



LiveTable Programmer’s Guide

© 2009 Quest Software, Inc. ALL RIGHTS RESERVED.

This guide contains proprietary information 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.

If you have any questions regarding your potential use of this material, contact:

Quest Software World Headquarters LEGAL Dept

5 Polaris Way

Aliso Viejo, CA 92656 Email: [legal@quest.com](mailto:legal@quest.com)

Refer to our Web site (www.quest.com) for regional and international office information.

Trademarks

Quest, Quest Software, the Quest Software logo, Aelita, Akonix, AppAssure, Benchmark Factory, Big Brother, ChangeAuditor, DataFactory, DeployDirector, ERDisk, Foglight, Funnel Web, GPOAdmin, iToken, I/Watch, Imceda, InLook, IntelliProfile, InTrust, Invertus, IT Dad, I/Watch, JClass, Jint, JProbe, LeccoTech, LiteSpeed, LiveReorg, MessageStats, NBSpool, NetBase, Npulse, NetPro, PassGo, PerformaSure, Quest Central, SharePlex, Sitraka, SmartAlarm, Spotlight, SQL LiteSpeed, SQL Navigator, SQL Watch, SQLab, Stat, StealthCollect, Tag and Follow, Toad, T.O.A.D., Toad World, vAnalyzer, vAutomator, vControl, vConverter, vDupe, vEssentials, vFoglight, vMigrator, vOptimizer Pro, VPackager, vRanger, vRanger Pro, vReplicator, vSpotlight, vToad, Vintela, Virtual DBA, VizionCore, Vizioncore vAutomation Suite, Vizioncore vEssentials, Xaffire, and XRT are trademarks and registered trademarks of Quest Software, Inc in the United States of America and other countries. Other trademarks and registered trademarks used in this guide are property of their respective owners.

Disclaimer

The information in this document is provided in connection with Quest products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Quest products. EXCEPT AS SET FORTH IN QUEST'S TERMS AND CONDITIONS AS SPECIFIED IN THE LICENSE AGREEMENT FOR THIS PRODUCT, QUEST ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL QUEST BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF PROFITS, BUSINESS INTERRUPTION OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF QUEST HAS

BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Quest makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and product descriptions at any time without notice. Quest does not make any commitment to update the information contained in this document.

Third Party Contributions

See Third\_Party\_Contributions.htm in your Quest Software License Server installation directory.

LiveTable Programmer’s Guide January 2009

**Version 6.4.2**

**Table of Contents**

###### [Preface 9](#_bookmark0)

[Introducing JClass LiveTable 9](#_bookmark1)

[Assumptions 10](#_bookmark4)

[Typographical Conventions 10](#_bookmark5)

[Overview of this Guide 10](#_bookmark7)

[API Documentation (Javadoc) 11](#_bookmark8)

[Licensing 11](#_bookmark10)

[Related Documents 12](#_bookmark11)

[About Quest Software, Inc. 12](#_bookmark12)

[JClass Community 13](#_bookmark14)

##### [Part](#_bookmark18) [I: Using JClass LiveTable](#_bookmark16)

###### [‘Hello Table’ –](#_bookmark19)

###### [JClass LiveTable Tutorial 17](#_bookmark21)

* 1. [The Basic Table 18](#_bookmark22)
  2. [Overview of Table Changes 20](#_bookmark29)
  3. [Improving the Table’s Appearance 21](#_bookmark31)

[Adding and Formatting Labels 21](#_bookmark34)

[Introduction to Cell Styles 22](#_bookmark37)

[Changing Foreground and Background Colors 23](#_bookmark41)

[Changing Alignment 24](#_bookmark45)

[Changing the Fonts 24](#_bookmark47)

[Adding Color to an Individual Cell 26](#_bookmark52)

[Changing the Cell Borders and Spacing 27](#_bookmark54)

[Displaying More of the Cells 28](#_bookmark56)

* 1. [Adding Interactivity 29](#_bookmark59)

[Making the Cells Editable 29](#_bookmark61)

[Enabling Cell Selection 30](#_bookmark63)

[Resizing Using Labels Only 30](#_bookmark65)

[Enabling Column Sorting 31](#_bookmark68)

* 1. [Proceeding from Here 32](#_bookmark71)
  2. [Internationalization 33](#_bookmark72)

###### [Building a Table 35](#_bookmark73)

* 1. [Table Anatomy 101 35](#_bookmark78)
  2. [Setting and Getting Properties 36](#_bookmark84)

[Table Contexts 37](#_bookmark86)

[Setting Table Properties with Java Code 39](#_bookmark97)

[Setting Properties with a Java IDE at Design-Time 40](#_bookmark101)

