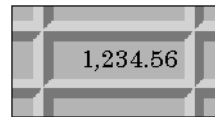
sets the bordercell width for all cells and labels to a value of two pixels



A table with a single cell containing the number "1,234.56". The cell and its label are surrounded by a thin, light gray border, demonstrating a border width of two pixels.

```
table.setCellBorderWidth(5);
```

sets the bordercell width for all cells and labels to a value of five pixels



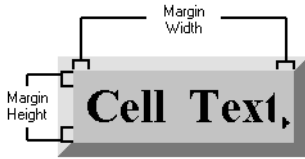
A table with a single cell containing the number "1,234.56". The cell and its label are surrounded by a thick, light gray border, demonstrating a border width of five pixels.

2.4.7 Cell and Label Margins

The `MarginWidth` and `MarginHeight` properties alter the space between the cell borders and the contents of cells.

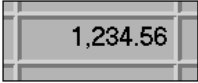

The `MarginWidth` property sets the distance (in pixels) between the inside edge of the cell border and the left and right edge of the cell's contents (default: 2). The `MarginHeight` property specifies the margin (in pixels) between the inside edge of the cell border and the top and bottom edge of the cell's contents (default: 1).

These properties affect all cells and labels in the table – **margins cannot be set for individual cells.**



The following table demonstrates the effect of different margin height and width settings:

Cell and Label Margin Examples

<pre>table.setMarginHeight(2); table.setMarginWidth(5);</pre> <p>sets the margin height to 2 and the width to 5</p>	
<pre>table.setMarginHeight(10); table.setMarginWidth(10);</pre> <p>sets the margin height and width to 10</p>	

2.4.8 Component Borders

The `ComponentBorderWidth` property sets the spacing between the border of a table's cells and components that are inserted into them. By default, this property is set to 0.

2.4.9 Frame Border Attributes

The `FrameBorder` property is an instance of `CellBorderModel`, and sets the border surrounding the cell and label areas.

The `FrameBorderWidth` property specifies the thickness of the border surrounding the cell and label areas. Its default value is 0 (no frame border).

Border colors are calculated using the table's background color.

The following table outlines all the valid frameborder types, and demonstrates frameborder widths. The `FrameBorderWidth` property, which specifies the thickness of the border surrounding the cell and label areas, has been set to a value of 6. The code in each

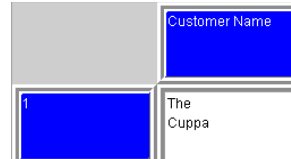
cell is the `CellBorderModel` value, which is used in the statement:
`table.setFrameBorder(new JCellBorder(value)).`

FrameBorder Attribute Examples

`JTableEnum.BORDER_ETCHED_IN`
creates a border that appears set in



`JTableEnum.BORDER_ETCHED_OUT`
creates a raised border



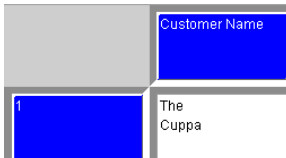
`JTableEnum.BORDER_FRAME_IN`
creates a frame border whose enclosed cells
and labels appear set in



`JTableEnum.BORDER_FRAME_OUT`
creates a frame border whose enclosed
cells and labels appear raised



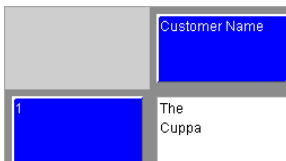
`JTableEnum.BORDER_IN`
creates a border whose enclosed cells and
labels appear set in



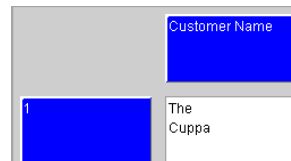
`JTableEnum.BORDER_OUT`
creates a border whose enclosed cells and
labels appear raised



`JTableEnum.BORDER_PLAIN`
creates a plain frame border

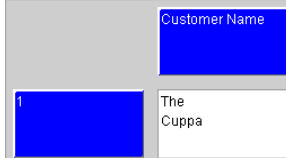


`JTableEnum.BORDER_THIN`
creates a thin frame border



FrameBorder Attribute Examples

JCTableEnum.BORDER_NONE
creates no frame border (default)



2.4.10 Row and Column Definition

Determining the Number of Rows/Columns

The `NumRows` and `NumColumns` properties are set using methods in the data source. To retrieve these values, use the `JCVectorDataSource.getNumRows()` and `JCVectorDataSource.getNumColumns()` methods. Please see [Setting Stock Data Source Properties](#), in Chapter 3, for information on setting these properties in the data source.

The number of rows/columns must be greater than the number of frozen rows/columns. For more information on frozen rows/columns, see [‘Freezing’ Rows and Columns](#).

Setting and Getting Visible Rows and Columns

The number of rows and columns currently visible in the window is specified by the `VisibleRows` and `VisibleColumns` properties.¹

You can force the table to display a particular number of rows or columns by calling `setVisibleRows()` and `setVisibleColumns()`.

To retrieve the values of `VisibleRows` or `VisibleColumns`, call the `getVisibleRows()` and `getVisibleColumns()` methods. These methods return the number of visible non-frozen rows or columns. These values determine the preferred size of the table and *are not updated dynamically* as a user resizes the table.

To get live values of the table, use `getNumVisibleRows()` and `getNumVisibleColumns()`, which return the total number of visible rows or columns.

To work with cells instead of rows or columns, use the `getVisibleCells()` method, which returns the range of non-frozen visible cells.

1. Rows/columns that are only partially visible are also included in the value of these properties.

Displaying the Entire Table

To display the entire table, set `VisibleRows` and `VisibleColumns` to `JCTableEnum.NOVALUE`. Setting either property to `NOVALUE` sets a special flag that causes the table to attempt to resize to make all rows or columns visible.

Swapping Rows or Columns

You can make two rows or columns switch places by using the `swapRows()` and `swapColumns()` methods. For example, to swap rows 3 and 9:

```
table.swapRows(3,9)
```

These methods do not affect the data source, but use an internal mapping table to keep track of row and column locations.

To reset the rows or columns to their original locations, based on the data source, use the `resetSwappedRows()` or `resetSwappedColumns()` methods.

'Freezing' Rows and Columns

An application can make rows and columns non-scrollable by using the `FrozenRows` and `FrozenColumns` properties. You can use frozen rows or columns to hold important information on the screen as a user scrolls through the table (such as totals at the bottom of a table). You could also use frozen rows or columns as additional rows or columns that act like labels; see Section 2.7.1, [Using Spanning to Create Multiline Headers](#) for an example.

- `setFrozenRows()` specifies the number of rows held at the top or bottom of the window and not scrolled. The default value is zero.
- `setFrozenColumns()` specifies the number of columns held at the left or right side of the window and not scrolled. The default is zero.

Frozen rows/columns always start from the beginning of the table. By default, they are editable and traversable, but not sorted and cannot be dragged. The following figure shows an example of frozen rows.

Sample	Temperature	Rainfall	Humidity
Median Values	37.0	11.46	32.0
Average Values	32.6	14.9	34.4
Sample 34	42.3	15.9	23.4
Sample 35	33.2	10.9	23.0
Sample 36	53.3	7.7	40.2
Sample 37	53.2	7.9	42.2
Sample 38	54.7	19.6	23.2
Sample 39	35.0	11.9	44.3

Figure 5 Visible and Frozen Rows and Columns- note absence of scrollbar to right of frozen rows.

Setting frozen rows or columns sets the number of columns from the left or the number of rows from the top. For example:

```
table.setFrozenRows(2);
```

freezes the first two rows of the table, and

```
table.setFrozenColumns(4);
```

freezes the first four columns of the table.

If you want to freeze a single column or row in the middle of the table, you can easily move the specified row or column to the beginning of the table by using the `swapRows()` or `swapColumns()` method (described above), then freeze the row or column.

To move and freeze more than one column or row, you will have to call the `moveRows()` or `moveColumns()` method *in the data source* (see [Using Stock Data Sources](#), in Chapter 3) to move the desired rows/columns to the beginning of the table, then set `FrozenRows` or `FrozenColumns` to the number of rows/columns that you want to freeze.

Placing Frozen Rows/Columns

You can place frozen rows at either the top or bottom of the table. Frozen columns can be placed at either the left or right of the table. The placement of frozen rows/columns does not affect the location of the rows/columns in the data source.

To change the placement of the frozen rows, set the `FrozenRowPlacement` property to either `JTableEnum.PLACE_TOP` or `JTableEnum.PLACE_BOTTOM`.

To change the placement of all frozen columns, set the `FrozenColumnPlacement` property to either `JTableEnum.PLACE_LEFT` or `JTableEnum.PLACE_RIGHT`.

2.4.11 Controlling Cell Editor Size

The table can control the size of a cell editing component using the `EditHeightPolicy` and `EditWidthPolicy` properties. Each of these properties can take one of three values:

- `JTableEnum.EDIT_SIZE_TO_CELL`: resize the component to fit the Table's cell size.
- `JTableEnum.EDIT_ENSURE_MINIMUM_SIZE`: resize the component to its minimum size.
- `JTableEnum.EDIT_ENSURE_PREFERRED_SIZE`: resize the cell to editing component's preferred size.

These properties allow the table to have better control over cell editors created using the `com.klg.jclass.cell.JCCellEditor` interface. For more information about cell editors, see [Displaying and Editing Cells](#), in Chapter 4.

2.5 Column Width and Row Height Properties

By default, JClass LiveTable sets the height of rows to display one line of text. The width of columns is set by default to display 10 characters of text. If a cell value, image file, or component does not fit in its cell, the cell displays clipping arrows by default. Each row can have its own height, and each column its own width.

JClass LiveTable provides two different ways to specify row height and column width: *character* and *pixel*. Character specification determines the height/width by the number of characters or lines that the row/column can display. Pixel specification determines the height/width by the explicit number of pixels.

Only one method can be used for a row or column. Pixel specification overrides character specification.

Note: When users interactively resize rows/columns, the row height/column width is specified by pixel regardless of how your application specified it.

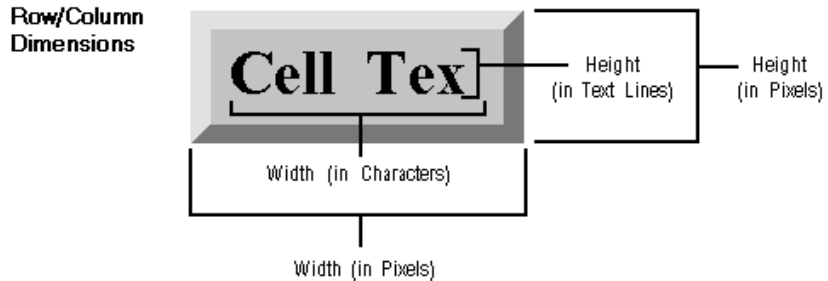


Figure 6 The difference between Character and Pixel Row/Column specification.

2.5.1 Character Height and Width

The `CharWidth` property specifies the number of characters a column can display. `CharHeight` specifies the number of lines of text a row can display. For these properties to control row height/column width, `PixelWidth` and `PixelHeight` must be set to `JTableEnum.NOVALUE`.

To determine the pixel dimensions of a row or column whose height/width was set by `CharWidth` or `CharHeight`, use the `getColumnPixelWidth()` or `getColumnPixelHeight()` methods.

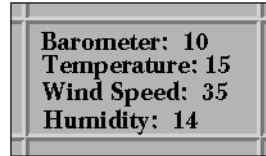
The following table demonstrates different character height and width settings:

For Column Width:

`table.setCharHeight(0,1);`
sets the first row's height to 1 character



`table.setCharHeight(0,4);`
sets the first row's height to four characters

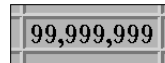


For Row Height:

`table.setCharWidth(4,3);`
sets the fifth column's width to 3 characters



`table.setCharWidth(4,10);`
sets the fifth column's width to 10 characters



Character specification is convenient when you know how many characters you want a row/column to display. It works best with non-proportional¹ fonts because JClass LiveTable uses the widest character along with the largest ascender/descender to guarantee that the specified number of characters will fit in the cell or label.

2.5.2 Absolute Pixel Height and Width

`PixelWidth` and `PixelHeight` specify column width and row height in pixels. You can set these properties to an explicit pixel value using `JCTableEnum.NOVALUE` or `JCTableEnum.VARIABLE` (this value is discussed in detail in the following section).

Unless set to `JCTableEnum.NOVALUE` (default), these properties override the `CharWidth` and `CharHeight` properties. The next illustration shows setting `PixelHeight` to a pixel value.

Absolute Pixel Height Examples

`table.setPixelHeight(4,15);`
sets the fifth row's height to 15 pixels



1. All of the characters in a fixed-width font have the same width

Absolute Pixel Height Examples

`table.setPixelHeight(4,30);`
sets the fifth row's height to 30 pixels



2.5.3 Variable Pixel Height and Width

An application can have `JClass LiveTable` automatically size rows and columns to fit the contents of the table by setting `PixelWidth` and `PixelHeight` to `JCTableEnum.VARIABLE`. As your application changes table attributes affecting the cells' contents, the table will resize the rows and columns to fit.¹

When a cell contains a component, `JClass LiveTable` sizes the cell to fit the component's preferred size.

To determine the pixel dimensions of a row or column with variable height or width, call the `getRowPixelHeight()` and `getColumnPixelWidth()` methods.

Defining How Much of the Table is Used in Pixel Estimates

By default, the `JCTableEnum.VARIABLE` value, when used with `PixelHeight` and `PixelWidth`, uses the entire row or column to calculate pixel dimensions.

Using `VARIABLE` with large tables can result in general table slowdowns due to the large number of cells involved in the height calculation. For large tables, use the `JCTableEnum.VARIABLE_ESTIMATE` value instead, which sets the pixel dimension to the highest value found in a range that you define.

You can explicitly control the range of cells used in the variable height calculation by using `setVariableEstimateCount()`. Typically, this value is set to the number of cells expected to be visible at any time.

Changing Variable Row and Height Dimensions to Fixed Values

Setting `PixelHeight` and `PixelWidth` to `JCTableEnum.AS_IS` does not change the pixel dimensions, and makes the current height and width settings fixed values.

Additionally, if you have set your row and column dimensions to be of variable height and width, and the user interactively resizes a row or column, the `PixelWidth` and `PixelHeight` values are converted to fixed values.

1. When width or height are set to zero, the row/column becomes hidden.

2.5.4 Maximum and Minimum Pixel Height and Width

While you can work with varying pixel height and width dimensions, you can still set the absolute maximum and minimum pixel dimensions for a table.

Use `setMaxHeight()` and `setMinHeight()` to determine the maximum height of any or all rows, all column labels or the whole table. Likewise, use `setMaxWidth()` and `setMinWidth()` to determine the minimum width of any or all columns, all row labels, or the whole table.

2.5.5 Displaying and Editing Multiple Lines in Cells

When you set the height and width of your cells, you adjust how much of the data can be displayed in the cell. If your cell contains text, then `JClass LiveTable` makes it possible for you to display and edit multiple lines.

For cell rendering, if the data displayed in the cells contains a newline character (`\n`), the cell is automatically displayed as a multiline cell.

For cell editing, by default, text is edited on a single line. For multiline editing, you must set the multiline editor. To set a multiline editor, you need to set the `Cell Style`'s editor properties. Create a `Cell Style` and set the editor for it, then call `setCellStyle()` with a row, column, or range. Please refer to Section 2.6, [Cell Styles](#), for more information about setting editor properties.

2.5.6 Using Row Height and Width to Hide Rows and Columns

An application can “hide” rows and columns from the end-user by setting the `PixelHeight/PixelWidth` properties to zero pixels (the current cell should not be in the hidden row/column). Though the row/column appears to have vanished, the application can set attributes or change cell values.

Note: The recommended way of hiding rows and columns is to set the boolean value of `setRowHidden()` and `setColumnHidden()` to `true`.

Customer Name	Order Date	Item	Quantity (lbs.)	Price/lb.
The Cuppa	11/11/97	French Mocha	60	\$7.01
The Underground Cafe	11/14/97	Brazilian Medium	112	\$6.80
RocketFuel and Cake	10/30/97	Espresso Dark	300	\$8.02
WideEyes Coffee House	11/12/97	Colombian/Irish Cream	120	\$5.30
Jitters Caffeine Cavern	10/01/97	Ethiopian Medium Roast	80	\$7.50
Twitchy's on the Mall	12/06/97	French Roast Kona	160	\$14.50
Quest Software Inc.	12/12/97	Colombian	22,000	\$5.28

Customer Name	Item	Quantity (lbs.)	Price/lb.
The Cuppa	French Mocha	60	\$7.01
The Underground Cafe	Brazilian Medium	112	\$6.80
RocketFuel and Cake	Espresso Dark	300	\$8.02
WideEyes Coffee House	Colombian/Irish Cream	120	\$5.30
Jitters Caffeine Cavern	Ethiopian Medium Roast	80	\$7.50
Twitchy's on the Mall	French Roast Kona	160	\$14.50
Quest Software Inc.	Colombian	22,000	\$5.28

Figure 7 Hiding the “Order Date” column.

2.6 Cell Styles

While the classes and properties mentioned in previous sections define table-wide or row/column properties, you can use Cell Styles to set the properties of individual cells or labels, or ranges of cells.

Every cell in a table is associated with a style that defines how the cell looks, how the data is edited, and whether the cell is traversable and editable.

2.6.1 Cell Style Properties and Implementation

A Cell Style is any object that implements the `CellStyleModel` interface. With this interface, the style properties that you can define are:

- background colors and foreground colors
- repeating background and foreground color settings
- font attributes
- horizontal and vertical text alignment
- cell border types
- cell border sides
- clip hints
- boolean editable

- editor (JCCellEditor)
- renderer (JCCellRenderer)
- data types
- cell traversal

Cell Styles make it easier to define and manage the appearance of a table. Instead of working with a myriad of visual properties for ranges of cells, you can define a particular Cell Style (which encompasses all of these properties), and then apply the style to any cells or labels.

Getting and setting Cell Styles

In order to set a Cell Style, you can use one of two methods:

```
// this applies a style to a cell
setCellStyle(int row, int column, CellStyleModel csm);
// this applies a style to a range of cells
setCellStyle(JCCellRange cr, CellStyleModel csm);
```

To retrieve the style for a cell, use:

```
CellStyleModel getCellStyle(int row, int column);
```

2.6.2 Defining Your Own or Changing Built-In Cell Styles

You can easily modify Cell Styles by making property changes to the JCCellStyle implementation, as well as default cell and label styles.

Changing Cell Styles

You can change a Cell Style by creating a new JCCellStyle object, modifying the desired properties, and applying these changes with the setCellStyle() method. For example, the style for cell (2, 2) is changed by using this code:

```
JCCellStyle cs = new JCCellStyle();
cs.setBackground(Color.blue);
table.setCellStyle(2, 2, cs);
```

You can also use the getCellStyle() method to retrieve the style properties from a particular cell. Consider this example, which gets the properties of cell (0, 0), then sets the background color to red:

```
JCCellStyle cs = new JCCellStyle();
cs.setBackground(Color.blue);
table.setCellStyle(JCTableEnum.ALL, JCTableEnum.ALL, cs);

CellStyleModel csm = table.getCellStyle(0, 0);
csm.setBackground(Color.red);
```

The problem with using getCellStyle() is that the style obtained from an individual cell may not be unique to that cell. Styles can also be applied to ranges, or an entire table. In the above example, you might expect the code to produce a table whose cells have a blue background, with the exception of cell (0, 0) which should have a red background.

However, since the style you are retrieving from cell (0, 0) is used for the whole table, all cell backgrounds will be red.

If you wanted to change the background color for cell (0, 0) to red, even though that cell's style is also being used for the whole table, you can work with a unique Cell Style:

```
CellStyleModel csm = table.getUniqueCellStyle(0, 0);
csm.setBackground(Color.red);
table.setCellStyle(0, 0, csm);
```

The bottom line is that you do not need to apply specific style changes with `setCellStyle()` if you want to change all the cells that share the style. In other words, the first example which used:

```
CellStyleModel csm = table.getCellStyle(0, 0);
csm.setBackground(Color.red);
```

is correct if your intention is to set the background color to red for all cells that share the same style as cell (0, 0).

Retrieving all Styles Used in a Table

You can easily work through all Cell Styles found in a table (even without knowing what they all are) by calling `Collection getCellStyles()`. You can use this to change a property for all styles in your table. The following example performs this operation, as it retrieves all the table's styles, and changes the foreground color to blue:

```
Collection col = table.getCellStyles();
Iterator it = col.iterator();
while(it.hasNext()){
    CellStyleModel csm = (CellStyleModel)it.next();
    csm.setForeground(Color.blue);
}
```

Creating "Parent" Styles

`JClass LiveTable` allows you to create styles that inherit property values from a parent style. For example, imagine you have a style (`mySimpleStyle`) with white background and black foreground (text) settings. If you want to change the style properties for a particular cell, or cell range, but retain the original properties for the other cells, you have two choices.

- The first choice involves the creation of a copy of the style in which you are interested, changing the property, and applying it back to the cell you want changed:

```
CellStyleModel myNewStyle = (CellStyleModel)(mySimpleStyle.clone());
myNewStyle.setBackground(Color.red);
table.setCellStyle(0, 0, myNewStyle);
```

The problem with this approach is that if `mySimpleStyle` changes (for example, the font is changed), `myNewStyle` will not pick up this change. Updating styles to match changes in other styles can be tedious.

- The second option makes updates automatic, as you implement `mySimpleStyle` as the “parent” of `myNewStyle`.

```
CellStyleModel myNewStyle = new JCCellStyle(mySimpleStyle);
myNewStyle.setBackground(Color.red);
table.setCellStyle(0, 0, myNewStyle);
```

By creating a `JCCellStyle` with another style as an argument, you create a link between the new style and the original one. Any property that is changed to the new style overrides the setting that comes from the original style, and any changes made to the original style (that are not overridden) are picked up by the new style.

The following example demonstrates the relationship between parent and child styles. Here, both styles end up using the `anotherFont` typeface. However, since the foreground color in `myNewStyle` was changed to yellow, setting the `myOldStyle` foreground color to white will not affect `myNewStyle`.

```
myNewStyle.setForeground(Color.yellow);
myOldStyle.setFont(anotherFont);
myOldStyle.setForeground(Color.white);
```

The `CellStyleModel` has `getParentStyle()` and `setParentStyle()` methods in addition to using the special constructor.

2.6.3 Using and Modifying JClass LiveTable’s Built-In Styles

To define Cell Styles, use the `CellStyleModel` interface. `CellStyleModel` is an interface that defines the methods required by an object to specify the attributes of a cell.

JCCellStyle: the Default CellStyleModel Implementation

`JClass LiveTable` provides an implementation of the `CellStyleModel` interface in the `JCCellStyle` class. Creating an instance of this class in your table program is a quick way of setting up a Cell Style. It has the following defaults:

Background	System control color
Border	BORDER_IN
BorderSides	BORDERSIDE_ALL
CellBorderColor	based on the background color of the cell
ClipHints	SHOW_ALL
Data Type	null
Editable	true
Editor	null
Foreground	System control text color
Font	Dialog-Plain-12

HorizontalAlignment	LEFT
Renderer	null
RepeatBackground	NONE
RepeatForeground	NONE
Traversable	true
VerticalAlignment	TOP

JCellStyle does not specify any `DataType`, `CellEditor`, or `CellRenderer` properties. Editors and renderers are determined by the type of data in the data source if an editor/renderer is not set. Please see [Displaying and Editing Cells](#), in Chapter 4, for more information.

Pluggable Look and Feel (PLAF) Styles

There are two default styles that are used and changed by the current system's PLAF. Use these if you want your table to look and behave in accordance with the host machine's properties (e.g. `Swing Metal`, `Windows`). `DefaultLabelStyle` is automatically applied to labels, while `DefaultCellStyle` is applied to all cells. Access the default styles as follows:

```
CellStyleModel csm = table.getDefaultLabelStyle();
csm = table.getDefaultCellStyle();
```

JClass `LiveTable` handles PLAF through a “parenting” mechanism. When the PLAF changes, `JTableUI` updates `DefaultLabelStyle` and `DefaultCellStyle` as required.

If you create a cell that uses this default style, but want to change a cell property while maintaining PLAF support for all other cell properties, you have to create a unique style for a cell.

For example, you can create a style for your table that has PLAF support, but changes the text alignment:

```
CellStyleModel csm = new JCellStyle(table.getDefaultCellStyle());
csm.setHorizontalAlignment(JCTableEnum.CENTER);
table.setCellStyle(0, 0, csm);
```

Here, you have created a new style based on `DefaultCellStyle`, and changed one property `HorizontalAlignment`. Applying this to cell (0, 0) changes the text alignment, but the other properties (background/foreground color, font, border type) will only change if the host machine's look and feel changes.

To gain a better understanding of how `JTableUI` works with default styles, imagine that you are applying this style change:

```
CellStyleModel csm = table.getDefaultCellStyle();
csm.setBackground(Color.blue);
```

You might think that when the user changes the PLAF, the blue background color will be cancelled out with the new PLAF defaults. This will not happen because `JTableUI` uses special wrapper objects to set values (e.g. `ColorUIResource`), and checks the current value to see if it is an instance of a UI Resource. If so, the property value is changed because `JTableUI` assumes the PLAF logic set it. If it is a regular object (in this case, `Color`), the value will not be updated by `JTableUI`.

2.6.4 Working with Colors

Setting Foreground and Background Colors

The foreground and background colors used for cells are specified by the `Foreground` and `Background` properties. The following example sets the background color of column 2 to blue:

```
JCCellStyle cell = new JCCellStyle();
cell.setBackground(Color.blue);
table.setCellStyle(JCTableEnum.ALL,2,cell);
```

In that example, the `JCCellStyle` default Cell Styles are used, with one overriding change for the background color.

The same applies in the next example, in which the foreground color value for cell (1, 4) is set to the color white:

```
JCCellStyle cell2 = new JCCellStyle();
cell2.setForeground(Color.white);
table.setCellStyle(0,3,cell2)
```

In addition to the row, column indexed contexts, you can set the `Foreground` and `Background` properties for a range of cells specified by a `JCCellRange` object:

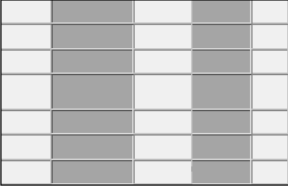
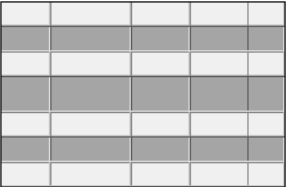
```
JCCellRange range = new JCCellRange(0,3,2,4);
JCCellStyle cell = new JCCellStyle();
cell.setBackground(Color.red);
table.setCellStyle(range,cell);
```

Repeating Colors

`JClass LiveTable` makes it easy to create rows or columns whose background and foreground colors alternate or cycle in a repeating pattern. To create a repeating pattern of background colors, set the `RepeatBackgroundColors` and `RepeatBackground` properties as shown in the following example:



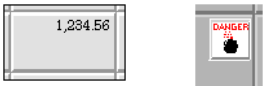

```
JCCellStyle colors = new JCCellStyle();
Color[] repeating = {Color.orange, Color.green, Color.white};
colors.setRepeatBackgroundColors(repeating);
colors.setRepeatBackground(JCTableEnum.REPEAT_COLUMN);
table.setCellStyle(JCTableEnum.ALLCELLS,
    JCTableEnum.ALLCELLS, colors);
```

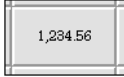

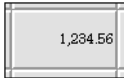

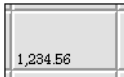

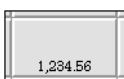




You can define as many repeating colors as you like. The colors are always selected in the order listed.

Repeating Color Property	Example
<p>JTableEnum.REPEAT_COLUMN</p> <p>sets repeating background or foreground colors by rows</p> <p>use as value for setRepeatBackground or setRepeatForeground methods</p>	
<p>JTableEnum.REPEAT_ROW</p> <p>sets repeating background or foreground colors by columns</p> <p>use as value for setRepeatBackground or setRepeatForeground methods</p>	

2.6.5 Text and Image Alignment

The horizontal and vertical alignment of text and images within cells and labels is specified by the `HorizontalAlignment` and `VerticalAlignment` properties. Cell/label values can be centered or positioned along any side of the cell/label.

Alignment Property	Examples
<p>setVerticalAlignment(JTableEnum.TOP)</p> <p>setHorizontalAlignment(JTableEnum.LEFT)</p>	
<p>setVerticalAlignment(JTableEnum.TOP)</p> <p>setHorizontalAlignment(JTableEnum.CENTER)</p>	
<p>setVerticalAlignment(JTableEnum.TOP)</p> <p>setHorizontalAlignment(JTableEnum.RIGHT)</p>	
<p>setVerticalAlignment(JTableEnum.CENTER)</p> <p>setHorizontalAlignment(JTableEnum.LEFT)</p>	

Alignment Property	Examples
setVerticalAlignment(JCTableEnum.CENTER) setHorizontalAlignment(JCTableEnum.CENTER)	 
setVerticalAlignment(JCTableEnum.CENTER) setHorizontalAlignment(JCTableEnum.RIGHT)	 
setVerticalAlignment(JCTableEnum.BOTTOM) setHorizontalAlignment(JCTableEnum.LEFT)	 
setVerticalAlignment(JCTableEnum.BOTTOM) setHorizontalAlignment(JCTableEnum.CENTER)	 
setVerticalAlignment(JCTableEnum.BOTTOM) setHorizontalAlignment(JCTableEnum.RIGHT)	 

2.6.6 Cell and Label Fonts

You can specify the font for the text in a cell or label with the `Font` property.

JClass LiveTable supports the use of one or more fonts in each cell/label. The example below sets a bold, serif font for all labels:

```
JCellStyle labelfont = new JCellStyle();
labelfont.setFont(new Font("TimesRoman", Font.BOLD, 10));
table.setCellStyle(JCTableEnum.LABEL, JCTableEnum.LABEL, labelfont);
```

JClass LiveTable can use any of the fonts available to Java. See your Java documentation for details on finding and setting fonts, or refer to [Appendix D](#).

2.6.7 Border Types

All cells and labels have a border around them, and the appearance of the cell or label border can be customized for individual cells and labels.

The border width, as well as the border around the table's frame, are not part of the Cell Style, as they are specified for the entire table. Please refer to Section 2.4, [Global Table Properties](#), for information about setting table-wide properties.

Cell and Label Border Types

Cell and label border types are defined by the `JCCellBorder` class. `JCCellBorder` implements the `CellBorderModel` interface, and can be set like any other Cell Style property.

The following table outlines all the valid cell and label border types. The code in each cell is the `JCCellBorderModel` value, which is used in the statement:
`border.setCellBorder(new JCCellBorder(value)).`

Note: In order for any different cell or label border type to be visible, the width of the border must be 5 pixels or greater (default: 1).

Cell or Label Border Type Properties

`JCTableEnum.ETCHED_IN`
 sets an etched border whose enclosed cells appear raised.



`JCTableEnum.ETCHED_OUT`
 sets an etched border whose enclosed cells appear set in.



`JCTableEnum.FRAME_IN`
 sets a framed border whose enclosed cells appear set in.



`JCTableEnum.FRAME_OUT`
 sets a framed border whose enclosed cells appear raised.



`JCTableEnum.IN`
 sets a plain border whose enclosed cell appears set in.



`JCTableEnum.OUT`
 sets a plain border whose enclosed cell appears raised.



`JCTableEnum.PLAIN`
 set a plain border.



`JCTableEnum.THIN`
 sets a plain border that appears thin.



`JCTableEnum.NONE`
 sets no border.



The following example sets a blank border for all cells in the first row:

```
JCellStyle border = new JCellStyle();
border.setCellBorder(new JCellBorder(JCTableEnum.BORDER_NONE));
table.setCellStyle(0, JCTableEnum.ALLCELLS, border);
```

To retrieve the border style for a cell, use the `getCellBorderStyle()` method. This returns a `CellBorder` object (see below).

Custom Cell and Label Borders

JClass `LiveTable` includes an interface that allows you to define your own cell borders and backgrounds for cells and labels. The `CellBorderModel` interface has a single method called `drawBackground()`. The `drawBackground()` method allows you to specify the border width, the sides of the cell on which to draw the border, the colors of the border sides, and the dimensions of the rectangle that gets drawn.

To define a new type of border, you have to create an Object that implements the `CellBorderModel` interface. The following (from the *BorderTypes.java* example in *examples/table/style*) defines a single-line border object called `LiteBorder`:

```
class LiteBorder implements CellBorderModel {
    Color color;

    public LiteBorder(Color color) {
        this.color = color;
    }

    public void drawBackground(Graphics gc, int border_thickness, int
        border_sides, int x, int y, int width, int height,
        Color top_color, Color bottom_color, Color plain_color) {

        gc.setColor(color);
        gc.drawRect(x, y, width, height);
    }
};
```

Now that the new type of border has been defined, you can use it as you would any cell style property:

```
JCellStyle cellborder = new JCellStyle();
cellborder.setCellBorder(new LiteBorder(Color.gray));
table.setCellStyle(JCTableEnum.ALLCELLS,
    JCTableEnum.ALLCELLS, cellborder);
```

The *examples/table/style* directory also contains a program called *TextureTable.java*, which illustrates how you can use the custom border features to insert a background graphic into cells.

Caution: If you create many different `CellBorder` objects, it will have an impact on your table's performance.

2.6.8 Cell and Label Border Sides

The `CellBorderSides` property specifies the sides of a cell or label that display the border type (specified by the `JCCellBorder` class). By default, the border type is displayed on all sides of a cell or label. The following figure illustrates one of the visual effects that can be achieved.

15 1/8	9 1/8	ISG Tech	ISGTF	10 1/2	10	10 1/4		16
28 1/2	9 1/4	Immune Rsp	IMNR	12 5/8	12	12 1/8	-1/4	823
27 1/4	13 3/8	Informix	IFMX	22	21	22	+1/2	6311

Figure 8 Customized Cell Borders.

The valid values for `CellBorderSides` are:

- `JCTableEnum.BORDERSIDE_LEFT`
- `JCTableEnum.BORDERSIDE_RIGHT`
- `JCTableEnum.BORDERSIDE_TOP`
- `JCTableEnum.BORDERSIDE_BOTTOM`
- `JCTableEnum.BORDERSIDE_ALL`
- `JCTableEnum.BORDERSIDE_NONE`

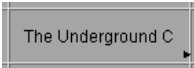
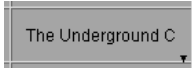
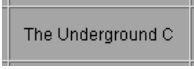

Specifying border sides is accomplished by OR-ing together all desired `CellBorderSides` values. The following example establishes cell borders on the left and top sides for all cells in column 3:

```
JCCellStyle borderside = new JCCellStyle();
borderside.setCellBorderSides(JCTableEnum.BORDERSIDE_LEFT |
    JCTableEnum.BORDERSIDE_TOP);
table.setCellStyle(JCTableEnum.ALL, 2, borderside);
```

2.6.9 Text and Image Clipping

When cell and label contents do not fit in their defined area, `JClass LiveTable` can clip the display of the cell value. The `ClipHints` property determines which method is used. The `setClipHints()` method can take the following values:

Clip Hints Properties

<p><code>JCTableEnum.SHOW_HORIZONTAL</code></p> 	<p><code>JCTableEnum.SHOW_VERTICAL</code></p> 
<p><code>JCTableEnum.SHOW_NONE</code></p> 	<p><code>JCTableEnum.SHOW_ALL (default)</code></p> 

2.6.10 Displaying Images in Table Cells

JClass LiveTable can display an image in each cell or label in the table. The image appears inside the margin of the cell. Images are displayed using the `JCImageCellRenderer` class in the `com.klg.jclass.cell` package. For more information, please see [Displaying and Editing Cells](#), in Chapter 4.

JClass LiveTable supports the image file formats supported by the Java AWT: GIF and JPEG. For more information on available file formats, see your Java documentation.

The position of the image within the cell is specified in the same way as Strings, using `HorizontalAlignment` and `VerticalAlignment`. This aspect of displaying images is handled by the `Styles` property. This is covered in Section 2.6.5, [Text and Image Alignment](#).

2.7 Cell and Label Spanning

Spanning is a way to join a range of cells or labels together and treat them as a single cell/label. A spanned range looks and acts like one cell/label that covers several rows and/or columns. There are many potential uses for spanning, including designing complex forms, displaying large images or components, and creating multiline headers.

When you create a spanned range, the top-left cell in the range is extended over the entire range. The top-left cell is the source cell, and its value and attributes apply over the entire span, overriding any values or attributes set for the other cells/labels in the range. Spanned ranges must begin at the top-left corner of the range. A span cannot contain both cells and labels, or frozen and non-frozen elements. There must also be more than one cell/label in a spanned range. When a single-cell range is specified, it is removed from the list.

The next figure shows an example of a table containing spanned ranges.

	Cable1	Cable2	7:00	7:30	8:00	8:30	9:00	9:30	10:00
2	1E	5	The Outer Limits	The Pretender	Puffler				
3	4	20	Babylon 5	Buffy Vampire Slayer	Baywatch			News	
4	1C	6	News	Medicine Woman	Early Edition			Cheers	
5	6	6	Sat. Repo't	Empty Nest	Liberty St	Liberty St	Movie: Naked Lunch		
6	3	3	Wilderness	Psi Factor	Early Edition			News	
7	14	4	News	College Football					
8	0	0	Entertainment Now	Medicine Woman	High Incident			Gain City	
11	11	1	Pensacola Wings	The Pretender	Puffler			Operation	
12	2L	58	Bingo	Sportsbest	Art - arce	I BA	Movie: Naked Lunch		
13	22	22	Entertainment Now	Medicine Woman	High Incident			Operation	

Print

Figure 9 Table design using spanned cells.

JClass LiveTable handles spanning cells with the `SpanHandler` class. This class contains the `setSpannedRanges()` method, which sets a `Collection` of ranges of cells or labels. (Please note that a `Collection` is typically a `Vector`.) Each element of the `Vector` is an instance of a `JCCellRange`.

A spanned range is a range of cells or labels that appear joined and can be treated as one cell. The top-left cell (specified by the `start_row` and `start_column` members) is the source cell for the spanned range. The cell/label value and attributes of the source cell are displayed in the spanned cell. Attributes for the spanned range must be set on the source cell.

Note: Spanned ranges may not overlap. If you have overlapping Spans, you will get a `System.err` message similar to the following:

```
spanlist.overlap: Range R1C2:R1C4 overlaps R1C1:R1C2
```

Overlaps are determined by the order of cell ranges in the Span Vector.

To remove all of the spanned ranges, use the `clearSpannedRanges()` method.

The following example defines a cell that spans three columns and two rows (columns 2 through 4, and rows 2 through 3):

```
JCCellRange spanrange = new JCCellRange(1,1,2,3);
table.addSpannedRange(spanrange);
```

	Customer N	Order Date	Item	Quantity (lbs)
1	The Cuppa	11/11/97	French Mocc	60
2	The Underg	11/14/97	Brazilian Me	112
3	RocketFuel	10/30/97	Espresso D	300
4	WideEyes C	11/12/97	Colombian/I	120

	Customer N	Order Date	Item	Quantity (lbs)
1	The Cuppa	11/11/97	French Mocc	60
2	The Underg	11/14/97		
3	RocketFuel			
4	WideEyes C	11/12/97	Colombian/I	120

Figure 10 Color properties of source cell (1,1) in the original table (left) are retained over the spanned cells in the table after the listed code has been added (right).

2.7.1 Using Spanning to Create Multiline Headers

You may want to create tables that contain multiline column headers where a top header is divided into two columns by sub-headers, as in the following illustration.

Customer Name	Order Date	Item	Order Info.	
			Quantity	Price/lb.
The Cuppa	11/11/97	French Mocha	60	\$7.01
The Underground	11/14/97	Brazilian	112	\$6.80
RocketFuel and	10/30/97	Espresso Dark	300	\$8.02
WideEyes Coffee	11/12/97	Colombian/Irish Cream	120	\$5.30
Jitters	10/01/97	Ethiopian	80	\$7.50
Twitchy's on	12/06/97	French Roast	160	\$14.50
Quest Software	12/12/97	Colombian	22,000	\$5.28

Figure 11 Multiline headers.

While JClass LiveTable does not support multi-row column labels, this effect can be achieved by setting some table-wide cell appearance and behavior properties, and some Cell Style properties. Use a frozen row at the top of the table to mimic the appearance of the column labels as follows:

- The right-most column label has been set to span columns 3 and 4. This produces a heading for both columns.
- The cell values for columns 3 and 4 in row zero have been set to contain the “subheadings” of the spanned label heading.
- The cells in row zero, columns 0 to 2 are empty.
- Row zero has been frozen using `setFrozenRows(1)` so that it stays at the top of the table and acts like a label.
- Row zero’s cells are not editable (using the `setEditable(false)` method) and not traversable (using `setTraversable(false)`).
- The `FrameBorderWidth` property of the table must be set to zero, so that the labels blend seamlessly into the frozen row.
- Finally, using Cell Styles, the `CellBorderSides`, `Background`, and `Foreground` properties for the column labels and row zero are all set to blend the two together.

Working with Table Data

Overview: Data Handling in JClass LiveTable ■ *Getting Data into your Table Using Stock Data Sources* ■ *Setting Stock Data Source Properties Loading Data from an XML Source* ■ *Creating your own Data Sources* ■ *Dynamically Updating Data*

3.1 Overview: Data Handling in JClass LiveTable

JClass LiveTable is a Java component that creates a table-formatted view of a given set of data. Data can come from many different types of sources; different applications can have different data storage needs. Since applications can generally store data more efficiently than a component, it is more practical for JClass LiveTable to use an external data object rather than storing the data internally. An external data model organizes the data in a way that is more convenient for the application, rather than for the component.

Consequently, JClass LiveTable uses a Model-View-Controller (MVC) architecture for data handling. The data in the table cells is stored in an external data source rather than the JCTable object itself. Either you create the data source object, or the data source can be a database. To use the latter, you need to use one of the LiveTable data-binding Beans. For more information about these Beans, please see [JClass LiveTable Beans and IDEs](#), in Chapter 9.

With LiveTable's MVC architecture, the data source object is the Model, which manages the underlying data being displayed and manipulated. The JCTable object acts as both the View (the object displaying the data to the user) and the Controller (the object that manipulates and modifies the data).

Because the JCTable object and the data source are separated, you are free to use whatever data storage mechanism you want; the JCTable object doesn't need to know anything about the mechanism itself. The MVC architecture also helps improve the performance of JClass LiveTable programs by removing the need to load all of the table's data into memory, then copy it to the JCTable object. The data source is able to copy only the data that is currently displayed by the JCTable object. An external data source can also manage large sets of data more efficiently than the JCTable object can.

3.1.1 How the Table and Data Source Communicate

Between the JCTable object and the data source lies an object that implements the `DataViewModel` and/or the `SortableDataViewModel`. The default implementation in the

table is `TableDataView`. While most developers will never have to work with it directly, it's important to realize that the `TableDataView` monitors the data source for changes and notifies the `JCTable` object when they occur. Additionally, the `TableDataView` has a set of translation tables that allow it to re-map rows or columns from the data source to the table. This is how `JClass LiveTable` can support features like column sorting and row or column swapping, where the appearance of the table changes, without manipulating the data source itself.

3.2 Getting Data into your Table

To display data in a `JClass LiveTable` application or applet, you need to create a data source object. Any object that implements the `TableDataModel` interface can be a data source. This can either be one of the stock data sources included with `LiveTable` (see Section 3.3, [Using Stock Data Sources](#)) or one of your own data sources (see Section 3.6, [Creating your own Data Sources](#)).

The `TableDataModel` interface is as follows:

```
public interface TableDataModel {
    public Object getTableDataItem(int row, int column);
    public int getNumRows();
    public int getNumColumns();
    public Object getTableRowLabel(int row);
    public Object getTableColumnLabel(int column);
    public void addTableDataListener(TableDataListener l);
    public void removeTableDataListener(TableDataListener l);
}
```

The primary method in the `TableDataModel` interface is `getTableDataItem()`, which retrieves the value of a specified cell. For more information on the types of cell data objects that `Table` understands, see [Displaying and Editing Cells](#), in Chapter 4. In short, you can have any type of object (usually one of `Integer`, `Double`, `String`, or `Image`) in a cell.

Table Size

The size of the table is also specified by the data source, using the `getNumRows()` and `getNumColumns()` methods.

Row and Column Labels

If you want to display row or column labels, their values are provided using the `getTableRowLabel()` and `getTableColumnLabel()` methods. These methods return the same types of objects as `getTableDataItem()`, but labels are never editable or traversable.

Data Format Detection

When using the `JCInputStream` stock data source, `LiveTable` automatically detects whether a data stream is in standard table or CSV format. So by default, `JCInputStreamDataSource` and `JCFileDataSource` attempt to determine the format of the

data source. To remove this automatic detection (and the overhead it creates), set a preferred data format type.

Data Source Listeners

Any time the data inside the data source changes, it should notify all of its listeners. To add and remove listeners to and from the data source, use the methods `addTableDataListener()` and `removeTableDataListener()`.

3.2.1 Making the Data Source Editable

If you want users to be able to edit the data, you must implement the `EditableTableModel` interface. `EditableTableModel` is derived from `TableModel` and adds one new method: `setTableDataItem()`.

```
public interface EditableTableModel extends TableModel {
    public boolean setTableDataItem(Object o, int row, int column);
}
```

When the user edits a cell in the table, the cell editor validates the data (for more information about cell editing, see [Displaying and Editing Cells](#), in Chapter 4), and passes the new data to the data source using the `setTableDataItem()` method. If the data source does not accept the value of the object (for example, if the value is invalid in some way), it will return `false` to indicate that the edit has been rejected. If the new value is valid, then `setTableDataItem()` will return `true` and the data source will store the value.

The `setEditable()` Method

You can use the `setEditable()` method, which is part of the `CellStyleModel` implementation, to turn editing on and off for specific cells and ranges of cells. `setEditable()` has no effect on labels, as they can never be edited. For `setEditable(true)` to have any effect, the data source must be editable.

3.3 Using Stock Data Sources

While it isn't hard to create a data source for a table, `JClass LiveTable` includes several stock data sources to save you the work of writing data sources for the most common data types. The following data sources are found in the `com.klg.jclass.table.data` package:

Data Source	Description
<code>JCAppletDataSource</code>	Reads in data from the <code>DATA</code> tag of an applet.
<code>JCCachedDataSource</code>	Caches previously read data from the data source.
<code>JCEditableCachedDataSource</code>	Allows users to edit cell values in tables with the above data source.

Data Source	Description
JCEditableFileDataSource	Allows users to edit cell values in tables with the above data source.
JCEditableVectorDataSource	Allows users to edit cell values in tables with the above data source.
JCFileDataSource	Creates an input data stream from a file.
JCInputStreamDataSource	Base class for any data source that relies on streamed input. This data source type can handle comma-separated value (CSV) data files.
JCResultSetDataSource	Simple read-only JDBC database source.
JCSpreadLabel	Contains convenience methods for creating spreadsheet labels.
JCTableModelDataSource	Enables users to display and edit Swing <code>TableModel</code> data objects in <code>JClass LiveTable</code> . Swing <code>TableModel</code> objects are typically used by the Swing <code>JTable</code> component.
JCURLDataSource	Uses URLs to create a data source object.
JCVectorDataSource	General purpose data source: extended by almost all stock data sources.

Most of the stock data sources extend the `JCVectorDataSource` class. Please see Appendix E, [JClass LiveTable Inheritance Hierarchy](#), for a complete hierarchy diagram that outlines the relationship between the stock data source classes.

3.3.1 JCVectorDataSource: the Data Source Workhorse

A `JCVectorDataSource` simply stores all of its data in memory using vectors. The `JCVectorDataSource` class contains methods that allow you to set individual elements, or to set all of the data in the data source from a vector or an array of objects.

Since `JCVectorDataSource` implements `TableDataModel`, it can't be edited by the `JTable` object. If you want users to be able to edit the cell values through the table, you should use `JCEditableVectorDataSource`. The `JCEditableVectorDataSource` class is a subclass of `JCVectorDataSource` that implements the `EditableTableDataModel` interface model.

3.3.2 Getting Data from an Input Stream

`JClass LiveTable` provides the `JCInputStreamDataSource` class to read data in through a standard `java.io.InputStream`, and since it is derived from `JCVectorDataSource`, it has all of the same capabilities as a `JCVectorDataSource` (see Section 3.4, [Setting Stock Data](#)

[Source Properties](#)). `JCInputStreamDataSource` accepts both CSV and table format data files, and items read into the data source are stored as either `String` or `double` objects. The data format for a simple table would be similar to the following (the `#` symbol denotes the beginning of a comment):

```
TABLE 2 4 NOLABEL # 2 rows, 4 columns
1 2 3 4 # row 1
1 2 3 4 # row 2
```

If you want to include labels, the data format would be:

```
TABLE 3 4
'Column 1' 'Column 2' 'Column 3' 'Column 4'
'Row 1' 1 2 3 4
'Row 2' 1 4 9 16
'Row 3' 1 16 81 256
```

The `JCInputStreamDataSource` class has the following subclasses that provide convenient constructors to create an `InputStream` from various sources:

- `JCFileDataSource`, for reading data from a file.
- `JCURLDataSource`, for reading data from a URL.
- `JCAppletDataSource`, for reading data from the `DATA <PARAM>` tag associated with the specified applet.

3.3.3 Getting Data from a Database

The `JCResultSetDataSource` uses a `JDBC` database connection and an `SQL` query to create a data source. The `JCResultSetDataSource` is a rudimentary implementation of a data-bound data source to demonstrate that `JClass LiveTable` can be used with database applications quite easily.

Note: The `JCResultSetDataSource` is not a data source that can be edited; that is, it will not write to the database.

`JClass LiveTable` also comes with data-binding Beans that allow you to bind your table to any `JDBC` data source. For information about the data binding Beans, please refer to [Data Binding with IDEs](#), in Chapter 9.

3.3.4 Caching Data with `JCCachedDataSource`

While `JCVectorDataSource` stores its memory using vectors, the `JCCachedDataSource` class stores its data in a vector of vectors. `JCCachedDataSource` uses another `TableDataModel` class to contain table cell and label information (“in between” the table and the data source). It will reference the cache first to see if the required data exists; if it does not, the call passes through to the original `TableDataModel` class, and the value is taken. When this happens, the retrieved value is also stored in `JCCachedDataSource`’s other `TableDataModel` class.

This method saves time by creating a second instance of previously retrieved data, outside of the actual data source. `JCCachedDataSource` should only be used when the

`TableModel`'s `getItem`, `getRowLabel`, and/or `getColumnLabel` are calculation-intensive or expensive to retrieve.

Use `JCEditableCachedDataSource` to bind to an editable data source and be able to edit the cell contents.

Note: A non-editable data source bound to `JCEditableCachedDataSource` will display an editor but reject all changes.

3.3.5 Using Swing TableModel Data Objects

The `JTableModelDataSource` enables you to use any type of `TableModel` data object in `JClass LiveTable`. `JTableModelDataSource` is an editable data source.

`JTableModelDataSource` interprets and reformats the `TableModel` data to the layout used by `JClass LiveTable`. This makes it easier to replace the `SwingJTable` component with `JClass LiveTable` because you do not have to reformat your data.

When you create a `JTableModelDataSource`, you need to pass the constructor a valid `TableModel` object.

3.4 Setting Stock Data Source Properties

The following properties are set using methods of the `JCVectorDataSource` class. Since the stock data sources are derived from the `JCVectorDataSource` class, you can set these properties from any of the stock data sources (though all of the properties may not be applicable to the specific data source).

Note: The `JCVectorDataSource` class contains properties that are not inherent to the `TableModel` interface. If you create your own data source, you will have to produce your own methods for such operations as adding and deleting rows and columns.

3.4.1 Working with Rows and Columns

Setting the Number of Rows/Columns

The `setNumRows()` and `setNumColumns()` methods specify the number of rows and columns in the data source (default is 5 columns and 10 rows). These values do not affect the internal `CellValues` vector of the data source. The values of the `NumRows` and `NumColumns` properties are updated by the `addRow()`, `addColumn()`, `deleteRows()`, and `deleteColumns()` methods (see below).

Specifying Row and Column Labels

You can set row and column labels by calling:

- `setRowLabel()` and `setColumnLabel()` for individual labels.
- `setRowLabels()` and `setColumnLabels()` for all of the labels.

Column and row labels can be set as an array of Strings, or as a vector. Each element of the labels' vector may be an instance of a String, Image, Component, or other object. To clear column or row labels, call the method with a null argument.

```
String clabels[] = { "Name", "Address", "Phone" };
...
JcVectorDataSource vds;
vds.setColumnLabels(clabels);
```

To retrieve the values, use:

- `getTableRowLabel()` and `getTableColumnLabel()` for individual labels.
- `getRowLabels()` or `getColumnLabels()` for all of the labels.

Adding Rows and Columns

You can insert new rows and columns into the data source using the `addColumn()` and `addRow()` methods. The `addColumn()` method inserts a new column into the data source, shifting any cell values to the right of the insertion. The `addRow()` method inserts a new row into the data source, shifting any cell values down.

The `addColumn()` and `addRow()` methods are identical:

- `public boolean addRow(int position,
Object label,
Vector values)`
- `public boolean addColumn(int position,
Object label,
Vector values)`

In the previous methods,

- The *position* parameter is the initial column (or row) index, and the new columns or rows are added prior to this position. If the position is set to `JcTableEnum.MAXINT`, the column or row is added after the final existing column or row.
- The *label* parameter refers to the column or row label. This parameter can have a null value.
- The *values* parameter refers to the array of objects that comprises the cell values. This parameter can have a null value.
- Both the `addColumn()` and `addRow()` methods return `false` if any of the parameters are invalid; if they return `false`, the row or column will not be added.

When calling `addRow()` and `addColumn()`, note the following:

- If you do not supply values for the new cells within the method, the cells are blank. Values for the new row or column labels must be specified separately.
- The initial row or column index cannot be greater than the values of `NumRows` or `NumColumns`.

Deleting Rows and Columns

Use the `deleteRows()` and `deleteColumns()` methods to remove rows and columns from the data source. When you delete a column, remaining cell values shift to the left; when you delete a row, existing cell values shift up.

The `deleteRows()` and `deleteColumns()` methods are identical:

- `public boolean deleteRows(int position,
int num_rows)`
- `public boolean deleteColumns(int position,
int num_columns)`

In the previous methods,

- The *position* parameter specifies the first row or column number to delete from the data source.
- The *num_rows* or *num_columns* parameter specifies the number of rows or columns to be deleted, starting from the row or column specified by *position*.

When calling `deleteRows()` and `deleteColumns()`, note the following:

- The starting row or column cannot be greater than the `NumRows` or `NumColumns` properties.
- Both the `deleteRows()` and `deleteColumns()` methods return `false` if any of the parameters are invalid.

Moving Rows and Columns

To move a range of rows or columns in the data source, use the `moveRows()` and `moveColumns()` methods. The `moveRows()` and `moveColumns()` methods take the following forms:

- `public boolean moveRows(int source,
int num_rows,
int destination)`
- `public boolean moveColumns(int source,
int num_columns,
int destination)`

In the previous methods,

- The *source* parameter specifies the first row or column to move.
- The *num_rows* and *num_columns* parameters specify the number of rows or columns to move.
- The *destination* parameter specifies the row number above which, or the column number to the left of which, to move the rows or columns.

When calling `moveRows()` and `moveColumns()`, note the following:

- The starting (*source*) row or column cannot be greater than the value of the `NumRows()` or `NumColumns()` properties.

- Both the `moveRows()` and `moveColumns()` methods return `false` if any of the parameters are invalid.

3.4.2 Working with Other Properties

Setting Cell Values

To set the cell values in the data source, use the `setCell()` or `setCells()` methods. The `setCells()` method can be a matrix of Strings or a vector of vectors. To remove all values, call `clearCells()`.

Adding and Removing TableDataListeners

The `JCVectorDataSource` class contains methods for adding and removing listeners to the data source: `addTableDataListener()` and `removeTableDataListener()`. These methods monitor the data source for changes. For more information, see Section 3.7, [Dynamically Updating Data](#).

3.5 Loading Data from an XML Source

3.5.1 XML Primer

XML – eXtensible Markup Language – is a scaled-down version of SGML (Standard Generalized Markup Language), the standard for creating a document structure. XML was designed especially for Web documents, and allows designers to create customized tags (“extensible”), thereby enabling a flexible approach to create common information formats for sharing both the format and the data on the Internet, intranets, and so on.

XML is similar to HTML in that both contain markup tags to describe the contents of a page or file. But HTML describes the content of a Web page (mainly text and graphic images) only in terms of how it is to be displayed and interacted with. XML, however, describes the content in terms of what data is being described. This means that an XML file can be used in various ways. For instance, an XML file can be utilized as a convenient way to exchange data across heterogeneous systems. As another example, an XML file can be processed (for example, via XSLT [Extensible Stylesheet Language Transformations]) in order to be visually displayed to the user by transforming it into HTML.

Here are links to more information on XML.

- <http://www.w3.org/XML/> – another W3C site; contains exhaustive information on standards. Of particular note are the XML schema 1 (structures) and XML schema 2 (datatypes) working drafts. They make up an extension that specifies how to constrain XML documents to particular schema. This is important if you want to represent database data or object-oriented data as *XML*.
- http://www.javasoft.com/xml/tutorial_intro.html – Sun’s XML site

- <http://www.oasis-open.org/cover/xml.html> – thorough list of links to XML papers and ongoing work

3.5.2 Using XML in JClass

In order to work with XML in your programs, or even to compile our XML examples, you will need to have the JAR files *jaxp.jar* and *crimson.jar*¹ in your CLASSPATH. These files are distributed with JClass LiveTable – you can find them in *JCLASS_HOME/lib/*.

JClass LiveTable can accept XML data formatted to the specifications outlined in `com.klg.jclass.util.xml.JCTableXMLParser`. This class takes in a stream of data and parses it under the assumption that it is in the defined XML format that JClass LiveTable uses. It then populates the specified table with the resulting data.

Examples of XML in JClass

For XML data source examples, see the `XMLFileData` and `XMLTableModelData` examples in *JCLASS_HOME/examples/table/datasource*. These both use the *colors.xml* file in *JCLASS_HOME/examples/table/datasource/*.

You can also specify your own data parsing format. There are now constructors in the `JCInputStreamDataSource`, `JCFileDataSource`, `JCURLDataSource`, and `JCAppletDataSource` classes that take an object that implements the `com.klg.jclass.table.data.JCFileFormatParser` interface.

Interpreter

The interpreter, which lets JClass LiveTable interpret the incoming data via the defined XML format, must be explicitly set by the user. The interpreter to use for JClass LiveTable is `com.klg.jclass.table.data.JCXMLFormatParser`.

Many constructors in the various data sources in JClass LiveTable take the `JCFileFormatParser` interface that this class (`JCXMLFormatParser`) implements.

Here are a few code examples that load XML data using this interpreter:

```
TableDataModel tdm = new JCFileDataSource(fileName,  
    new JCXMLFormatParser());
```

```
TableDataModel tdm = new JCURLDataSource(codeBase, fileName,  
    new JCXMLFormatParser());
```

Note: A user can create a custom data format and create a custom data interpreter by implementing `JCFileFormatParser`.

1. You may substitute for *crimson.jar* any parser that is compliant with Sun's JAXP 1.1 specification. See Sun's JAXP documentation for more information:

3.5.3 Example XML Files for JClass LiveTable

Here is an XML file that contains data formatted to the specifications detailed in `com.klg.jclass.util.xml.JCTableXMLParser`:

```
<?xml version="1.0"?>
<!DOCTYPE JCTableData SYSTEM "JCTableData.dtd">
<JCTableData>
  <Row>
    <Cell>1</Cell> <Cell>2</Cell> <Cell>3</Cell> <Cell>4</Cell>
  </Row>
  <Row>
    <Cell>1</Cell> <Cell>2</Cell> <Cell>3</Cell> <Cell>4</Cell>
  </Row>
</JCTableData>
```

Here is another example XML file that contains data formatted to the specifications detailed in `com.klg.jclass.util.xml.JCTableXMLParser`, this one with row and column labels:

```
<?xml version="1.0"?>
<!DOCTYPE JCTableData SYSTEM "JCTableData.dtd">
<JCTableData>
  <ColumnLabel>Column 1</ColumnLabel>
  <ColumnLabel>Column 2</ColumnLabel>
  <ColumnLabel>Column 3</ColumnLabel>
  <ColumnLabel>Column 4</ColumnLabel>
  <Row>
    <RowLabel>Row 1</RowLabel>
    <Cell>1</Cell> <Cell>2</Cell> <Cell>3</Cell> <Cell>4</Cell>
  </Row>
  <Row>
    <RowLabel>Row 2</RowLabel>
    <Cell>1</Cell> <Cell>4</Cell> <Cell>9</Cell> <Cell>16</Cell>
  </Row>
  <Row>
    <RowLabel>Row 3</RowLabel>
    <Cell>1</Cell> <Cell>16</Cell> <Cell>81</Cell> <Cell>256</Cell>
  </Row>
</JCTableData>
```

3.5.4 Tags

`<ColumnLabel>` and `<RowLabel>` tags are optional. Every `<Row>` tag can contain any number of `<Cell>` tags. These `<Cell>` tags define the value of one cell within the row.

3.5.5 Creating a Swing TableModel class

For details on how to use the above XML format to create a `Swing TableModel` class instead of a standard `JClass LiveTable` data source, please look at the `com.klg.jclass.util.xml.JCXMLTableModel` class. The user can pass an XML input stream to this object and use the resulting table model to populate a `JClass LiveTable`, a `Swing JTable`, a `JClass Chart`, or any other object that takes a `Swing TableModel` class.

Also, the XMLTableModelData example in *JCLASS_HOME/examples/table/datasource* shows this.

3.6 Creating your own Data Sources

If the stock data sources provided with JClass LiveTable do not meet your needs, you can easily create your own data source objects by implementing the `TableModel` interface, as in the following example from *examples/table/datasource/StaticDataSource.java*:

```
import com.klg.jclass.table.TableDataModel;
import com.klg.jclass.table.JTableDataListener;

public class StaticDataSource implements TableDataModel {

    protected String data[];

    public StaticDataSource(String strings[]) {
        if(strings == null)
            data = new String[0];
        else
            data = strings;
    }

    public Object getTableDataItem(int row, int column) {
        if(column == 0)
            return data[row];
        else
            return null;
    }

    public int getNumRows() {
        return data.length;
    }

    public int getNumColumns() {
        return 1;
    }

    public Object getTableRowLabel(int row) {
        return Integer.toString(row);
    }

    public Object getTableColumnLabel(int column) {
        return "Some Data";
    }

    public void addTableDataListener(JTableDataListener l) {
    }

    public void removeTableDataListener(JTableDataListener l) {
    }
}
```

The `StaticDataSource` class takes a one-dimensional array of Strings and turns it into a read-only data source. The constructors take the array of Strings; the `getTableDataItem()` method supplies the data as it is needed. Note that the `addTableDataListener()` and `removeTableDataListener()` methods have been left empty because this data source is not going to be changing dynamically, and thus does not need to keep track of its listeners. You can attach this data source to a table quite easily. To see a demonstration of this, run the *StaticTest.java* file, found in the *examples/table/datasource* directory.

To make the items in the table editable, you must implement the `EditableTableModel` interface, as in *examples/table/datasource/StaticEditableDataSource.java*:

```
import com.klg.jclass.table.EditableTableModel;
import com.klg.jclass.table.JTableDataListener;

public class StaticEditableDataSource implements EditableTableModel {
    protected String data[];

    public StaticEditableDataSource(String strings[]) {
        if(strings == null)
            data = new String[0];
        else
            data = strings;
    }

    public Object getTableDataItem(int row, int column) {
        if(column == 0)
            return data[row];
        else
            return null;
    }

    public boolean setTableDataItem(Object o, int row, int column) {
        if(column == 0) {
            if (o instanceof String)
                data[row] = (String)o;
            else
                data[row] = o.toString();
        }

        return true;
    }

    public int getNumRows() {
        return data.length;
    }

    public int getNumColumns() {
        return 1;
    }

    public Object getTableRowLabel(int row) {
```

```

        return Integer.toString(row);
    }

    public Object getTableColumnLabel(int column) {
        return "Some Data";
    }

    public void addTableDataListener(JCTableDataListener l) {
    }

    public void removeTableDataListener(JCTableDataListener l) {
    }
}

```

The `StaticEditableDataSource` class could have been a subclass of `StaticDataSource`, adding only the `setTableDataItem()` method, but in this example it was shown as a standalone class to make sure everything is as clear as possible. Note that the object that is passed back to the data source in `setTableDataItem()` is not a `String`.

To see a demonstration of the `StaticEditableDataSource` class, run the *StaticEditableTest.java* file, found in the *examples/table/datasource* directory.

3.7 Dynamically Updating Data

Sometimes the data in the data source changes all by itself – for example, you may have a table displaying stock prices with data arriving in real-time over a network socket. As new prices arrive, your users would like the table to update the values of the appropriate cells.

To notify the table that the data has changed, send a `JCTableDataEvent` to all of the `JCTableDataListener` objects that have registered themselves with the data source.

The following is a simple example that creates a background thread that automatically updates cell values. It can be found in the file *examples/table/datasource/DynamicDataSource.java*:

```

import java.util.Enumeration;
import java.util.Random;
import com.klg.jclass.table.TableDataModel;
import com.klg.jclass.table.JCTableDataEvent;
import com.klg.jclass.table.JCTableDataListener;
import com.klg.jclass.util.JCListenerList;

public class DynamicDataSource implements TableDataModel, Runnable {

    protected int data[] = {
        1, 2, 3, 4, 5, 6, 7, 8, 9,
    };

    protected JCListenerList listeners;
    protected Thread kicker;
}

```

```

public DynamicDataSource() {
    kicker = new Thread(this);
    kicker.start();
}

public Object getTableDataItem(int row, int column) {
    if (column == 0) {
        return new Integer(data[row]);
    }
    return null;
}

public int getNumRows() {
    return data.length;
}

public int getNumColumns() {
    return 1;
}

public Object getTableRowLabel(int row) {
    return Integer.toString(row);
}

public Object getTableColumnLabel(int column) {
    return "Some Data";
}

public void addTableDataListener(JCTableDataListener l) {
    listeners = JCListenerList.add(listeners,l);
}

public void removeTableDataListener(JCTableDataListener l) {
    listeners = JCListenerList.remove(listeners,l);
}

public void run() {
    Random random = new Random();
    Enumeration e;
    JCTableDataListener l;
    JCTableDataEvent event;
    int i;

    for(;;) {
        i = random.nextInt() % data.length;
        if (i < 0) {
            i = -i;
        }
        data[i] += (int)(random.nextGaussian()*10);
        event = new JCTableDataEvent(this,i,0,0,0,
            JCTableDataEvent.CHANGE_VALUE);

        for(e = JCListenerList.elements(listeners); e.hasMoreElements(); ) {
            l = (JCTableDataListener)e.nextElement();
            l.dataChanged(event);
        }
    }
}

```

```

        try {
            Thread.sleep(100);
        }
        catch(Exception ex) {
        }
    }
}
}

```

The `DynamicDataSource` class sends `CHANGE_VALUE` messages to all of its listeners whenever a value changes. When the `JTable` object receives this message it retrieves the new value from the data source and repaints the appropriate cell. There are several other update commands available on the `JTableDataEvent` class:

- | | |
|------------------------------------|------------------------------|
| ■ <code>CHANGE_VALUE</code> | ■ <code>NUM_ROWS</code> |
| ■ <code>CHANGE_ROW</code> | ■ <code>NUM_COLUMNS</code> |
| ■ <code>CHANGE_COLUMN</code> | ■ <code>ADD_COLUMN</code> |
| ■ <code>CHANGE_ROW_LABEL</code> | ■ <code>REMOVE_COLUMN</code> |
| ■ <code>CHANGE_COLUMN_LABEL</code> | ■ <code>MOVE_ROW</code> |
| ■ <code>ADD_ROW</code> | ■ <code>MOVE_COLUMN</code> |
| ■ <code>REMOVE_ROW</code> | ■ <code>RESET</code> |

All of the `CHANGE_` messages cause the Table to reload the specified data and repaint the intersection of the data that has been changed and the data that is being shown on screen.

The file `examples/table/datasource/DynamicTest.java` demonstrates the simple technique used in `DynamicDataSource.java`.

Easy Listener Management

If you do not want to have to manage the listeners, JClass LiveTable includes a class called `AbstractDataSource`. `AbstractDataSource` is an object provided by `JTable` that implements `TableDataModel`, and has methods for adding and removing `JTableDataListeners`. In addition, it contains several convenience methods for firing events, such as `fireValueChanged()` and `fireRowLabelChanged()` method.

As an example, the `DynamicDataSource.java` program could be implemented again to use the `AbstractDataSource` as follows:

```

import java.util.Enumeration;
import java.util.Random;
import com.klg.jclass.table.data.AbstractDataSource;
import com.klg.jclass.table.JTableDataEvent;
import com.klg.jclass.table.JTableDataListener;
import com.klg.jclass.util.JCListenerList;

```



```

public class DynamicDataSource2 extends AbstractDataSource
implements Runnable {

protected int data[] = {
    1, 2, 3, 4, 5, 6, 7, 8, 9,
};

protected Thread kicker;

public DynamicDataSource2() {
    kicker = new Thread(this);
    kicker.start();
}

public Object getTableDataItem(int row, int column) {
    if(column == 0)
        return new Integer(data[row]);
    else
        return null;
}

public int getNumRows() {
    return data.length;
}

public int getNumColumns() {
    return 1;
}

public Object getTableRowLabel(int row) {
    return Integer.toString(row);
}

public Object getTableColumnLabel(int column) {
    return "Some Data";
}

public void run() {
    Random random = new Random();
    Enumeration e;
    JTableDataListener l;
    JTableDataEvent event;
    int i;

    for(;;) {
        i = random.nextInt() % data.length;
        if (i < 0) {
            i = -i;
        }

        data[i] += (int)(random.nextGaussian()*10);

        event = new JTableDataEvent(this,i,0,0,0,
            JTableDataEvent.CHANGE_VALUE);

        fireTableDataEvent(event);
    }
}
}

```

```

        try {
            Thread.sleep(100);
        }
        catch(Exception ex) {
        }
    }
}

```

Running *examples/table/datasource/DynamicTest2.java* demonstrates that the same results can be achieved more easily by using a subclass of `AbstractDataSource`.

3.7.1 Adding and Removing Columns and Rows

`ADD_ROW`, `REMOVE_ROW`, `ADD_COLUMN`, and `REMOVE_COLUMN` notify the table that a row or column has been added or removed so that the table can update its internal list of cell attributes. For example, if all your rows are different colors, and you delete a row, the remaining rows will still have the correct colors if you send a `REMOVE_ROW` message to the `JTable`. Some of the event parameters may be ignored for row or column operations. For example, when you do an operation on an entire row or column, if you create an `ADD_ROW` event, the *column* parameter is ignored by the table. With the exception of the `MOVE_` events, all of the events ignore the *num_affected* and *destination* parameters of the `JTableDataEvent`.

The `MOVE_ROW` and `MOVE_COLUMN` commands are the only commands that make use of the *num_affected* and *destination* parameters in the `JTableDataEvent`. When you have a `MOVE_` event, you can move multiple rows/columns (the *num_affected* parameter) and you must specify to which row/column you are moving (*destination*).

The `RESET` message causes the `JTable` object to re-initialize itself by re-reading the number of rows, number of columns and all the data from the data source. The table's visual attributes, such as fonts and colors, are not affected.

Note: When a user edits a cell in the table and the value is put back into the data source via `setTableDataItem()`, the table will automatically repaint the cell with a new value.

Displaying and Editing Cells

Overview ■ *Default Cell Rendering and Editing*
Rendering Cells ■ *Editing Cells* ■ *The JCellInfo Interface*

4.1 Overview

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

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

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

JClass Cell Package	Contents
<code>com.klg.jclass.cell</code>	<p>Contains editor/renderer interfaces and support classes, including these interfaces:</p> <p>JCellEditor: used to define an editor JCellRenderer: the common and basic interface for renderers JComponentCellRenderer: allows the creation of renderers that are based on JComponent JLightCellRenderer: allows the creation of renderers based on direct drawing</p>
<code>com.klg.jclass.cell.editors</code>	<p>Contains editors for common data types.</p> <p>Please see Section 4.4.1, Default Cell Editors, for details.</p>

JClass Cell Package	Contents
<code>com.klg.jclass.cell.renderers</code>	Contains renderers for common data types, including expressions. Please see Section 4.3.1, JClass Cell Renderers , for details.
<code>com.klg.jclass.cell.validate</code>	Contains data validation interfaces and support classes.

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

Note: For the JClass Field component to work as a renderer, you need to use a particular instance from the `com.klg.jclass.field.cell` package.

JClass LiveTable has been designed to identify the type of data being retrieved from the data source and to provide the appropriate cell renderer and cell editor for that data type. For example, if JClass LiveTable encounters an expression in a cell (for example, any formula from `com.klg.jclass.util.formulae`), the default `JCEXpressionCellRenderer` will be used.

Often, however, you will want to control the way data in a particular area of the table is rendered, or assign a specific type of editor for that data. An example of this is rendering String data in multiple lines and using `javax.swing.JTextArea` as the editor, rather than rendering and editing single line Strings.

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

4.2 Default Cell Rendering and Editing

Basic Editors and Renderers

When the table draws itself, it accesses the data source and attempts to paint the contents of each cell. In doing so, it works through a two-stage process:

1. The table checks to see if a renderer has been assigned to the cell or a series of cells by the `CellRenderer` property in the cell's style.
2. If the table can't find a specific `CellRenderer` for the data, it uses the default mapping for that data type.

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

Data Type	Renderer	Editor
Boolean	JCStringCellRenderer	JCBooleanCellEditor
Date	JCStringCellRenderer	JCDateCellEditor
Double	JCStringCellRenderer	JCDoubleCellEditor
Expression	JCExpressionCellRenderer	none
Float	JCStringCellRenderer	JCFloatCellEditor
Image	JCImageCellRenderer	none
Integer	JCStringCellRenderer	JCIntegerCellEditor
Object	JCStringCellRenderer	none
String	JCStringCellRenderer	JCStringCellEditor

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

4.3 Rendering Cells

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

More information about included renderers is found in the next section, and information about the two rendering models on which you can base customized renderers is found in Section 4.3.4, [Creating your own Cell Renderers](#).

4.3.1 JClass Cell Renderers

As shown in the table above, `JClass LiveTable` maps standard data types to specific renderers when the program does not specify a renderer for that data type (either by setting for a series or mapping). This means that most tables are easily rendered without any special coding. The renderers are internally assigned. `JClass LiveTable` also contains several cell renderers for specific data types that you can set for a series (see Section 4.3.2, [Setting a Cell Renderer for a Series](#)) or as a mapping (see Section 4.3.3, [Mapping a Data](#)

[Type to a Cell Renderer](#)). These cell renderers are described in the following table and all of them are in `com.klg.jclass.cell.renderers` package.

Name	Data Type	Description
JCheckBoxCellRenderer	boolean	Defines a <code>JComponentCellRenderer</code> object that paints boolean objects in a table cell using Swing's <code>JCheckBox</code> .
JComboBoxCellRenderer	integer	Defines a <code>JComponentCellRenderer</code> that paints integer objects in a table using Swing's <code>JComboBox</code> .
JImageCellRenderer	image	Defines a <code>JLightCellRenderer</code> object that paints <code>Image</code> objects in a table cell.
JCEXpressionCellRenderer	expression	Defines the result of a formula (<i>com.klg.jclass.util.formulae.Expression</i>).
JLabelCellRenderer	String and/or image	Defines a <code>JLabelCellRenderer</code> object that uses Swing's <code>JLabel</code> to render cell contents.
JCRawImageCellRenderer	image	Defines a <code>JLightCellRenderer</code> object that paints unconverted <code>Image</code> objects in a table cell (extends <code>JCScaledImageCellRenderer</code>).
JCScaledImageCellRenderer	image	Defines a <code>JLightCellRenderer</code> object that paints scaled <code>Image</code> objects in a table cell.
JCStringCellRenderer	String, boolean, double, float, integer, object	Defines a <code>JLightCellRenderer</code> object that can draw Strings.
JCWordWrapCellRenderer	String	Defines word-wrapping logic for multiline display of Strings in cells.

The default mappings and these special renderer classes should provide rendering for most data types. Few programmers work under ideal conditions, however, and you may need to extend the capability of these renderers. `JClass LiveTable` includes ways for you to customize cell rendering as described in Section 4.3.4, [Creating your own Cell Renderers](#).

4.3.2 Setting a Cell Renderer for a Series

Often, the rows and columns that comprise a table are grouped by the type of data they contain. You may be creating an order form that has a product name (a `String`) in one

column, a part number (an Integer) in another, and a check box (a special type of object) in the final column to indicate that you want that product. For example:

Contents	Product Name	Part Number	Order Checkbox
Data Type	String	Integer	Boolean

All of these columns take a different data type, so their data is all rendered differently. LiveTable will automatically detect the type of data found, and use one of the default renderers for that column (please see Section 4.2, [Default Cell Rendering and Editing](#), for a list of default renderers). However, you can use your own renderer if the default does not suit your needs.

In the case of the Order check box, the default renderer for its Boolean data type will be the `JCStringCellRenderer`. With this default renderer, since the data type is boolean, instead of having a check (or no check) painted onto the cell, “true” or “false” will appear. This is not desirable, so you need to deviate from `JClass LiveTable`’s default renderer.

To set a new cell renderer for a range of cells, use a cell style, which has its own cell renderer property (for more information, please refer to [Cell Styles](#), in Chapter 2). Inserting these lines of code into your program will do this:

```
CellStyleModel style = table.getUniqueCellStyle(0,3);
style.setCellRenderer(new JCheckBoxCellRenderer());
table.setCellStyle(JCTableEnum.ALL, 3, style);
```

The `JCheckBoxCellRenderer` class defines an object that paints boolean objects in a table cell as checks. This way, the first two columns render automatically with the defaults, and the third column will use your defined renderer.

4.3.3 Mapping a Data Type to a Cell Renderer

Even though you can set the renderer series, your table may be designed in such a way that the data types within a row or column are not consistent, or will change depending on the data source. In this case you could decide not to set the renderer series at all, and allow the container to evaluate the data type and provide the appropriate renderer. Unfortunately, this means you have to use the default renderers for a given data type.

To use your own renderers without sacrificing flexibility, you can create a *mapping*. The mapping takes a data type and associates it with a `JCCellRenderer` object; whenever the container encounters that type of data, it uses the mapped `JCCellRenderer` object to render the data object in the cell.

Mapping a `JCCellRenderer` object to a data type takes the following construction:

```
table.setCellRenderer(Class cellType, Class renderer);
```

For example, in the following code fragment (from *TriangleTable.java* in the *examples/table/cell* directory of the JClass distribution), the cell renderer is set for a particular data type, defined by `java.awt.Polygon`.

```
try {
    table.setCellRenderer(Class.forName("java.awt.Polygon"),
        Class.forName
        ("examples.table.cell.TriangleCellRenderer"));
    ....
}
catch (ClassNotFoundException e) {
    e.printStackTrace(System.out);
}
```

The `table.setCellRenderer()` method takes a class to define the data type and a class to define the renderer. In the case below, we have created a class called `TriangleCellRenderer`, which is identified using the `Class.forName()` method imported from `java.lang.Class`. (Creating your own cell renderers is explained in the next section.)

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

To “unmap” a renderer, set the renderer class parameter to null.

Alternatively, you can map a particular cell renderer instance to a data type using:

```
table.setCellRenderer(Class cellType, JCCellRenderer renderer);
```

This method is useful if you want to reuse the same renderer instance, or if your renderer does not have a default construction.

4.3.4 Creating your own Cell Renderers

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

The *examples/table/cell* directory and the *demos/table* directories of your JClass LiveTable distribution contain a wide array of sample programs that use different approaches to cell rendering. You can use these examples and demos to help you refine your own renderers for whatever purpose you require.

Subclassing the Default Renderers

A simple way to create your own renderer objects is to subclass one of the renderers provided with JClass LiveTable. For example, *CurrencyRenderer.java*, found in the *examples/table/cell* directory, is an example of subclassing from the `JCStringCellRenderer` in the `com.klg.jclass.cell.renderers` package:

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

import java.awt.Graphics;

public class CurrencyRenderer extends JCStringCellRenderer {

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

Creating a Drawing-based Cell Renderer with JCLightCellRenderer

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

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

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

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

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

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

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

public class TriangleCellRenderer implements JCLightCellRenderer {

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

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

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

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

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

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

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

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

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

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

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

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

Creating a Component-based Cell Renderer with `JCComponentCellRenderer`

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

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

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

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

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

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

```
import com.klg.jclass.cell.JCComponentCellRenderer;
import com.klg.jclass.cell.JCCellInfo;
import javax.swing.JLabel;
import java.awt.Component;

public class JLabelCellRenderer extends JLabel
    implements JCComponentCellRenderer {

    public JLabelCellRenderer() {
        super();
    }

    public Component getRendererComponent(JCCellInfo cellInfo, Object o,
                                           boolean selected) {

        if (o != null) {
            if (o instanceof String) {
                setText((String)o);
            }
            else {
                setText(o.toString());
            }
        }
        else {
            setText("");
        }
        setFont(cellInfo.getFont());
        setBackground(selected ? cellInfo.getSelectedBackground() :
                        cellInfo.getBackground());
        setForeground(selected ? cellInfo.getSelectedForeground() :
                        cellInfo.getForeground());
        setHorizontalAlignment(cellInfo.getHorizontalAlignment());
        setVerticalAlignment(cellInfo.getVerticalAlignment());
        return this;
    }
}
```

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

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

`JComponentCellRenderer` is a very powerful rendering model. While it is not as flexible as `JLightCellRenderer`, it allows the reuse of code by using a lightweight component as a rubber stamp for painting in a cell. Any existing lightweight container can be used to render data inside of a cell – even other `JClass` components.

4.4 Editing Cells

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

A typical cell edit works through the following process:

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

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

4.4.1 Default Cell Editors

Cell editors are typically Swing components with extended functionality provided by the `com.klg.jclass.table.cell.JCCellEditor` interface. Although every data object is guaranteed to have a cell renderer, not every object is guaranteed to have an editor. Unless an object has an editor, the cell is not editable, regardless of whether the `table.setEditable()` method has a `true` value for that cell. Most of the standard data types have default editors which are internally associated with that data type. If the program does not specify an editor for a series or map a data type to an editor, the `Table` uses the default. The following editors are provided in the `com.klg.jclass.cell.editors` package:

Editor	Description
<code>BaseCellEditor</code>	Provides a base editing component for other editors.
<code>JCBigDecimalCellEditor</code>	An editor using a simple text field for <code>BigDecimal</code> objects.
<code>JCBooleanCellEditor</code>	Provides a simple text editing component that allows the user to set the boolean value as <code>true</code> , <code>false</code> , <code>t</code> , or <code>f</code> .
<code>JCByteCellEditor</code>	An editor using a simple text field for <code>Byte</code> objects.
<code>JCCheckBoxCellEditor</code>	An editor for boolean data that automatically changes the checked state.
<code>JCComboBoxEditor</code>	An editor using a simple Swing <code>JComboBox</code> for editing an enum.
<code>JCDateCellEditor</code>	An editor using a simple text field for <code>Date</code> objects
<code>JCDoubleCellEditor</code>	An editor using a simple text field for <code>Double</code> objects.
<code>JCFloatCellEditor</code>	An editor using a simple text field for <code>Float</code> objects.
<code>JCImageCellEditor</code>	An editor using a simple text field for <code>Image</code> objects.
<code>JCIntegerCellEditor</code>	An editor using a simple text field for <code>Integer</code> objects.
<code>JCLongCellEditor</code>	An editor using a simple text field for <code>Long</code> objects.
<code>JCMultilineCellEditor</code>	A simple text editing component for multiline data.
<code>JCShortCellEditor</code>	An editor using a simple text field for <code>Short</code> objects.
<code>JCSqlDateCellEditor</code>	An editor using a simple text field for <code>SQL Date</code> objects.
<code>JCSqlTimeCellEditor</code>	An editor using a simple text field for <code>SQL Time</code> objects.

Editor	Description
JCSqlTimestampCellEditor	An editor using a simple text field for SQL Timestamp objects.
JCStringCellEditor	Provides a simple text editing component.
JCWordWrapCellEditor	Provides a simple text editing component that wraps text.

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

4.4.2 Setting a Cell Editor for a Series

As mentioned above, JClass LiveTable contains logic that will map data types to their default editors. If you want to override these defaults, you can set a specific editor for a series of cells in your table by setting the `CellEditor` property on a cell style, for a range of cells:

```
CellStyleModel style = table.getUniqueCellStyle(0,3);
style.setCellEditor(new JCStringCellEditor());
table.setCellStyle(JCTableEnum.ALL, 3, style);
```

This code uses the same `CellEditor` (the default `String` editor in the `com.klg.jclass.cell.editors` package) for all of the cells in the fourth column in the table.