* 1. [Preset Table Styles 40](#_bookmark103)
  2. [Global Table Properties 40](#_bookmark110)

[Focus Rectangle Appearance 41](#_bookmark112)

[Screen Cursor Type 41](#_bookmark115)

[Scrollbars 41](#_bookmark117)

[Cell Selection Colors 42](#_bookmark129)

[Row and Column Labels 43](#_bookmark135)

[Cell and Label Border Width 45](#_bookmark143)

[Cell and Label Margins 45](#_bookmark146)

[Component Borders 46](#_bookmark150)

[Frame Border Attributes 46](#_bookmark153)

[Row and Column Definition 48](#_bookmark157)

[Controlling Cell Editor Size 50](#_bookmark172)

* 1. [Column Width and Row Height Properties 51](#_bookmark174)

[Character Height and Width 51](#_bookmark180)

[Absolute Pixel Height and Width 52](#_bookmark182)

[Variable Pixel Height and Width 53](#_bookmark184)

[Maximum and Minimum Pixel Height and Width 54](#_bookmark188)

[Displaying and Editing Multiple Lines in Cells 54](#_bookmark190)

[Using Row Height and Width to Hide Rows and Columns 54](#_bookmark193)

* 1. [Cell Styles 55](#_bookmark197)

[Cell Style Properties and Implementation 55](#_bookmark200)

[Defining Your Own or Changing Built-In Cell Styles 56](#_bookmark203)

[Using and Modifying JClass LiveTable’s Built-In Styles 58](#_bookmark208)

[Working with Colors 60](#_bookmark213)

[Text and Image Alignment 61](#_bookmark218)

[Cell and Label Fonts 62](#_bookmark221)

[Border Types 62](#_bookmark223)

[Cell and Label Border Sides 65](#_bookmark228)

[Text and Image Clipping 65](#_bookmark230)

[Displaying Images in Table Cells 66](#_bookmark233)

* 1. [Cell and Label Spanning 66](#_bookmark243)

[Using Spanning to Create Multiline Headers 68](#_bookmark247)

###### [Working with Table Data 69](#_bookmark249)

* 1. [Overview: Data Handling in JClass LiveTable . .](#_bookmark252) [How the Table and Data Source Communicate .](#_bookmark256)

|  |  |
| --- | --- |
| [. . . .](#_bookmark252) | [. 69](#_bookmark252) |
| [. . . .](#_bookmark256) | [. 70](#_bookmark256) |
| [. . . .](#_bookmark260) | [. 70](#_bookmark260) |
| [. . . .](#_bookmark271) | [. 71](#_bookmark271) |
| [. . . .](#_bookmark275) | [. 71](#_bookmark275) |
| . . | . 72 |
| [. . . .](#_bookmark281) | [. 73](#_bookmark281) |
| [. . . .](#_bookmark285) | [. 73](#_bookmark285) |
| [. . . .](#_bookmark287) | [. 73](#_bookmark287) |
| [. . . .](#_bookmark290) | [. 74](#_bookmark290) |
| [. . . .](#_bookmark294) | [. 74](#_bookmark294) |
| [. . . .](#_bookmark296) | [. 74](#_bookmark296) |
| [. . . .](#_bookmark309) | [. 77](#_bookmark309) |
| [. . . .](#_bookmark312) | [. 77](#_bookmark312) |
| [. . . .](#_bookmark314) | [. 77](#_bookmark314) |
| [. . . .](#_bookmark316) | [. 78](#_bookmark316) |
| [. . . .](#_bookmark320) | [. 79](#_bookmark320) |
| [. . . .](#_bookmark321) | [. 79](#_bookmark321) |
| [. . . .](#_bookmark323) | [. 80](#_bookmark323) |
| [. . . .](#_bookmark325) | [. 80](#_bookmark325) |
| [. . . .](#_bookmark329) | [. 82](#_bookmark329) |
| [. . . .](#_bookmark336) | [. 86](#_bookmark336) |

* 1. [Getting Data into your Table . . . . . . . . .](#_bookmark260)

[Making the Data Source Editable . . . . . .](#_bookmark271)

* 1. [Using Stock Data Sources . . . . . . . . . .](#_bookmark275)

[JCVectorDataSource: the Data Source Workhorse](#_bookmark278) [Getting Data from an Input Stream . . . . .](#_bookmark281)

[Getting Data from a Database . . . . . . . .](#_bookmark285)

[Caching Data with JCCachedDataSource . . .](#_bookmark287)

[Using Swing TableModel Data Objects . . . .](#_bookmark290)

* 1. [Setting Stock Data Source Properties . . . . . .](#_bookmark294)