4.4.3 Mapping a Data Type to a Cell Editor

Even though you can set the editor series, your table may be designed in such a way that the data types within a row or column are not consistent, or will change depending on the data source. In this case you can create a *mapping*. The mapping takes a data type and associates it with a cell editor; whenever the container encounters that type of data, it uses the mapped `JCCellEditor`.

Mapping a `CellEditor` object to a data type takes the following construction:

```
table.setCellEditor(Class cellType, Class Editor);
```

Consider the following sample from *TriangleTable.java* in the *examples/table/cell* directory of the JClass LiveTable distribution:

```
try {
    table.setCellEditor(Class.forName("java.awt.Polygon"),
        Class.forName
            ("examples.table.cell.TriangleCellEditor"));
}
catch (ClassNotFoundException e) {
    e.printStackTrace(System.out);
}
```

The `table.setCellEditor()` method takes a class to define the data type and a class to define the editor. In the case above, we have created a class called `TriangleCellEditor`, which is identified using the `Class.forName()` method imported from `java.lang.Class`. (Creating your own cell editors is explained in the next section).

To “unmap” an editor, set the *editor* class parameter to `null`.

Alternatively, you can map a cell editor to a data type using:

```
table.setCellEditor(Class cellType, JCCellEditor editor);
```

This method is useful if you want to reuse the same editor instance, or if your editor does not have a default constructor.

Note: If the value for a particular cell is null, JClass LiveTable has no way of determining its type. This can cause problems if mapping a null value to an editor. To work around this, use the `DataType` property that is used with cell styles. LiveTable refers to `DataType` when it encounters a null in the data source.

4.4.4 Creating Your Own Cell Editors

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

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


This chart describes each of the methods in `JCCellEditor`:

Method and Description

```
public void initialize(AWTEvent ev, JCCellInfo info, Object o);
```

The table calls `initialize()` before the edit starts to let the editor know what kind of event started the edit, using `java.awt.AWTEventObject`. The size of the cell comes from the `JCCellInfo` interface (detailed below). The `initialize()` method also provides the data object (`Object o`).

```
public Component getComponent();
```

Returns the AWT component that does the editing. The component should be lightweight.

```
public Object getCellEditorValue();
```

Returns the value contained in the editor. This method is called by the table when the edit is complete. The value will be sent to the data source.

```
public boolean stopCellEditing();
```

When this method is called by the table, the editor can refuse to commit invalid values by returning `false`. This tells the container that the edit is not valid.

```
public boolean isModified();
```

The container uses this method to check whether the data has changed. This can save unnecessary access to the data source when the data has not actually changed.

```
public void cancelCellEditing();
```

Called by the table to stop editing and restore the cell's original contents.

```
public JCKeyModifier[] getReservedKeys();
```

Retrieves the keys the editor would like to reserve for itself. In order to avoid the container overriding key processing in the editor, the editor can pass back a list of keys it wishes to reserve. The container can refuse the editor's request to reserve keys. Most editors can simply return `null` for this method.

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

Method and Description

```
public abstract void addCellEditorListener(JCellEditorListener l);
```

Adds a listener to the list that's notified when the editor starts, stops, or cancels editing.

```
public abstract void removeCellEditorListener(JCellEditorListener l);
```

Removes the listener.

In addition to implementing the methods of `JCellEditor`, an editor is responsible for monitoring events and sending `editingStopped` and `editingCanceled` events to the table. This functionality is further explained in [Creating Your Own Cell Editors](#).

Subclassing the Default Editors

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

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

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

public class MoneyCellEditor extends JCStringCellEditor {

    Money initial = null;

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

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

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

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

        return new_data;
    }

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

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

Writing Your Own Editors

Of course, you may not want to subclass any of the editors provided with the `com.klg.jclass.cell.editors` package. The following is from an editor that was written without subclassing an existing editor. By implementing the `JCCellEditor` interface, we have written an editor that will edit triangles. The code is in *examples/table/cell/TriangleCellEditor.java*. You can see it work by running `examples.table.cell.TriangleTable`.

The editor handles both `Integer` and `Polygon` data types. It initializes the editor with the object to be edited, either a `Number` or a `Polygon`:

```
....

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

    new_poly = null;

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

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

```
....

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

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

```
....

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

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

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

taking place, so the `stopCellEditing()` method will be unconditionally obeyed. The `TriangleCellEditor` also defines a `cancelCellEditing()` method, which resets the new **Polygon**.

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

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

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

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

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

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

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

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

    int x, y;

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

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

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

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

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

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

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

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

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

Handling Editor Events

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

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

`JCCellEditorSupport` methods include:

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

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

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

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

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

Editor Key Control

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

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

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

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

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

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

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

4.5 The `JCellInfo` Interface

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

The `JCellInfo` interface gives the renderer and editor access to cell formatting information from the `Table`, including:

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

This information is fairly generic. The `com.klg.jclass.table` package also contains an object called `TableCellInfoModel`, which extends `JCellInfo` to include more detailed information from the `Table`. `TableCellInfoModel` is useful for retrieving `Table`-specific information for use in the editor or renderer.

Note that editors and renderers that rely on `TableCellInfoModel` can only be used with `JClass LiveTable`.

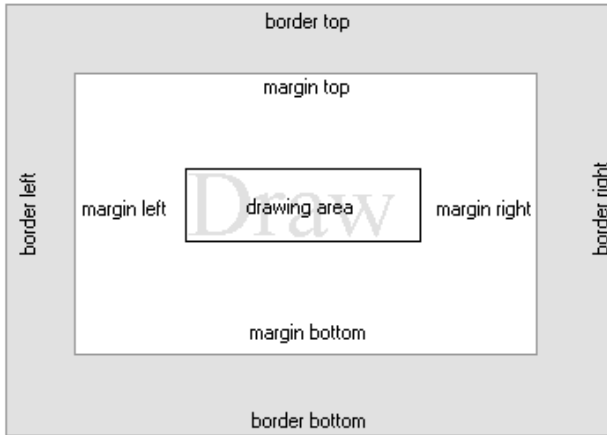


Figure 12 The relationship of border sides, margins, and drawing area provided by JCellInfo.

For more information, please see the `JCellInfo` API documentation.

Adding Formulae to JClass LiveTable

Introduction ■ *com.klg.jclass.util.formulae's Hierarchy* ■ *Expressions and Results* ■ *Math Values*
Operations ■ *Exceptions* ■ *Using Formulae in JClass LiveTable*

5.1 Introduction

The `formulae` package in `com.klg.jclass.util` has special capabilities for evaluating mathematical objects. Similar to the way that objects such as `java.lang.Double` wrap a primitive type, those in `com.klg.jclass.util.formulae` encapsulate mathematical expressions (operators) whose operands may be scalars, vectors (in the mathematical sense), and matrices. These objects may then be stored as the generalized values of cells in a JClass LiveTable, or in a JClass PageLayout table, where they may be evaluated at run time to produce results based on the then-current data.

In addition, subclasses of `MathValue`, which are wrappers for generalized scalars, vectors, and matrices, provide several methods for converting an expression to a value and to a `String`, as well as other methods useful when dealing with these objects.

5.2 `com.klg.jclass.util.formulae's Hierarchy`

The interfaces, abstract classes, and derived classes, including possible exception classes, are shown in Figure 13.

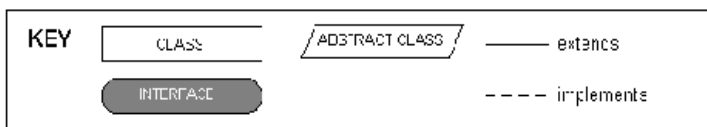
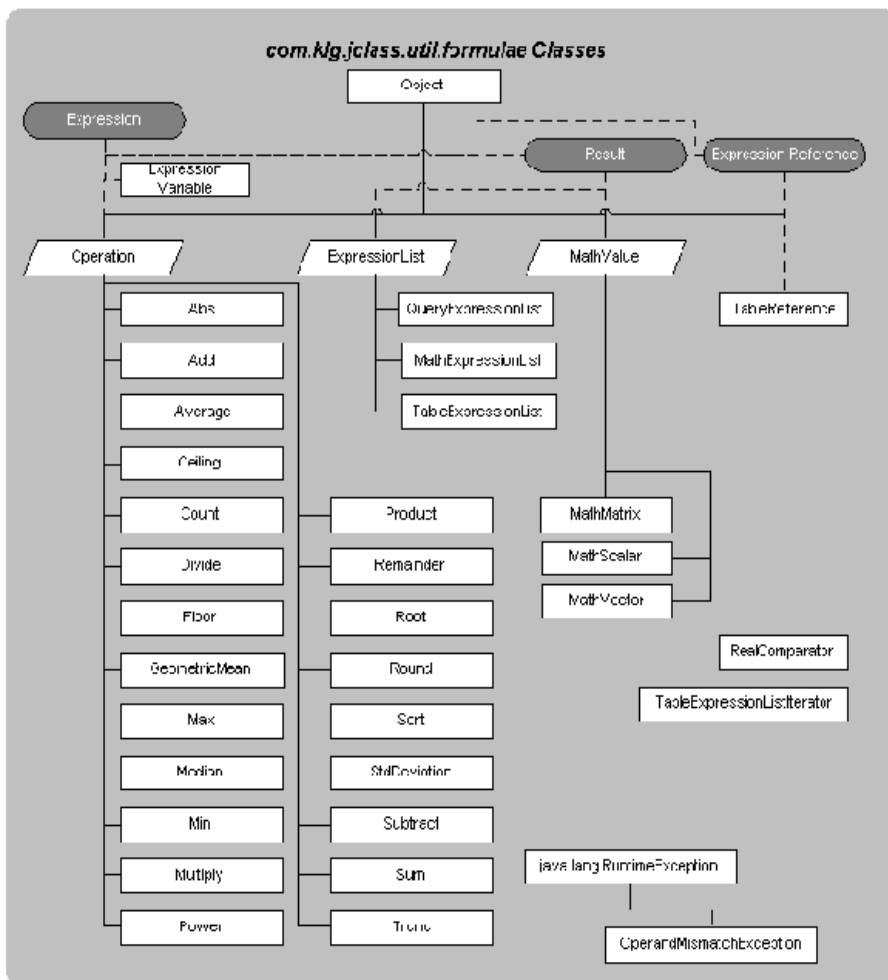


Figure 13 The inheritance hierarchy for `com.klg.jclass.util.formulae`.

The diverse set of mathematical operations permit you to compose complex mathematical formulas and provide references to them. Dynamic updating of the value

represented by the expression is made possible through callbacks to the mathematical expression object.

5.3 Expressions and Results

The top-level interface for the `com.klg.jclass.util.formulae` package is `Expression`, whose sole method is `evaluate()`. Any object that functions as an expression must have an `evaluate()` method that knows how to operate on data that might be a scalar, a vector, or a matrix. Applying the `evaluate()` method to an `Expression` produces a `Result`, which is a marker interface that identifies `Expression` types that are valid return types from the evaluation of other `Expressions`.

An `Expression` may be an `Operation`, as in:

```
Expression f = new Add(op1, op2);
```

which, after evaluation, returns a `Result`.

5.4 Math Values

The abstract class `MathValue` forms the root for all derived constant-based result/data classes. It satisfies the `Expression` interface by defining an `evaluate()` method, which simply returns the `MathValue` as a `Result`. Its concrete subclasses are `MathMatrix`, `MathScalar`, and `MathVector`. Because `MathValue` has an `evaluate()` method it is an `Expression`. Thus, `MathValues` may be passed as `Expression` objects.

MathValue Methods

MathValue Method	Description
<code>evaluate()</code>	Satisfies the <code>Expression</code> interface by returning the stored value. No evaluation is required because no operation is implied.
<code>getDataFormat()</code>	Retrieves the <code>NumberFormat</code> associated with this data.
<code>matrixValue()</code>	Gets the contents of this <code>MathValue</code> as a matrix of <code>Numbers</code> .
<code>numberValue()</code>	Gets the contents of this <code>MathValue</code> as a <code>Number</code> .
<code>setDataFormat()</code>	Sets a <code>NumberFormat</code> to use on the contents of this <code>MathValue</code> .
<code>vectorValue()</code>	Gets the contents of this <code>MathValue</code> as a vector of <code>Numbers</code> .

Note: The subclasses of `MathValue` override all but the first method. Since, for example, `matrixValue()` is not appropriate to a `MathScalar`, it throws an

UnsupportedOperationException if it is called. Other method-data type mismatches also throw UnsupportedOperationExceptions. The method tables for the subclasses indicate which methods are data type mismatches for the given class.

5.4.1 MathScalar

MathScalar is a scalar constant represented as a MathValue. By encapsulating it in this fashion it can support integer and real numbers, and it can be extended if necessary to support other types of scalar numbers. Its data field is a realValue, a Number that is output based on the current dataFormat kept in MathValue.

Example:

```
double s1 = 10.0; MathValue ss1 = new MathScalar(s1);
```

MathScalar Constructors

The no-argument constructor MathScalar() creates an instance that encapsulates the value 0.0, while the other three constructors take a double, an int, or a java.lang.Number argument.

MathScalar Methods

MathScalar Method	Description
matrixValue()	Throws an UnsupportedOperationException.
numberValue()	Gets the contents of this MathValue as a Number.
toString()	Returns a String representation of this value.
vectorValue()	Throws an UnsupportedOperationException.

5.4.2 MathVector

MathVector is a representation of the class of vectors in a linear algebra sense. They may also be used as operands in matrix multiplication. A MathVector encapsulates a list of values which may be integers, doubles, or more generally, objects of type Number. It has methods for retrieval or modification of a value at a particular index, and for outputting the list as a String. The operators discussed in the next section accept these objects as operands.

Example:

```
double[] ed = {2.71828, 3.1415927, 1.6020505};  
MathValue mv = new MathVector(ed);
```

MathVector Constructors

The constructors for MathVector parallel those for MathScalar, except they take arrays as parameters rather than single values.

MathVector Methods

MathVector Method	Description
getValueAt()	Retrieves the value at a particular index in the vector.
matrixValue()	Throws an UnsupportedOperationException.
numberValue()	Throws an UnsupportedOperationException.
setValueAt()	Sets the value at a particular index in the vector.
toString()	Outputs the value of this vector as a String.
vectorValue()	Gets the contents of this MathValue as a vector of Numbers.

5.4.3 MathMatrix

`MathMatrix` is a representation of the class of matrices, again in the sense of linear algebra. The package implements the basic addition and multiplication operations in matrix algebra, including multiplying a matrix by a vector. It has methods for retrieval or modification of a value at a particular pair of indices, and for outputting the matrix as a String. The operators discussed in the next section accept these objects as operands.

Example:

```
double[][] m1 = {{1.1, 1.2, 1.3},
                 {2.1, 2.2, 2.3},
                 {3.1, 3.2, 3.3}};
MathValue mm = new MathMatrix(m1);
```

MathMatrix Constructors

The constructors for `MathMatrix` parallel those for `MathScalar`, except they take 2D arrays as parameters rather than single values.

MathMatrix Methods

MathMatrix Method	Description
<code>getValueAt()</code>	Retrieves the value at a particular row, column pair of index values in the matrix.
<code>matrixValue()</code>	Gets the contents of this <code>MathValue</code> as an array of <code>Numbers</code> .
<code>numberValue()</code>	Throws an <code>UnsupportedOperationException</code> .
<code>setValueAt()</code>	Sets the value at a particular row, column pair of index values in the matrix.
<code>toString()</code>	Outputs the value of this vector as a <code>String</code> .
<code>vectorValue()</code>	Throws an <code>UnsupportedOperationException</code> .

5.5 Operations

The abstract `Operation` class defines the basic elements of an operator. Binary operators have a left and right operand, which enables the correct ordering to be applied to matrix operations and any other non-commutative operators. Unary operators have a single operand.

Example:

```
double[] ed = {2.71828, 3.1415927, 1.6020505};
double[] rd = {(Math.sqrt(5.0) + 1.0) / 2.0, 4.0, 32.0};

MathValue e = new MathVector(ed);
MathValue r = new MathVector(rd);

Expression add = new Add(e, r);
```

Operation Constructors

There is a no-argument constructor that creates a generic operator, and there are constructors for every unary and binary permutation of `Expressions` and `Numbers`. A sample constructor is: `Operation(Expression left, Expression right)`.

Operation Methods

The two non-inherited methods in `Operation` are `evaluate()`, which returns a `Result` containing the evaluation of the expression, and `clone()`, which returns a deep-copy clone of the operation and of all operands.

5.5.1 The Defined Mathematical Operations

Unary Operators

Unary operators take one parameter, which is either an Expression or a Number. Because they are Expressions they all have an `evaluate()` method which returns a Result.

Operator	Description
Abs	The class for the absolute value operation. The operand may be a Number or an Expression, which may be a MathScalar or an ExpressionList, but not a vector or a matrix.
Ceiling	Ceiling is defined as the least integer greater than or equal to the operand, which may be a MathValue.
Floor	Floor is defined as the greatest integer less than or equal to the operand.
Root	Returns the positive square root of its operand.
Round	Round is defined as nearest integer to the operand. Rounding is done to an even number if the operand is exactly midway between two integers.
Trunc	Takes the integer part of a number. Equivalent to rounding to the nearest integer closer to zero. Example: <code>trunc(-3.5) = -3</code> .

Binary Operators

Binary operators take two parameters, which are Expressions, Numbers, or one of each. Because they are Expressions they all have an `evaluate()` method which returns a Result.

Operator	Description
Add	Adds two Expressions. If the Expressions are vectors of the same length, pairwise addition is performed. Matrices may be added providing the two operands have the same number of rows and columns. Unary addition is possible, and returns the evaluated operand.
Average	Average (arithmetic mean) is defined as the sum of all elements divided by the number of elements. Its one-parameter constructor is an Expression, usually a list. Its two-parameter constructors are combinations of Expressions and Numbers.

Operator	Description
Count	Count determines the total number of elements in its operands. Its one- and two-parameter constructors take one or two Expressions (usually a list or lists) and count their elements.
Divide	Division is the ratio of two operands. The left operand is the numerator and the right operand is the denominator.
GeometricMean	Geometric mean is defined as the n th root of the product of a set of n numbers. Its one-parameter constructor takes an Expression, usually a list. Its two-parameter constructors take combinations of Expressions and Numbers, multiplying all elements together and taking the n th root.
Max	Max is defined for a pair of elements or across a list. It selects the largest element. Its one-parameter constructor takes an Expression, usually a list. Its two-parameter constructors take combinations of Expressions and Numbers, examining all elements and selecting the largest.
Median	The Median of a list is the middle element of a sorted list, or the average of the two middle values if the list has an even number of elements. Its one- and two-parameter constructors take one or two Expressions.
Min	Min is defined for a pair of elements or across a list. It selects the smallest element. Its one-parameter constructor takes an Expression, usually a list. Its two-parameter constructors take combinations of Expressions and Numbers, examining all elements and selecting the smallest.
Multiply	Multiplication is the product of a pair of elements. Its two-parameter constructors take combinations of Expressions and Numbers, examining all elements and selecting the smallest.
Power	The exponentiation (^) operation. Its two-parameter constructors take combinations of Expressions and Numbers. The left operand is the base and the right operand is the exponent.
Product	A product can be performed on a pair of elements or across a list. The product of an ExpressionList is the product of its individual members. Multiplication order is left-to-right, and first element of a list to last element. The result of a matrix multiplication may depend on the order of the operands.

Operator	Description
Sort	This operation returns a sorted list of the given elements. Any secondary or nested lists are flattened.
StdDeviation	The sample standard deviation, given by $sd = \sqrt{\frac{\sum_{i=1}^n (\text{element} - \text{average})^2}{n - 1}}$, where n is the number of samples and average is the sample average. It has one- and two-parameter constructors consisting of Expressions.
Subtract	The difference between two numbers. It has two-parameter constructors that take combinations of Expressions and Numbers.
Sum	A sum can be performed on a pair of elements or across a list. Its two-parameter constructors take combinations of Expressions and Numbers. Its one-parameter constructor usually takes an ExpressionList.

5.5.2 Reducing Operations to Values

Since Operations are Expressions, they all have an `evaluate()` method. Evaluation returns a `Result`, which may be converted to a `String` for printing. Here is an example:

```
double edd = 2.0;
double exp = 8.0;
MathValue eddy = new MathScalar(edd);
MathScalar expy = new MathScalar(exp);

double[] ed = {2.71828, 3.1415927, 1.6020505};
MathValue e = new MathVector(ed);

Expression pow = new Power(eddy, expy);
Expression powr = pow.evaluate();
// Either one of these has a toString() method
System.out.println("Power without evaluate(): " + pow);
System.out.println("Power with evaluate(): " + powr);
```

After which the following is written on the output:

```
Power without evaluate(): com.klg.jclass.util.formulae.Power@eb4f3b8c
Power with evaluate(): 256.0
```

You see that calling `evaluate()` is necessary to have a value returned by the implicit `toString()` call.

5.6 Expression Lists

Expression lists are handy containers that permit you to perform an operation on a group of values.

MathExpressionList

The example shown here uses the binary form of `Add` to find the grand total of all the elements in two `ExpressionList`s.

```
// Expression Lists
Expression[] exprs1 = {null, null, null, null, null, null, null,
                      null, null, null};
for (int i = 0; i < 10; i++){
    exprs1[i] = new MathScalar(95 + i);
}
ExpressionList explist1 = new MathExpressionList(exprs1);

Expression[] exprs2 = {null, null, null, null, null, null, null,
                      null, null, null};
for (int i = 0; i < 10; i++){
    exprs2[i] = new MathScalar(95 + i);
}
ExpressionList explist2 = new MathExpressionList(exprs2);

sssl = new Sum(explist1, explist2);
ssssl = sssl.evaluate();
System.out.println(
    "Summing ExpressionLists with    evaluate(): " + ssssl);
```

Here's the output:

```
Summing ExpressionLists with    evaluate(): 1990
```

QueryExpressionList

A `QueryExpressionList` is designed as a wrapper for a set of `Expressions` stored in a JDBC-type `ResultSet`; that is, the result of a database query. Users of `JClass DataSource` may also use this facility.

TableExpressionList

Expression lists may be used to extend data from portions of a `JClass LiveTable` to produce summary reports. For details, see Section 5.9, [Using Formulae in JClass LiveTable](#).

5.7 Events and Listeners

TableListenerPropagator

The `TableListenerPropagator` listener, which implements the `JCTableDataListener` interface, wraps a formula and listens for changes to table cells that are operands for this formula, and propagates the changes so that other interested listeners can re-evaluate

themselves. The `TableListenerPropagator` listener automatically updates the whole dependency hierarchy of `com.klg.jclass.util.formulae` when a suboperation has been modified.

5.8 Exceptions

OperandMismatchException

Various operations such as adding a number to a vector are not defined, whereas other operations, for example, multiplying a vector by a number, can be interpreted as a scaling operation. At compile time, numbers, vectors, and matrices can be declared as generic `Expressions`, making it impossible to predetermine which operations are not permitted. A run time check of the validity of an operation must be made. If a mathematical construct is evaluated and found to be illegal, the class throws an `OperandMismatchException`.

ClassCastException

There are cases where a run time class cast exception may occur. While most of these should be avoidable by selecting the correct class (such as using `Product` rather than `Multiply` when multiplying two vectors) the fact that both take `Expressions` as their parameters makes it difficult to avoid the possibility of an end-user passing in an incorrect type if your application permits flexible user input. You may permit substitution of one arithmetic class for another, since they are all `Operations`. This also opens the door to class cast exceptions.

If the possibility exists for either of these exceptions, your code should attempt to handle it.

5.9 Using Formulae in JClass LiveTable

5.9.1 Registering a Cell Editor and a Cell Renderer with the JClass Central Registry

If you are planning to allow your end-users to specify mathematical operations, you may make use of the editor/renderer registry in `com.klg.jclass.cell`. Note that if a cell is placed in a `JClass LiveTable`, the `ExpressionCellRenderer` will be used by default.

The following code snippet, taken from the *SpreadSheet* demo, registers a cell editor that takes a value in the form of a `util.formulae.Expression` from a table cell and copies its String equivalent in a text box. In the *SpreadSheet* demo, formulas are entered by beginning them with an equal sign (=), for example, `=SUM(A1:A5)`. The class called

MyFormulaCellEditor recognizes this syntax and translates a String of this form to an Expression, then stores it in a table cell.

```
EditorRendererRegistry.getCentralRegistry().addClass(  
    "java.lang.String",  
    null,  
    "demos.table.spreadsheet.MyFormulaCellEditor",  
    "com.klg.jclass.cell.renderers.JCStringCellRenderer");
```

See the [SpreadSheet demo](#) for a complete code listing.

5.9.2 Performing a Mathematical Operation on a Range of Cells

Expression Lists and Expression References

Expression list objects hold a group of Expressions. ExpressionList is an abstract class whose methods permit the inclusion of additional elements to those already present, a method for removing elements or clearing all elements, for retrieving an element, and for comparing with another list. These operations are common to the concrete classes MathExpressionList, QueryExpressionList, and TableExpressionList.

Expression lists may be used as arguments for all mathematical operations. When given an expression list, evaluating a unary operator such as ABS returns a list containing the absolute values of its input list. Binary operators may return a single result or a list. Given expression lists, the mathematical operators Abs, Add, Ceiling, Divide, Floor, Multiply, Power, Remainder, Root, Round, Sort, and Subtract return lists, while Average, Count, GeometricMean, Max, Median, Min, Product, and Sum all return a single result after evaluate() has been called.

Use TableExpressionList to perform an operation over a range of cells in a table. The following code snippet shows that the required parameters are a table data model and a block of cells.

```
Expression expression = new TableExpressionList(  
    table.getDataSource(),  
    new MathScalar(startRow),           // first row  
    new MathScalar(endRow),           // last row  
    new MathScalar(startColumn), // first column  
    new MathScalar(endColumn)       // last column  
);  
Sum sum = new Sum(expression);
```

The next code fragment places the formula for the sum in the last column, just below the last row. With the proper cell renderer/editor combination, such as the one listed in the previous section, the formula or the numerical value of the sum is shown, depending on whether the cell is selected or not.

```
((EditableTableModel)table.getDataSource()).setTableDataItem(sum,  
    endRow + 1, endColumn);
```

The advantage of using TableExpressionLists is that the formulas containing them know to update themselves when a cell's value is altered.

Programming User Interactivity

Cell Traversal ■ *Cell Selection* ■ *Resizing Rows and Columns* ■ *Table Scrolling*
Dragging Rows and Columns ■ *Sorting Columns* ■ *Custom Mouse Pointers*

JClass LiveTable makes it easy to allow users to interact with the tables you create. You can control how users can manipulate the table, and how a JClass LiveTable application can control this interaction. The following sections describe the types of user interactivity supported by JClass LiveTable, its default behavior, and how to customize that behavior. Note that programming cell editing behavior is discussed separately in [Displaying and Editing Cells](#), in Chapter 4.

6.1 Cell Traversal

Traversal is the act of moving the current cell indicator from one location to another. A traversal passes through three stages: validating the edited current cell, determining the new current cell location, and entering that cell.

The `Traversable` property, which is part of the `CellStyleModel` interface, determines whether or not a cell is traversable. You set this property when you are setting a cell style (for more information about cell styles, please see [Cell Styles](#), in Chapter 2).

6.1.1 Default Cell Traversal

Users can traverse cells by clicking the primary mouse button when the mouse pointer is over a cell. This changes the focus to that cell (a focus rectangle appears around the inside of the cell borders). Users can traverse cells from the keyboard by using the cursor keys (up, down, left, and right) and the **Tab** key to traverse right and **Shift+Tab** key to traverse left.

6.1.2 Customizing Cell Traversal

By default, all cells are traversable. To prevent users from traversing to a cell, set `Traversable` cell style property to `false`. Making a cell non-traversable also prevents it from being traversed to programmatically.

Disabling traversal also disables cell editability regardless of whether the cell's data source is editable.

The following code fragment sets all cells in row 3 to be non-traversable:

```
JCCellStyle traverserow = new JCCellStyle  
traverserow.setTraversable(false);  
table.setCellStyle(3, JCTableEnum.ALLCELLS, traverserow);
```

You can also set the `Traversable` property for a range of cells specified by a `JCCellRange` object:

```
JCCellRange range = new JCCellRange(2, 3, 2, 8);  
JCCellStyle traverserange = new JCCellStyle;  
traverserange.setTraversable(false);  
table.setCellStyle(range, traverserange);
```

Use the `setTraverseCycle()` method, part of the `JCTable` class, to determine whether the traversal moves to the opposite side when the left, top, right or bottom cell is reached (that is, when the user traverses to the bottom of the table, the next traversal down will bring them to the top of the table). The `TraverseCycle` property takes a boolean value, and the default is `true`.

6.1.3 Minimum Cell Visibility

By default, when a user traverses to a cell that is not currently visible, `JClass LiveTable` scrolls the table to display the entire cell.

The `setMinCellVisibility()` method sets the minimum amount of a cell made visible when it is entered. When the table scrolls to edit a non-visible cell, the `MinCellVisibility` property determines the percentage of the cell that is scrolled into view. When `MinCellVisibility` is set to 100, the entire cell is made visible. When `MinCellVisibility` is set to 10, only 10% of the cell is made visible. If `MinCellVisibility` is set to 0, the table will not scroll to reveal the cell.

The value of the `MinCellVisibility` property also affects the behavior of the `makeVisible()` methods described in Section 6.3.2, [Managing Table Scrolling](#).

6.1.4 Forcing Traversal

An application can force the current cell to traverse to a particular cell by calling `traverse()`. If the cell is non-traversable (specified by `Traversable`), this method returns `false`.

Calling the `traverse()` method to force cell traversal requires that you define these parameters:

- `row`: the row to which the current cell will traverse
- `column`: the column to which the current cell will traverse

- `show_editor`: a boolean value that determines if the editing component will be displayed in the cell. The default is `false`.
- `select`: a boolean value that determines the cell will be selected (if the `SelectionPolicy` allows it). The default is `false`.

6.1.5 Controlling Interactive Traversal

You can use the `TRAVERSE_CELL` action in `JCTraverseCellEvent` to control interactive traversal. As a user traverses from one cell to another, this event is posted after a user has committed a cell edit, and before moving to the next cell. Each event listener is passed an object of type `JCTraverseCellEvent`.

`JCTraverseCellEvent` uses the `getTraverseType()` method to retrieve information on the direction of the traversal. `getTraverseType()` retrieves one of the following integers indicating the direction of traversal:

- `TRAVERSE_POINTER` – traverse to the cell the user clicked.
- `TRAVERSE_LEFT` – traverse left to the first traversable cell.
- `TRAVERSE_RIGHT` – traverse right to the first traversable cell.
- `TRAVERSE_UP` – traverse up to the first traversable cell.
- `TRAVERSE_DOWN` – traverse down to the first traversable cell.
- `TRAVERSE_HOME` – traverse to the top-left corner of the table (0, 0).
- `TRAVERSE_END` – traverse to the bottom-right corner of the table.
- `TRAVERSE_TOP` – traverse to the top of the table column.
- `TRAVERSE_BOTTOM` – traverse to the bottom of the table column.
- `TRAVERSE_PAGEUP` – traverse up to the next off-screen or partially visible row.
- `TRAVERSE_PAGEDOWN` – traverse down to the next off-screen or partially visible row.
- `TRAVERSE_TO_CELL` – traverse programmatically.

The `getColumn()` and `getRow()` methods get the column and row of the current cell respectively. Finally, the `NextColumn` and `NextRow` properties respectively set or retrieve the column and row of the cell to traverse to.

The `TRAVERSE_CELL` action attempts to traverse to the cell specified by these members. Note that if `NextColumn` and `NextRow` reference a non-traversable cell, the traversal

attempt will be unsuccessful. The following example code prevents the user from traversing outside of column 0:

```
public void traverseCell(JCTraverseCellEvent ev) {
    if (ev.getNextColumn() > 0) {
        if (ev.getRow() >= table.getNumRows()) {
            ev.setNextRow(0);
        }
        else {
            ev.setNextRow(ev.getRow() + 1);
        }
        ev.setNextColumn(0);
    }
}
```

6.2 Resizing Rows and Columns

6.2.1 Default Resizing Behavior

JClass LiveTable allows a user to interactively resize a row and/or column (when allowed by `AllowCellResize`). This action routine alters the `PixelHeight` property when resizing rows, and the `PixelWidth` property when resizing columns.

Users can position the mouse pointer over a cell/label border and click-and-drag to resize the row/column. If users position the mouse pointer over the corner of a cell/label, the mouse drag will resize the row and column simultaneously.

6.2.2 Disallowing Cell Resizing

Use the `setAllowCellResize()` method to control interactive row/column resizing over the entire table. The valid parameters of the `AllowCellResize` property are:

- `JCTableEnum.RESIZE_ALL`: user resizing of cell permitted (default).
- `JCTableEnum.RESIZE_NONE`: no row/column resizing is allowed.
- `JCTableEnum.RESIZE_COLUMN`: only columns may be resized.
- `JCTableEnum.RESIZE_ROW`: only rows may be resized.

6.2.3 Controlling Resizing

You can use a `JCResizeCellListener` (registered with `addResizeCellListener(JCResizeCellListener)`) to control interactive row/column resizing on a case-by-case basis. `JCResizeCellEvent` is the event posted as a user resizes a row and/or column, with valid stages being `BEFORE_RESIZE`, `RESIZE`, and `AFTER_RESIZE`.

The `getColumn()` method gets the column being resized. The `getCurrentColumnWidth()` and `getCurrentRowHeight()` methods get the current column width and the current row

height respectively. The `NewColumnWidth` and `NewRowHeight` properties can set and retrieve information on the new column width and the new row height respectively.

As a cell is resized by the user, a `JCResizeCellEvent` is triggered, which passes objects to `JCResizeCellMotionListener` during the event.

`beforeResizeCell(JCResizeCellEvent)` is sent the initial values (as specified by `getCurrentColumnWidth()` and `getCurrentColumnHeight()`). When the user commits the change by releasing the mouse button, the end value from

`resizeCell(JCResizeCellEvent)` is available for retrieval (by `getNewColumnWidth()` or `getNewRowHeight()`) or changing (by `setNewColumnWidth()` and `setNewRowHeight()`), and `afterResizeCell()` is called with final results.

Note: Interactively resizing cannot exceed the set minimum and maximum cell sizes.

The following example event listener routine sets the width of any resized column to an increment of 10 pixels:

```
public class MyTable extends Frame implements JCResizeCellListener {
    ...
    public void beforeResizeCell(JCResizeCellEvent ev) {}
    public void resizeCell(JCResizeCellEvent ev) {
        ev.setNewColumnWidth(ev.getNewColumnWidth() / 10 * 10);
    }
    public void afterResizeCell(JCResizeCellEvent ev) {};
```

To register the above event listener routine, use the following call (where this refers to the class `MyTable`, which implements the `JCResizeCellListener` interface):

```
table.addResizeCellListener(this);
```

Resizing all Rows or Columns at Once

You can configure your `JClass LiveTable` program so that when a user interactively resizes a row or column, all of the other rows or columns in the table resize to the same value. This is achieved by setting the `ResizeEven` property to `true` using the following method:

```
table.setResizeEven(true);
```

Setting this property overrides row and column height and width properties, since the rows and columns are all set to the same value as the row and column the user resized.

Resizing Using Only Labels or Cells

As you've seen above, you can control how users can resize cells, rows, columns, and labels. `JClass LiveTable` also allows you to set the resizing capability so that users can only resize rows and/or columns using the row and column labels.

The `setAllowResizeBy()` method determines how table rows and columns are resized. Use `RESIZE_BY_LABELS` to allow resizing only with labels. The mouse pointer will not change to a resize arrow over cell borders in the body of the table.

Using `RESIZE_BY_CELLS` achieves the opposite, while the default, `RESIZE_BY_ALL` allows resizing with both cells and labels.

6.3 Table Scrolling

6.3.1 Default Scrolling Behavior

When a table is larger than the rows/columns visible on the screen, an end-user can scroll through the table with the mouse or keyboard. JClass LiveTable uses two scrollbar components (one horizontal, one vertical) to implement table scrolling.

JClass LiveTable can also scroll the table when requested by other interactions, such as cell traversal, mouse dragging, or cell selection. Scrolling does not change the location of the current cell.

You can control how and where scrollbars are attached to the component, when they are displayed, and how they behave. The following sections outline programming scrollbar behavior. For information about displaying scrollbars, and setting scrollbar display properties, please refer to [Scrollbars](#), in Chapter 2.

6.3.2 Managing Table Scrolling

Jump Scrolling

You can configure the table to scroll smoothly (by pixel) through the table or to use jump scrolling, which is scrolling the table one whole row or column at a time. This behavior is controlled by calling `setJumpScroll()` with one of the following parameters:

- `JCTableEnum.JUMP_NONE`: neither horizontal nor vertical scrollbars will use jump scrolling (default)
- `JCTableEnum.JUMP_HORIZONTAL`: only the horizontal scrollbar will use jump scrolling
- `JCTableEnum.JUMP_VERTICAL`: only the vertical scrollbar will use jump scrolling
- `JCTableEnum.JUMP_ALL`: both the vertical and horizontal scrollbars will use jump scrolling

Using Automatic Scrolling

You can configure the table to scroll automatically whenever a user selects cells or drags the mouse past the edge of the visible table area. To do this, you must call the `setAutoScroll()` method, specifying one of the following parameters:

- `JCTableEnum.AUTO_SCROLL_NONE` (default)
- `JCTableEnum.AUTO_SCROLL_ROW`
- `JCTableEnum.AUTO_SCROLL_COLUMN`
- `JCTableEnum.AUTO_SCROLL_BOTH`

Note that automatic scrolling is disabled when no scrollbars are visible and when jump scrolling is enabled.

Disabling Interactive Scrolling

Scrolling can be disabled in one or both directions. Mouse and keyboard scrolling cannot be disabled separately.

Remove the scrollbars from the screen by setting `HorizSBDisplay` and/or `VertSBDisplay` to `JCTblEnum.SCROLLBAR_NEVER`.

To fully disable any and all scrolling, an application should also ensure that the user cannot select cells or traverse to cells outside the visible area.

Forcing Scrolling

An application can force the table to scroll in any of the following four ways.

- First, to scroll a particular row to the top of the display, set the `TopRow` property to the number of the row you want to display at the top. For example, to display the fifth row at the top of the table:
`setTopRow(4)`
- Second, to scroll a particular column to the left side of the display, set the `LeftColumn` property to the column number that you want to display. For instance, to display the thirteenth column at the left of the table:
`setLeftColumn(12)`
- Third, to determine whether a row or column is visible, call the `JCTable.isRowVisible()` or `JCTable.isColumnVisible()` methods. To check whether a particular cell is visible, use `JCTable.isCellVisible()`.
- Finally, to scroll to display a particular cell, call the `makeVisible()` method for that cell's context. For example: `makeVisible(4, 21)`
(You can also call the `makeRowVisible()` and `makeColumnVisible()` methods for entire rows and columns.)

Mouse Wheel Support

`JClass LiveTable` has built in mouse wheel support, if mouse wheel support is available in the underlying JDK (JDK 1.4 or higher). By default, a table adds a `TableMouseWheelListener` which listens for `MouseWheelEvents` and changes the value of the vertical or horizontal scrollbar, depending on which ones are visible. The vertical scrollbar is used if visible; otherwise, the horizontal scrollbar is used if visible. Mouse wheel support can be disabled by calling `removeTableMouseWheelListener()`, or the default listener can be replaced by calling `addTableMouseWheelListener()` with a new `MouseWheelListener`.

A table scrolls one unit for every click of a scrollbar arrow. This unit value can be set by calling `getVertSB()` or `getHorizSB()` and setting the appropriate property on the `JScrollbar` object that is returned. By default, `JClass LiveTable` sets the unit to 20 pixels for a horizontal scrollbar and 21 pixels for a vertical scrollbar. If mouse wheel support is

enabled, rolling the mouse wheel one click will scroll the table the number of units that your mouse software has been configured to scroll. For example, if this value is set to three and the unit value of the scrollbar is set to 20 pixels, rolling the mouse one click will cause the table to scroll 60 pixels, the equivalent of clicking the corresponding scrollbar arrow three times.

If `jumpScroll` is set on the scrollbar, scrolling the mouse wheel one click will cause exactly one row or column to be scrolled in the appropriate direction. In this case, one mouse wheel click is exactly the same as clicking once on the corresponding scrollbar arrow.

Tracking Scrollbars

The behavior of scrollbars during tracking can be set by using `setHorizSBTrack()` for horizontal scrollbars and `setVertSBTrack()` for vertical scrollbars. Scrolling behavior can be set two ways.

Using `JTableEnum.TRACK_LIVE`, the table redisplay while the user scrolls. This type of scrollbar tracking can be resource intensive, particularly with larger tables.

An alternate way of tracking scrollbars is to use `JTableEnum.TRACK_COLUMN_NUMBER` for horizontal scrollbar tracking, and `JTableEnum.TRACK_ROW_NUMBER` for vertical scrollbar tracking. In these cases, the table does not redisplay until scrollbar tracking is complete, but an indicator appears beside the scrollbar that informs the user where in the table the scrolling has taken them.

When using this kind of tracking, the indicator's appearance is set using `setTrackBackground()`, `setTrackForeground()`, and `setTrackSize()`. The contents of the indicator can either be the row/column number, or the contents of a cell or label.

To display the contents of a cell or label, use `setHorizSBTrackRow()` and `setVertSBTrackColumn()` to specify which row or column's data will be used in the scroll tracking indicator, and use `JTableEnum.TRACK_ROW`, `JTableEnum.TRACK_COLUMN`, or `JTableEnum.LABEL` to specify the specific row number (for vertical scrolling), column number (for horizontal scrolling), or label whose data will be used in the indicator.

For example, the following line of code sets the vertical tracking so that it displays the contents of the second column:

```
table.setVertSBTrackColumn(1);
```

6.3.3 Scroll Listener Methods

`JClass LiveTable` provides a way for your application to be notified when the table is scrolled by either the end-user or the application. The `JCScrollListener` (registered with `addScrollListener(JCScrollListener)`) allows you to define a procedure to be called when the table scrolls; this is useful if your application is drawing into the table. The method is sent an instance of `JCScrollEvent`.

The example below shows how to use the `scroll(JCScrollEvent)` and `afterScroll(JCScrollEvent)` scrollbar interface methods to store an internal state:

```
public MyClass extends Frame implements JCScrollListener {
    ....
    public void scroll(JCScrollEvent ev) {
        if (ev.getDirection() == TableScrollbar.HORIZONTAL)
            hScrollingActive = true;
        else if (ev.getDirection() == TableScrollbar.VERTICAL)
            vScrollingActive = true;
    }
    public void afterScroll(JCScrollEvent ev) {
        if (ev.getDirection() == TableScrollbar.HORIZONTAL)
            hScrollingActive = false;
        else if (ev.getDirection() == TableScrollbar.VERTICAL)
            vScrollingActive = false;
    }
}
```

To register the above event listener routine, use the following call (where `(this)` refers to the class `MyClass`, which implements the `JCScrollListener` interface):

```
table.addScrollListener(this);
```

6.4 Cell Selection

6.4.1 Default Cell Selection

Cell selection is not enabled by default. When cell selection is enabled (see Section 6.4.3, [Customizing Cell Selection](#)), the default selection behavior is as follows:

- Clicking a cell, holding the mouse button down, and dragging selects those cells.
- Clicking a label selects all the cells in the column or row.
- Holding down the **Shift** key while clicking and dragging modifies the selection (that is, it does not clear the previous selection).
- Holding down the **Ctrl** key and making a sequence of selections adds the selections together.
- Clicking a cell, traversing out of the cell, then traversing back to the clicked cell selects the cell without editing it.

`JClass LiveTable` allows a user to interactively select one or more ranges of cells. An application can retrieve each range to manipulate the cells within it. An application can also be notified of each user selection to control what and how the user selects cells.

`JClass LiveTable` supports a number of selection policies, including:

- `JCTableEnum.SELECT_MULTIRANGE`: multirange selection (selecting multiple ranges of cells)
- `JCTableEnum.SELECT_RANGE`: single range
- `JCTableEnum.SELECT_SINGLE`: single cell





- `JTableEnum.SELECT_NONE`: no selection.

6.4.2 Selection Colors

By default, selected cells and labels display with reversed colors, that is, the background and foreground colors are inverted under selection. When programming the appearance of your table, you can set the colors for selected cells. For more information, please see [Cell Selection Colors](#), in Chapter 2.

6.4.3 Customizing Cell Selection

The `SelectionPolicy` property controls the amount of selection allowed on the table, both by end-users and by the application. Changing the selection policy affects subsequent selection attempts; it does not affect current selections. The following illustration shows the valid values, and the amount of selection they allow.

Selection Policy	Example
selection disabled <code>JTableEnum.SELECT_NONE</code>	
single cell selection <code>JTableEnum.SELECT_SINGLE</code>	
single range selection <code>JTableEnum.SELECT_RANGE</code>	
multiple range selection <code>JTableEnum.SELECT_MULTIRANGE</code>	

When `SelectionPolicy` is set to `JTableEnum.SELECT_NONE` (default), `JCSelectEvent` events are not posted as a user edits or attempts to select cells. Note that setting this property does not change the selected cell list – this means that if a cell is already selected, then changing this property won’t clear the list. As an example, if your selection policy was set to `MULTI_RANGE` and you selected multiple ranges of cells, a change to `RANGE`,

SINGLE or NONE will **not** modify the current selection, that is, the current selection will not honour the selection policy.

Selecting Row/Column Labels

By default, when a user clicks on a row or column label, the entire row or column, including the label is highlighted. To change it so that the label is not highlighted with the rest of the cells, set `SelectIncludeLabels` to `false`:

```
table.setSelectIncludeLabels(false);
```

6.4.4 Selected Cell List

The `SelectedCells` property specifies the collection of all currently selected ranges in the table, where each element is an instance of a `JCCellRange`. `SelectedCells` is updated dynamically as a user selects cells. It is also updated when an application programmatically selects or deselects cells. Labels cannot be part of a selected range.¹

Each range in the selected cell list is a `JCCellRange` structure. Its variables include:

- `start_column`
- `start_row`
- `end_column`
- `end_row`

The `start_column` and `start_row` variables represent the first cell in the range (top-left corner), while the `end_column` and `end_row` variables represent the last cell in the range (bottom-right corner).

All members of the `JCCellRange` structure can be a row and column index. `end_row` and `end_column` can also be set to `MAXINT`, which specifies all of the cells in a row or column. Because the user can make a selection at any point and in any direction within a table, the start point is not necessarily the top-left corner of the range – it may be anywhere within the table.

6.4.5 Working with Selected Ranges

To get a selected range, use `getSelectedCells()`. A table's set of selected cells is a collection of `JCCellRange` instances. This method has the following prototype:

```
public Collection getSelectedCells()
```

Each element of the `Collection` is an instance of a `JCCellRange`. This value is updated dynamically as a user selects cells. The selection policy controls the amount of selection allowed on the table, both by users and by the application.

1. Clicking a label selects all of the cells in the row or column, including the label.

Adding to the current selection requires the use of `addRowSelection()`, `addColumnSelection()`, or `addSelection()`.

An application can add a selection to the selected cell list by adding the new range to the `SelectedCells` Collection, as shown by the following code fragment:

```
Collection col = table.getSelectedCells();
col.add(new JCCellRange(1, 1, 3, 3));
```

6.4.6 Removing Selections

To remove all selections from the table, call `clearSelection()`.

6.4.7 Runtime Selection Control

You can use `JCSelectListener` (registered with `addSelectListener(JCSelectListener)`) to control interactive cell selection at each stage, on a case-by-case basis. `JCSelectEvent` has a number of methods and properties, enabling the programmer to modify the `JCSelectEvent`. The `getAction()` method retrieves one of the following to determine how the cell was selected:

- `SELECT` – selects the cell if `SelectionPolicy` is not `SELECT_NONE`.
- `EXTEND` – extends the selected region to include cell if `SelectionPolicy` is `SELECT_RANGE` or `SELECT_MULTIRANGE`.
- `ADD` – selects the cell if `SelectionPolicy` is to `SELECT_MULTIRANGE`.
- `END` – finishes a selection.
- `DESELECT` – cancels the cell selection.

The `setCancelled()` method determines whether the selection (or unselection) should be allowed (default is `false`). The `Row` and `Column` properties set or retrieve the respective value of the row or column being selected or unselected.

`JCSelectListener` is called before selection begins (`beforeSelect(JCSelectEvent)`), after the user's selection is complete (`select(JCSelectEvent)`) and after all listeners have been notified that the selection is complete (`afterSelect(JCSelectEvent)`).

6.5 Dragging Rows and Columns

You can configure your `JClass LiveTable` program to allow users to drag rows and columns to a new position in the table. This feature is implemented using the `RowTrigger` and `ColumnTrigger` properties to specify a key-mouse-click combination for dragging a row or column by its label. For example, you can specify that when a user holds the **Shift** key and clicks on a row label, the user can drag that row to another location in the table. When dragging is enabled, the mouse pointer turns into a hand to indicate that the row or column can be dragged.

To enable users to drag rows and columns by holding down the **Shift** key and clicking on row or column labels, first call `addAction()`, with which you define the action's initiation, as well as the action itself.

Here is a code snippet showing the `addAction()` method in use:

```
// Action for dragging columns
table.addAction(new TableAction(ini, JCTableEnum.COLUMN_DRAG_ACTION));
// Action for dragging rows
table.addAction(new TableAction(ini, JCTableEnum.ROW_DRAG_ACTION));
```

For dragging, the settings for `TableAction` are `JCTableEnum.COLUMN_DRAG_ACTION` and `JCTableEnum.ROW_DRAG_ACTION`.

Dragging a row or column affects only the data view. It does not change the data source.

6.6 Sorting Columns

You can easily program your JClass LiveTable applications and applets to allow users to sort columns in the table. Sorting columns rearranges the rows in the table display, but *does not* affect the data source of the table. By default, sort behavior does not sort frozen rows set with the `setFrozenRows()` method (see [‘Freezing’ Rows and Columns](#), in Chapter 2).

The `sortByColumn()` method compares objects based on the type of data found in the data source. As such, in some cases, sorting results may vary. For example, using `sortByColumn(0, Sort.ASCENDING)`, where the data used for column 1 are Strings, the String “14” will be considered greater than “110.” However, if these same numerical values are represented as integers, 110 will be greater than 14.

Sorting a single column

To sort a single column in the data view, call the `sortByColumn()` method, specifying the column number to sort, and the direction (`Sort.ASCENDING` or `Sort.DECENDING`):

```
sortByColumn(2, Sort.DECENDING);
```

You can specify that only a particular range of rows is sorted using this variation on the `sortByColumn()` method with the following construction:

```
table.sortByColumn(int col,
                  int direction,
                  int start_row,
                  int end_row)
```

The following code sorts rows 2 to 18 in column 2 in descending order.

```
sortByColumn(1, Sort.DECENDING, 1, 17);
```

Sorting Based on Multiple Columns

You can sort columns based on the values of cells in more than one column using the following method construction:

```
table.sortByColumn(int col[],
                  int direction[])
```

This method requires that you specify an array of columns on which to base the sorting, and an array of directions in which to sort the columns.

When the sort begins, the rows are sorted based on the first column in the array. If two or more rows contain the same value at the first column, the second column in the array is used to sort the identical values. This process continues until there are no duplicate values in a column, or until the end of the column array is reached.

Consider the following example:

	Column 0	Column 1	Column 2	Column 3
Row 0	A	20	Z	2
Row 1	G	7	A	4
Row 2	Z	8	B	5
Row 3	B	11	Z	4
Row 4	A	10	C	1

To sort based on the cell values in columns 0, 1, and 3, use the following code:

```
int [] columns = {0, 1, 3};
int [] direction = {Sort.ASCENDING, Sort.ASCENDING, Sort.ASCENDING};
table.sortByColumn(columns, direction);
```

In this case, the sort is first based on the data in the rows in column 0. Since column 0 contains two cells with values 'A' (Rows 0 and 4), the sort moves to the next column (1) in the array to determine how to sort the two 'A' rows. Row 0 at Column 1 has a value of 20 and Row 4 at Column 1 has a value of 10. Since these are sorted in ascending order, the outcome of the sort is:

	Column 0	Column 1	Column 2	Column 3
Row 4	A	10	C	1
Row 0	A	20	Z	2
Row 3	B	11	Z	4

	Column 0	Column 1	Column 2	Column 3
Row 1	G	7	A	4
Row 2	Z	8	B	5

If there had been duplicate values in column 1, these would have been sorted based on the values in the third column in the array (3).

You can also specify that the sorting operation affect a given range of rows using the following method:

```
table.sortByColumn(int col[],
                  int direction[],
                  int start_row,
                  int end_row)
```

To sort the example above from row 2 to row 4, use the following code:

```
int [] columns = {0, 1, 3};
int [] direction = {Sort.ASCENDING, Sort.ASCENDING, Sort.ASCENDING};
table.SortByColumn(columns, direction, 2, 4);
```

6.6.1 Sort by Clicking on a Column Label

With JClass LiveTable you can easily configure your table to sort columns based on a key-mouse-click combination on the column's label. For example, you can specify that when a user holds the **Ctrl** key and clicks the column label, that column gets sorted in ascending order.

To enable sorting by clicking, call `addAction()`, with which you define the action's initiation, as well as the action itself. For dragging, the settings for `TableAction` are `JCTableEnum.COLUMN_DRAG_ACTION` and `JCTableEnum.ROW_DRAG_ACTION`.

6.6.2 Resetting the Table after Sorting

To clear all of the changes to the display resulting from column sorting, call the `resetSortedRows()` method, which resets the display to match the data source.

6.7 Custom Mouse Pointers

When tracking the mouse pointer, JClass LiveTable considers the current settings of `AllowCellResize` properties. The `getAllowCellResize()` method retrieves the table's `AllowCellResize` value. The `setAllowCellResize()` method sets how an end-user can interactively resize rows/columns; valid values are `JCTableEnum.RESIZE_ALL` (default), `JCTableEnum.RESIZE_NONE`, `JCTableEnum.RESIZE_COLUMN`, and `JCTableEnum.RESIZE_ROW`.

Disabling Pointer Tracking

To use an application-defined mouse pointer over the entire component, set `TrackCursor` to `false`; `JClass LiveTable` will not track the position of the mouse over the component. By default, `TrackCursor` is set to `true`.

Events and Listeners

Displaying Cells ■ *Editing Cells* ■ *Painting Tables* ■ *Printing Tables* ■ *Resizing Cells*
Scrolling in Tables ■ *Selecting Cells* ■ *Sorting Table Data* ■ *Table Data Changes* ■ *Traversing Cells*

The following sections explain how to generate and receive events in your JClass LiveTable programs.

The descriptions are listed in sets of events and event listeners, with examples of when you would use the event and listener, and sample code.

In order to register an event listener in your program, it must implement the listener's interface.

7.1 Displaying Cells

JCellDisplayEvent

This event is posted for every cell that is displayed in the table. When you receive a `JCellDisplayEvent` object, you can call following methods:

- `getCellData()` returns the object to be passed to the renderer of the given cell.
- `getRow()` retrieves the row number of the cell or label displayed.
- `getColumn()` retrieves the column number of the cell or label displayed.
- `setDisplayData()` lets you change the object that is displayed.
- `getDisplayData()` retrieves the object to be displayed.

A display request from `JTable` generates a `JCellDisplayEvent` and notifies any `JCellDisplayListeners` that they can customize the display object by calling `setDisplayData()` on the event. `JTable` does not generate the event if there are no listeners registered with the table.

JCellDisplayListener

To register the above event listener routine, use the following call (where `this` refers to the class, which implements the `JCellDisplayListener` interface):

```
table.addCellDisplayListener(this);
```

JCellDisplayListener requires the following method to be implemented:

```
public void cellDisplay(JCellDisplayEvent e)
```

Calling JCellDisplayEvent and JCellDisplayListener methods

JCellDisplayListener's method is called before each cell is rendered, and all JCellDisplayEvent methods are available at all times during the display process. For more information, please refer to [Appendix A](#), which provides a complete event summary.

Using JCellDisplay Events and Listeners

JCellDisplayListener can be used to format the display String. Changing displayed data does not affect either data source values or values passed to editors. As such, JCellDisplayEvent does not provide any mechanism to store the displayed data in the data source. The following example (see *examples/table/listeners/BooleanDisplay.java*) displays objects as yes/no. Setting the display object does not have any effect during edit.

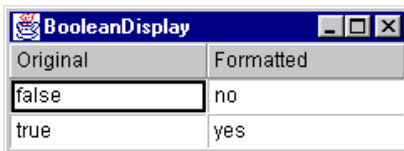


Figure 14 Using JCellDisplayEvent to display BooleanCellData objects as yes/no Strings.

```
import com.klg.jclass.table.JTable;
import com.klg.jclass.table.JTableEnum;
import com.klg.jclass.table.data.JCEditableVectorDataSource;
import com.klg.jclass.table.JCellDisplayListener;
import com.klg.jclass.table.JCellDisplayEvent;
import com.klg.jclass.util.swing.JCExitFrame;
import java.awt.Color;
import java.awt.GridLayout;
import javax.swing.JPanel;

public class BooleanDisplay extends JPanel implements JCellDisplayListener
{

    // Table instance
    protected JTable table;

    // Editable table data source
    protected JCEditableVectorDataSource evds;

    public BooleanDisplay() {
        setBackground(Color.lightGray);

        // Create table instance
        table = new JTable();

        // Create and set up data source
```



```

evds = new JCEditableVectorDataSource();
evds.setNumRows(2);
evds.setNumColumns(2);
evds.setColumnLabel(0, "Original");
evds.setColumnLabel(1, "Formatted");
// Note that BooleanCellEditor will be automatically chosen by Table
evds.setCell(0, 0, new Boolean(false));
evds.setCell(0, 1, new Boolean(false));
evds.setCell(1, 0, new Boolean(true));
evds.setCell(1, 1, new Boolean(true));

// Connect table data source
table.setDataSource(evds);

// Turn off row labels because they are ugly.
table.setRowLabelDisplay(false);

// Add everything to the panel
setLayout(new GridLayout(1,1));
add(table);

// Add cell display listener.
table.addCellDisplayListener(this);
}

public void cellDisplay(JCCellDisplayEvent e) {
    if(e.getColumn() == 1 && e.getRow() != JCTableEnum.LABEL) {
        // Grab displayed data, in this case a boolean
        Boolean dd = (Boolean)e.getDisplayData();
        if(dd.equals(Boolean.TRUE)) {
            e.setDisplayData("yes");
        }
        else {
            e.setDisplayData("no");
        }
    }
}

public static void main(String args[]) {
    JCExitFrame frame = new JCExitFrame("BooleanDisplay");
    BooleanDisplay bd = new BooleanDisplay();
    frame.getContentPane().add(bd);
    frame.pack();
    frame.setVisible(true);
}
}

```

7.2 Editing Cells

JCEditCellEvent

This event is posted whenever a user traverses into and edits a cell. When you receive a `JCEditCellEvent` object, you can call the following methods:

- `getRow()` – retrieves the row number of the cell that is being edited.
- `getColumn()` – retrieves the column number of the cell that is being edited.
- `getType()` – retrieves the type of edit event, where valid types are `BEFORE_EDIT_CELL`, `EDIT_CELL`, and `AFTER_EDIT_CELL`.
- `getEditingComponent()` – returns the editing component.
- `isCancelled()` – retrieves the cancelled value.
- `setCancelled()` – determines whether to allow an edit.

JCEditCellListener

To register the above event listener routine, use the following call (where `(this)` refers to the class, which implements the `JCEditCellListener` interface):

```
table.addEditCellListener(this);
```

`JCEditCellListener` requires the following methods to be implemented:

```
public void beforeEditCell(JCEnterCellEvent e)
public void editCell(JCEnterCellEvent e)
public void afterEditCell(JCEnterCellEvent e)
```

Calling JCCellDisplayEvent and JCCellDisplayListener methods

`JCEditCellListener`'s `beforeEditCell()` method is used before any cell edits by the user occur. This is the only time an edit can be cancelled. `editCell()` is used when the editor is displayed to the user, and at this point, you cannot cancel the edit.

Once the user's edit action has been completed, `afterEditCell()` is used, committing the final changes on your part. For more information, please refer to [Appendix A](#), which provides a complete event summary.

Using JCEditCell Events and Listeners

The following example (see *examples/table/listeners/EditCell.java*) displays a status comment whenever a user edits a cell.

```
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import com.klg.jclass.table.JCTable;
import com.klg.jclass.table.JCEditCellListener;
import com.klg.jclass.table.JCEditCellEvent;
import com.klg.jclass.table.data.JCEditableVectorDataSource;
import com.klg.jclass.util.swing.JCExitFrame;
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Component;

public class EditCell extends JPanel implements JCEditCellListener {

    // Table instance
    protected JCTable table;
```

```

// Editable data source for table
protected JEditableVectorDataSource evds;

// Label to track table column #
protected JLabel message;

// Messages to appear in the JLabel.
protected String messages[] = {
    "This is the first column",
    "This is the second column",
    "This is the third column",
    "This is the forth column" };

public EditCell() {
    setBackground(Color.lightGray);

    // Create table instance
    table = new JTable();

    // Create and set up data source
    evds = new JEditableVectorDataSource();
    evds.setNumRows(10);
    evds.setNumColumns(4);
    evds.setColumnLabel(0, "First");
    evds.setColumnLabel(1, "Second");
    evds.setColumnLabel(2, "Third");
    evds.setColumnLabel(3, "Forth");
    for(int r = 0; r < evds.getNumRows(); r++)
        for(int c = 0; c < evds.getNumColumns(); c++)
            evds.setCell(r, c, "R"+r+"C"+c);

    // Connect table data source
    table.setDataSource(evds);

    // Turn off row labels because they are ugly
    table.setRowLabelDisplay(false);

    // Add everything to the panel
    this.setLayout(new BorderLayout());
    this.add("North", table);
    this.add("South", message = new JLabel());

    // Add cell Edit event listener
    table.addCellListener(this);
}

public void beforeEditCell(JCEditCellEvent event) {
    message.setText(messages[event.getColumn()]);
}

public void editCell(JCEditCellEvent event) {
    // get the editing component and select all of the text if it
    // is a JTextField component
    Component c = event.getEditingComponent();
    if(c instanceof JTextField) {
        ((JTextField)c).selectAll();
    }
}

```

```

    }
}
public void afterEditCell(JCEditCellEvent event) {
}

public static void main(String args[]) {
    JCExitFrame frame = new JCExitFrame("EditCell");
    EditCell ec = new EditCell();
    frame.getContentPane().add(ec);
    frame.pack();
    frame.setVisible(true);
}
}

```

7.3 Painting Tables

JCPaintEvent

This event is posted before and after a portion of the table is painted. When you receive a `JCPaintEvent` object, you can call the following methods:

- `getStartRow()` – retrieves the start row of the repainted region.
- `getStartColumn()` – retrieves the start column of the repainted region.
- `getEndRow()` – retrieves the end row of the repainted region.
- `getEndColumn()` – retrieves the end column of the repainted region.
- `getType()` – retrieves the paint event type, where valid types are `BEFORE_PAINT` and `AFTER_PAINT`.
- `getCellRange()` – returns a `JCCellRange` containing the painted area.

JCPaintListener

To register the above event listener routine, use the following call (where `this` refers to the class, which implements the `JCPaintListener` interface):

```
table.addPaintListener(this);
```

`JCPaintListener` requires the following methods to be implemented:

```
public void beforePaint(JCPaintEvent e)
public void afterPaint(JCPaintEvent e)
```

Calling JCPaintEvent and JCPaintListener Methods

`JCPaintListener`'s methods, `beforePaint()` and `afterPaint()`, can call `JCPaintEvent` methods at any time, as you are not able to interrupt the cell painting process. For more information, please refer to [Appendix A](#), which provides a complete event summary.

Using JCPaint Events and Listeners

`JCPaintListener` allows you to monitor the repainting of table cells. Labels, frozen cells, and scrollable cells are painted independently.

7.4 Printing Tables

JCPrintEvent

This event is posted when your table is printed. When you receive a `JCPrintEvent` object, you can call the following methods:

- `getGraphics()` – retrieves the current graphics object.
- `getPage()` – retrieves the page number.
- `getPageDimensions()` – retrieves the page dimensions.
- `getPageMargins()` – retrieves the page margins.
- `getPageResolution()` – retrieves the page dpi resolution.
- `getMarginUnits()` – retrieves the margin units (pixels or inches).
- `getNumPages()` – retrieves the total number of pages (handy for *page x of x* footers).
- `getNumHorizontalPages()` – retrieves the number of pages needed to print all of the columns in the table.
- `getNumVerticalPages()` – retrieves the number of pages needed to print all of the rows in the table.
- `getTableDimensions()` – retrieves the dimension needed to print the table on the current page.
- `getType()` – retrieves the print event type, where valid types are `PRINT_HEADER`, `PRINT_BODY`, and `PRINT_FOOTER`.

JCPrintListener

To register the above event listener routine, use the following call (where `this` refers to the class, which implements the `JCPrintListener` interface):

```
table.addPrintListener(this);
```

`JCPrintListener` requires the following methods to be implemented:

```
public void printPageHeader(JCPrintEvent e)
public void printPageFooter(JCPrintEvent e)
public void printPageBody(JCPrintEvent e)
```

Using JCPrint Events and Listeners

`JCPrintListener` allows you to customize the header and footer regions for each page of the printout. [Table Printing](#), in Chapter 8, has details and examples for using the `JCPrintListener`.

7.5 Resizing Cells

JCResizeCellEvent

This event is posted when a cell or label is resized. When you receive a `JCResizeCellEvent` object, you can call the following methods:

- `getRow()` – retrieves the row being resized. Returns `JCTableEnum.NOVALUE` if only a column is being resized.
- `getColumn()` – retrieves the column being resized. Returns `JCTableEnum.NOVALUE` if only a row is being resized.
- `getCurrentRowHeight()` – retrieves the current row height. Returns `JCTableEnum.NOVALUE` if only a column is being resized.
- `getCurrentColumnWidth()` – retrieves the current column width. Returns `JCTableEnum.NOVALUE` if only a row is being resized.
- `getType()` – retrieves the type where valid types are `BEFORE_RESIZE`, `RESIZE`, `RESIZE_DRAG` and `AFTER_RESIZE`.
- `getNewRowHeight()` – retrieves the new row height. Returns `JCTableEnum.NOVALUE` if only a column is being resized.
- `setNewRowHeight()` – sets the new row height.
- `getNewColumnWidth()` – retrieves the new column width. Returns `JCTableEnum.NOVALUE` if only a row is being resized.
- `setNewColumnWidth()` – sets the new column width.
- `isCancelled()` – retrieves the cancelled value.
- `setCancelled()` – determines whether to allow an interactive resize.

JCResizeCellListener

To register the above event listener routine, use the following call (where `(this)` refers to the class, which implements the `JCResizeCellListener` interface):

```
table.addResizeCellListener(this);
```

`JCResizeCellListener` requires the following methods to be implemented:

```
public void beforeResizeCell(JCResizeCellEvent e)
public void resizeCell(JCResizeCellEvent e)
public void afterResizeCell(JCResizeCellEvent e)
```

JCResizeCellMotionListener

To register the above event listener routine, use the following call (where `(this)` refers to the class, which implements the `JCResizeCellMotionListener` interface):

```
table.addResizeCellMotionListener(this);
```

`JCResizeCellMotionListener` requires the following methods to be implemented:

```
public void resizeCellDragged(JCResizeCellEvent e)
```

Calling JCResizeCellEvent and JCResizeCellListener Methods

JCResizeCellListener's `beforeResizeCell()` method is called once cell resizing begins, and allows the opportunity to programmatically cancel the resize. `resizeCell()` is called once the user releases the mouse button, and the resize is complete from their perspective. Programmatically, you can cancel the resize, or set new column widths or row heights if the cell resize dimension is invalid, or outside the boundaries of predefined maximum/minimum cell sizes.

Once the resize values have been set, `afterResizeCell()` is used, committing the final resize changes. For more information, please refer to [Appendix A](#), which provides a complete event summary.

Using JCResizeCell Events and Listeners

JCResizeCellListener allows you to customize how table resizes on a per-cell basis. The following example (see *examples/table/listeners/ResizeCell.java*) restricts resize so that row labels cannot be resized and no cell can be less than 100 pixels or greater than 200 pixels.

```
import com.klg.jclass.table.JCTable;
import com.klg.jclass.table.JCTableEnum;
import com.klg.jclass.table.data.JCEditableVectorDataSource;
import com.klg.jclass.table.JCResizeCellListener;
import com.klg.jclass.table.JCResizeCellMotionListener;
import com.klg.jclass.table.JCResizeCellEvent;
import com.klg.jclass.util.swing.JCExitFrame;
import java.awt.Color;
import java.awt.GridLayout;
import javax.swing.JPanel;

public class ResizeCell extends JPanel implements JCResizeCellListener,
                                                JCResizeCellMotionListener {

    // Table instance
    protected JCTable table;

    // Editable table data source
    protected JCEditableVectorDataSource evds;

    public ResizeCell() {
        setBackground(Color.lightGray);

        // Create table instance
        table = new JCTable();

        // Create and set up data source
        evds = new JCEditableVectorDataSource();
        evds.setNumRows(100);
        evds.setNumColumns(2);

        evds.setColumnLabel(0, "End-Point");
        evds.setColumnLabel(1, "Drag");

        for(int r = 0; r < evds.getNumRows(); r++) {
            evds.setRowLabel(r, "Row: "+r);
        }
    }
}
```

```

        for(int c = 0; c < evds.getNumColumns(); c++)
            evds.setCell(r,c,"Cell: R"+r+"C"+c);
    }
    // Connect table data source
    table.setDataSource(evds);

    // Add everything to the panel
    this.setLayout(new GridLayout(1,1));
    this.add(table);

    // Add resize cell listener
    table.addResizeCellListener(this);
    // Add resize cell motion listener
    table.addResizeCellMotionListener(this);
}

public void beforeResizeCell(JCResizeCellEvent event) {
    if (event.getColumn() == JCTableEnum.LABEL) {
        event.setCancelled(true);
        return;
    }
}

public void resizeCell(JCResizeCellEvent event) {
    // Width must be between 100 and 200
    int width = event.getNewColumnWidth();
    if (width < 100) {
        event.setNewColumnWidth(100);
    }
    else if (width > 200) {
        event.setNewColumnWidth(200);
    }

    // Height must be between 30 and 70
    int height = event.getNewRowHeight();
    if (height != JCTableEnum.NOVALUE && height < 30) {
        event.setNewRowHeight(30);
    }
    else if (height > 70) {
        event.setNewRowHeight(70);
    }
}

public void afterResizeCell(JCResizeCellEvent event) {
}

public void resizeCellDragged(JCResizeCellEvent event) {
    // restrict the range of motion for column 1 to 100 to 200
    if(event.getColumn() == 1) {
        int width = event.getNewColumnWidth();
        if(width < 100) {
            event.setNewColumnWidth(100);
        }
        else if(width > 200) {
            event.setNewColumnWidth(200);
        }
    }
}

```



```

    }
}

public static void main(String args[]) {
    JCExitFrame frame = new JCExitFrame("ResizeCell");
    ResizeCell rc = new ResizeCell();
    frame.getContentPane().add(rc);
    frame.pack();
    frame.setSize(600, 400);
    frame.setVisible(true);
}
}

```

7.6 Scrolling in Tables

JCScrollEvent

This event is posted when the table is scrolled by either the user or the application. When you receive a `JCScrollListener` object, you can call the following methods:

- `getAdjustable()` – retrieves the affected adjustable object.
- `getDirection()` – retrieves the scrolling direction (either `Adjustable.HORIZONTAL` or `Adjustable.VERTICAL`).
- `getEvent()` – retrieves the event that initiated the action.
- `getType()` – retrieves the scroll event type, where valid types are `JCScrollEvent.SCROLL` and `JCScrollEvent.AFTER_SCROLL`.
- `getValue()` – retrieves the scrollbar’s current value.
- `setValue()` – sets the scrollbar’s current value.

The `JCScrollListener` (registered with `addScrollListener(JCScrollListener)`) allows you to define a procedure to be called when the table scrolls; this is useful if your application is drawing into the table. The method is sent an instance of `JCScrollEvent`.

JCScrollListener

To register the above event listener routine, use the following call (where `this` refers to the class, which implements the `JCScrollListener` interface):

```
table.addScrollListener(this);
```

`JCScrollListener` requires the following methods to be implemented:

```
public void scroll(JCScrollEvent e)
public void afterScroll(JCScrollEvent e)
```

Calling JCScrollEvent and JCScrollListener Methods

`JCScrollListener`’s `scroll()` method is invoked when the user begins to scroll, during which all `JCScrollEvent` methods are available. `afterScroll()` is called when the user

has finished scrolling. For more information, please refer to [Appendix A](#), which provides a complete event summary.

Using JCSroll Events and Listeners

JCSrollListener allows you to synchronize table scrolling with another object. The following example (see *examples/table/listeners/TwoTables.java*) links two tables together with one scrollbar. This example uses two tables inside another table to simulate a splitter window.

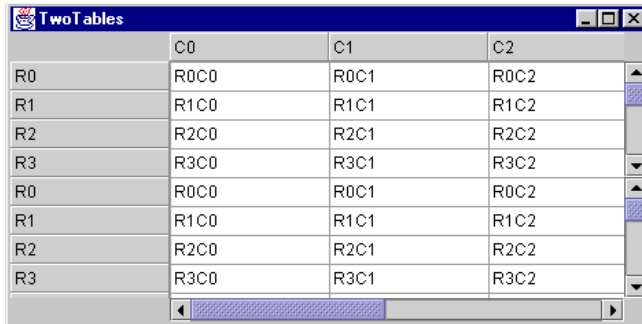


Figure 15 Example using JCSrollListener to synchronize scrolling between two tables.

```
import com.klg.jclass.table.JTable;
import com.klg.jclass.table.JTableEnum;
import com.klg.jclass.table.data.JCVectorDataSource;
import com.klg.jclass.table.data.JCEditableVectorDataSource;
import com.klg.jclass.table.JCSrollListener;
import com.klg.jclass.table.JCSrollEvent;
import com.klg.jclass.util.swing.JCExitFrame;
import java.awt.Color;
import java.awt.GridLayout;
import java.awt.Adjustable;
import java.awt.Component;
import java.awt.Scrollbar;
import javax.swing.JPanel;

public class TwoTables extends JPanel implements JCSrollListener {

    // First table
    protected JTable table1;

    // Second table
    protected JTable table2;

    // Common data source
    protected JCEditableVectorDataSource evds1;

    // Local variable used to avoid infinite loops in
    // scroll event handler
    protected boolean forcedScroll = false;
```

```

public TwoTables() {
    setBackground(Color.lightGray);

    // Create first table
    table1 = new JTable();

    // Create and set up data source for first table
    evds1 = new JEditableVectorDataSource();
    evds1.setNumRows(100);
    evds1.setNumColumns(6);

    for (int c = 0; c < evds1.getNumColumns(); c++)
        evds1.setColumnLabel(c, "C"+c);
    for (int r = 0; r < evds1.getNumRows(); r++) {
        evds1.setRowLabel(r, "R"+r);
        for (int c = 0; c < evds1.getNumColumns(); c++)
            evds1.setCell(r,c,"R"+r+"C"+c);
    }

    // Connect data source to first table.
    table1.setDataSource(evds1);

    // Set up visuals and interactions for table 1.
    table1.setAllowCellResize(JTableEnum.RESIZE_NONE);
    table1.setHorizSBDisplay(JTableEnum.SBDISPLAY_NEVER);
    table1.getDefaultCellStyle().setTraversable(false);
    table1.setVisibleRows(2);
    table1.setVisibleColumns(3);

    // Create second table
    table2 = new JTable();

    // Connect second table to same data source as first table.
    table2.setDataSource(evds1);

    // Set up visuals and interactions for table 2.
    table2.setAllowCellResize(JTableEnum.RESIZE_NONE);
    table2.setColumnLabelDisplay(false);
    table2.setTopRow(2);
    table2.getDefaultCellStyle().setTraversable(false);
    table2.setVisibleRows(5);
    table2.setVisibleColumns(3);

    // Add to panel
    setLayout(new GridLayout(2,1));
    add(table1);
    add(table2);

    // Add scroll listeners for both tables
    table1.addScrollListener(this);
    table2.addScrollListener(this);
}

public void scroll(JCScrollEvent event) {

```

```

// use forcedScroll to prevent an infinite loop, since
// calling setValue() on the scrollbar will generate another
// event.
if (event.getDirection() == Scrollbar.HORIZONTAL) {
    if (forcedScroll == false) {
        // Scroll event not forced by this method, okay
        // to continue

        // Grab adjustable object
        Adjustable adj = event.getAdjustable();
        // We need for it to be a component. Should be -
        // scroll events come from LiveTable's scrollbar
        if (adj != null && adj instanceof Component) {
            Component c = (Component)adj;
            if (c.getParent() == table2) {
                // If table 2 scrolled, synchronize table 2
                forcedScroll = true;
                table1.getHorizSB().setValue(event.getValue());
            }
            else if (c.getParent() == table1) {
                // If table 1 scrolled, synchronize table 2
                forcedScroll = true;
                table2.getHorizSB().setValue(event.getValue());
            }
        }
    } else {
        forcedScroll = false;
    }
}

}

public void afterScroll(JCScrollEvent event) {
}

public static void main(String args[]) {
    JCExitFrame frame = new JCExitFrame("TwoTables");
    TwoTables tt = new TwoTables();
    frame.getContentPane().add(tt);
    frame.pack();
    frame.setVisible(true);
}
}

```

7.7 Selecting Cells

This event is posted when the user selects cells, or cells are selected programmatically. When you receive a `JCSelectEvent` object, you can call the following methods:

- `getType()` – returns the type of selection event, where valid types are `BEFORE_SELECTION`, `SELECTION`, and `AFTER_SELECTION`.
- `getStartRow()` – retrieves the start row of the selected or deselected cell range.

- `getStartColumn()` – retrieves the start column of the selected or deselected cell range.
- `getEndRow()` – retrieves the end row of the selected or deselected cell range.
- `getEndColumn()` – retrieves the end column of the selected or deselected cell range.
- `isCancelled()` – returns true if any listener has rejected the selection.
- `setCancelled()` – determines if selection is allowed.
- `getAction()` – returns the type of selection action, where valid types are SELECT, ADD, EXTEND, DESELECT, and END.
- `getActionString()` – returns a String representation of the action.

JCSelectListener

To register the above event listener routine, use the following call (where `(this)` refers to the class, which implements the `JCSelectListener` interface):

```
table.addSelectListener(this);
```

`JCSelectListener` requires the following methods to be implemented:

```
public void beforeSelect(JCSelectEvent e)
public void select(JCSelectEvent e)
public void afterSelect(JCSelectEvent e)
```

Calling JCSelectEvent and JCSelectListener Methods

`JCSelectListener`'s methods are called when the user begins cell selection in a table. `beforeSelect()` is invoked when the user selects or deselects a cell, and all `JCSelectEvent` methods are available. For example, it is possible to cancel a selection in `beforeSelect()` by calling `setCancelled(true)`.

The `select()` and `afterSelect()` methods are called during and after the selection process, meaning that at that point, the cell is now visually selected from the user's perspective. For more information, please refer to [Appendix A](#), which provides a complete event summary.

Using JCSelect Events and Listeners

`JCSelectListener` allows you to monitor scrolling actions in your table, either before or after the scrolling event. The following example (see *examples/table/listeners/SelectListener.java*) demonstrates the use of `JCSelectListener` notifications to cancel out cell selection.

```
import java.awt.Color;
import java.awt.GridLayout;
import java.applet.Applet;
import javax.swing.JPanel;
import com.klg.jclass.table.JCTable;
import com.klg.jclass.table.JCSelectListener;
import com.klg.jclass.table.JCSelectEvent;
import com.klg.jclass.table.JCTableEnum;
import com.klg.jclass.table.data.JCVectorDataSource;
```

```

import com.klg.jclass.util.swing.JCExitFrame;

public class SelectListener extends JPanel implements JCSelectListener {
    // Table instance
    protected JCTable table;

    // Table data source
    protected JCVectorDataSource ds;

    public SelectListener() {
        setBackground(Color.lightGray);

        // Create table instance
        table = new JCTable();
        // Create and set up data source
        ds = new JCVectorDataSource();
        ds.setNumRows(10);
        ds.setNumColumns(4);

        for(int c = 0; c < ds.getNumColumns(); c++)
            ds.setColumnLabel(c, "Column: "+c);

        ds.setColumnLabel(1, "Non Selectable Column");
        for(int r = 0; r < ds.getNumRows(); r++) {
            ds.setRowLabel(r, "Row: "+r);
            for(int c = 0; c < ds.getNumColumns(); c++)
                ds.setCell(r,c,"Cell: R"+r+"C"+c);
        }

        // Connect table data source
        table.setDataSource(ds);
        table.setSelectionPolicy(JCTableEnum.SELECT_RANGE);

        // Add everything to the panel
        setLayout(new GridLayout(1,1));
        add(table);

        table.addSelectListener(this);
    }

    public void beforeSelect(JCSelectEvent e) {
        if (e.getStartColumn() == 1) {
            e.setCancelled(true);
            System.out.println("We don't want selection starting from this
column");
            return;
        }
        System.out.println("beforeSelect: startRow="+e.getStartRow()+
            ", startColumn="+e.getStartColumn());
    }

    public void select(JCSelectEvent e) {
        if (e.getAction() == JCSelectEvent.EXTEND &&
            Math.abs(e.getStartRow()-e.getEndRow())>1) {
            e.setCancelled(true);
        }
    }
}

```

```

        System.out.println("We don't want selection extending for more than
2 rows");
        return;
    }
    System.out.println("select: startRow="+e.getStartRow()+
        ", startColumn="+e.getStartColumn()+", endRow="+e.getEndRow()+
        ", endColumn="+e.getEndColumn());
}

public void afterSelect(JCSelectEvent e) {
    System.out.println("afterSelect: startRow="+e.getStartRow()+
        ", startColumn="+e.getStartColumn()+", endRow="+e.getEndRow()+
        ", endColumn="+e.getEndColumn());
}

public static void main(String args[]) {
    JCExitFrame frame = new JCExitFrame("SelectListener");
    SelectListener sn = new SelectListener();
    frame.getContentPane().add(sn);
    frame.pack();
    frame.setSize(600, 150);
    frame.setVisible(true);
}
}

```

7.8 Sorting Table Data

JCSortEvent

This event is posted when the table is sorted. When you receive a `JCSortEvent` object, you can call the following methods:

- `getColumns()` – retrieves an array of column indices that were sorted.
- `getNewRows()` – retrieves the newly sorted order.

JCSortListener

To register the above event listener routine, use the following call (where `(this)` refers to the class, which implements the `JCSortListener` interface):

```
table.addSortListener(this);
```

`JCSortListener` requires the following method to be implemented:

```
public void sort(JCSortEvent e)
```

Using JCSort Events and Listeners

`JCSortListener` allows you to synchronize the sorted rows with another object (or to sort the data source). The following example (see *examples/table/listeners/Sorter.java*) uses the row sort array to pull out the top value.

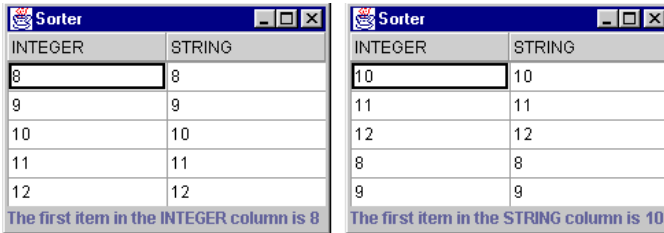


Figure 16 Sorter.java, illustrating how to use JCSort Events and Listeners.