[Working with Rows and Columns . . . . . .](#_bookmark296)

[Working with Other Properties . . . . . . .](#_bookmark309)

* 1. [Loading Data from an XML Source . . . . . .](#_bookmark312)

[XML Primer . . . . . . . . . . . . . .](#_bookmark314)

[Using XML in JClass . . . . . . . . . . .](#_bookmark316)

[Example XML Files for JClass LiveTable . . .](#_bookmark320)

[Tags . . . . . . . . . . . . . . . . . .](#_bookmark321)

[Creating a Swing TableModel class . . . . .](#_bookmark323)

* 1. [Creating your own Data Sources . . . . . . . .](#_bookmark325)
  2. [Dynamically Updating Data . . . . . . . . .](#_bookmark329)

[Adding and Removing Columns and Rows . .](#_bookmark336)

###### [Displaying and Editing Cells 87](#_bookmark339)

* 1. [Overview 87](#_bookmark343)
  2. [Default Cell Rendering and Editing 88](#_bookmark348)
  3. [Rendering Cells 89](#_bookmark351)

[JClass Cell Renderers 89](#_bookmark353)

[Setting a Cell Renderer for a Series 91](#_bookmark354)

[Mapping a Data Type to a Cell Renderer 91](#_bookmark356)

[Creating your own Cell Renderers 92](#_bookmark362)

* 1. [Editing Cells 97](#_bookmark368)

[Default Cell Editors 97](#_bookmark370)

[Setting a Cell Editor for a Series 99](#_bookmark373)

[Mapping a Data Type to a Cell Editor 99](#_bookmark375)

[Creating Your Own Cell Editors 100](#_bookmark377)

* 1. [The JCCellInfo Interface 107](#_bookmark384)

1. [Adding Formulas to JClass LiveTable 109](#_bookmark386)
   1. [Introduction 109](#_bookmark389)
   2. [com.klg.jclass.util.formulae’s Hierarchy 109](#_bookmark394)
   3. [Expressions and Results 111](#_bookmark397)
   4. [Math Values 111](#_bookmark400)

[MathScalar 112](#_bookmark410)

[MathVector 112](#_bookmark418)

[MathMatrix 113](#_bookmark428)

* 1. [Operations 114](#_bookmark438)

[The Defined Mathematical Operations 115](#_bookmark442)

[Reducing Operations to Values 117](#_bookmark468)

* 1. [Expression Lists 118](#_bookmark470)
  2. [Events and Listeners 119](#_bookmark475)
  3. [Exceptions 119](#_bookmark478)
  4. [Using Formulae in JClass LiveTable 119](#_bookmark482)

[Registering a Cell Editor and a Cell Renderer with the JClass Central Registry 119](#_bookmark484)

[Performing a Mathematical Operation on a Range of Cells . 120](#_bookmark486)

1. [Programming User Interactivity 123](#_bookmark489)
   1. [Cell Traversal 123](#_bookmark493)

[Default Cell Traversal 123](#_bookmark495)

[Customizing Cell Traversal 123](#_bookmark497)

[Minimum Cell Visibility 124](#_bookmark499)

[Forcing Traversal 124](#_bookmark501)

[Controlling Interactive Traversal 125](#_bookmark503)

* 1. [Resizing Rows and Columns 126](#_bookmark506)

[Default Resizing Behavior 126](#_bookmark508)

[Disallowing Cell Resizing 126](#_bookmark514)

[Controlling Resizing 126](#_bookmark517)

* 1. [Table Scrolling 128](#_bookmark523)

[Default Scrolling Behavior 128](#_bookmark526)

[Managing Table Scrolling 128](#_bookmark528)

[Scroll Listener Methods 130](#_bookmark545)

* 1. [Cell Selection 131](#_bookmark549)

[Default Cell Selection 131](#_bookmark551)

[Selection Colors 132](#_bookmark553)

[Customizing Cell Selection 132](#_bookmark555)

[Selected Cell List 133](#_bookmark560)

[Working with Selected Ranges 133](#_bookmark562)

[Removing Selections 134](#_bookmark564)

[Runtime Selection Control 134](#_bookmark566)

* 1. [Dragging Rows and Columns 134](#_bookmark568)
  2. [Sorting Columns 135](#_bookmark570)

[Sort by Clicking on a Column Label 137](#_bookmark575)

[Resetting the Table after Sorting 137](#_bookmark577)

* 1. [Custom Mouse Pointers 137](#_bookmark580)