```

import javax.swing.JLabel;
import com.klg.jclass.table.JTable;
import com.klg.jclass.table.JTableEnum;
import com.klg.jclass.table.data.JCEditableVectorDataSource;
import com.klg.jclass.table.JCSortListener;
import com.klg.jclass.table.JCSortEvent;
import com.klg.jclass.table.MouseActionInitiator;
import com.klg.jclass.util.swing.JCExitFrame;
import java.awt.Color;
import java.awt.event.InputEvent;
import java.awt.BorderLayout;
import javax.swing.JPanel;

public class Sorter extends JPanel implements JCSortListener {

    // Table instance
    protected JTable table;

    // Table data source
    protected JCEditableVectorDataSource ds;

    // Label that will display column top value
    protected JLabel toItem;

    public Sorter() {
        setBackground(Color.lightGray);

        // Create table instance
        table = new JTable();

        // Create and set up data source
        ds = new JCEditableVectorDataSource();
        ds.setNumRows(5);
        ds.setNumColumns(2);

        // Column labels
        ds.setColumnLabel(0, "INTEGER");
        ds.setColumnLabel(1, "STRING");

        // Populate data source with generated data.
        int numRows = ds.getNumRows();
        for(int r = 0; r < numRows; r++) {

```



```

        ds.setCell(r, 0, new Integer(r+8));
        ds.setCell(r, 1, "" + (r+8));
    }

    // Connect table data source
    table.setDataSource(ds);

    // Turn off row labels because they are ugly.
    table.setRowLabelDisplay(false);

    // Allow column sorting using a shift-click combination
    table.addAction(new
        MouseActionInitiator(MouseActionInitiator.ANY_BUTTON_MASK,
            InputEvent.SHIFT_MASK),
            JTableEnum.COLUMN_SORT_ACTION);

    // Add everything to the panel
    setLayout(new BorderLayout());
    add(table, BorderLayout.CENTER);
    add(topItem = new JLabel("Shift-click the label to sort numeric or
        string data"),
        BorderLayout.SOUTH);

    // Add sort listener
    table.addSortListener(this);
}

public void sort(JCSortEvent event) {
    int columns[] = event.getColumns();
    int rows[] = event.getNewRows();
    topItem.setText("The first item in the " +
        ds.getTableColumnLabel(columns[0]) + " column is " +
        ds.getTableDataItem(rows[0], columns[0]));
}

public static void main(String args[]) {
    JCExitFrame frame = new JCExitFrame("Sorter");
    Sorter s = new Sorter();
    frame.getContentPane().add(s);
    frame.pack();
    frame.setVisible(true);
}
}

```

7.9 Table Data Changes

JCTableDataEvent

Unlike previous events, `JCTableDataEvent` objects are not thrown by `JCTable`; instead, they come from the table's data source. This event is posted when the `TableDataModel` object has been modified. When you receive a `JCTableDataEvent` object, you can call the following methods:

- `getColumn()` – retrieves the column of the current cell.
- `getRow()` – retrieves the row of the current cell.
- `getNumAffected()` – retrieves the number of rows affected by `TableModel` object changes.
- `getDestination()` – indicates the destination location for MOVE events.
- `getCommand()` – returns the command that initiated the event. Valid commands include:

<code>CHANGE_VALUE</code>	Single cell value changed.
<code>CHANGE_ROW</code>	Single row changed.
<code>ADD_ROW</code>	New row added.
<code>REMOVE_ROW</code>	Row removed.
<code>CHANGE_COLUMN</code>	Single column changed.
<code>ADD_COLUMN</code>	New column added.
<code>REMOVE_COLUMN</code>	Column removed.
<code>CHANGE_ROW_LABEL</code>	Single row label changed.
<code>CHANGE_COLUMN_LABEL</code>	Single column label changed.
<code>MOVE_ROW</code>	Row moved, new location in <code>getDestination</code> .
<code>MOVE_COLUMN</code>	Column moved, new location in <code>getDestination</code> .
<code>NUM_ROWS</code>	Number of rows changed.
<code>NUM_COLUMNS</code>	Number of columns changed.
<code>RESET</code>	Data source significantly changed, should be re-read.

JCTableDataListener

To register the above event listener routine, use the following call (where `(this)` refers to the class `MyClass`, which implements the `JCTableDataListener` interface):

```
table.addTableDataListener(this);
```

`JCDataListener` requires the following method to be implemented:

```
public void dataChanged(JCTableDataEvent e)
```

Using JCTableData Events and Listeners

`JCTableDataListener` allows you to monitor any changes made to the `TableModel` object. Valid changes are listed above with the `getCommand()` method.

There are examples included with your JClass LiveTable distribution that demonstrate the use of data with a table. See [Creating your own Data Sources](#), in Chapter 3, to see descriptions of the included sample code.

7.10 Traversing Cells

JCTraverseCellEvent

This event is posted when cells in the table are traversed. When you receive a `JCTraverseCellEvent` object, you can call the following methods:

- `getColumn()` – retrieves the column of the current cell.
- `getNextColumn()` – retrieves the targeted column for traversal.
- `getRow()` – retrieves the row of the current cell.
- `getNextRow()` – retrieves the targeted row for traversal.
- `setNextRow()` – sets the row of the cell to which the user will traverse.
- `setNextColumn()` – sets the column of the cell to which the user will traverse.
- `getTraverseType()` – returns the action that caused the traverse. Valid `JCTableEnum` action values are: `TRAVERSE_POINTER`, `TRAVERSE_DOWN`, `TRAVERSE_UP`, `TRAVERSE_LEFT`, `TRAVERSE_RIGHT`, `TRAVERSE_PAGEUP`, `TRAVERSE_PAGEDOWN`, `TRAVERSE_HOME`, `TRAVERSE_END`, `TRAVERSE_TOP`, `TRAVERSE_BOTTOM`, and `TRAVERSE_TO_CELL`.
- `getTraverseTypeString()` – returns a `String` value for the traverse type.
- `isCancelled()` – retrieves the cancelled value.
- `setCancelled()` – determines whether to allow an interactive resize.
- `getType()` – retrieves valid event types, which are `TRAVERSE_CELL` and `AFTER_TRAVERSE_CELL`.
- `toString()` – returns a `String` representation of the event.

JCTraverseCellListener

To register the above event listener routine, use the following call (where `(this)` refers to the class, which implements the `JCTraverseCellListener` interface):

```
table.addTraverseCellListener(this);
```

`JCTraverseCellListener` requires the following methods to be implemented:

```
public void traverseCell(JCTraverseCellEvent e)
public void afterTraverseCell(JCTraverseCellEvent e)
```

Calling JCTraverseCellEvent and JCTraverseCellListener Methods

`JCTraverseCellListener`'s `traverse()` method is invoked when the user begins to traverse to a neighboring cell. Since this method is called before the actual traversal, all `JCTraverseCellEvent` methods are available. For example, if `setCancelled()` is called,

and is set to true, the cell traversal is cancelled. You can also call methods that permit cell skipping during traversal.

`afterTraverseCell()` is invoked after valid cell traversal. The `setNextRow()`, `setNextColumn()`, and `setCancelled()` methods are unavailable during `afterTraverseCell()`. For more information, please refer to [Appendix A](#), which provides a complete event summary.

Using JCTraverse Events and Listeners

`JCTraverseCellListener` allows you to control how cell traversal occurs in `JClass LiveTable`. The following example (see *examples/table/listeners/SkipNavigation.java*) uses a `JCTraverseCellListener` to skip the second column if navigating from the first column. The column is not skipped if navigating from the third column.

```
import com.klg.jclass.table.JTable;
import com.klg.jclass.table.JTableEnum;
import com.klg.jclass.table.data.JCVectorDataSource;
import com.klg.jclass.table.JCTraverseCellListener;
import com.klg.jclass.table.JCTraverseCellEvent;
import com.klg.jclass.util.swing.JCExitFrame;
import java.awt.Color;
import java.awt.GridLayout;
import javax.swing.JPanel;

public class SkipNavigation extends JPanel implements
JCTraverseCellListener {

    // Table instance
    protected JTable table;

    // Table data source
    protected JCVectorDataSource ds;

    public SkipNavigation() {
        setBackground(Color.lightGray);

        // Create table instance
        table = new JTable();

        // Create and set up data source
        ds = new JCVectorDataSource();
        ds.setNumRows(10);
        ds.setNumColumns(4);

        for(int c = 0; c < ds.getNumColumns(); c++)
            ds.setColumnLabel(c, "Column: "+c);

        ds.setColumnLabel(1, "Skip from 0 to 2");
        for(int r = 0; r < ds.getNumRows(); r++) {
            ds.setRowLabel(r, "Row: "+r);
            for(int c = 0; c < ds.getNumColumns(); c++)
                ds.setCell(r,c,"Cell: R"+r+"C"+c);
        }
    }
}
```

```

// Connect table data source
table.setDataSource(ds);

// Add everything to the panel
setLayout(new GridLayout(1,1));
add(table);

// Add traverse cell listener
table.addTraverseCellListener(this);
}

public void traverseCell(JCTraverseCellEvent event) {
// Skip second column when approaching from the left
if (event.getColumn() == 0 && event.getNextColumn() == 1) {
    event.setNextColumn(2);
}

// Skip second column in both directions.
// if(event.getNextColumn() == 1)
//     event.setNextColumn(1 + event.getNextColumn() -
//         event.getColumn());
}

public void afterTraverseCell(JCTraverseCellEvent e) {
}

public static void main(String args[]) {
    JCExitFrame frame = new JCExitFrame("SkipNavigation");
    SkipNavigation sn = new SkipNavigation();
    frame.getContentPane().add(sn);
    frame.pack();
    frame.setSize(600, 150);
    frame.setVisible(true);
}
}

```


Table Printing

Printing ■ *Print Preview*

Although `JClass LiveTable` is a grid/table component, it still allows end users to print and print–preview table applications. By using the `JCPrintTable` class, you can control layout, formatting, and header/footer information.

8.1 Printing

The `JCPrintTable` class offers print functionality in `JCTable`. The following code creates a `JCPrintTable` and prints `JCTable` with the default print options:

```
JCPrintTable pt = new JCPrintTable(myTable);  
pt.print();
```

Default printed pages consist of:

- 1” margins
- no header information
- a footer message: page x of y
- all table pages printed (which cannot be changed)

The `JCPrintTable` class creates a copy of the table’s visible properties and retrieves cell contents from the data source. Cell height and width are copied by actual pixel size. Scrollbars are not part of the printed table.

Using `JCPrintTable` offers controls over various aspects of your printed pages.

8.1.1 Setting Page Layout Properties

The `JCPrintTable` class provides methods for detailed control of print output from a `JClass LiveTable` application or applet.

Page Size

The following methods define printed page sizes in pixels:

```
setPageDimensions();  
setPageWidth();  
setPageHeight();
```

Use the `getPageDimensions()`, `getPageWidth()`, and `getPageHeight()` methods to retrieve page sizes by retrieving page information from the printer. By default, the standard A4 page (8½" x 11") is used.

Page Margins

Page margins are set using the `setPageMargins()` method. This method uses the `java.awt.Insets` class to set the margins as in the following example:

```
printtable.setPageMargins(new Insets(54,36,36,54));
```

By default, using the `Insets` object to respectively specify top, left, bottom and right insets will set the margins in pixels. To specify margin units in inches, use the variable `MARGIN_IN_INCHES` in the `getMarginUnits()` method:

```
setMarginUnits(JCPrintTable.MARGIN_IN_INCHES);
```

You can retrieve page margins based on the `Insets` of the page using the `getPageMargins()` method. Use the `getDefaultPageMargins()` to retrieve the default `Insets`.

Page Numbering

To control page numbering, use `getNumHorizontalPages()` and `getNumVerticalPages()` to determine the number of pages across and down. Use `getNumPages()` to determine the total number of pages required to print the table, based on how you have defined the page and margin sizes.

8.1.2 Page Resolution

Use the `getPageResolution()` method to get the printer page resolution. The default is 72 pixels per inch. Use `setPageResolution()` to set the printer page resolution.

8.1.3 Printing Headers and Footers

Headers and footers are applied using `JCPrintListener` receiving `JCPrintEvent` events. A `JCPrintEvent` is posted for each page during printing, and provides a graphic object clipped to the allowable paint region, the page number of the current page, and the total number of pages:

```
public JCPrintEvent
    (Table table, Graphics gc, int page, int numPages, int Type);
public Graphics getGraphics();
public Insets getPageMargins();
public int getMarginUnits();
public int getNumHorizontalPages();
public int getNumPages();
public int getNumVerticalPages();
public int getPage();
public Dimension getPageDimensions();
public int getPageResolution();
public Dimension getTableDimensions();
```


`getTableDimensions()` can be used in the `printPageBody()` method of `JCPrintListener()` to determine the size the table occupies on the page.

The `JCPrintListener` requires that three methods are defined:

```
public void printPageHeader(JCPrintEvent e);
public void printPageFooter(JCPrintEvent e);
public void printPageBody(JCPrintEvent e);
```

The `printPageBody` method is called after `JClass LiveTable` has finished setting up the print of the page body, but just before it is actually sent to the printer.

The following code produces the footer illustrated below:

```
public void printPageFooter(JCPrintEvent e) {
    Graphics gc = e.getGraphics();
    Rectangle r = gc.getClipRect();

    FontMetrics fm = gc.getFontMetrics();

    String page = "Page " + e.getPage();
    String note = "Use JCPrintListener to customize the footer!";
    // Pad the footer text to the right
    gc.drawString(page, 0, r.height/2);
    gc.drawString(note, r.width - fm.stringWidth(note), r.height/2);
}
```

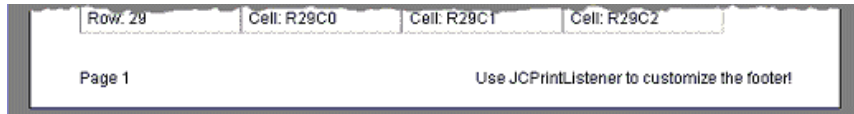


Figure 17 A Page Footer.

If you don't register a `JCPrintListener` for the table, the print engine will default to printing a centered footer containing the text **Page x**, where **x** is the page number. If you do register a `JCPrintListener`, however, then you are responsible for the placing the page number either in the header or footer of the page.

8.2 Print Preview

Using `JCPrintPreview`

`JClass LiveTable` provides a class that displays a preview of the print job in a separate frame. Using the print preview frame, end-users can flip through the pages of the print job, and send the current page or all of the pages to the printer.

To add the print preview functionality, use `JCPrintPreview`:

```
JCPrintPreview(String title, JCPrintTable table)
showPage(int page)
```

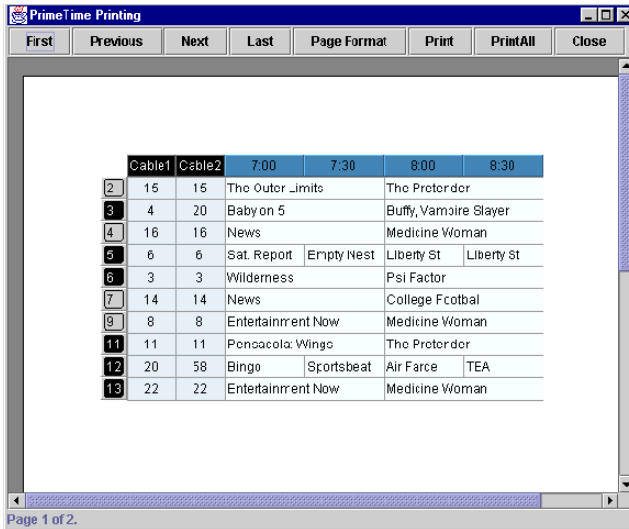
For example, the following provides a preview beginning at the first page of the job:

```
JCPrintPreview pf = new JCPrintPreview("Table Print  
Preview",printtable);  
pf.showPage(0);
```

Using JCPrintTable

Alternatively, to allow users to preview a print job, you can use JCPrintTable's showPrintPreview() method.

An example of print preview exists in the *PrimeTime.java* demo, located in the *demos/table/primetime* directory.



The screenshot shows a window titled "PrimeTime Printing" with a menu bar containing "First", "Previous", "Next", "Last", "Page Format", "Print", "PrintAll", and "Close". The main content area displays a table of TV program listings. The table has columns for "Cable1", "Cable2", "7:00", "7:30", "8:00", and "8:30". The rows are numbered 2 through 13. The table content is as follows:

	Cable1	Cable2	7:00	7:30	8:00	8:30
2	15	15	The Outer Limits		The Pretender	
3	4	20	Baby on 5			
4	16	16	News			
5	6	6	Sat. Report	Empty Nest	Liberty St	Liberty St
6	3	3	Wilderness		Psi Factor	
7	14	14	News		College Football	
9	8	8	Entertainment Now			
11	11	11	Pensacola Wings		The Pretender	
12	20	58	Bingo	Sportsbeat	Air Force	TEA
13	22	22	Entertainment Now		Medicine Woman	

Page 1 of 2.

Figure 18 The JClass LiveTable Print Preview Window.

JClass LiveTable Beans and IDEs

- An Introduction to JavaBeans* ■ *JClass LiveTable and JavaBeans*
- Setting Properties for the LiveTable Bean* ■ *Tutorial: Building a Table in an IDE*
- Data Binding with IDEs* ■ *Interacting with Data Bound Tables*
- Property Differences Between the JClass LiveTable Beans*

JClass LiveTable complies with the JavaBeans specification and includes several Beans that make it easy to create JClass LiveTable applications in an Integrated Development Environment (IDE). The following sections outline some principles of JavaBeans, and provide information about using JClass LiveTable in an IDE. All illustrations display the BeanBox, JavaSoft's test container for Beans included in the Beans Development Kit (BDK).

9.1 An Introduction to JavaBeans

Introduced in JDK 1.1, JavaBeans is a specification for reusable, pre-built Java software components. It is designed to be a fully platform-independent component model written for the Java programming language. The JavaBeans specification (available at <http://java.sun.com/beans/index.html>) enables developers to write components that can be combined in applications, reducing the total time needed to write entire applications.

The three main features of a Bean are:

- the set of properties it exposes
- the set of methods it allows other components to call
- and the set of events it fires

9.1.1 Properties

Under the JavaBeans model, *properties* are public attributes that affect a Bean's appearance or behavior. Properties can be read only, read/write, or write only. Properties that are readable have a `get` method which enables you to retrieve the property's value, and those properties which are writable have a `set` method which allows you to change their values.

For example, `JClass LiveTable` has a property called `FrameBorderType`. This property specifies the kind of border displayed around the table. To set the property value, use the `setFrameBorderType()` method. To obtain the property value, use the `getFrameBorderType()` method.

The main advantage of following the JavaBeans specification is that it makes it easy for a Java IDE to “discover” the set of properties belonging to an object. Developers can then manipulate the properties of the object easily through the graphical interface of the IDE when constructing a program.

There are two ways to set (and retrieve) `JClass LiveTable` Bean properties; use the method that applies best to your application:

- By using a Java IDE at design-time
- By calling property set and get methods in Java code

Each method changes the same table property. This manual, therefore, uses *properties* to discuss how features work, rather than using the method, Property Editor, or HTML parameter you might use to set that property.

9.1.2 Setting Properties in a Java IDE at Design-Time

`JClass LiveTable` can be used with a Java Integrated Development Environment (IDE), and its properties can be manipulated at design time. If you install your IDE after you have installed `JClass LiveTable`, you will have to manually add `LiveTable` to the IDE’s component manager. Refer to the `JClass` and `Your IDE` section in the [Installation Guide](#) for more information. Also, consult your IDE documentation for information on working with third-party components.

Please see Section 9.7, [Property Differences Between the `JClass LiveTable` Beans](#), for information on the differences between the `LiveTable` Bean, and the data binding Beans.

9.1.3 Setting Properties using Methods in the API

With the exception of read-only properties (which only have a `get` method), every property in `JClass LiveTable` has a `set` and `get` method associated with it. For example, to retrieve the value of the `FrameBorderType` property of a given cell and label area:

```
getFrameBorderType();
```

To set the `FrameBorderType` property in the same object:

```
setFrameBorderType(JCTbleEnum.BORDER_IN);
```

9.2 `JClass LiveTable` and JavaBeans

The JavaBeans included with `JClass LiveTable` make it easy to create applications and applets in an Integrated Development Environment. `JClass LiveTable` provides the following Beans:

- `LiveTable`: the core `JClass LiveTable` Bean.

- **JBuilder Bean (JBdbTable):** the same as the **LiveTable Bean**, but can bind **LiveTable** to a database using **Borland JBuilder's DataSet** (version 3.0 or greater).
- **LiveTable DataSource Bean (DSdbTable):** the same as the **LiveTable Bean**, but can bind **LiveTable** to a database using **Quest's JClass DataSource**.

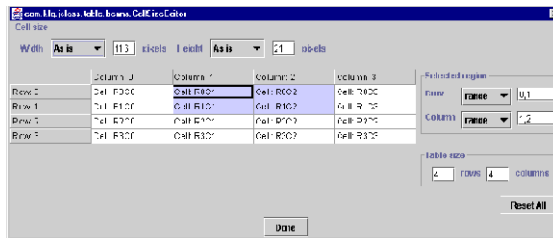
9.3 Setting Properties for the LiveTable Bean

At design-time, most **LiveTable** properties are set using simple menu choices or text entry boxes on the property sheet. Some properties that are set for individual cells or labels, or ranges of cells or labels, are set using a property editor. The **LiveTable** property editors provide a visual interface for setting the properties using a model of the table you are creating, and a number of ways for selecting the cell(s) or ranges that you want to set the property for.

To make it easier to use, the **LiveTable Bean** combines some properties into special property groups that are set using a single editor. For example, the **Style** property combines the **Foreground**, **Background**, and **Font** properties and presents them in a single editor.

9.3.1 JClass LiveTable Property Editors

The following is a typical property editor with elements common to other **LiveTable** property editors:



Each property editor has the same interface for selecting the cells to which a specific property is applied. On the left is a view of the table (the data reflects the properties you are setting). On the right are two groups of controls: **Selected region** provides an alternate control of part of the table selected; **Table Size** controls the size of the table view in the editor. Both of these interact with the table on the left.

It is important to note that the table view is provided only as a visual guide for setting properties. Its size and contents may not necessarily reflect those of the actual table you are building.

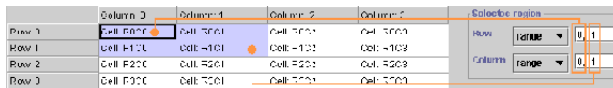
Selecting a Cell or Cell Range

The purpose of the property editors is to apply a given property to a single cell or label, or to a range of cells or labels. You can select cells interactively using the mouse or by using the **Selected region** controls.

To select cells using the mouse:

- Click an individual cell with the mouse to select that cell.
- Click and drag the mouse to select a range of cells.
- Click on a row or column label to select that row or column.
- Click on a cell, hold down the **Shift** key and click another cell to select a range of cells between the two.

Note that when you make selections with the mouse, the ranges you select are displayed in the **Selected region** controls, as shown in the following diagram:



The screenshot shows a table with 4 columns and 4 rows. The cells are labeled as follows:


	Column 0	Column 1	Column 2	Column 3
Row 0	Cell: R0C0	Cell: R0C1	Cell: R0C2	Cell: R0C3
Row 1	Cell: R1C0	Cell: R1C1	Cell: R1C2	Cell: R1C3
Row 2	Cell: R2C0	Cell: R2C1	Cell: R2C2	Cell: R2C3
Row 3	Cell: R3C0	Cell: R3C1	Cell: R3C2	Cell: R3C3

Selected region controls:

Row	range	0, 1
Column	range	0, 1

To select cells using the **Selected region** controls:

To select a single cell, choose **Range** from both the row and column pull-down menus, then type the row index and column index for the cell. For example, the cell that intersects the fourth row and the third column would be selected by typing 3 for the **Row** range and 2 for the **Column** range (remember, the rows and columns start at 0).



Selected region

Row	range	3
Column	range	2

To select a range of cells, you must specify the row and column index for the top-left and bottom-right cells in the range (typically specifying a range is easier to do with the mouse). Choose **Range** from both the row and column pull-down menus, then type the numbers of the top and bottom rows of the range separated by a comma; then type the left and right columns of the range separated by a comma. This has specified a bounding box for the range.



Selected region

Row	range	0,2
Column	range	1,3

Selecting Labels

To select labels using the mouse:

- Row labels: click the row label or select a range of row labels and choose Label from the **Column** pull-down menu in the **Selected region** controls.

- Column labels: click the column label or select a range of column labels and choose Label from the **Row** pull-down menu in the **Selected region** controls.

It may seem odd to be choosing in the **Column** box for *row* labels and in the **Row** box for *column* labels, but it is easier to understand if you consider that you are really specifying the row of column labels or the column of row labels.

To select labels using the **Selected region** controls:

To select all row labels, choose **all** in the **Row** pull down menu, and choose **label** in the **Column** pull down menu.



To select a row label, choose **label** from the **Column** pull-down menu. Then choose **range** in the **Row** pull down menu, and type the desired row number in the text field.



To select all column labels, choose **all** in the **Column** pull down menu, and choose **label** in the **Row** pull down menu.



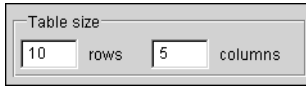
To select a column label, choose **label** from the **Row** pull down menu. Then choose **range** in the **Column** pull down menu, and type the desired row number in the text field.



Changing the Property Editor Table Size

The **Table size** controls set the working size of the table view *in the editor*. By default, the property editors display a 10 row, 5 column table, which is sufficient for most selection. If you need to edit properties for a specific row or column beyond this limit, use the **Table size** controls to enlarge the working area in the property editor. To change table size,

enter a new value in the **Row** or **Column** text field and press **Enter** (or traverse out of the field); the table view will update to the new dimensions.



Note: To change the *actual* size of the table you're building, use the **Table size** controls on the DataEditor. The DataEditor is the *only* editor that uses the **Table size** controls this way.

You can undo any of your changes and reset the properties to the values they had when you opened the editors by clicking the **Reset All** button.

9.3.2 LiveTable Properties

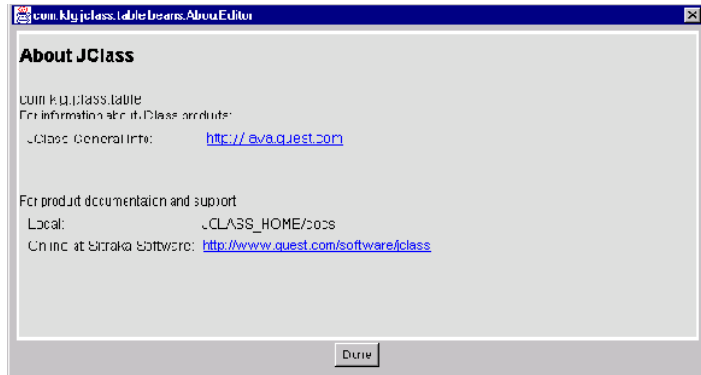
The LiveTable Bean exists because the current generation of Java IDEs do not support properties in contained objects. While current Java IDEs allow properties in contained objects to be modified, they cannot yet properly generate code for the property change. LiveTable works around this problem by exposing many JClass LiveTable properties in one object. While not all properties are provided, the most common properties are available.

The following sections list the properties exposed in the LiveTable Bean. Many of the properties can be set for individual cells/labels or ranges; these properties are set using visual property editors (see Section 9.3.1, [JClass LiveTable Property Editors](#), for a description of a typical editor and how to select cells). Note that the illustrations are from Sun's BeanBox in the Beans Development Kit (BDK). The properties and editors are

listed in alphabetical order; the BeanBox unfortunately does not list properties in alphabetical order.

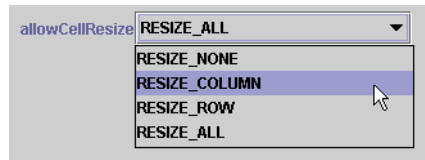
about

about displays the component version and points to other sources of JClass information



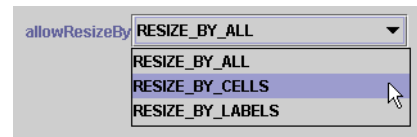
allowCellResize

Setting `allowCellResize` determines which part of a cell users can click and drag to resize cells. The default setting, `RESIZE_ALL`, allows resizing by clicking and dragging both row and column cell borders.



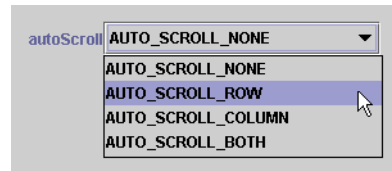
allowResizeBy

Setting `allowResizeBy` determines which part of the table can be used to resize rows and columns: cells, labels or both. This property works in conjunction with `allowCellResize`.



autoScroll

The `autoScroll` property determines if the table automatically scrolls when the user drags the mouse or traverses past the borders of the table.

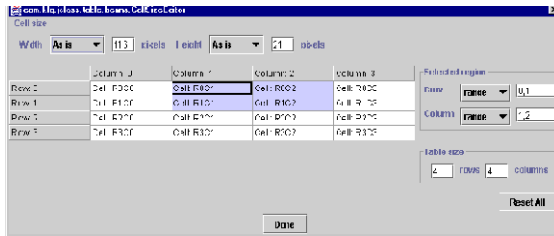


cellBorderWidth

Entering a number into the `cellBorderWidth` field specifies the thickness of the border around each cell and label.

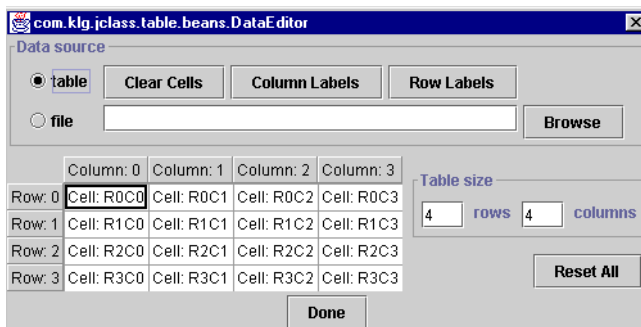
cellSize

The `cellSize` editor lets you set row height and column width values as either fixed pixel values or variable values. Changes made to cell dimensions are only applied to selected cells, and cells found in the same rows and columns as the selection.



data

The `data` property, exclusive to the LiveTable Bean and not available in the data binding Beans, enables you to add and customize table data and row/column labels. There are two ways to get data in a table – by entering data directly using the `DataEditor`, or by specifying a data file (which can contain label information).



The data file format is a space-delimited text file that can contain Strings, doubles, or integers. This example file contains 4 rows and 4 columns with no labels:

```
TABLE 4 4 NOLABEL
'0,0' '0,1' '0,2' '0,3'
'1,0' '1,1' '1,2' '1,3'
'2,0' '2,1' '2,2' '2,3'
'3,0' '3,1' '3,2' '3,3'
```

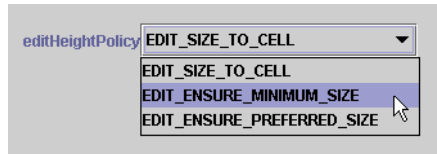
To include reserved characters (like spaces), enclose the data item in single quotation marks, for example 'The Cuppa'. This example shows how to specify labels:

```
TABLE 4 3
'Col 0' 'Col 1' 'Col 2'
'Row 0' '0,0' '0,1' '0,2'
'Row 1' '1,0' '1,1' '1,2'
'Row 2' '2,0' '2,1' '2,2'
'Row 3' '3,0' '3,1' '3,2'
```

editHeightPolicy

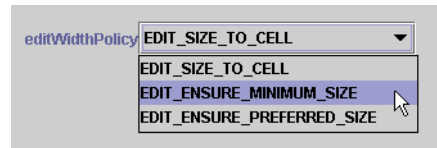
The table can control the height of a cell editing component using the `editHeightPolicy` property. This property can take one of three values:

- `EDIT_SIZE_TO_CELL` resizes the component to fit the table's cell size.
- `EDIT_ENSURE_MINIMUM_SIZE` resizes the component to its minimum size.
- `EDIT_ENSURE_PREFERRED_SIZE` resizes the cell to the editing component's preferred size.



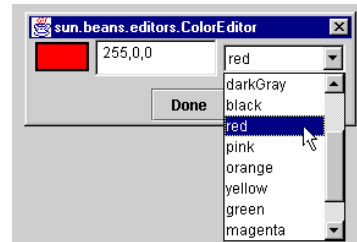
editWidthPolicy

The table can control the width of a cell editing component using the `editWidthPolicy` property. The valid values this property takes are the same as those associated with `editHeightPolicy`.



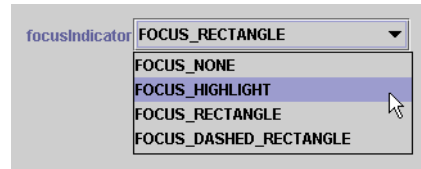
focusColor

Choosing a color for the `focusColor` property determines the color of the focus rectangle. The focus rectangle is the line drawn around the inside of the cell that currently has focus.



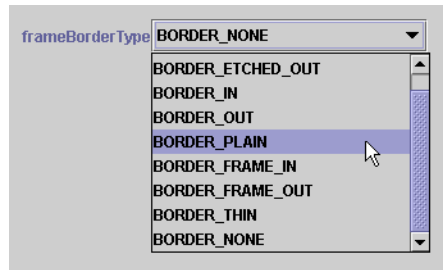
focusIndicator

The `focusIndicator` property determines how cell focus is shown to the user. This indicator appears inside the cell that currently has focus. The default is `FOCUS_RECTANGLE`.



frameBorderType

Setting the `frameBorderType` property determines what type of border is used for the frame enclosing the cell and label areas. Choose a border type from the pull-down menu. Note that the `FrameBorderWidth` property must be set to greater than 5 in order for the etched border types to be visible.

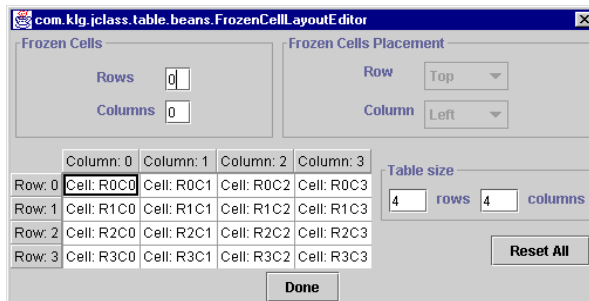


frameBorderWidth

Enter a value in the `frameBorderWidth` property field to determine the thickness of the frame around the cell and label areas of the table in pixels.

frozenCellLayout

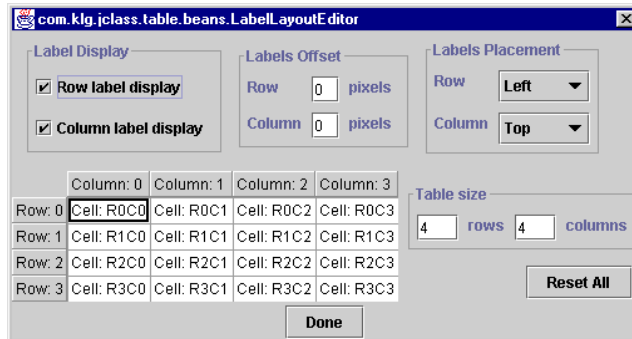
Working with the `frozenCellLayout` property editor sets whether there are any frozen rows or columns, and specifies their position in the table. Frozen rows can be placed at the top or bottom of the table, and frozen columns can be placed on the left or right side of the table.



labelLayout

Working with the `labelLayout` property editor offers full control over label attributes. You determine if row or column labels exist, then specify offsets and label placement.

Label offset is the distance in pixels between the edge of the table and the labels. You can place row labels at the top or bottom of the table, and column labels at the right or left side of the table.



leftColumn

Enter the column number in this field to specify which is the left-most column when displaying the table.

marginHeight

Enter a value in the `marginHeight` field to determine the height of the top and bottom margins of each cell.

marginWidth

Enter a value in the `marginWidth` field to determine the width of the left and right margins of each cell.

minCellVisibility

By default, when a user traverses to a cell that is not currently visible, `LiveTable` scrolls the table to display the entire cell. Enter a percentage value in the `minCellVisibility` field to set the percentage of the cell that is scrolled into view when it is the target of a traversal.

When `MinCellVisibility` is set to 100, the entire cell is made visible. When `MinCellVisibility` is set to 10, only 10% of the cell is made visible. If `MinCellVisibility` is set to 0, the table will not scroll to reveal the cell.

sBLayout

Working with the `sBLayout` property editor offers full control over visual and behavioral scrollbar attributes:

The Scrollbar Display settings determine if horizontal or vertical scrollbars exist. If so, you can also determine if the scrollbars are displayed at all times, or only when the table's contents exceed the table's size.

The Scrollbar Position settings determine whether scrollbars are positioned **by cells** or **at sides**. The former option places the scrollbar beside the cell/label viewport, while the latter places the scrollbar beside the edge of the table area. Note that there is no visual difference between the two options unless the cell/label area is smaller than the table area.

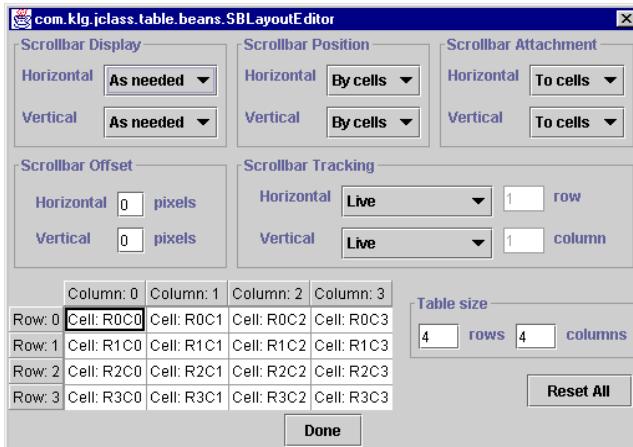
The Scrollbar Attachment settings determine how far along the side of the table the scrollbars extend. **To cells** (default) places the scrollbar along the edge of all visible, non-frozen cells, while **To table** places the scrollbar along the edge of the entire table.

Scrollbar Offset sets the amount of space, in pixels, between the scrollbar and the table.

Scrollbar Tracking determines the type of feedback a user receives when they click and drag a scrollbar. **Live** tracking refreshes the table as the user scrolls. The **Column number/Row number** and **Row/Column** options do not redraw the table until the user stops scrolling by releasing the mouse button.

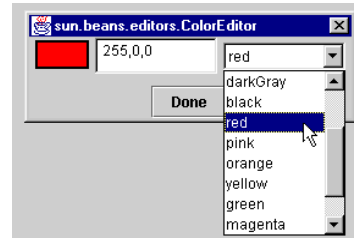
However, to offer the user feedback, the **Column number/Row number** option displays a box with the current cell number, while the **Row/Column** option displays the

actual contents of the current cell. The row or column from which the displayed contents are taken is determined by the number entered in the **Row** or **Column** fields.



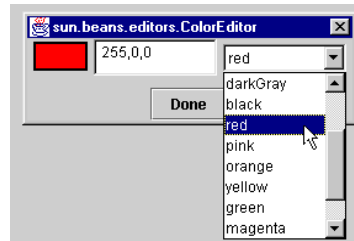
selectedBackground

Selecting a color for the `selectedBackground` property determines the background (highlight) color for cells that have been selected. The default is the foreground color for the cells.



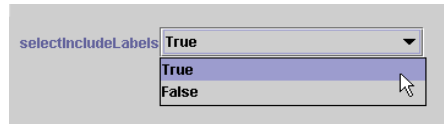
selectedForeground

Selecting a color for the `selectedForeground` property determines the foreground (highlight) color for cells that have been selected. The default is the background color for the cells.



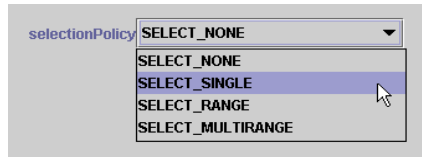
selectIncludeLabels

By default, when a user clicks a label, the entire row or column, including the label, is highlighted. To change it so that the label is not highlighted with the rest of the cells, set this property to false.



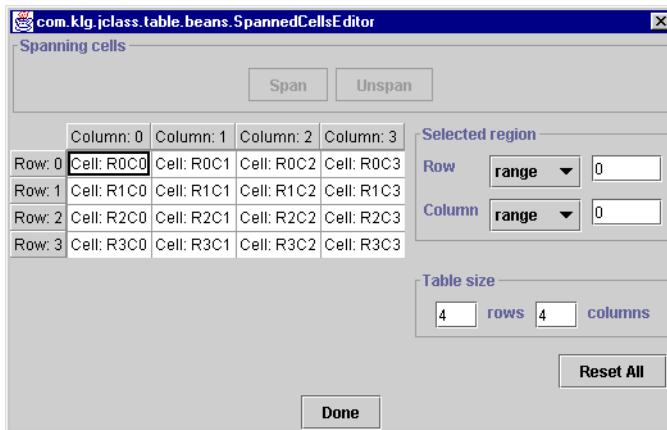
selectionPolicy

The `SelectionPolicy` property controls the amount of selection allowed on the table, both by end-users and by the application.



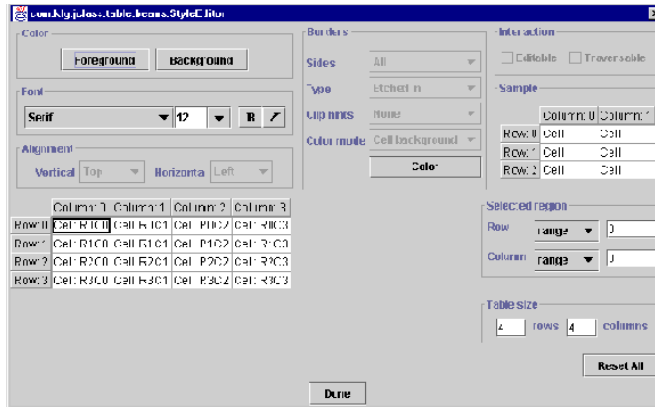
spannedCells

The `spannedCells` property editor lets you visually implement cell spanning with your table. Use the displayed table to select which cells are to be combined. The cell found at the top left corner of the spanned range becomes the new cell.

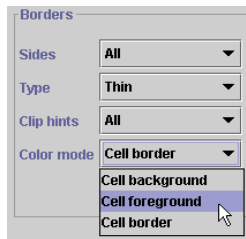


styles

The styles property editor gives you control over cell style options, including: cell colors, font attributes, editable and traversable states, and alignment in cells.

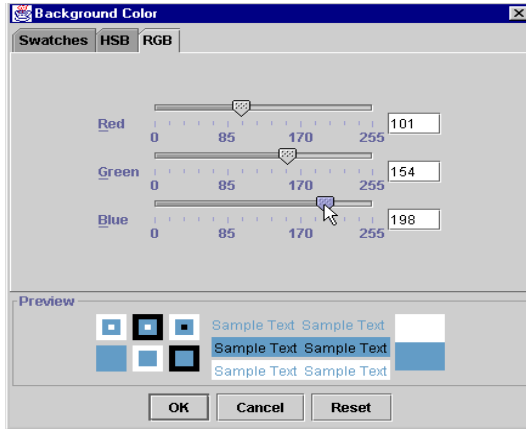


You can also use the styles property editor to work with and set all border properties for cells in the selected area. By using the pull-down menus, you can set which cell sides have borders, what type of border is used, whether clip hints are displayed (should the cell's contents exceed its size) and what the selected cells border color is.



When choosing border or cell background/foreground colors, the color selection options offer precise control. You can select a color by using the Hue/Saturation/Brightness

(HSB) panel, the Red/Green/Blue (RGB) Panel, or the color swatch. All three methods offer color previews for both text and objects.



topRow

Enter the row number in this field to specify which is the topmost row when displaying the table.

9.4 Tutorial: Building a Table in an IDE

The following exercise will guide you through the steps to produce a JClass LiveTable program in an IDE. The exercise is the same as the one in ‘Hello Table’ – [JClass LiveTable Tutorial](#), in Chapter 1, which explains how to build a table using the API. The example uses JavaSoft’s BeanBox IDE in the Java BDK. This tutorial assumes that you have some experience working with a Java IDE. If you are unsure how to get the LiveTable Bean into your IDE, please consult the IDE documentation.

This program displays information about orders for ‘The Musical Fruit’¹, a fictional wholesale coffee distributor, based on the following data:

Customer Name	Order Date	Item	Quantity (lbs.)	Price/lb.
The Cuppa	11/11/97	French Mocha	60	\$7.01
The Underground Cafe	11/14/97	Brazilian Medium	112	\$6.80
RocketFuel and Cake Cafe	10/30/97	Espresso Dark	300	\$8.02

1. We apologize for the addition of yet another coffee reference in an already crowded pantheon.

Customer Name	Order Date	Item	Quantity (lbs.)	Price/lb.
WideEyes Coffee House	11/12/97	Colombian/Irish Cream Flavored	120	\$5.30
Jitters Caffeine Cavern	10/01/97	Ethiopian Medium	80	\$7.50
Twitchies on the Mall	12/06/97	French Roast Kona	160	\$14.50
Quest Software	12/12/97	Colombian	22,000	\$5.28

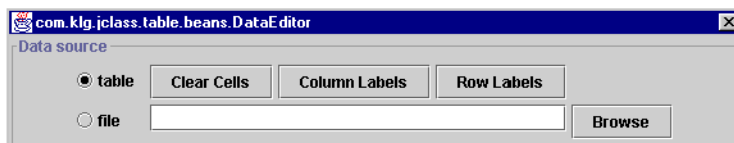
9.4.1 The Basic Table

The first step is to create a default table. In the BeanBox, click the `LiveTable` component displayed in the ToolBox, then to insert the table component, click in the BeanBox window. The BeanBox displays a default-sized (10 rows by 5 columns) table with visible row and column labels:

	Column 1	Column 2	Column 3	Column 4
Row 1	Cell: R1C1	Cell: R1C2	Cell: R1C3	Cell: R1C4
Row 2	Cell: R2C1	Cell: R2C2	Cell: R2C3	Cell: R2C4
Row 3	Cell: R3C1	Cell: R3C2	Cell: R3C3	Cell: R3C4
Row 4	Cell: R4C1	Cell: R4C2	Cell: R4C3	Cell: R4C4
Row 5	Cell: R5C1	Cell: R5C2	Cell: R5C3	Cell: R5C4
Row 6	Cell: R6C1	Cell: R6C2	Cell: R6C3	Cell: R6C4
Row 7	Cell: R7C1	Cell: R7C2	Cell: R7C3	Cell: R7C4
Row 8	Cell: R8C1	Cell: R8C2	Cell: R8C3	Cell: R8C4
Row 9	Cell: R9C1	Cell: R9C2	Cell: R9C3	Cell: R9C4

Supplying the Data

The data for the table is contained in the data source. You can provide the data by entering it into the table using the `DataEditor`, or by specifying that a file is the data source. Start the `DataEditor` by clicking the data property on the property sheet. Notice that the editor defaults to using the table as the data source.

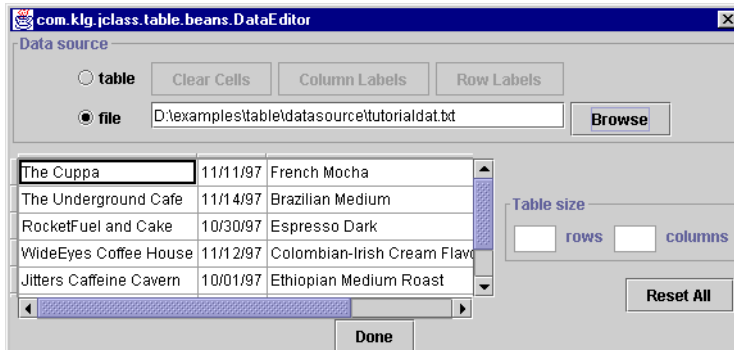


The data we want to display is stored in a file. To use this file as the data source:

1. Click **Browse**.
2. Navigate to the `examples/table/datasource` directory of your JClass LiveTable distribution.

- Choose the *tutorialdat.txt* file.

When you choose the file, the table display in the editor populates with the data from the data source. Also, the **Table size** fields should be disabled, as shown in the following illustration:



- To close the editor, click **Done**. The table displayed in the BeanBox now shows the values from the data source:

The Cuppa	11/11/97	French Mocha	60		\$7.21
The Underground Cafe	11/14/97	Brazilian Medium	112		\$8.30
RocketFuel and Cake	10/30/97	Espresso Dark	300		\$6.32
WideEyes Coffee House	11/12/97	Colombian-Irish Cream	120		\$6.30
Jitters Caffeine Cavern	10/01/97	Ethiopian Medium Roast	80		\$7.50
Twitches on the Mail	12/06/97	French Press - Crema	160		\$14.50
Quest Software Inc.	12/2/97	Colombian	21,000		\$6.20

9.4.2 Improving the Table's Appearance

Using some of the properties for modifying a table's appearance, you can easily move from the basic table to a more interesting table that's easier to understand, and easier to use. The following sections explain how to set these properties using an IDE.

Adding Column Labels

The table currently displayed in the BeanBox is not very useful to an end-user. Not only is it not interesting to look at, but you cannot tell what kinds of information the various cells contain because there are no column labels. In the original data outline for the table, we indicated that we wanted the following column labels:

- Customer Name
- Order Date
- Item
- Quantity (lbs)
- Price/lb

Labels are cells that can never be edited and can contain any Object (Strings, images, Integers, and so on). Notice that since our data source contained no data for the labels, the `LiveTable Bean` does not display any labels in the `BeanBox`.

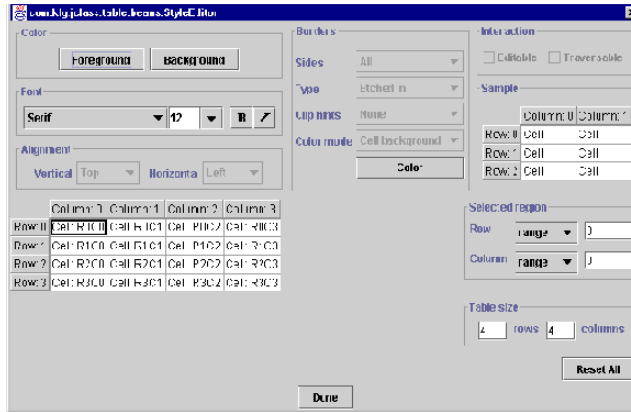
If you had entered the table data directly into the Bean, you could add the labels using the `DataEditor`. But since the file is the data source, adding the labels must be done by editing the file. For convenience we have included the labels in another data file. Load this data file using the `DataEditor` as before. The new file is called *tutorialdat-labels.txt*, located in the *examples/table/datasource* directory of your JClass `LiveTable` distribution.

By default, the table displays row and column labels that have values. This is controlled by the `labelLayout` property. Now that you have column labels, the table in the IDE should update to look something like the following illustration:

Customer Name	Order Date	Item	Quantity (lbs.)	Price/lb.
The Cuppa	11/11/97	French Mocha	60	\$7.01
The Underground C	11/14/97	Brazilian Medium	112	\$6.00
RocketFuel and Cak	10/30/97	Espresso Dark	300	\$8.02
WideEyes Coffee H	11/12/97	Colombian-Irish Cr	120	\$5.00
Jitters Caffeine Cav	10/01/97	Ethiopian Medium R	80	\$7.00
Twitchys on the Mall	12/06/97	French Roast Kona	160	\$14.50
Ques: Software Inc.	12/12/97	Colombian	22,000	\$5.28

Notice that if you click a label in your table, you do not get the focus rectangle the way you do if you click a cell: labels cannot be edited and cannot be the target of a traversal. In certain circumstances, clicking a label performs an action (see Section 9.4.3, [Adding Interactivity](#)), but in this case the labels do not perform any interactive function.

The labels have a default border and color set to make them stand out from the table. In this exercise, we will take it one step further by changing the colors and fonts of the labels using the StyleEditor, accessed by clicking the styles property on the property sheet:



To begin, select the column labels. In the **Selected region** box:

1. Choose **label** from the **Row** pull-down menu.
2. Choose **all cells** from the **Column** pull-down menu.

This applies any settings you choose to the column labels. Next, choose the font and size of the label text. In the **Font** box:

3. Choose **Times New Roman** from the font pull-down menu.
4. Choose **14** from the point size pull-down menu.

Note: The type of font displayed on a user's system depends entirely on the fonts that are local to that user's computer. If a font name specified in a Java program is not found on a user's system, the closest possible match is used (as determined by the Java AWT).

Finally, change the color of the label text. In the **Color** box:

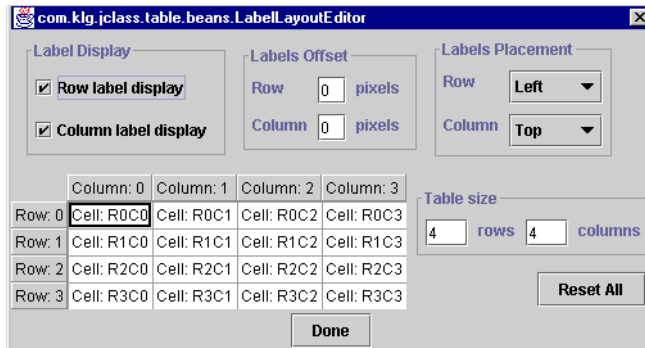
5. Choose **Foreground**, then select a white color from the **Swatches** tab.
6. Choose **Background**, then select a blue color from the **Swatches** tab.
7. You should be able to see these changes in the sample table in the StyleEditor. Click **Done** to commit the changes and return to the BeanBox.

Your changes are now visible in the BeanBox. You now have a basic table with labels colored and text formatted to differentiate them from the rest of the table cells.

Customer Name	Order Date	Item	Quantity (lbs.)	Price/lb.
The Cuppa	11/11/97	French Mocha	80	\$7.01
The Underground Cafe	11/14/97	Brazilian Medium	112	\$6.90
RocketFuel and Cake	10/30/97	Espresso Dark	300	\$6.02
WideEyes Coffee House	11/12/97	Colombian-Iris-1 Cream	120	\$5.30
Jitters Coffee Cavern	10/01/97	Ethiopian Medium Roast	80	\$7.50
Twitchys on the Mall	12/06/97	French Roast Kona	130	\$14.50
Quiet Software Inc	12/12/97	Colombian	22,000	\$5.28

Label Layout

You can change the position of the labels relative to the table, and control their distance from the table to help make the labels even more distinctive. By default, labels are displayed right against the table border. You can make it stand off by using the LabelLayout editor, accessed by clicking the LabelLayout property on the property sheet:



For this exercise, you are going to add some space between the column labels and the top of the table and take out the row labels. In the LabelLayout editor:

1. In the **Label Display** box, both row and column labels are selected by default. Clear the **Row label display** check box.
2. In the **Labels Offset** box, change the value in the **Column** field from 0 to 2 pixels.

The changes are immediately reflected in the editor and the BeanBox.

3. Click **Done** to commit the changes and return to the BeanBox.

Having changed the alignment and font, your table should now look something like the following illustration:

Customer Name	Order Date	Item	Quantity (lbs.)	Price/lb
The Cuppa	1/11/97	French Roast	60	\$7.01
The Underground Cafe	1/11/97	Brazilian Medium	112	\$6.00
RocioFuel and Cafe	1/10/97	Espresso Dark	300	\$6.02
WideTree Coffee Int...	1/12/97	Caribbean-Irish Crea...	120	\$6.00
Jitters Coffee Tavern	1/09/97	Ethiopian Medium Ro...	60	\$7.50
Twilights on the Wall	12/06/97	French RoastKona	150	\$74.50
Guest Software Inc.	12/12/97	Colombian	22,000	\$4.28

Changing the Cell Borders and Thickness

JClass LiveTable has properties that you can use to change the way the cell borders and cell spacing appears.

There are a number of choices for cell borders, outlined earlier in the description of the `style` property. For the example program, you're going to thicken the cell borders and change the border style. This involves working with the `cellBorderWidth` and `styles` properties on the property sheet.

1. First, to change the cell border width value, simply edit the value in the text box for the `cellBorderWidth` property. Set the value to 2 instead of the default (1).

To change the cell border type for the table cells and labels, you need to call the `StyleEditor` (again, click `styles` in the property sheet).

2. To begin, select all non-label cells. In the **Selected region** box, choose **all cells** from the **Row** pull-down menu and choose **all cells** from the **Column** pull-down menu.

Now that you have selected all cells, change their border:

3. Click the **Types** drop-down list in the **Borders** box and select **border in**.

To change the border type for the column labels, you now need to select all column labels. In the **Selected Region** box:

4. Choose **label** from the **Row** pull-down menu and choose **all cells** from the **Column** pull-down menu.

This applies any settings you choose to the column labels. Now, change their border:

5. Click the **Types** drop-down list in the **Borders** box and select **border out**.

The table should now resemble the following in the BeanBox:

Customer Name	Order Date	Item	Quantity (lbs.)	Price/lb.
The Uddops	11/1/1991	French Mocha	50	\$1.11
The Underground Cafe	11/2/1997	Orsatian Medium	112	\$5.10
Rockwell and Cake	10/0/007	Espresso Dark	300	\$3.32
Wildfires Coffee Hut...	11/2/297	Columbian-Hispanic...	123	\$5.30
Jill's Coffee Corner	10/01/97	Elfinador Medium Ro...	80	\$7.50
Twitchy on the Mall	12/0/007	French Roast/Kona	160	\$17.60
Quest Software Inc.	12/2/297	Columbian	22000	\$5.28

9.4.3 Adding Interactivity

In a real-world situation, our example table would likely be used to track orders and accounts with a large number of customers. Your users will likely want to update the data, sort the information displayed in the table, and select sections of the table to perform operations on them.

We'll add some basic user-interactivity to our example table to give you a sense of some of the things JClass LiveTable can do. You can explore user-interactivity further in [Programming User Interactivity](#), in Chapter 6.

Controlling Cell Editability

Using the LiveTable Bean, the data source is editable by default. You can change the editability of cells using the EditState property. Note that in the data source, Quest Software has ordered 22,000 pounds of coffee. This is obviously a typographical error, but we're going to make sure Quest Software gets all 22,000 pounds of coffee by not allowing that cell to be edited.

Invoke the style editor by clicking `styles` on the property sheet.

In our original data, the cell containing the value 22,000 was located at row 6, column 3. (Recall that arrays in Java are zero-based – thus, the row and column indexes begin at zero.) You can either select this cell with the mouse, or type these values in **Row** and **Column** fields in the **Selected region** box.

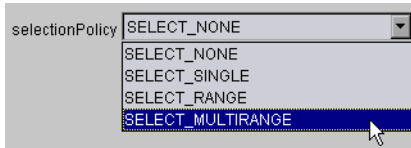
Note that each cell in the editor's table reflects its current traversable and editable state. A cell that is editable must also be traversable, but a cell that is traversable does not necessarily have to be editable. For this particular cell, leave traversability on, and simply unset its editability:

1. In the **Interaction** box, clear the **Editable** checkbox. This makes the cell traversable but not editable, as is displayed in the editor's table.
2. Click **Done**.

Now we can be sure that nobody will change Quest Software's coffee order!

Enabling Cell Selection

JClass LiveTable provides methods that set how users can select cells, ranges of cells, and entire rows and columns. Selection is enabled by setting the `SelectionPolicy` property. By default, cell selection reverses the foreground and background colors of the cells to highlight the selection. You can enable selection by choosing a value from the `SelectionPolicy` pull-down menu in the LiveTable property sheet.



1. Choose `SELECT_MULTIRANGE`. This allows users to select one or more cells in rows or columns by clicking and dragging the mouse, or using keyboard combinations.

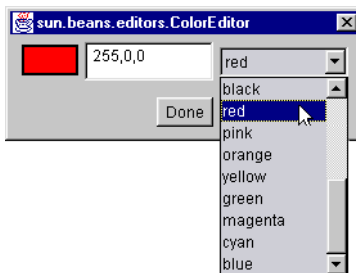
By default, setting the `SelectionPolicy` property enables selection of entire rows or columns by clicking on the row or column label. When the user clicks on the column label, the column display, including the label, is reversed to highlight the selection. You can configure the table not to highlight the label by setting the `SelectIncludeLabels` property to `false`.

Resizing using Labels Only

By default, users can resize rows, columns, and labels by clicking on their borders and dragging to resize. You can change this functionality to have the resize capability available only from the label; to resize a column, the user resizes its label rather than its cells. LiveTable provides the `allowResizeBy` property to enable this feature. In the property sheet, change the `allowResizeBy` property to `RESIZE_BY_LABELS`.

Changing the Focus Rectangle Color

Finally, some of your users have complained that it's hard for them to see what cell currently has focus because the focus rectangle is plain black. You can change the color of the focus rectangle easily by setting the `focusColor` property:



When you click on the `FocusRectColor` property, the default color chooser appears.

Choose red from the color chooser. Now your users should be able to see the focus rectangle clearly.

9.4.4 The Final Program

Your simple table program has evolved into an interactive, easy-to-understand utility. Although it's far from being a real order-tracking system, using a few more `JClass LiveTable` features, it soon could be. The following illustration shows all of the visual changes that you've accomplished. From here you can try out other properties and see how they affect the table's appearance and behavior.

Customer Name	Order Date	Item	Quantity (lbs.)	Price/lb.
The Cuppa	11/11/97	French Mocha	00	\$7.01
The Underground Cafe	11/14/97	Brazilian Medium	112	\$6.81
RockaFuel and Cafe	12/01/97	Espresso Dark	203	\$9.01
WideEyes Coffee Hou...	11/12/97	Columbian-Hish Crea...	123	\$5.91
Jitters Caffeine Caram	11/11/97	Ethiopian Medium Inc...	80	\$7.51
TwitCys on the Mtl	12/06/97	French Roast Kona	163	\$4.40
Quest Software Inc	12/12/97	Columbian	22000	\$6.21

9.5 Data Binding with IDEs

If you are using an IDE to develop Java applets and applications, the `LiveTable` data binding Beans allow you to bind a table with a `JDBC`-compliant data source, an `ODBC` data source (by using the `JDBC-ODBC` bridge), or an IDE-specific data source. The data binding Beans are loosely based on a `Model-View-Controller (MVC)` data mechanism. The direct link between the table component and the IDE's data source offers an easy and efficient way of representing and modifying data in your tables.

As outlined earlier in Section 9.2, `JClass LiveTable` and `JavaBeans`, `LiveTable` includes data binding Beans: `JDbTable` is used with `JBuilder's DataSet` and `DSdbTable` is used with any `JDBC` data source (and `JClass DataSource`) in any IDE.

The principles of data binding and connecting to a database in any environment are similar. The following sections assume that you are:

- familiar enough with your IDE or other development environment to create and work with basic application projects
- familiar with setting up database connectivity in your development environment's projects

Note: The examples used in the following sections use a sample `JClassDemo` database (*demo.mdb*) that can be found in the `JCLASS_HOME/demos/common/databases` directory. As such, these examples are primarily meant to illustrate data binding with IDEs, as you will not be able to duplicate them if you do not have the sample database.

9.5.1 Data Binding LiveTable with a JBuilder Data Source

To data bind to a JBuilder's DataSet using `JBdbTable`, you require:

- Borland JBuilder 3.0 or greater
- JDK 1.3.1 or greater
- JClass LiveTable's `JBdbTable` Bean
- a data source properly set up in Windows' ODBC Data Source Administrator

Creating a Java application that contains a data bound table in JBuilder requires an understanding of database connectivity in a JBuilder project. Binding your table with a database in this IDE involves:

- creating the project and laying out the UI
- adding and configuring the Database component
- adding and configuring the `QueryDataSet` component
- adding and configuring the LiveTable data binding Bean (`JBdbTable`)

There are different methods and components with which a JBuilder project with database connectivity can be created. The following provides a general overview of data binding, as it is assumed that you are familiar with working with your IDE. For specific information, please refer to your JBuilder documentation, where comprehensive tutorial and reference information can be found about setting up an application project, adding the Database and `QueryDataSet` components to manage the JDBC connection, and communicating with the database.

Let's begin with a basic project in JBuilder, where the UI components are set up, readying the project for the addition of the database and data binding components.

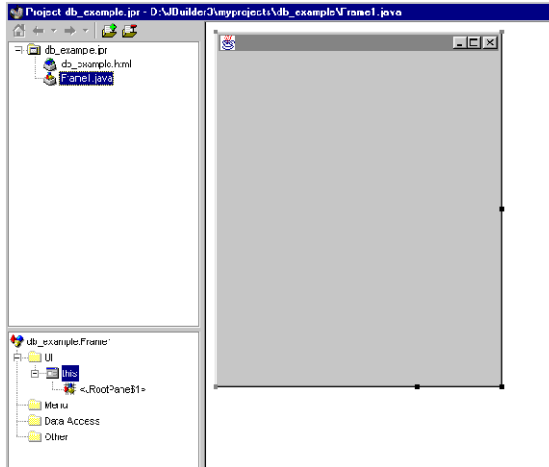


Figure 19 Example project work space with defined UI components, ready for database components.

Adding the Database Component to the Project

In the Data Express tab of the Component Palette, click the `borland.sql.dataset.Database` object, designating it as the component to be added. Next, click an empty area of the Component Tree. The database object is added to your project.



Figure 20 Selecting the Database component on the Component Palette.

Setting the Connection Property for the Database Component

Now that the database object has been inserted into your project, you need to define the JDBC Connection information for this object. This is done by setting the connection property in the Inspector, when the database object is selected.

It is here that you select the data source that you want bound to your table component. All available data sources and DataGateway sources recognized by JBuilder are listed and available to be set as the main data source. These data sources are defined with Windows' ODBC Administrator. In this example, the *demo.mdb* database (JClassDemo) is selected.

Adding the Database component and setting the connection properties adds the following lines of code to your project:

```
import borland.sql.dataset.*;
...
Database database1 = new Database();
...
database1.setConnection(new com.borland.dx.sql.dataset.
    ConnectionDescriptor("jdbc:odbc:JClassDemo", "", "", false,
        "sun.jdbc.odbc.JdbcOdbcDriver"));
```

This code introduces the database component (in this example, it is named *database1*) and points it to the data source that you define (in this example, the sample *demo.mdb* database, *JClassDemo* is used).

Adding the QueryDataSet Component to the Project

Now that you have set up the link between your project and the desired database, you need to interact with that database. In the Data Express tab of the Component Palette, click the `borland.sql.dataset.QueryDataSet` object, designating it as the component to be added. Next, click an empty area in the Component Tree. The database query component is added to your project.



Figure 21 Selecting the *QueryDataSet* component on the Component Palette.

Setting the Query Property for the QueryDataSet Component

Now that the *QueryDataSet* component has been added, you need to define which parts of which database will be used. To do this, you need to query the database with an SQL statement. This is done by setting the query property in the Inspector, when the *QueryDataSet* object is selected.

In the query property area, select the database you just added, and browse the tables if you have to get the proper information for your table. Enter the SQL statement that represents your needs. Test it to be sure that such a query will be successful.

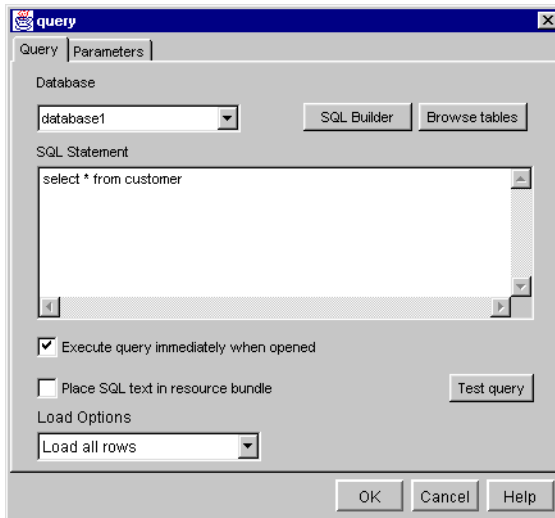


Figure 22 Entering the query statement to the selected database.

Adding the `QueryDataSet` component and setting the query property adds the following lines of code to your project:

```
QueryDataSet queryDataSet1 = new QueryDataSet();  
...  
queryDataSet1.setQuery(new com.borland.dx.sql.dataset.QueryDescriptor  
(database1, "select * from customer", null,true,Load.ALL));
```

This code defines a new `QueryDataSet` component (in this example, named *queryDataSet1*), which lets you read and write to and from the database by way of SQL statements. It also defines which part of the database is extracted, and bound to your table component. In this example, the `Customer` table is selected with the query statement.

It is here that you set the property to the `QueryDataSet` component name that is part of your project. In this example, the name of the component is `queryDataSet1`. This action adds these lines of code to your .java file:

```
import com.klg.jclass.table.db.jbuilder.*;
...
JDbTable jDbTable1 = new JDbTable();
...
jDbTable1.setDataSet(queryDataSet1);
...
this.getContentPane().add(jDbTable1, BorderLayout.NORTH);
```

This code introduces the `LiveTable` data binding Bean, and connects it to `JBuilder`'s `DataSet`.

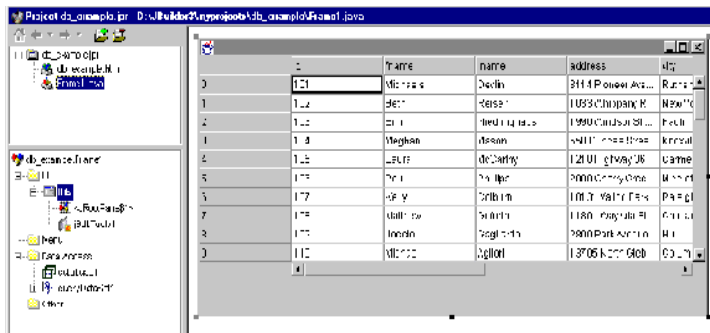


Figure 25 The project now contains a data bound table.

Now that the table component has been linked to the `QueryDataSet` component, the data bound table is part of the project. The project can now be compiled and run, or continued to be developed.

9.5.2 Data Binding Using JClass DataSource

Using the `DataSource` data binding Bean (`DSdbTable`), you can bind a table component with any `JDBC`-compliant data source. The `DSdbTable` Bean works in any IDE¹ but can also be used if you are developing an applet or application without one. Data binding with the `DSdbTable` Bean requires:

- Sun Microsystems' Beans Development Kit (BDK) or an IDE
- JDK 1.3.1 or greater
- JClass DataSource
- JClass LiveTable's `DSdbTable` Bean

1. If you are developing an application with `JBuilder`, you can use the specific data binding Bean, `JDbTable`, that was designed for use with it.

- a data source properly set up in Windows' ODBC Data Source Administrator

JClass DataSource is available as part of the JClass DesktopViews suite. Visit <http://www.quest.com> for information and downloads.

When data binding your table component with a database, JClass DataSource manages the connection and queries to the database in your development project. Using the DSdbTable Bean creates a table component that connects with JClass DataSource, thus completing the data binding link.

JClass DataSource uses two data binding Beans: the JCData Bean and the JCTreeData Bean. JCData allows data binding to flat data models, while JCTreeData allows data binding to hierarchical data models. The following example provides a general overview of data binding a LiveTable component to a database in Sun's Bean Development Kit.

It is assumed that you already are familiar with setting up a database connection with JClass DataSource. For specific information, please refer to your JClass DataSource documentation. Binding your table with a database involves:

- creating a project in an IDE or the Beans Development Kit
- establishing a database connection with JCData or JCTreeData
- inputting database query statements with either the JCData or the JCTreeData's NodePropertiesEditor or TreePropertiesEditor
- adding the DSdbTable data binding Bean to the work area

Establish a database connection with JCData

Insert JCData into the design area. Doing this will allow you to begin working with the NodePropertiesEditor in the BDK Properties window.

If desired, enter names in the **Description** and **Model Name** fields. In this example, we will leave the BDK's default names (**Node1** and **JCData1**). On the **Serialization** tab, click **Save As** to save your serialization file. Next, click the **Data Model** tab to specify which database you want to connect to.

You need to specify the **Server Name** and **Driver** on the **Data Model / JDBC / Connection** tab. For the purposes of this example, we are using the *demo.mdb* (JClass Demo) database. In the **Server Name** field, enter or select **jdbc:odbc:JClassDemo**, and in the **Driver** field, enter or select **sun.jdbc.odbc.JdbcOdbcDriver**. Ensure that the **Prompt User For Login** checkbox is not selected, and test the connection. When you

receive confirmation that the database connection is successful, you can begin to set up the query statements.

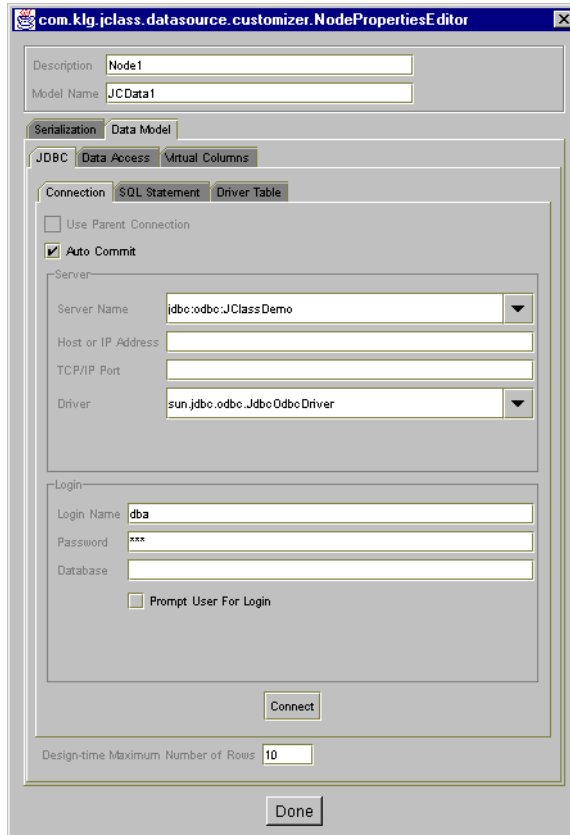
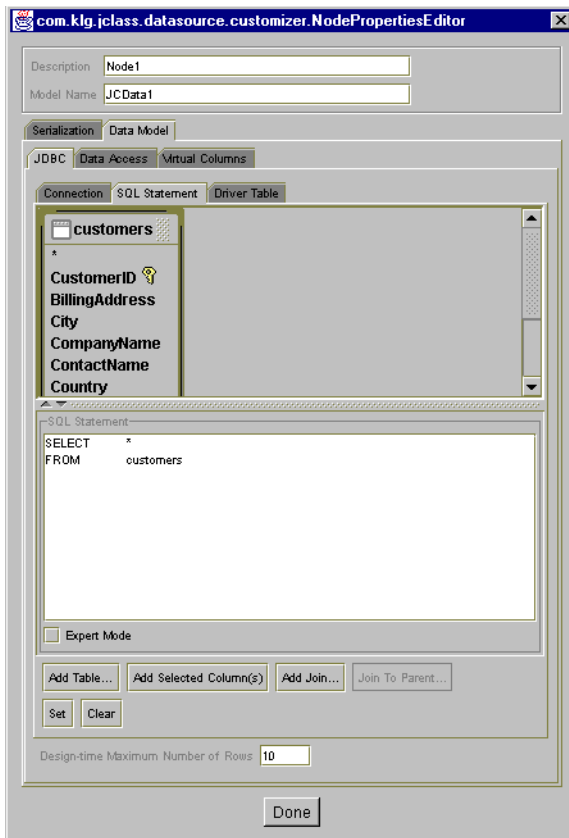


Figure 26 Connecting to the *demo.mdb* database in Sun's Beans Development Kit.

Inputting database query statements with the DataBean

In order to properly query the database you have connected to, you need to input your query statement in the fields found on the **Data Model / JDBC / SQL Statement** tab. For this example, the *demo.mdb* database contains various tables, one of which is *Customers*. Enter `select * from Customers` in the *SQL Statement* window to take all of the fields from the *demo.mdb*'s customers table, then click **Set**.

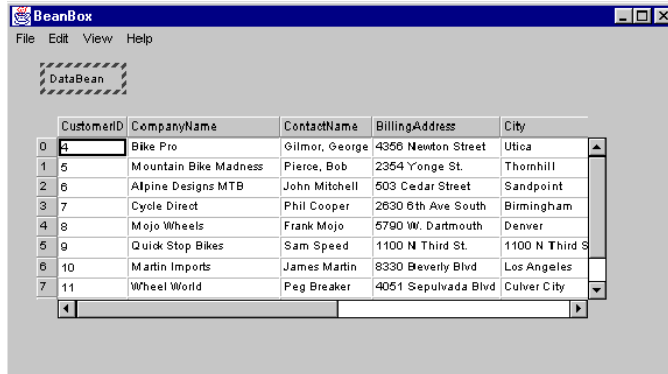
Now that you've successfully connected to, and queried the database, click **Done**.



Adding DSdbTable

The last step in creating a data bound table in your development project is adding the actual table component. In the BDK's Toolbox, click the DataSource data binding Bean (DSdbTable) and drop it into the BeanBox design area. Doing this will allow you to begin working with the DSdbTable properties in the Properties window. In the list of properties, click the **dataBinding** property, and set the connection to the appropriate data source. The data source is determined by what you entered in the **Description** and **Model Name** fields in the DataBeanComponentEditor (if you used the defaults in this example,

they will be **Node1** and **JCData1**). The table object will update to reflect the successful binding to the data source.



At this point, you have a table component in your design area, that is bound to the designated data source. You can now continue developing the rest of your application.

9.6 Interacting with Data Bound Tables

When a data bound table component has been successfully placed into your applet or application, you can interact with the table that takes advantage of the binding between the component and the data source.

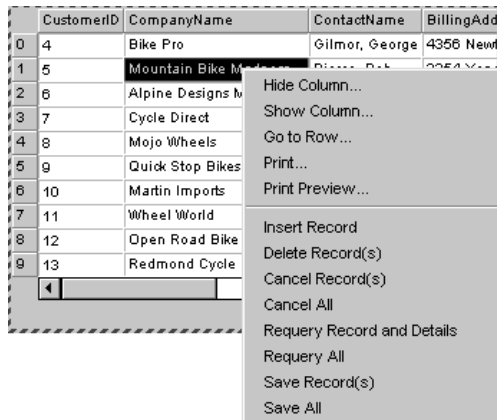


Figure 27 Interacting with the data bound table component.

These actions are accessible through the table component's pop-up menu. By right-clicking a record, or multiple selected records, a list of possible actions is presented to the user.

Pop-up Menu Item	Function
Insert Record	Adds a new record to the current table and bound data source.
Delete Record(s)	Removes the selected row(s) from the current table and bound data source.
Cancel Record(s)	Cancel changes made to selected records.
Cancel All	Cancel all changes made to all records.
Requery Record and Details	Requeries the selected record(s) and any of its children from the database.
Requery All	Requeries all records in the table from the database.
Save Record(s)	Commits changes to selected records in the table, and updates the bound data source.
Save All	Commits all changes made in the table to the database

9.7 Property Differences Between the JClass LiveTable Beans

Most of the common properties of the data binding Beans (`JBdbTable` and `DSdbTable`), are the same as the `LiveTable` Bean. By retaining most of the `LiveTable` Bean properties (outlined in Section 9.3.2, [LiveTable Properties](#)), the new Beans provide feature-rich data binding table components.

The following data binding Bean properties are either unavailable, or have a new editors.

Data binding Bean	Property difference from <code>LiveTable</code> Bean
<code>CellSize</code>	Unavailable in the data binding Beans.
<code>Data</code>	Unavailable: replaced by specific data binding properties.
<code>FrozenRow</code>	Unavailable in the data binding Beans.
<code>LeftColumn</code>	Unavailable: data bound table always starts at column 0.
<code>RowLabelDisplay</code>	Unavailable in the data binding Beans.

Data binding Bean	Property difference from LiveTable Bean
SpanningCells	Unavailable in the data binding Beans.
Style	Same property; new editor.
TopRow	Unavailable: data bound table always starts at row 0.
TraverseCycle	Unavailable: always on in the data binding Beans.

Part ***II***

Reference
Appendices

Appendix A

Event Summary

This table is a quick reference to JClass LiveTable's events and their corresponding event listeners. Event listeners may use up to three methods that are used during the process of executing the event. The standard naming convention for these methods are `before<Event>`, `<event>`, and `after<Event>`. For details on how to use events and listeners in your programs, see [Events and Listeners](#), in Chapter 7.

Event and Description	Event Methods	Action			Listener Interface
		before	on	after	
JCellDisplayEvent Posted when a cell's contents are to be displayed in a table.	getRow getColumn getCellData getDisplayData setDisplayData		● ● ● ● ●		JCellDisplayListener
JEditCellEvent Posted when a cell's contents are to be edited.	getRow getColumn getEditingComponent getType isCancelled setCancelled	● ● ● ● ● ●	● ● ● ● ● —	● ● ● ● ● —	JEditCellListener
JPaintEvent Posted when a portion of the table is painted.	getType getStartRow getStartColumn getEndRow getEndColumn getCellRange	● ● ● ● ● ●		● ● ● ● ● ●	JPaintListener

Event and Description	Event Methods	Action			Listener Interface
		before	on	after	
JCPrintEvent ^a The event posted for each page during the printing process.	getGraphics getMarginUnits getNumHorizontalPages getNumPages getNumVerticalPages getPage getPageDimensions getPageMargins getPageResolution getTableDimensions getType	● ● ● ● ● ● ● ● ● ● — ●	● ● ● ● ● ● ● ● ● ● ● ●	● ● ● ● ● ● ● ● ● ● ● ●	JCPrintListener
JCResizeCellEvent Posted when a cell in the table is resized.	getRow getColumn getCurrentRowHeight getNewRowHeight setNewRowHeight getCurrentColumnWidth getNewColumnWidth setNewColumnWidth isCancelled setCancelled	● ● ● ● — ● ● ● — ● ● ●	● ● ● ● ● ● ● ● ● ● ● ●	● ● ● ● — ● ● ● — ● ● —	JCResizeCellListener
JCResizeCellEvent Posted while a cell is being resized.	getRow getColumn getCurrentRowHeight getNewRowHeight setNewRowHeight getCurrentColumnWidth getNewColumnWidth setNewColumnWidth isCancelled setCancelled		● ● ● ● ● ● ● ● ● ● ●		JCResizeCellMotionListener ^b
JCScrollEvent Posted when a user resizes a row and/or column.	getAdjustable getDirection getEvent getType getValue setValue		● ● ● ● ● ● ●	● ● ● ● ● —	JCScrollListener

Event and Description	Event Methods	Action			Listener Interface
		before	on	after	
JCSelectEvent Posted when a user selects one or more cells.	getType getStartRow getStartColumn getEndRow getEndColumn isCancelled setCancelled getAction getActionString	● ● ● ● ● ● ● ● ● ●	● ● ● ● ● ● ● ● ● ●	● ● ● ● ● ● — ● ● ●	JCSelectListener
JCSortEvent Posted after a sortByColumn call.	getColumns getNewRows		● ●		JCSortListener
JCTableDataEvent Posted when the TableDataModel object is modified.	getRow getColumn getNumAffected getDestination getCommand		● ● ● ● ●		JCTableDataListener ^c
JCTraverseCellEvent Posted when a user traverses from one cell to another.	getRow getColumn getNextRow getNextColumn setNextRow setNextColumn getTraverseType isCancelled setCancelled		● ● ● ● ● ● ● ● ● ●	● ● ● ● — — ● ● ● —	JCTraverseCellListener

- a. JCSortEvent actions are not before<Event>, on<Event> and after<Event> as they are with other events; the action events are printPageHeader(), printPageBody() and printPageFooter().
- b. JCSortEvent has one method: sortByColumn(). It is called repeatedly during cell resizing.
- c. JCTableDataEvents are posted by the table's data source, and not by the table itself.

Appendix B

JClass LiveTable Property Listing

Properties of com.klg.jclass.table.JCTable ■ *Properties of com.klg.jclass.table.CellStyleModel*
Properties of com.klg.jclass.table.beans.LiveTable ■ *Properties of com.klg.jclass.table.db.jbuilder.JBdbTable*
Properties of com.klg.jclass.table.db.datasource.DSdbTable

The following lists summarize all of the JClass LiveTable properties. Each of these properties have two *accessor methods*: `set` and `get`. Methods are instantiated using `set(PropertyName)`, and you can retrieve the current value of any property using the property's `get` method.

The lists below are organized by the class that their accessor methods are called in, and further by the type of property. The lists show the property, a brief description, and either its enumerable value, defined by `JCTableEnum` or an example of a value for setting the property. Default values are marked with an asterisk (*).

B.1 Properties of com.klg.jclass.table.JCTable

Name	Description	Values/Examples
<code>AllowCellResize</code>	The <code>AllowCellResize</code> property specifies whether and how an end user is able to resize rows and columns.	<code>JCTableEnum.RESIZE_ALL*</code> <code>JCTableEnum.RESIZE_NONE</code> <code>JCTableEnum.RESIZE_COLUMN</code> <code>JCTableEnum.RESIZE_ROW</code>
<code>AllowResizeBy</code>	This property determines whether row heights and column widths can be resized by labels.	<code>JCTableEnum.RESIZE_BY_LABELS*</code> <code>JCTableEnum.RESIZE_BY_CELLS</code> <code>JCTableEnum.RESIZE_BY_ALL</code>
<code>AutoEdit</code>	Determines whether an editor is automatically displayed in a cell when it is entered.	boolean value: default <code>false</code>

Name	Description	Values/Examples
AutoScroll	This property sets how the table scrolls when the user moves out of the bounds of the displayed table.	JTableEnum.AUTO_SCROLL_NONE* JTableEnum.AUTO_SCROLL_ROW JTableEnum.AUTO_SCROLL_COLUMN JTableEnum.AUTO_SCROLL_BOTH
CellBorderWidth	Sets the shadow thickness around each cell.	integer: number of pixels
CharHeight	Height in characters of individual cells.	specific row number, JTableEnum.LABEL, or JTableEnum.ALL, plus the number of characters
CharWidth	Width of column in characters.	specific column number, JTableEnum.LABEL, or JTableEnum.ALL, plus the number of characters
ColumnHidden	Determines if the column is hidden.	boolean value: default false
ColumnLabelDisplay	Determines whether the column labels display in the table.	boolean value: default false
ColumnLabelOffset	Distance between column labels and table cells.	pixels (default: 0). For example: setColumnLabelOffset(4)
ColumnLabelPlacement	Location of the column labels (top or bottom of the table).	JTableEnum.PLACE_TOP* JTableEnum.PLACE_BOTTOM
ColumnSelection	Selects a range of columns.	int column range
Component	Swing and lightweight AWT components in individual cells.	row and column index, component
ComponentBorderWidth	This property determines the spacing between the border of a cell and the cell component.	integer value (pixels)

Name	Description	Values/Examples
Cursor	Creates a cursor and determines the cursor type.	Cursor.CROSSHAIR_CURSOR Cursor.DEFAULT_CURSOR Cursor.E_RESIZE_CURSOR Cursor.HAND_CURSOR Cursor.MOVE_CURSOR Cursor.N_RESIZE_CURSOR Cursor.NE_RESIZE_CURSOR Cursor.NW_RESIZE_CURSOR Cursor.S_RESIZE_CURSOR Cursor.SE_RESIZE_CURSOR Cursor.SW_RESIZE_CURSOR Cursor.TEXT_CURSOR Cursor.W_RESIZE_CURSOR Cursor.WAIT_CURSOR
EditHeightPolicy	Vertical sizing policy for cell editors.	JTableEnum.EDIT_SIZE_TO_CELL* JTableEnum.EDIT_ENSURE_ MINIMUM_SIZE JTableEnum.EDIT_ENSURE_ PREFERRED_SIZE
EditWidthPolicy	Horizontal sizing policy for cell editors.	JTableEnum.EDIT_SIZE_TO_CELL* JTableEnum.EDIT_ENSURE_ MINIMUM_SIZE JTableEnum.EDIT_ENSURE_ PREFERRED_SIZE
FocusColor	Determines the color of the focus indicator.	Any java.awt.Color object.
FocusIndicator	This property sets the focus indicator.	JTableEnum.FOCUS_RECTANGLE* JTableEnum.FOCUS_DASHED_ RECTANGLE JTableEnum.FOCUS_NONE JTableEnum.FOCUS_HIGHLIGHT
Font	Sets the font for the entire table.	array of colors
Foreground	Sets the foreground color for the entire table.	array of colors
FrameBorder	Sets the border for the frame around the table.	CellBorderModel

Name	Description	Values/Examples
FrameBorderWidth	Sets the thickness of the frame around the entire table.	pixels (default:0). For example: setFrameBorderWidth(5);
FrozenColumn Placement	Sets the location of all frozen columns within the component display. Changing the placement of frozen columns does not change the location of the columns in the table's internal CellValues.	JTableEnum.PLACE_LEFT* JTableEnum.PLACE_RIGHT
FrozenColumns	Specifies the number of columns from the start of the table that are not horizontally scrollable.	number of columns to freeze. For example: setFrozenColumns(3);
FrozenRow Placement	Specifies the location of all frozen rows.	JTableEnum.PLACE_TOP* JTableEnum.PLACE_BOTTOM
FrozenRows	Specifies the number of rows from the start of the table that are not vertically scrollable.	number of rows to freeze. For example: setFrozenRows(2);
HorizSBAttachment	Attach point for horizontal scrollbar. When set to SIZE_TO_CELLS, the scrollbar ends at the edge of the visible cells. When set to SIZE_TO_TABLES, the scrollbar is attached to the whole side of the table.	JTableEnum.SIZE_TO_CELLS* JTableEnum.SIZE_TO_TABLE
HorizSBDisplay	Determines when to display horizontal scrollbar.	JTableEnum.SBDISPLAY_ALWAYS JTableEnum.SBDISPLAY_NEVER JTableEnum.SBDISPLAY_AS_NEEDED*
HorizSBOffset	Distance between the table and horizontal scrollbar in pixels.	integer: number of pixels. For example: setHorizSBOffset(3);

Name	Description	Values/Examples
HorizSBPosition	Position of horizontal scrollbar. When set to POSITION_BY_CELLS, the scrollbar is attached to the visible cells. When set to POSITION_AT_SIDE, the scrollbar is attached to the whole side of the table.	JTableEnum.POSITION_BY_CELLS* JTableEnum.POSITION__AT_SIDE
HorizSBTrack	Determines how the horizontal scrollbar acts during scroll tracking.	JTableEnum.TRACK_LIVE* JTableEnum.TRACK_COLUMN_ NUMBER JTableEnum.TRACK_ROW
HorizSBTrackRow	Determines the row number whose text is displayed when JTableEnum.TRACK_ROW is used with setHorizSBTrack()	JTableEnum.LABEL or integer value (row number's cell data)
JumpScroll	Determines whether the table will visually scroll smoothly or whether the display will 'jump' to display the cells scrolled to.	JTableEnum.JUMP_NONE* JTableEnum.JUMP_HORIZONTAL JTableEnum.JUMP_VERTICAL JTableEnum.JUMP_ALL
LeftColumn	Indicates the non-frozen column at least partially visible at the left side of the window.	integer: column number
MarginHeight	Specifies the distance (in pixels) between the inside edge of the cell border.	integer: pixels. For example: setMarginHeight(4);
MarginWidth	Specifies the distance (in pixels) between the inside edge of the cell border and the left/right edge of the cell's contents.	integer: pixels. For example: setMarginWidth(3);

Name	Description	Values/Examples
MaxHeight	Sets the maximum number of pixels for a row's height.	integer value (pixels)
MaxWidth	Sets the maximum number of pixels for a column's width.	integer value (pixels)
MinCellVisibility	Minimum visible percentage of a cell.	integer: 1 to 100
MinHeight	Sets the minimum number of pixels for a row's height.	integer value (pixels)
MinWidth	Sets the minimum number of pixels for column's width.	integer value (pixels)
PixelHeight	Row height in pixels. This property controls the height unless set to JCTableEnum.NOVALUE.	integer value (pixels) Special values: JCTableEnum.VARIABLE JCTableEnum.AS_IS JCTableEnum.VARIABLE_ESTIMATE
PixelWidth	Column width in pixels. This property controls the width unless set to JCTableEnum.NOVALUE.	integer value (pixels) Special values: JCTableEnum.VARIABLE JCTableEnum.AS_IS JCTableEnum.VARIABLE_ESTIMATE
PopupMenuEnabled	Determines whether or not to display the table popup menu.	boolean value default: false for LiveTable default: true for JDbTable and DSdbTable
RepaintEnabled	Sets whether the table should be redrawn and recomputed whenever one of its properties is set.	boolean value (default: true)
ResizeEven	Specifies that when a user resizes a row or column, all of the rows or columns also resize the same amount.	boolean value (default: false)

Name	Description	Values/Examples
ResizeInteractive	Determines if a table is repainted to reflect column width or row height changes if they are resized interactively.	boolean value (default: false)
RowHidden	Determines if the row is hidden.	boolean value (default: false)
RowLabelDisplay	This property has a boolean value to determine whether the row labels display in the table	boolean value (default: true)
RowLabelOffset	Offset between row labels and table.	integer: number of pixels
RowLabelPlacement	Location of the row labels.	JCTableEnum.PLACE_LEFT* JCTableEnum.PLACE_RIGHT
RowSelection	Selects a range of rows.	integer range
SelectIncludeLabels	Sets the selection behavior for row and column labels. When true, full column or row selections do not change the visible properties of the label. When false, the row or column label is changed.	boolean (default: true)
SelectedBackground	Background color for cells that have been selected. The default is the cells' foreground color.	Color value. For example: setSelectedBackground(Color. yellow);
SelectedBackground Mode	Sets the table's background color for selected cells.	JCTableEnum.USE_SELECTED_ BACKGROUND JCTableEnum.USE_CELL_ BACKGROUND JCTableEnum.USE_CELL_ FOREGROUND
SelectedCells	List of selected cells.	Vector or JCCellRange

Name	Description	Values/Examples
TrackSize	Returns the size of the track component.	Dimension value
TraverseCycle	Specifies that when a user traverses to past the top, bottom, left, or right of the table, the traversal wraps to the opposite side.	boolean (default: true)
VariableEstimateCount	Sets the number of cells to use in estimating variable pixel calculations.	default: JTableEnum.ALL
VertSBAttachment	Attach point for vertical scrollbar. When set to SIZE_TO_CELLS, the scrollbar ends at the edge of the visible cells. When set to SIZE_TO_TABLE, the scrollbar is attached to the whole side of the table.	JTableEnum.SIZE_TO_CELLS JTableEnum.SIZE_TO_TABLE
VertSBDisplay	Determines when to display vertical scrollbar.	JTableEnum.SBDISPLAY_AS_NEEDED* JTableEnum.SBDISPLAY_ALWAYS JTableEnum.SBDISPLAY_NEVER
VertSBOffset	Distance between the table and vertical scrollbar.	integer: number of pixels. For example: setVertSBOffset(4);
VertSBPosition	Position of vertical scrollbar. When set to POSITION_BY_CELLS, the scrollbar is attached to the visible cells. When set to POSITION__AT_SIDE, the scrollbar is attached to the whole side of the table.	JTableEnum.POSITION_BY_CELLS JTableEnum.POSITION_AT_SIDE

Name	Description	Values/Examples
VertSBTrack	This property determines how the vertical scrollbar acts during scroll tracking.	JTableEnum.TRACK_LIVE* JTableEnum.TRACK_ROW_NUMBER JTableEnum.TRACK_COLUMN
VertSBTrackColumn	Determines the column number whose text is displayed when JTableEnum.TRACK_ROW is used with setVertSBTrack().	JTableEnum.LABEL or integer value (row number's cell data)
VisibleColumns	Sets the number of columns used to determine the initial table size. This value is not updated when columns or the table are resized.	integer: number of visible columns
VisibleRows	Sets the number of rows used to determine the initial table size. This value is not updated when rows or the table are resized.	integer: number of visible rows

B.2 Properties of com.klg.jclass.table.CellStyleModel

Name	Description	Values/Examples
Background	Sets the background color of the cell.	Color value
CellBorder	Sets the cell border object for the cell.	CellBorderModel
CellBorderColor	Sets the cell's border color.	Color value
CellBorderColorMode	Sets the mode used to determine cell border color.	JTableEnum.USE_CELL_BORDER_COLOR JTableEnum.BASE_ON_BACKGROUND JTableEnum.BASE_ON_FOREGROUND

Name	Description	Values/Examples
CellBorderSides	Visible border sides (defined by CellBorderType) for individual cells.	JTableEnum.BORDERSIDE_NONE JTableEnum.BORDERSIDE_ALL* JTableEnum.BORDERSIDE_LEFT JTableEnum.BORDERSIDE_RIGHT JTableEnum.BORDERSIDE_TOP JTableEnum.BORDERSIDE_BOTTOM
ClipHints	Determines whether clip arrows are shown, and where, when the contents of the cell do not fit in the cell frame.	JTableEnum.SHOW_NONE JTableEnum.SHOW_HORIZONTAL JTableEnum.SHOW_VERTICAL JTableEnum.SHOW_ALL
Editable	Editable attribute for individual cells.	boolean value (default: true)
Font	Sets the cell's font.	Font value
Foreground	Sets the foreground color of the cell.	Color value
HorizontalAlignment	Sets the horizontal alignment for the contents of the cell.	JTableEnum.LEFT* JTableEnum.CENTER JTableEnum.RIGHT
RepeatBackground	Determines if the background color repeats for rows or columns.	JTableEnum.REPEAT_NONE* JTableEnum.REPEAT_ROW JTableEnum.REPEAT_COLUMN
RepeatBackground Colors	Repeats pattern for background colors.	array of colors
RepeatForeground	Determines if the foreground color repeats for rows or columns.	JTableEnum.REPEAT_NONE* JTableEnum.REPEAT_ROW JTableEnum.REPEAT_COLUMN
RepeatForeground Colors	Repeats pattern for foreground colors.	array of colors
Traversable	Allows traversal of individual cells.	boolean (default: true)
VerticalAlignment	Sets the vertical alignment for the contents of the cell.	JTableEnum.TOP* JTableEnum.CENTER JTableEnum.RIGHT

B.3 Properties of `com.klg.jclass.table.beans.LiveTable`

Name	Description
<code>about</code>	Displays component version and contact information.
<code>allowCellResize</code>	Determines whether end-user can resize cells at runtime.
<code>allowResizeBy</code>	Sets how cells can be resized.
<code>autoEdit</code>	Determines whether the table automatically displays an editor when entering a cell.
<code>autoScroll</code>	Determines whether table scrolls during selection/traversal.
<code>cellBorderWidth</code>	Specifies width of cell/label borders.
<code>cellSize</code>	Specifies row heights and column widths.
<code>data</code>	Specifies table data, data source, and row/column labels.
<code>editHeightPolicy</code>	Determines height control of cell editing components.
<code>editWidthPolicy</code>	Determines width control of cell editing components.
<code>frameBorderType</code>	Specifies frame border type.
<code>frameBorderWidth</code>	Specifies frame border width.
<code>frozenCellLayout</code>	Determines the position of frozen rows/columns.
<code>focusColor</code>	Sets the color of the focus indicator.
<code>focusIndicator</code>	Determines the type of focus indicator used.
<code>jumpScroll</code>	Determines whether jump scrolling is turned on.
<code>labelLayout</code>	Determines the position of row/column labels.
<code>leftColumn</code>	Specifies first column displayed on screen.
<code>marginHeight</code>	Specifies top and bottom cell margins.
<code>marginWidth</code>	Specifies left and right cell margins.
<code>minCellVisibility</code>	Determines amount of cell scrolled into view during traversal.
<code>popupMenuEnabled</code>	Determines whether the pop-up menu is enabled.
<code>resizeEven</code>	Determines whether cell resizing is applied evenly to non-label cells.

Name	Description
<code>resizeInteractive</code>	Determines whether cell resizing is interactively displayed.
<code>sBLayout</code>	Determines the space between scrollbars and cells.
<code>selectedBackground</code>	Determines the background color of selected cells.
<code>selectedForeground</code>	Determines the foreground color of selected cells.
<code>selectIncludeLabels</code>	Determines whether selection includes row and column labels.
<code>selectionPolicy</code>	Determines type of cell selection allowed.
<code>spannedCells</code>	Specifies cell ranges to treat as spanned cells.
<code>styles</code>	Sets the styles property, which defines visual aspects of the table.
<code>swingDataModel</code>	Sets the table's data source to use a specified Swing <code>TableModel</code> object, instead of using the <code>data</code> property.
<code>topRow</code>	Specifies first row displayed on screen.
<code>trackCursor</code>	Sets whether the mouse pointer movements are tracked.
<code>traverseCycle</code>	Determines whether cell traversal cycles on the same row or moves to the next row.

B.4 Properties of `com.klg.jclass.table.db.jbuilder.JBdbTable`

Name	Description
<code>about</code>	Displays component version and contact information.
<code>allowCellResize</code>	Determines whether end-user can resize cells at run-time.
<code>allowResizeBy</code>	Sets how cells can be resized.
<code>autoEdit</code>	Determines whether the table automatically displays an editor when entering a cell.
<code>autoScroll</code>	Determines whether table scrolls during selection/traversal.
<code>cellBorderWidth</code>	Specifies width of cell/label borders.

Name	Description
cellSize	Specifies row heights and column widths.
dataSet	Specifies the table data source.
editHeightPolicy	Determines height control of cell editing components.
editWidthPolicy	Determines width control of cell editing components.
frameBorderStyle	Specifies frame border type.
frameBorderWidth	Specifies frame border width.
frozenCellLayout	Determines the position of frozen rows/columns.
focusColor	Sets the color of the focus indicator.
focusIndicator	Determines the type of focus indicator used.
labelLayout	Determines the position of row/column labels.
leftColumn	Specifies first column displayed on screen.
marginHeight	Specifies top and bottom cell margins.
marginWidth	Specifies left and right cell margins.
minCellVisibility	Determines amount of cell scrolled into view during traversal.
popupMenuEnabled	Determines whether the pop-up menu is enabled.
resizeEven	Determines whether cell resizing is applied evenly to non-label cells.
resizeInteractive	Determines whether cell resizing is interactively displayed.
sBLayout	Determines the space between scrollbars and cells.
selectedBackground	Determines the background color of selected cells.
selectedForeground	Determines the foreground color of selected cells.
selectIncludeLabels	Determines whether selection includes row and column labels.
selectionPolicy	Determines type of cell selection allowed.
spannedCells	Specifies cell ranges to treat as spanned cells.
styles	Sets the styles property, which defines visual aspects of the table.

Name	Description
swingDataModel	Sets the table's data source to use a specified Swing <code>TableModel</code> object, instead of using the <code>data</code> property.
topRow	Specifies first row displayed on screen.
trackCursor	Sets whether the mouse pointer movements are tracked.
traverseCycle	Determines whether cell traversal cycles on the same row or moves to the next row.

B.5 Properties of `com.klg.jclass.table.db.datasource.DSdbTable`

Name	Description
about	Displays component version and contact information.
allowCellResize	Determines whether end-user can resize cells at run-time.
allowResizeBy	Sets how cells can be resized.
autoEdit	Determines whether the table automatically displays an editor when entering a cell.
autoScroll	Determines whether table scrolls during selection/traversal.
cellBorderWidth	Specifies width of cell/label borders.
cellSize	Specifies row heights and column widths.
dataBinding	Specifies the table data source.
editHeightPolicy	Determines height control of cell editing components.
editWidthPolicy	Determines width control of cell editing components.
frameBorderType	Specifies frame border type.
frameBorderWidth	Specifies frame border width.
frozenCellLayout	Determines the position of frozen rows/columns.
focusColor	Sets the color of the focus indicator.
focusIndicator	Determines the type of focus indicator used.
labelLayout	Determines the position of row/column labels.
leftColumn	Specifies first column displayed on screen.

Name	Description
marginHeight	Specifies top and bottom cell margins.
marginWidth	Specifies left and right cell margins.
minCellVisibility	Determines amount of cell scrolled into view during traversal.
popupMenuEnabled	Determines whether the pop-up menu is enabled.
resizeEven	Determines whether cell resizing is applied evenly to non-label cells.
resizeInteractive	Determines whether cell resizing is interactively displayed.
sBLayout	Determines the space between scrollbars and cells.
selectedBackground	Determines the background color of selected cells.
selectedForeground	Determines the foreground color of selected cells.
selectIncludeLabels	Determines whether selection includes row and column labels.
selectionPolicy	Determines type of cell selection allowed.
styles	Sets the styles property, which defines visual aspects of the table.
swingDataModel	Sets the table's data source to use a specified Swing <code>TableModel</code> object, instead of using the <code>data</code> property.
topRow	Specifies first row displayed on screen.
trackCursor	Sets whether the mouse pointer movements are tracked.
traverseCycle	Determines whether cell traversal cycles on the same row or moves to the next row.
useDataSourceEditable	Determines whether the editable column state is defined by the data source or the table.

Appendix C

Porting JClass 3.6.x Applications

[Overview of Changes](#) ■ [Porting Strategies](#) ■ [Highlights of Main Changes](#)

JClass LiveTable for Java2 is significantly different from previous versions. The focus of JClass LiveTable for Java2 is to make any changes necessary to take full advantage of Swing, and to restructure the product for future expansion.

C.1 Overview of Changes

The major changes are listed in the following table. Each change is discussed in more detail later in this appendix.

Change	Rationale
package name change (com.klg.jclass.table)	Old package name pre-dated naming standard.
Swing-like API	JClass 4 is Swing-based.
data subpackage	Makes it easier to find stock data sources. Stock data sources now include the <code>JC</code> prefix.
beans subpackage	Makes it easier to find Beans. Important for users who wish to remove the Beans from deployment JARs.
no more JCString	JCString has been replaced by HTML in cells.
JCTable and Table	In LiveTable 3.*, Table was the core class and JCTable was a backwards-compatibility class for LiveTable 2.* customers. In LiveTable 4.*, JCTable is the core class, and Table is a backwards-compatibility class for LiveTable 3.* customers.
beans APIs	Various bean properties have been modified. Essentially, the LiveTable 4.* Beans are not backwards compatible. This porting guide does not talk about the Beans.
new events	The events fired by LiveTable have been rationalized based on user feedback. The listener methods have been renamed to be consistent across all methods.

Change	Rationale
cell changes	LiveTable now supports two different rendering models. Many renderers and editors were updated to make use of Swing and of the new rendering model. All stock editors and renderers use the <code>JC</code> prefix.

C.2 Porting Strategies

LiveTable 4.x comes with two tools designed to help you move from LiveTable 3.x to 4.x:

- `com.klg/jclass/util/scripts/table3to4.pl` is a Perl conversion script. It is designed to convert about 60-80% of table code.
- `com.klg.jclass.table.Table` is a subclass of `JCTable` that supports the old LiveTable 3 API.

C.3 Highlights of Main Changes

New Beans Subpackage

All the Beans have been moved to the `beans` subpackage. There have been many Bean property changes.

No More JCString

`JCStrings` have been replaced by `HTML` in cells. This is supported by Swing, and has been added to LiveTable where appropriate.

You can now put raw `HTML` into headers and footers, as long as the text starts with `<html>`. `HTML` is also valid in axis annotations, axis titles, and legend elements.

Style-based Property Setting

Styles are objects that encapsulate all the visual attributes of cells. You set the property for the style object, then apply it to a range of cells.

In LiveTable 3.*, each visual attribute was set individually on a range of cells.

For example, the following code sets the foreground and background color on a range of cells:

```
table.setForeground(1, JCTableEnum.ALL, Color.blue);
table.setBackground(1, JCTableEnum.ALL, Color.black);
```

Using styles, the attribute is set on the style, and the style is applied to the cells:

```
JCCellStyle style;
style.setForeground(Color.blue);
style.setBackground(Color.blue);
table.setCellStyle(1, JCTableEnum.ALL, style);
```


In general, styles are easier to use for tables that tend to set multiple attributes on ranges of cells (the majority of cases for table users).

Please refer to [Building a Table](#), in Chapter 2, for details on style-based property setting.

JCTable and Table

In LiveTable 4.x, JCTable is the core class. Table is a subclass of JCTable that is to be used if you want to use LiveTable 3.x API calls. Table does not support all the LiveTable 3.x API.

Beans APIs

Many of the Bean properties are the same. The Appearance property has been replaced by Styles.

New events

The new event structure has been rationalized, and is documented in [Events and Listeners](#), in Chapter 7. It is recommended that you rewrite your event handling code based on the new events.

Cell Editors and Renderers

Most classes now begin with JC.

The drawing-based rendering model (formerly called CellRenderer) is now called JCLightCellRenderer. The getPreferredSize() method now takes a Graphics object:

Old	New
<code>jclass.cell.CellRenderer</code>	<code>com.klg.jclass.cell.JCLightCellRenderer</code>
<code>getPreferredSize(CellInfo, Object)</code>	<code>getPreferredSize(Graphics, JCCellInfo, Object)</code>

There is now a new rendering model based on a component called JCComponentCellRenderer. Some of the default renderers now use this component instead of JCLightCellRenderer.

JCComponentCellRenderer and JCLightCellRenderer have a common base class called JCCellRenderer.

Not all of the renderers are still around.

Old	New
<code>ButtonCellRenderer</code>	None
<code>CheckboxCellRenderer</code>	<code>JCCheckboxCellRenderer</code>

Old	New
ChoiceCellRenderer	JCComboBoxCellRenderer
EllipsisCellRenderer	None
ImageCellRenderer	JCImageCellRenderer
RawImageCellRenderer	JCRawImageCellRenderer
ScaledImageCellRenderer	JCScaledImageCellRenderer
StringCellRenderer	JCStringCellRenderer
WordWrapCellRenderer	JCWordWrapCellRenderer
None	JCHTMLCellRenderer
None	JCLabelCellRenderer

The advanced cell editors and renderers have been removed. Users are expected to use `JClass Field` for this purpose.

The editing interface has changed slightly:

Old	New
<code>initialize(InitialEvent, CellInfo, Object)</code>	<code>initialize(AWTEvent, JCCellInfo, Object)</code>
<code>getPreferredSize(CellInfo, Object)</code>	None. Use component preferred size.
<code>KeyModifier[] getReservedKeys()</code>	<code>JCKeyModifier[] getReservedKeys()</code>

Appendix D

Colors and Fonts

Colorname Values ■ *RGB Color Values* ■ *Fonts*

This section provides information on common colorname values, specific RGB color values, and fonts applicable to all Java programs.

D.1 Colorname Values

The following lists all the colornames that can be used within Java programs. The majority of these colors will appear the same (or similar) across different computing platforms.

- black
- blue
- cyan
- darkGray
- darkGrey
- gray
- grey
- green
- lightGray
- lightGray
- lightBlue
- magenta
- orange
- pink
- red
- white
- yellow

D.2 RGB Color Values

The following lists all the main RGB color values that can be used within JClass LiveTable. RGB color values are specified as three numeric values representing the red, green, and blue color components; these values are separated by dashes (“-”).

The following RGB color values describe the colors available to Unix systems. It is recommended that you test these color values in a JClass program on a Windows or Macintosh system before utilizing them.

The list begins with all of the variations of white, then blacks and greys, and then describes the full color spectrum ranging from reds to violets.

Example code from an HTML file:

```
<PARAM NAME=backgroundList VALUE="(4, 5 255-255-0)">
```

RGB Value	Description
255-250-250	Snow
248-248-255	Ghost White
245-245-245	White Smoke
220-220-220	Gainsboro
255-250-240	Floral White
253-245-230	Old Lace
250-240-230	Linen
250-235-215	Antique White
255-239-213	Papaya Whip
255-235-205	Blanched Almond
255-228-196	Bisque
255-218-185	Peach Puff
255-222-173	Navajo White
255-228-181	Moccasin
255 248-220	Cornsilk
255-255-240	Ivory
255-250-205	Lemon Chiffon
255-245-238	Seashell
240-255-240	Honeydew
245-255-250	Mint Cream
240-255-255	Azure
240-248-255	Alice Blue
230-230-250	Lavender
255-240-245	Lavender Blush
255-228-225	Misty Rose
255-255-255	White
0-0-0	Black

RGB Value	Description
47-79-79	Dark Slate Grey
105-105-105	Dim Gray
112- 128-144	Slate Grey
119- 136-153	Light Slate Grey
190- 190-190	Grey
211- 211-211	Light Gray
25-25-112	Midnight Blue
0-0-128	Navy Blue
100- 149 237	Cornflower Blue
72-61-139	Dark Slate Blue
106-90-205	Slate Blue
123- 104 238	Medium Slate Blue
132-112- 255	Light Slate Blue
0-0-205	Medium Blue
65-105-225	Royal Blue
0-0-255	Blue
30-144-255	Dodger Blue
0-19 -255	Deep Sky Blue
135-206-235	Sky Blue
135-206-250	Light Sky Blue
70-130-180	Steel Blue
176-196- 222	Light Steel Blue
173-216-230	Light Blue
176-224-230	Powder Blue
175-238-238	Pale Turquoise
0-206-209	Dark Turquoise
72-209-204	Medium Turquoise
64-224-208	Turquoise
0-255-255	Cyan
224-255-255	Light Cyan
95-158-160	Cadet Blue
102-205-170	Medium Aquamarine

RGB Value	Description
127-255-212	Aquamarine
0-100-0	Dark Green
85-107-47	Dark Olive Green
143-188-143	Dark Sea Green
46-139-87	Sea Green
60-179-113	Medium Sea Green
32-178-170	Light Sea Green
152-251-152	Pale Green
0-255-127	Spring Green
124-252- 0	Lawn Green
0-255-0	Green
127-255- 0	Chartreuse
0-250-154	Medium Spring Green
173-255-47	Green Yellow
50-205-50	Lime Green
154-205-50	Yellow Green
34-139-34	Forest Green
107-142-35	Olive Drab
189-183-107	Dark Khaki
240-230-140	Khaki
238-232-170	Pale Goldenrod
250-250-210	Light Goldenrod Yellow
255-255-224	Light Yellow
255-255-0	Yellow
255-215-0	Gold
238-221-130	Light Goldenrod
218-165-32	Goldenrod
184-134-11	Dark Goldenrod
188-143-143	Rosy Brown
205-92-92	Indian Red
139-69-19	Saddle Brown
160-82-45	Sienna

RGB Value	Description
205-133-63	Peru
222-184- 135	Burlywood
245-245-220	Beige
245-222-179	Wheat
244-164-96	SandyBrown
210-180-140	Tan
210-105-30	Chocolate
178-34-34	Firebrick
165-42-42	Brown
233-150-122	Dark Salmon
250-128-114	Salmon
255-160-122	Light Salmon
255-165- 0	Orange
255-140-0	Dark Orange
255-127-80	Coral
240-128-128	Light Coral
255-99-71	Tomato
255-69-0	Orange Red
255-0-0	Red
255-105-180	Hot Pink
255-20-147	Deep Pink
255-192-203	Pink
255-182-193	Light Pink
219-112-147	Pale Violet Red
176-48-96	Maroon
199-21-133	Medium Violet Red
208-32-144	Violet Red
255-0-255	Magenta
238-130-238	Violet
221-160-221	Plum
218-112-214	Orchid
186-85-211	Medium Orchid

RGB Value	Description
153-50-204	Dark Orchid
148-0-211	Dark Violet
138-43-226	Blue Violet
160- 32-240	Purple
147-112-219	Medium Purple
216-191-216	Thistle

D.3 Fonts

There are five different font names that can be specified in any Java program. They are:

- Courier
- Dialog
- DialogInput
- Helvetica
- TimesRoman

Note: Font names are case-sensitive.

There are also four standard font style constants that can be used. The valid Java font style constants are:

- bold
- bold+italic
- italic
- plain

These values are strung together with dashes (“-”) when used with the `VALUE` attribute. You must also specify a point size by adding it to other font elements. To display a text using a 12-point italic Helvetica font, use the following:

```
Helvetica-italic-12
```

All three elements (font name, font style, and point size) must be used to specify a particular font display; otherwise, the default font is used instead.

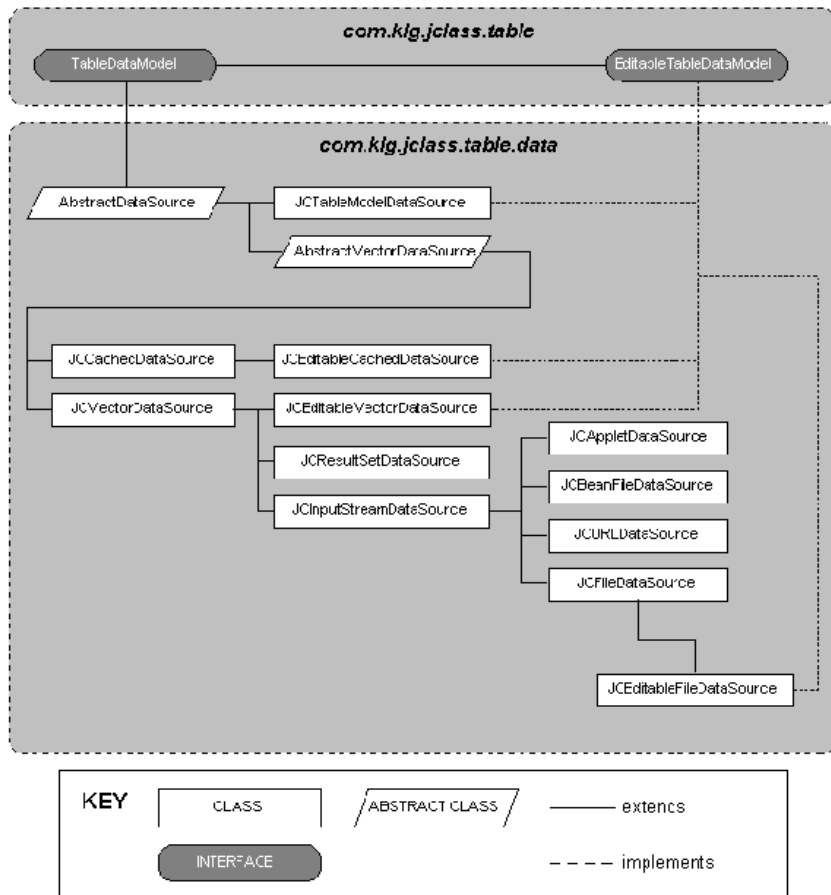
Note: Font display may vary from system to system. If a font does not exist on a system, the default font is displayed instead.

Appendix E

JClass LiveTable Inheritance Hierarchy

General JClass LiveTable Classes

The following figure gives an overview of class inheritance for table creation in JClass LiveTable.

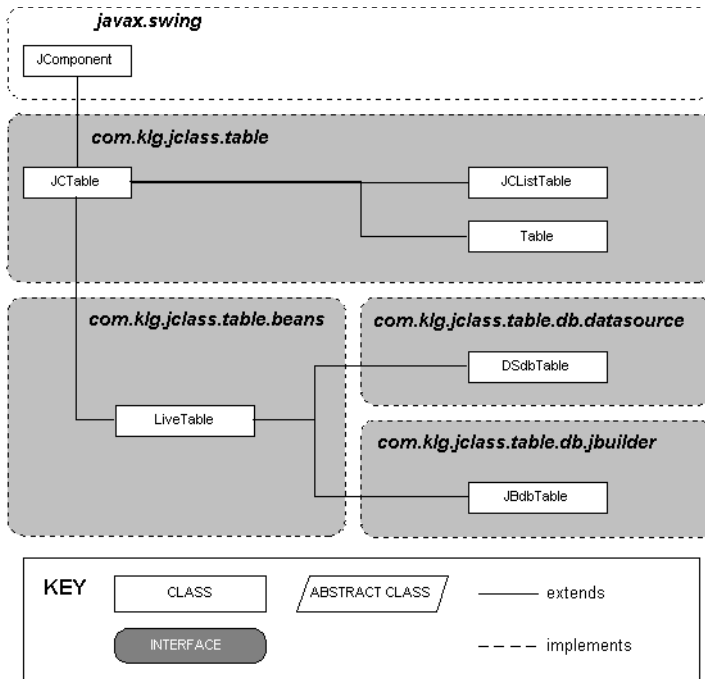


JCTable is the core JClass LiveTable class, with which most table programming is performed, and from which all LiveTable Beans are extended. The JListTable class

extends `JCTable` and provides a quick way of formatting a table to look and act like a list. The data binding Beans allow you to bind your table application to an IDE-specific or ODBC/JDBC-compliant data source.

JClass LiveTable Data Classes

The following figure provides an overview of class inheritance for data handling in JClass LiveTable.



`TableDataModel` is the core data source interface, and `EditableTableDataModel` extends this interface to allow editing of the data. `JCVectorDataSource` stores table data in a series of vectors. `JCInputStreamDataSource` extends `JCVectorDataSource` to read from any stream, and `JCAppletDataSource`, `JCBeanFileDataSource`, `JCFileDataSource`, and `JCURLDataSource` further extend it to read from specific stream types. The other stock data sources exist for other, more specific, situations.

Appendix F

Distributing Applets and Applications

Using JarMaster to Customize the Deployment Archive

F.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 *jctable.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 in the JClass DesktopViews suite of products. For more details please refer to [Quest's Web site](#).

Appendix G

Overview of Examples and Demos

[JClass LiveTable Examples](#) ■ [JClass LiveTable Demos](#)

G.1 JClass LiveTable Examples

The following sections offer an overview of the examples included with JClass LiveTable. These examples demonstrate the various concepts that can be used to create a table. The examples vary in depth and complexity, but all are helpful in showing you how to implement some of JClass LiveTable's features.

Note: Please consult the [Installation Guide](#) to ensure that you are properly set up to run these examples.

G.1.1 Introductory Examples

The introductory JClass LiveTable examples are part of the tutorial found in 'Hello Table' – [JClass LiveTable Tutorial](#), in Chapter 1, that walk you through the construction and modification of a basic table. These examples are part of the `examples.table.intro` package, and are found in the `examples/table/intro` directory:

<i>ExampleTable1.java</i>	A table with basic visual and interactive properties.
<i>ExampleTable2.java</i>	A table based on the previous example, but with labels.
<i>ExampleTable3.java</i>	The previous example table's label colors have been changed.
<i>ExampleTable4.java</i>	The label fonts and text alignment have been modified.
<i>ExampleTable5.java</i>	Cell and frame borders and spacing have been changed. One table cell's colors have been changed.
<i>ExampleTable6.java</i>	Cell editing has been enabled. Cell width and height have been changed.
<i>ExampleTable7.java</i>	Table resizing only with labels has been set.
<i>ExampleTable8.java</i>	Column sorting has been enabled.

G.1.2 Table Layout Examples

The layout examples demonstrate how to build tables that go beyond the basic grid design for tables. These are part of the `examples.table.layout` package, and are found in the `examples/table/layout` directory:

<i>Cars.java</i>	A table that contains tables. This example also demonstrates cell spanning and visual property settings.
<i>Flexible.java</i>	Columns resize dynamically, depending on how the whole table is resized.

G.1.3 Cell Style Examples

The cell style examples demonstrate how style properties affect the appearance of individual or groups of cells. You can find more information about cell style properties in [Cell Styles](#), in Chapter 2. These examples are part of the `examples.table.styles` package, and are found in the `examples/table/styles` directory:

<i>Animated.java</i>	Displays animated cells.
<i>BorderTypes.java</i>	Showcases the cell border options available, including some customized cell borders.
<i>CellBorders.java</i>	Demonstrates various cell border appearance attributes, including border type, border width, and cell side coverage. The table updates with each selection.
<i>RepeatColor.java</i>	Uses alternating background colors for rows or columns, to improve readability.
<i>TextureTable.java</i>	Uses a customized border type that tiles a background image onto table cells.
<i>UpdateStyle.java</i>	Continuously updates the background color property for a particular table row's style.

G.1.4 Cell Examples

The cell editing and rendering examples demonstrate specific applications for JClass LiveTable's editors and renderers. For more information about cell editors and renderers, please refer to [Displaying and Editing Cells](#), in Chapter 4. These examples are part of the `examples.table.cell` package, and are found in the `examples/table/cell` directory:

<i>CurrencyTable.java</i>	Uses <code>CurrencyRenderer</code> to take all data, regardless of its original type, and render it as dollar values in the table.
---------------------------	--

<i>Histogram.java</i>	Uses a custom cell renderer to take randomized integer data, and render them as horizontal bars in a table.
<i>MoneyTable.java</i>	Uses a custom cell editor and a custom data type.
<i>TriangleTable.java</i>	A demonstration of a non-text based editor and renderer for Integer and Polygon types.
<i>WordWrap.java</i>	Uses a word wrapping renderer to handle long String data in cells.

G.1.5 Table Listener Examples

The event and listener examples demonstrate how you can work with events that your table receives. For more information about the types of events and listeners available, please refer to [Events and Listeners](#), in Chapter 7. These examples are part of the `examples.table.listeners` package, and are found in the `examples/table/listeners` directory:

<i>BooleanDisplay.java</i>	Uses the <code>JCCellDisplay</code> listener to intercept the display event and change it before rendering (changes the boolean from true/false to yes/no).
<i>CancelEdit.java</i>	Shows how to use <code>JCTableDataEdit</code> and <code>JCTableDataListener</code> to cancel active edits when the data source changes.
<i>DoubleClickEdit.java</i>	Demonstrates the use of <code>JCEditCell</code> events by requiring a user to double click a cell in order to edit it. A single click, or click and drag, only selects cells.
<i>EditCell.java</i>	Demonstrates the use of <code>JCEditCell</code> events by placing a message in the table's pane, telling you which column's cell is being edited.
<i>ResizeCell.java</i>	Uses <code>JCResizeCell</code> events to ensure that the maximum and minimum set values for row heights and column widths are adhered to when the user resizes rows and columns. These maximum and minimum values are defined in the table's code.
<i>SelectListener.java</i>	Uses <code>JCSelectListener</code> events to make sure that the portion of the table that is defined as non-selectable are not included in cell range selections, and cannot be initially selected.
<i>SkipNavigation.java</i>	Cell traversal events are used to listen for and skip a column. When the user traverses to the right from column 0, cell focus skips column 1 and moves to column 2.

<i>Sorter.java</i>	Demonstrates column sorting by performing a String and numerical sort on a column of integers.
<i>TwoTables.java</i>	JCScroll events are used to sync the horizontal scrolling for two separate tables.

G.1.6 Table Interaction Examples

The interaction examples show you how to improve the usability of your table applications by enhancing some of your table's interactive features. For information about table interactions, please refer to [Programming User Interactivity](#), in Chapter 6. These examples are part of the `examples.table.interactions` package, and are found in the `examples/table/interactions` directory:

<i>ColumnLabelPopUp.java</i>	Shows how to use column labels as tool tips.
<i>DragDrop.java</i>	Demonstrates the use of drag and drop by allowing the user to interactively reorder rows and columns. This is done by clicking and dragging labels.
<i>ExcelTableExample.java</i>	Demonstrates how to copy and paste selected cells from a JCTable to a Microsoft Excel spreadsheet and vice-versa.
<i>TableAutoColumnResize.java</i>	Depicts how to emulate JTable auto column resizing in JCTable.
<i>TraverseOnEnter.java</i>	Displays how cell traversal (across rows) can optionally be handled by using the Enter key.

G.1.7 Data Source Examples

The data source examples demonstrate how you can customize and use data sources with your table application. These examples are part of the `examples.table.datasource` package, and are found in the `examples/table/datasource` directory:

<i>DynamicTest.java</i>	This example table continually updates the data it displays, as its data source's values randomly change.
<i>DynamicTest2.java</i>	This example is the same as <i>DynamicTest.java</i> , except that it uses <code>AbstractDataSource</code> .
<i>FileData.java</i>	Demonstrates how to load data from an external file by using <code>JCFileDataSource</code> . Both a CSV and table format file are loaded and displayed side by side.

<i>FontList.java</i>	Uses the <code>getFont()</code> method to access AWT's available fonts and use it as the data source. Displays the name, appearance, and other information pertaining to the available fonts.
<i>Pivot.java</i>	Shows how to create a data source instance that can transpose itself. You can use a customized data source or create your own.
<i>StaticEditableTest.java</i>	A basic, custom data source that is built on editable String values.
<i>StaticTest.java</i>	This example is the same as the previous example, except that the table data is not editable.
<i>XMLFileData.java</i>	Demonstrates how to load XML-formatted data from an external file by using <code>JCFileDataSource</code> . Please note that in order to run this example, you will need to have the <i>jaxp.jar</i> and <i>crimson.jar</i> files in your CLASSPATH. For more information, please see Loading Data from an XML Source , in Chapter 3.
<i>XMLTableModelData.java</i>	Demonstrates how to load XML-formatted data into a Swing <code>TableModel</code> class. Please note that in order to run this example, you will need to have the <i>jaxp.jar</i> and <i>crimson.jar</i> files in your CLASSPATH. For more information, please see Loading Data from an XML Source , in Chapter 3.

G.1.8 DataSource Data Binding Examples

The data binding examples demonstrate the use of data binding in tables. Binding your table to a ODBC, JDBC, or IDE-specific data source gives you live control over a robust data source. These examples are part of the `examples.table.db.*` package, and are based in the `examples/table/db/` directory:

<i>datasource/SimpleData.java</i>	Shows how to bind <code>DSdbTable</code> to <code>JClass DataSource</code> .
<i>jbuilder/JBuilderDBTable.java</i>	This example shows how to bind <code>JBdbTable</code> to a <code>JBuilder QueryDataSet</code> .

G.1.9 Advanced JClass LiveTable Examples

The advanced examples combine two or more concepts demonstrated in previous examples. They are found in the `examples.table.advanced` package, and are in the `examples/table/advanced` directory:

DecimalTableCellDisplay.java This example assigns each column to take on a different format of the same number. It uses `CellDisplayListener` to format the rendered text, and it uses `UserData` to store the numeric format of the cells.

DecimalTableCellStyle.java This example shows how to achieve the same results as *DecimalTableCellDisplay.java*, except that `CellStyleModel` object is used.

Gradient.java Creates a color whose value is incrementally changed from cell to cell. This example uses the cell style defaults, but shows how to bypass some properties for particular cells.

G.2 JClass LiveTable Demos

The JClass LiveTable demos showcase different types of complete table solutions. They combine several table concepts that are explained in this manual, including cell spanning, user interaction, and editors and renderers.

The following table offers an overview of each demo that comes with JClass LiveTable, including demo name, package, description, and sample screen shot.

Note: Please consult the Installing JClass Products section in the [Installation Guide](#) to ensure that you are properly set up to run these demos.

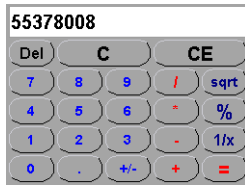
BeanSweeper.java
`demos.table.beanSweeper`



This demo recreates the well known minesweeper game. In the creation of this table, cell styles are used extensively. Style properties that affect cell and border colors, border types, and image use, give the game its look.

Calculator.java

demos.table.calculator



This basic calculator demo is a simple yet effective demonstration of how cell styles can be used to customize the look of your table. In this case, cell border and color properties are set to make this look like an authentic \$2 calculator. This demo also implements a data source that gathers information that is input by the user.

CustomCells.java

demos.table.customCells

String	Date	Double	Bar	Check	Color	ComboBox
April 1970	java.util.Gregory	79.159475577514	█	<input type="checkbox"/>	●	orange
April 1971	java.util.Gregory	46.001452631932	█	<input type="checkbox"/>	●	white
April 1972	java.util.Gregory	30.591596929942	█	<input type="checkbox"/>	●	green
April 1973	java.util.Gregory	15.065047896786	█	<input type="checkbox"/>	●	green
April 1974	java.util.Gregory	99.979000833444	█	<input type="checkbox"/>	●	purple
April 1975	java.util.Gregory	70.447350717092	█	<input type="checkbox"/>	●	orange
April 1976	java.util.Gregory	36.103457235362	█	<input type="checkbox"/>	●	orange
April 1977	java.util.Gregory	94.891049681174	█	<input type="checkbox"/>	●	yellow
April 1978	java.util.Gregory	70.883232279122	█	<input type="checkbox"/>	●	black
August 1970	java.util.Gregory	55.757691891626	█	<input checked="" type="checkbox"/>	●	blue
August 1971	java.util.Gregory	64.889148932562	█	<input type="checkbox"/>	●	blue
August 1972	java.util.Gregory	23.082718498802	█	<input type="checkbox"/>	●	green
August 1973	java.util.Gregory	60.267670594732	█	<input type="checkbox"/>	●	purple
August 1974	java.util.Gregory	55.303750395996	█	<input type="checkbox"/>	●	purple
August 1975	java.util.Gregory	76.725200613222	█	<input type="checkbox"/>	●	white

An effective demonstration of using custom cell renderers, which take various data types and converts them to the standard desired formats. Also, column sorting and JClass Field component integration are showcased.

Matrix.java

demos.table.matrix

MarketMatrix	Success Level											
SubMarket	Country1				Country2				Country3			
	Market Share	Market Share	Market Share	Market Share	Market Share	Market Share	Market Share	Market Share	Market Share	Market Share	Market Share	
Company 1	10	20	30	40	50	60	70	80	90	100	110	
Company 2	15	25	35	45	55	65	75	85	95	105	115	
Company 3	20	30	40	50	60	70	80	90	100	110	120	
Company 4	25	35	45	55	65	75	85	95	105	115	125	
Company 5	30	40	50	60	70	80	90	100	110	120	130	
Company 6	35	45	55	65	75	85	95	105	115	125	135	
Company 7	40	50	60	70	80	90	100	110	120	130	140	
Company 8	45	55	65	75	85	95	105	115	125	135	145	
Company 9	50	60	70	80	90	100	110	120	130	140	150	
Company 10	55	65	75	85	95	105	115	125	135	145	155	
Company 11	60	70	80	90	100	110	120	130	140	150	160	
Company 12	65	75	85	95	105	115	125	135	145	155	165	

This information matrix shows how a custom JCellRenderer can be used to add properties that are not built into JClass LiveTable. Here, an implementation of the CellStyleModel interface, RotatedCellStyle, adds the Rotation property to render text at any given angle.

PrimeTime.java

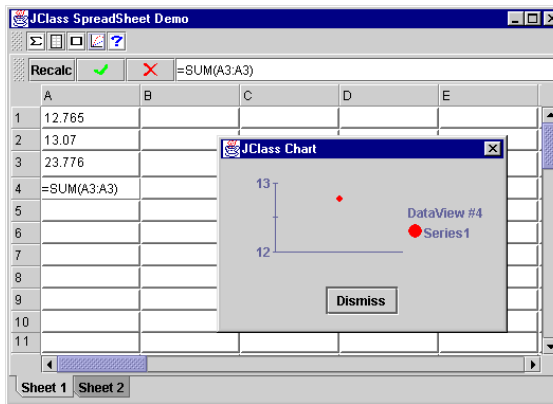
demos.table.primetime

Channel	Category	7:00	7:30	8:00	8:30	9:00	9:30	10:00
7	15	15	The Outer Limits	The Pretender	Profiler			
8	4	20	Babylon 5	Eury, Vampire Slayer	Baywatch		News	
9	16	16	News	Medicine Women	Early Edition		CFR98	
5	6	6	Sat. Report	Empty Nest	Liberty St	Joey's St	Movie: Naked Lunch	
6	3	3	Wifemans	Fa. Father	Party Fritin		News	
7	14	14	News	College Football				
8	8	8	Entertainment Now	Medicine Women	High Incident		Spin City	
11	11	11	Pennacric: Wings	The Pretender	Profiler		Operation	
15	20	50	Orange	Sportsbeat	Air Force	TDA	Movie: Naked Lunch	
16	22	22	Entertainment Now	Medicine Women	High Incident		Operation	

This demo presents television listings, and shows what can be achieved with cell styles and cell spanning. This demo also incorporates printing functionality into a table.

SpreadSheet.java

demos.table.spreadsheet



This spreadsheet demo emulates an Excel-style spreadsheet. It supports a subset of spreadsheet functionality and is intended to demonstrate formulae integration with table.

Within this demo you can also chart a selected region of the spreadsheet.

Stocks.java

demos.table.stocks

Stock Monitor					
Symbol: <input type="text"/>					Search
Symbol	High	Low	Close	Volume	Change
AA PR	12 1/2	4 3/4	15	1055.0	1/4
AEN	13	2 5/8	14 1/8	4469.0	+1 5/8
AEX	4	14 1/8	9 5/8	6155.0	1/2
AIA	7	6	4 1/2	6965.0	1/2
All	1/2	4 3/8	12 1/4	8373.0	1/8
ALC	12 1/2	4 1/4	6 1/8	9946.0	+1 3/8
ALE	19 7/8	10 1/4	4	8171.0	+1 1/2
AMS	8 1/2	3/4	5 7/8	3932.0	3/8
AMV	9 1/2	17	12 1/2	8309.0	-1 1/2
ARS	13 7/8	7 1/8	18 3/4	9578.0	3/8

Legend: 70% drop 50% drop 20% drop 20% gain

This stock information demo provides the user with quick data updates and colors rows according to changes in value. Custom cell rendering and cell styles are emphasized with this demo.

Index

A

- about property 165, 214–215, 217
- Abstract Windowing Toolkit, see AWT 15
- AbstractDataSource 76
- adding labels 35
- alignment
 - cells 53
 - changing, tutorial 16
- AllowCellResize property 118, 165, 203, 214–215, 217
 - effect on mouse pointers 129
- AllowResizeBy property 165, 203, 214–215, 217
- API 3
 - programming 27
 - setting properties 160
- applets 231
 - JarMaster 231
- applications
 - distributing 231
 - JarMaster 231
- attaching scrollbars 33
- autoEdit property 203, 214–215, 217
- automatic scrolling 120
- autoScroll property 165, 204, 214–215, 217
- AWT
 - color constants 15
 - font styles, tutorial 17
 - image file formats supported 58

B

- background
 - colors 15, 52
 - repeating 52
 - property 52, 212
- basic table 10
- Bean 159
 - LiveTable 161, 164
 - about property 165
 - allowCellResize property 165
 - allowResizeBy property 165
 - autoScroll property 165
 - CellBorderWidth property 166
 - CellSize property 166
 - data property 166

- editHeightPolicy property 167
- editWidthPolicy property 167
- focusColor property 167
- focusIndicator property 168
- frameBorderType property 168
- frameBorderWidth property 168
- frozenCellLayout property 168
- LabelLayout property 168
- leftColumn property 169
- marginHeight property 169
- marginWidth property 169
- minCellVisibility property 169
- sBLayout property 169
- selectedBackground property 171
- selectedForeground property 171
- selectIncludeLabels property 172
- selectionPolicy property 172
- spannedCells property 172
- styles property 173
- topRow property 174
- LiveTable, changing property editor table size 163
- LiveTable, property
 - editors 161
- LiveTable, selecting cell 162
- LiveTable, selecting labels 162
- property differences 194
- setting properties 161

- BeanBox 159
- Beans Development Kit 159
- borders
 - colors 38
 - component 38
 - custom 56
 - frame attributes 162, 172
 - sides
 - specifying 57
 - table frame 38
 - type 54
 - width 37
- Borland JBuilder
 - data binding 184
- built-in styles, using and modifying 50

C

- CellBorder property 212

- CellBorderColor property 212
- CellBorderColorMode property 212
- CellBorderSides property 57, 213
- CellBorderType property 19
- CellBorderWidth property 19, 37, 204
- cellBorderWidth property 166, 214–215, 217
- CellInfo interface 100
- cells
 - adding color to one cell, tutorial 18
 - alignment 53
 - alignment, tutorial 16
 - area
 - spacing from labels 36
 - border sides 57
 - border types 54
 - border width 37
 - border, IDE tutorial 180
 - borders, tutorial 19
 - CellEditor interface 79
 - CellInfo interface 100
 - CellRenderer interface 79
 - clipping arrows 43
 - controlling editor size 42
 - controlling selection at runtime 126
 - current 28
 - custom borders 56
 - customizing traversal 115
 - default editors 90
 - default selection 123
 - default traversal 115
 - definition 28
 - determining visibility 121
 - dimensions 43
 - displaying 79, 131
 - displaying images 58
 - displaying multiple lines 46
 - editable, tutorial 21
 - editing 28, 63, 79, 89, 133
 - default 80
 - editors
 - and CellInfo interface 100
 - controlling size 42
 - creating 92
 - default 90
 - defined 90
 - getting reserved keys 93
 - handling events 99
 - key control 99
 - mapping a data type 91
 - registering 113
 - reserving keys 93, 99
 - setting for a series 91
 - subclassing 94
 - writing 96
 - fonts 54
 - forcing traversal 116
 - image alignment 53
 - image layout 58
 - interactive traversal 117
 - making visible 121
 - margins 37
 - mathematical operation 114
 - maximum height/width 46
 - minimum height/width 46
 - minimum visibility 116
 - multiline 46
 - newline characters 46
 - preset styles, selection 32
 - range
 - referencing 29
 - referencing, all 30
 - referencing, one 29
 - removing a selection range 126
 - renderers 84
 - component based 87
 - creating 84
 - data type 83
 - mapping 83
 - mapping a data type 83
 - registering 113
 - setting 82
 - subclassing 85
 - writing 85
 - rendering 28, 81
 - default 80
 - reserving keys for editors 93
 - resizing 118
 - selected cell list 125
 - selecting 144
 - selecting ranges 125
 - selection
 - customizing 124
 - selection colors 34
 - selection, row and column labels 125
 - selection, tutorial 22
 - setting dimensions 43
 - setting properties 31
 - setting renderers 82
 - setting selection colors 34
 - setting values 69
 - size
 - absolute 44
 - character height/width 43
 - character width/height 43
 - pixel width/height 44
 - variable 45
 - changing to fixed values 45
 - size, tutorial 20
 - spacing 19
 - spanning 58
 - specific data types 80
 - styles 47

- built-in, modifying 50
- built-in, using 50
- changing default 48
- defining 48
- getting and setting 48
- parent styles 49
- pluggable look and feel 51
- properties 47
- retrieving from table 49
- setting properties 31
- tutorial 15
- text alignment 53
- thickness, IDE tutorial 180
- traversal 115
- traversing 151
- values
 - setting 69
 - variable dimensions 45
 - changing to fixed values 45
- cellSize property 166, 214, 216–217
- CellStyleModel 15, 50
- central registry 113
- change values 76
- changing default cells
 - styles 48
- character
 - determining cells
 - size 43
 - height 43
 - width 43
- CharHeight property 43, 204
 - tutorial 20
- CharWidth property 43, 204
 - tutorial 20
- ClassCastException 113
- clip arrows
 - tutorial 12
- clip hints
 - displaying 57
- ClipHints property 57, 213
- clipping
 - arrows 43
 - image 57
 - text 57
- colors 52
 - AWT constants 15
 - background 52
 - colname values 223
 - focus rectangle 33
 - foreground 52
 - repeating 52
 - RGB 223
 - RGB color value list 223
 - selection 124
 - setting 52
 - setting, tutorial 15
 - colors, selection 34
 - setting 34
 - ColumnHidden property 47, 204
 - ColumnLabelDisplay property 204
 - tutorial 14
 - ColumnLabelOffset property 36, 204
 - ColumnLabelPlacement property 35, 204
 - columns
 - adding 67, 78
 - adding labels, IDE tutorial 176
 - controlling resizing 118
 - default resizing behavior 118
 - deleting 68
 - determining number 40
 - determining visibility 121
 - disallowing resizing 118
 - displaying 41
 - dragging 126
 - freezing 41–42
 - hiding 46
 - labels 35, 62, 129
 - displaying 14
 - labels, placement 35
 - making visible 121
 - moving 68
 - placement of frozen 42
 - referencing, all 29–30
 - referencing, entire 30
 - referencing, one 29
 - removing 78
 - resizing 118
 - resizing all at once 119
 - resizing with labels 119
 - selecting labels 125
 - set as left 34
 - setting the number 66
 - sorting 127, 129
 - sorting frozen 127
 - sorting multiple 128
 - sorting, tutorial 24
 - specifying labels 66
 - swapping 41
 - visible, getting 40
 - visible, setting 40
 - width
 - pixel value 43
 - width property 43
 - width, setting 43
 - ColumnSelection property 204
 - ColumnTrigger property
 - and dragging 126
 - and sorting 129
 - com.klg.jclass.table.beans.LiveTable 214
 - com.klg.jclass.table.CellStyleModel 212
 - com.klg.jclass.table.db.datasource.DSdbTable 217
 - com.klg.jclass.table.db.builder.JBdbTable 215

- com.klg.jclass.table.JTable 203
- com.klg.jclass.util.formulae 103
- comments on product 6
- component borders 38
- Component property 204
- ComponentBorderWidth property 38, 204
- context 29
- creating a cell editor 92
- creating cell renderers 84
- CSV 62
- current cell 28
 - definition 28
- current context 29
- Cursor property 205
- cursor type 33
 - tracking 33

D

- data
 - caching 65
 - cell editor 80
 - cell renderer 80
 - data source 62
 - data storage 61
 - editing 63
 - format, detection 62
 - from an input stream 64
 - getting from database 65
 - getting into a table 62
 - handling 61
 - property 166, 214
 - storing 64–65
 - Swing TableModel objects 66
 - updating dynamically 74
- data binding
 - examples 237
 - IDE 183
 - JBuilder 184
 - JClass DataSource 189
- data bound
 - interacting 193
- data source
 - adding and removing listeners 63, 69
 - and table size 62
 - communication with the table 61
 - creating 72
 - editable 63
 - editable, tutorial 21
 - event listeners 63
 - examples 236
 - IDE tutorial 175
 - JBuilder 184
 - JClass DataSource 189
 - JCVectorDataSource 64

- model-view-controller 61
- object 61
- retrieving data 62
- setting cell values 69
- setting properties 66
- stock data properties 66
- stock data sources 63
- tutorial 11
- data type
 - cell renderers 83
 - mapping 83
 - mapping to a cell editor 91
- database
 - getting data 65
- dataBinding property 217
- dataSet property 216
- default
 - cell editors 90
 - scrolling 120
- defaultCellStyle 51
- defaultLabelStyle 51
- deleting rows and columns 68
- demos 238
 - BeanSweeper.java 238
 - Calculator.java 239
 - CustomCells.java 239
 - Matrix.java 239
 - overview 233
 - PrimeTime.java 240
 - SpreadSheet.java 240
 - Stocks.java 240
- destination parameter 68
- dimensions
 - cell 43
- displaying
 - cells 131
 - clip hints 57
 - rows and columns 41
- distributing 231
 - applets and applications 231
 - JarMaster 231
- dragging rows and columns 126
- drawBackground 56
- drawing cells 80
- dynamically updating data 74

E

- editable cells 63
- Editable property 63, 213
- EditableTableModel 63
- EditHeightPolicy property 205
- editHeightPolicy property 42, 167, 214, 216–217
- editing cells 63, 80, 89
- editors

- LiveTable 161
 - setting 91
- EditWidthPolicy property 42
- editWidthPolicy property 167, 205, 214, 216–217
- evaluate
 - method in MathValue 105
- event listeners 131
 - adding and removing 69
 - cell display 132
 - data source 63
 - entering cells 134
 - JCellDisplayListener 131
 - JCEnterCellListener 134
 - JCPaintListener 136
 - JCPrintListener 137
 - JCResizeListener 138
 - JCScrollListener 141
 - painting 136
 - printing 137
 - resizing 138
- events 112, 131
 - cell display events 132
 - cell editors 99
 - editing cells 133
 - JCellDisplayEvent 131
 - JCEditCellEvent 133
 - JCPaintEvent 136
 - JCPrintEvent 137
 - JCResizeEvent 138
 - JCScrollEvent 141
 - JCSortEvent 147
 - JCTraverseCellEvent 149, 151
 - painting 136
 - printing 137
 - resizing 138
 - scrolling 141
 - sorting 147
 - summary 199
 - TableListenerPropagator 112
 - traversal 149, 151
- examples 233
 - advanced 238
 - cell 234
 - cell style 234
 - data binding 237
 - data source 236
 - introductory 233
 - overview 233
 - table interaction 236
 - table layout 234
 - table listener examples 235
- exceptions 113
 - ClassCastException 113
 - OperandMismatchException 113
- expression 105
 - interface 105

- lists 112, 114
 - MathExpressionList 112
 - QueryExpressionList 112
 - TableExpressionList 112
- mathematical 103
- references 114

F

- FAQs 5
- feature overview 1
- fixed values 45
- focus rectangle
 - color 33
- focusColor property 167, 205, 214, 216–217
- focusIndicator property 168, 205, 214, 216–217
- FocusRectColor property 33
- Font property 205, 213
- font styles
 - AWT 17
- fonts
 - cells 54
 - labels 54
 - matched by AWT 18
 - names 228
 - point size 228
 - setting, tutorial 17
 - size, tutorial 17
 - style constants 228
- footers
 - printing 156
- foreground
 - colors 15, 52
 - repeating 52
 - property 52
- Foreground property 205, 213
- format, RGB 223
- formulae hierarchy 103
- formulae package 103
- formulas
 - adding to JClass LiveTable 103
 - using in JClass LiveTable 113
- frame
 - border 162, 172
- FrameBorder property 38, 205
- FrameBorderType property
 - in IDEs 168
- frameBorderType property 168, 214, 216–217
- frameBorderWidth property 38, 168, 206, 214, 216–217
- freezing
 - columns 42
 - rows 42
- frozen
 - column placement 42

- columns 41
 - and sorting 127
- row placement 42
- rows 41
- frozenCellLayout property 168, 214, 216–217
- FrozenColumnPlacement property 206
- FrozenColumns property 206
 - properties
 - FrozenColumns 41
- FrozenRowPlacement property 206
- FrozenRows property 41, 206

G

- get method 203
- getDataFormat
 - method in MathValue 105
- getValueAt
 - method in MathMatrix 108
 - method in MathVector 107
- GIF
 - image file formats supported 58
- global table properties 32

H

- headers
 - printing 156
- hiding
 - columns 46
 - rows 46
- HorizontalAlignment property 58, 213
- HorizSBAAttachment property 33, 206
- HorizSBDisplay property 121, 206
- HorizSBOffset property 206
- HorizSBPosition property 207
- HorizSBTrack property 207
- HorizSBTrackRow property 207

I

- IDEs 159
 - data binding 183
 - setting properties 32, 160
 - tutorial 174
- image
 - alignment in cell 53
 - clipping 57
 - displaying in cells 58
 - formats supported 58
 - layout 58
 - layout in cell 58
- inheritance hierarchy 229

- input stream
 - getting data from 64
- Integrated Development Environment (IDE) 160
- interactivity 115
 - IDE tutorial 181
 - interacting with data bound tables 193
 - tutorial 21
- internationalization 26
- introduction
 - JClass LiveTable 1

J

- JAR 231
 - JarMaster 231
- JarMaster 231
 - JAR 231
- JavaBeans
 - features of 159
- JBuilder 184
 - data binding 184
- JCCachedDataSource 65
- JCCellBorder class 54
- JCCellDisplayEvent 131, 199
- JCCellDisplayListeners 131
- JCCellRange
 - in cell selection 125
- JCCellRenderer 83
- JCCellStyle 15, 48, 50
- JCComponentCellRenderer 87
- JCEditCellEvent 199
- JCExpressionCellRenderer 80
- JCInputStream 62
- JCInputStreamDataSource 64
- JClass 3.6.x applications
 - porting 219
- JClass DataSource 189
- JClass JarMaster 231
- JClass LiveTable, introduction 1
- JClass technical support 5
 - contacting 5
- jclass.cell package 79
- JCListTable 32
- JCPaintEvent 136, 199
- JCPaintListener 136
- JCPrintEvent 137, 200
- JCPrintListener 137
- JCPrintPreview 157
- JCPrintTable 155, 158
- JCResizeCellEvent 118, 138, 200
- JCResizeCellListener 138
- JCResizeCellMotionListener 138
- JCScrollEvent 122, 141, 200
- JCScrollListener 122, 141
- JCSelectEvent 201

- JCSelectListener 145
- JCSortEvent 147, 201
- JCSortListener 147
- JCTable 27
- JCTableDataEvent 149, 201
- JCTableDataListener 150
- JCTraversalCellEvent 151
- JCTraverseCellEvent 117, 201
- JCTraverseCellListener 151
- JCVectorDataSource 11
 - editing 64
- JDBC 183
- JPEG
 - image file formats supported 58
- JumpScroll property 207, 214

K

- keys
 - control 99
 - reserving for cell editors 93, 99

L

- label parameter 67
- labelLayout property 168, 214, 216–217
- labels
 - adding 13, 35
 - border sides 57
 - border width 37
 - column, IDE tutorial 176
 - columns 62
 - custom borders 56
 - definition 28
 - displaying, tutorial 14
 - fonts 54
 - formatting 13
 - layout, IDE tutorial 179
 - margins 37
 - offset from table 36
 - placement 35
 - preset styles, selection 32
 - referencing, all 30
 - referencing, all columns 30
 - referencing, all rows 30
 - referencing, column 29
 - referencing, row 29
 - resize, tutorial 23
 - rows 62
 - selecting 125
 - setting properties 31
 - spacing from cell area 36
 - spanning 58
 - specifying row and column 66

- using for resizing 119
- leftColumn property 34, 121, 169, 207, 214, 216–217
- list of cell editors 90
- listeners 112, 131
 - examples 235
 - management 76
 - scroll 122
 - TableListenerPropagator 112
- LiveTable
 - general classes, inheritance 229
- LiveTable Bean
 - about property 165
 - allowCellResize property 165
 - allowResizeBy property 165
 - autoScroll property 165
 - CellBorderWidth property 166
 - CellSize property 166
 - data property 166
 - editHeightPolicy property 167
 - editWidthPolicy property 167
 - focusColor property 167
 - focusIndicator property 168
 - frameBorderType property 168
 - frameBorderWidth property 168
 - frozenCellLayout property 168
 - LabelLayout property 168
 - leftColumn property 169
 - marginHeight property 169
 - marginWidth property 169
 - minCellVisibility property 169
 - properties 164
 - sBLayOut property 169
 - selectedBackground property 171
 - selectedForeground property 171
 - selectIncludeLabels property 172
 - selectionPolicy property 172
 - setting properties 161
 - spannedCells property 172
 - styles property 173
 - topRow property 174
- LiveTable Data
 - classes, inheritance 230
- loading data 69
- localization 26

M

- mapping 83
 - a data type 83
- marginHeight property 37, 169, 207, 214, 216, 218
- margins
 - cell and label 37
 - setting 37
- marginWidth property 37, 169, 207, 214, 216, 218
- math values 105

- mathematical expressions 103
- Mathematical operations 109
 - binary 109
 - range of cells 114
 - unary 109
- MathExpressionList 112
- MathMatrix 107
 - constructors 107
 - getValueAt method 108
 - matrixValue method 108
 - methods 108
 - numberValue method 108
 - setValueAt method 108
 - toString method 108
 - VectorValue method 108
- MathScalar 106
 - constructors 106
 - matrixValue method 106
 - methods 106
 - numberValue method 106
 - toString method 106
 - vectorValue method 106
- MathValue 103
 - class 105
 - evaluate method 105
 - getDataFormat method 105
 - matrixValue method 105
 - methods 105
 - numberValue method 105
 - setDataFormat method 105
 - vectorValue method 105
- MathVector 106
 - constructors 106
 - getValueAt method 107
 - matrixValue method 107
 - methods 107
 - numberValue method 107
 - setValueAt method 107
 - toString method 107
 - vectorValue method 107
- matrixValue
 - method in MathMatrix 108
 - method in MathScalar 106
 - method in MathValue 105
 - method in MathVector 107
- MaxHeight property 208
- maximum pixel
 - height 46
 - width 46
- MaxWidth property 208
- methods
 - accessor 203
- minCellVisibility property 169, 208, 214, 216, 218
- MinHeight property 208
- minimum
 - cell visibility 116

- pixel height 46
- pixel width 46
- MinWidth property 208
- model-view-controller 11, 61
 - data source 61
- mouse pointers
 - custom 129
 - disabling tracking 130
- multiline 46
 - headers, spanning 60
- multiple
 - columns, sorting 128
 - lines in cells 46
- MVC, see model-view-controller 61

N

- newline character
 - and Multiline property 46
- num_columns parameter 68
- num_rows parameter 68
- numberValue
 - method in MathMatrix 108
 - method in MathScalar 106
 - method in MathValue 105
 - method in MathVector 107
- NumColumns property 40, 62
- NumRows property 40, 62

O

- ODBC 183
- offset of labels 36
- OperandMismatchException 113
- Operation class 108
- operations
 - constructor 108
 - mathematical 109
 - binary 109
 - unary 109
 - methods 108
 - reducing values 111
- operators
 - Abs 109
 - Add 109
 - Average 109
 - Ceiling 109
 - Count 110
 - Divide 110
 - Floor 109
 - GeometricMean 110
 - in com.klg.jclass.util.formulae 109
 - Max 110
 - Median 110

- Min 110
- Multiply 110
- Power 110
- Product 110
- Root 109
- Round 109
- Sort 111
- StdDeviation 111
- Subtract 111
- Sum 111
- Trunc 109

P

- page layout
 - page size 155
 - setting, for printing 155
- page margins
 - setting, for printing 156
- page numbering
 - setting, for printing 156
- page resolution 156
- painting 136
- parent cell styles
 - creating 49
- PixelHeight property 44, 208
 - setting, tutorial 20
 - user row resizing 118
 - using to hide rows 46
- pixels
 - absolute height and width 44
 - estimate 45
 - maximum height/width 46
 - minimum height/width 46
 - variable height and width 45
 - changing to fixed values 45
- PixelWidth property 44, 208
 - column resizing 118
 - setting, tutorial 20
 - using to hide columns 46
- pluggable look and feel (PLAF) 51
- popupMenuEnabled property 208, 214, 216, 218
- porting
 - JClass 3.6.x applications 219
- position parameter 67–68
- PreferredSize 87
- preset styles 32
- print preview 157
- printing 137, 155
 - events 137
 - headers and footers 156
 - page layout 155
 - page margins 156
 - page numbering 156
 - page resolution 156
 - page size 155
 - preview 157
- product feedback 6
- programming the API 27
- properties
 - about 214–215, 217
 - access in IDE 32
 - accessor methods 203
 - allowCellResize 203, 214–215, 217
 - effect on mouse pointers 129
 - allowResizeBy 203, 214–215, 217
 - autoEdit 203, 214–215, 217
 - autoScroll 204, 214–215, 217
 - Background 212
 - cell style 47
 - CellBorder 212
 - CellBorderColor 212
 - CellBorderColorMode 212
 - CellBorderSides 57, 213
 - CellBorderWidth 37, 204, 214–215, 217
 - cellSize 214, 216–217
 - CharHeight 20, 43, 204
 - CharWidth 20, 43, 204
 - ClipHints 57, 213
 - Color 223
 - column width 43
 - ColumnHidden 47, 204
 - ColumnLabelDisplay 204
 - ColumnLabelOffset 36, 204
 - ColumnLabelPlacement 35, 204
 - ColumnSelection 204
 - ColumnTrigger 126, 129
 - com.klg.jclass.table.beans.LiveTable 214
 - com.klg.jclass.table.CellStyleModel 212
 - com.klg.jclass.table.db.datasource.DSdbTable 217
 - com.klg.jclass.table.db.jbuilder.JBdbTable 215
 - com.klg.jclass.table.JCTable 203
 - ComponentBorderWidth 38, 204
 - Cursor 205
 - data 214
 - dataBinding 217
 - dataSet 216
 - Editable 213
 - EditHeightPolicy 42, 205, 214, 216–217
 - EditWidthPolicy 42, 205, 214, 216–217
 - focusColor 205, 214, 216–217
 - focusIndicator 205, 214, 216–217
 - Font 205, 213, 228
 - Foreground 205, 213
 - FrameBorder 38, 205
 - frameBorderType 160, 214, 216–217
 - frameBorderWidth 38, 206, 214, 216–217
 - frozenCellLayout 214, 216–217
 - FrozenColumnPlacement 206
 - FrozenColumns 206
 - FrozenRowPlacement 206

- FrozenRows 41, 206
- getting 28
- global 32
- HorizontalAlignment 213
- HorizSBAttachment 206
- HorizSBDisplay 206
- HorizSBOffset 206
- HorizSBPosition 207
- HorizSBTrack 207
- HorizSBTrackRow 207
- JumpScroll 207, 214
- labelLayout 214, 216–217
- leftColumn 34, 207, 214, 216–217
- marginHeight 37, 207, 214, 216, 218
- marginWidth 37, 207, 214, 216, 218
- MaxHeight 208
- MaxWidth 208
- minCellVisibility 208, 214, 216, 218
- MinHeight 208
- MinWidth 208
- NumColumns 40, 62
- NumRows 40, 62
- PixelHeight 20, 44, 208
 - using to hide rows 46
- PixelWidth 20, 44, 208
 - using to hide columns 46
- popupMenuEnabled 208, 214, 216, 218
- RepaintEnabled 208
- RepeatBackground 213
- RepeatBackgroundColors 213
- RepeatForeground 213
- RepeatForegroundColors 213
- ResizeByLabelsOnly 23, 119
- resizeEven 119, 208, 214, 216, 218
- resizeInteractive 209, 215–216, 218
- row height 43
- RowHidden 47, 209
- RowLabelDisplay 209
- RowLabelOffset 36, 209
- RowLabelPlacement 35, 209
- RowSelection 209
- RowTrigger 126
- sBLayout 215–216, 218
- selectedBackground 34, 209, 215–216, 218
- SelectedBackgroundMode 209
- SelectedCells 209
- selectedForeground 34, 210, 215–216, 218
- SelectedForegroundMode 210
- selectIncludeLabels 209, 215–216, 218
- SelectionModel 210
- selectionPolicy 22, 210, 215–216, 218
- SeriesDataSorted 210
- setting 28
- setting cell styles 31
- setting for a cell 29
- setting for a cell range 32

- setting for a range 30
- setting for all cells 30
- setting for all columns 29–30
- setting for all labels 30
- setting for all rows 29
- setting for cells and labels 31
- setting for column 29
- setting for entire column 30
- setting for entire row 30
- setting for entire table 30
- setting for labels 29–30
- setting for range of cells 29
- setting in the API 13
- SortBy Column 127
- spannedCells 215–216
- stock data source 66
- StoreImageEnabled 210
- styles 215–216, 218
- summary of 203
- swingDataModel 215, 217–218
- topRow 34, 210, 215, 217–218
- TrackBackground 210
- trackCursor 210, 215, 217–218
- TrackForeground 210
- TrackSize 211
- Traversable 213
- traverseCycle 211, 215, 217–218
- useDatasourceEditable 218
- VariableEstimateCount 211
- VerticalAlignment 213
- VertSBAttachment 211
- VertSBDisplay 211
- VertSBOffset 211
- VertSBPosition 211
- VertSBTrack 212
- VertSBTrackColumn 212
- VisibleColumns 40–41, 212
- VisibleRows 40–41, 212
- property
 - definition 159
 - difference between JClass LiveTable Beans 194
 - LiveTable properties 164
 - setting in Java IDE 160
 - setting using API 160
- property editors
 - LiveTable 161

Q

- QueryExpressionList 112
- Quest Software technical support
 - contacting 5

R

- ranges
 - in cell selection 125
 - referencing 30
 - selected 125
 - used in cell spanning 59
- references
 - expression 114
- registry
 - JClass central 113
- removing cell selections 126
- renderers 83–84
 - component based 87
 - default 85
 - editors 28
 - subclassing 85
 - unmap 84
 - writing 85
- rendering cells 80–81
- RepaintEnabled property 208
- RepeatBackground property 213
- RepeatBackgroundColors property 213
- RepeatForegroundColor property 213
- RepeatForegroundColors property 213
- repeating colors 52
- reset 78
- resetSortedRows() method 129
- ResizeByLabelsOnly property 119
 - tutorial 23
- ResizeEven property 119, 208
- resizeEven property 214, 216, 218
- ResizeInteractive property 209
- resizeInteractive property 215–216, 218
- resizing 138
 - columns 118
 - default behavior 118
 - disabling 118
 - events and listeners 138
 - pixel width 118
 - preset styles 32
 - rows 118
 - rows and columns 118–119
 - using labels only 119
 - using labels, tutorial 23
- results 105
- RGB color values 223
- RowHidden property 47, 209
- RowLabelDisplay property 209
- RowLabelOffset property 36, 209
- RowLabelPlacement property 35, 209
- rows
 - adding 67, 78
 - controlling resizing 118
 - default resizing behavior 118
 - definition

- columns
 - definition 40
- deleting 68
- determining number 40
- disallowing resizing 118
- displaying 41
- dragging 126
- freezing 41–42
- height
 - pixel value 43
- height property 43
- height, setting 43
- hiding 46
- labels 35, 62
- labels, placement 35
- making visible 121
- moving 68
- placement of frozen 42
- referencing, all 29–30
- referencing, entire 30
- referencing, one 29
- removing 78
- resizing 118
- resizing all at once 119
- resizing with labels 119
- selecting labels 125
- set as top 34
- setting the number 66
- specifying labels 66
- swapping 41
- visible, getting 40
- visible, setting 40

RowSelection property 209

RowTrigger property

- and dragging 126

runtime

- cell selection 126

S

- sBLayOut property 169, 215–216, 218
- scroll listener methods 122
- scrollbars 33
 - attaching 33
 - component 34
 - definition 28
 - disabling interactive 121
 - display 34
 - force scrolling by an application 121
 - jump scrolling 120
 - options 34
 - positioning 33
 - programming 120
 - tracking 122
- scrolling 120, 141

- automatic 120
- disabling interactive 121
- forcing 121
- jump scrolling 120
- listener 122
- managing 120
- mouse wheel support 121
- tracking scrollbars 122
- selected cells
 - list 125
- selectedBackground property 34, 171, 209, 215–216, 218
- SelectedBackgroundMode property 209
- SelectedCells property 209
- selectedForeground property 34, 171, 210, 215–216, 218
- SelectedForegroundMode property 210
- selectIncludeLabels property 172, 209, 215–216, 218
- selection
 - cells 144
 - cells, default 123
 - colors 34, 124
 - setting 34
 - customizing 124
 - enabling cell selection, tutorial 22
- SelectionModel property 210
- selectionPolicy property 124, 172, 210, 215–216, 218
 - tutorial 22
- SeriesDataSorted property 210
- set method 203
- setAutoScroll() method 120
- setDataFormat
 - method in MathValue 105
- setEditable method 63
- setting
 - cell renderers, for a series 82
 - properties in an IDE 32
 - scrollbar options 34
- setValueAt
 - method in MathMatrix 108
 - method in MathVector 107
- SortableDataViewModel 61
- SortByColumn property 127
- sorting
 - and ColumnTrigger property 129
 - columns 127
 - columns, tutorial 24
 - events and listeners 147
 - frozen columns 127
 - multiple columns 128
 - resetting 129
 - SortByColumn property 127
- source parameter 68
- SpanHandler 59
- spannedCells property 172, 215–216
- spanning

- cells 58
 - create multiline headers 60
 - using JCCellRange 59
- StaticDataSource 73
- stock data sources
 - using 63
- StoreImageEnabled property 210
- storing data 64–65
- styles
 - preset 32
 - property 173, 215–216, 218
- subclassing
 - cell editors 94
 - cells
 - renderers 85
 - summary of properties 203
- support 5
 - contacting 5
 - FAQs 5
- swapColumns method 41
- swapping rows and columns 41
- swapRows method 41
- Swing TableModel class 71
- Swing, using TableModel data objects 66
- swingDataModel property 215, 217–218

T

- table
 - basic 10
 - frame border 38
 - preset styles 32
 - printing 137
 - referencing, entire 30
 - resize events 138
 - resizing 138
 - scrolling 141
 - size defined by data source 62
 - sorting 147
- table anatomy
 - cell 28
 - current cell 28
 - current context 29
 - label 28
 - renderers 28
 - scrollbars 28
- table context 29
- table frame
 - border 162, 172
- table layout
 - examples 234
- table scrolling
 - attaching scrollbars 33
 - default 120
 - different component 34

- setting options 34
- Table.isRowVisible() 121
- TableDataEvent 74
- TableDataItem 62
- TableDataListener 74
- TableDataModel interface 62
- TableDataView 62
- TableExpressionList 112
- TableListenerPropagator 112
- TableModel, using in table 66
- TableSwingDataSource 66
- technical support 5
 - contacting 5
 - FAQs 5
- text
 - alignment in cell 53
 - clipping 57
- topRow property 34, 121, 174, 210, 215, 217–218
- toString
 - method in MathMatrix 108
 - method in MathScalar 106
 - method in MathVector 107
- TrackBackground property 210
- trackCursor property 130, 210, 215, 217–218
- TrackForeground property 210
- tracking
 - cursor type 33
 - mouse pointers, disabling 130
 - scrollbars 122
- TrackSize property 211
- Traversable property 213
- traversal
 - cell 115
 - customizing cell 115
 - default 115
 - events 149, 151
 - forcing 116
 - interactive 117
 - preset styles 32
- traverseCycle property 211, 215, 217–218
- trigger 24
- tutorial
 - adding color to an individual cell 18
 - adding interactivity 21
 - background colors 15
 - basic table 10
 - cell borders 19
 - cell selection 22
 - cell size 20
 - cell spacing 19
 - cell styles 15
 - changing alignment 16
 - clip arrows 12
 - fonts, setting 17
 - foreground color 15
 - improving table appearance 13

- making a table editable 21
- PixelHeight property 20
- PixelWidth property 20
- resize using labels 23
- ResizeByLabelsOnly property 23
- resizing cells 12
- SelectionPolicy property 22
- setting a data source 11
- setting colors 15
- setting properties in the API 13
- sorting columns 24
- table appearance 12
- table changes 12
- typographical conventions 2

U

- useDatasourceEditable property 218
- user interactivity 115

V

- values
 - change 76
 - parameter 67
- variable cell size 20
- VariableEstimateCount property 211
- vectorValue
 - method in MathMatrix 108
 - method in MathScalar 106
 - method in MathValue 105
 - method in MathVector 107
- VerticalAlignment 58
- VerticalAlignment property 213
- VertSBAttachment property 211
- VertSBDisplay 121
- VertSBDisplay property 211
- VertSBOffset property 211
- VertSBPosition property 211
- VertSBTrack property 212
- VertSBTrackColumn property 212
- visibility
 - cells 116, 121
 - columns 121
 - forcing 121
- VisibleColumns property 40–41, 212
- VisibleRows property 40–41, 212

W

- writing a cell renderer 85

X

XML

- examples 70
- in JClass 70
- interpreter 70
- loading data 69
- primer 69
- Swing TableModel class 71
- tags 71