1. [Events and Listeners 139](#_bookmark585)
   1. [Displaying Cells 139](#_bookmark589)
   2. [Editing Cells 142](#_bookmark596)
   3. [Painting Tables 144](#_bookmark599)
   4. [Printing Tables 145](#_bookmark603)
   5. [Resizing Cells 146](#_bookmark607)
   6. [Scrolling in Tables 149](#_bookmark613)
   7. [Selecting Cells 153](#_bookmark619)
   8. [Sorting Table Data 155](#_bookmark622)
   9. [Table Data Changes 158](#_bookmark626)
   10. [Traversing Cells 159](#_bookmark630)
2. [Table Printing 163](#_bookmark634)
   1. [Printing 163](#_bookmark637)

[Setting Page Layout Properties 163](#_bookmark640)

[Printing Headers and Footers 164](#_bookmark646)

* 1. [Print Preview 165](#_bookmark648)

1. [JClass LiveTable Beans and IDEs 167](#_bookmark652)
   1. [An Introduction to JavaBeans 167](#_bookmark658)

[Properties 167](#_bookmark660)

[Setting Properties in a Java IDE at Design-Time 168](#_bookmark664)

[Setting Properties using Methods in the API 168](#_bookmark666)

* 1. [JClass LiveTable JavaBean 168](#_bookmark668)
  2. [Setting Properties for the LiveTable Bean 168](#_bookmark669)

[JClass LiveTable Property Editors 169](#_bookmark671)

[LiveTable Properties 171](#_bookmark677)

* 1. [Tutorial: Building a Table in an IDE 181](#_bookmark708)

[The Basic Table 182](#_bookmark710)

[Improving the Table’s Appearance 183](#_bookmark712)

[Adding Interactivity 187](#_bookmark716)

[The Final Program 189](#_bookmark718)

[Part](#_bookmark721) [II: Reference Appendices](#_bookmark719)

1. [Event Summary 193](#_bookmark722)
2. [JClass LiveTable Property Listing 197](#_bookmark737)
   1. [Properties of com.klg.jclass.table.JCTable 198](#_bookmark743)
   2. [Properties of com.klg.jclass.table.CellStyleModel 208](#_bookmark823)
   3. [Properties of com.klg.jclass.table.beans.LiveTable 210](#_bookmark841)
3. [Colors and Fonts 213](#_bookmark878)
   1. [Colorname Values 213](#_bookmark882)
   2. [RGB Color Values 213](#_bookmark884)
   3. [Fonts 218](#_bookmark887)
4. [JClass LiveTable Inheritance Hierarchy 219](#_bookmark892)
5. [Distributing Applets and Applications 221](#_bookmark898)
6. [Overview of Examples and Demos 223](#_bookmark904)
   1. [JClass LiveTable Examples 223](#_bookmark908)
   2. [JClass LiveTable Demos 228](#_bookmark918)

[Index 231](#_bookmark927)

#### Preface

[*Introducing JClass LiveTable*](#_bookmark1)■ [*Assumptions*](#_bookmark4)■ [*Typographical Conventions*](#_bookmark5)[*Overview of this Guide*](#_bookmark7)■ [*API Documentation (Javadoc)*](#_bookmark8)■ [*Licensing*](#_bookmark10)[*Related Documents*](#_bookmark11)■ [*About Quest Software, Inc.*](#_bookmark12)

###### 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](#_bookmark11) later in this section of the manual for additional sources of Java-related information.

###### Typographical Conventions

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 this Guide

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

[Chapter 1, ‘Hello Table’ – JClass LiveTable Tutorial,](#_bookmark20) provides a tutorial exercise to familiarize new users with the basics of writing a JClass LiveTable program.

[Chapter 2, Building a Table,](#_bookmark74) explains how to set most JClass LiveTable properties to customize the appearance and display of JClass LiveTable applications.

[Chapter 3, Working with Table Data,](#_bookmark250) 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,](#_bookmark340) describes how to configure JClass LiveTable so users can edit cells of any data type.

[Chapter 5, Adding Formulas to JClass LiveTable,](#_bookmark387) outlines the formulae package in

*com.klg.jclass.util*, which has special capabilities for working with mathematical objects.

[Chapter 6, Programming User Interactivity,](#_bookmark490) explains how to control how users interact with your table application, including cell traversal, selection, sorting, and more.

[Chapter 7, Events and Listeners,](#_bookmark586) explains how to send events and register event listeners in your JClass LiveTable programs.

[Chapter 8, Table Printing,](#_bookmark635) describes the enhanced printing features of JClass LiveTable.

[Chapter 9, JClass LiveTable Beans and IDEs,](#_bookmark653) 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](#_bookmark723), lists events and corresponding event listeners.

[Appendix B, JClass LiveTable Property Listing](#_bookmark738), is a quick reference to properties, their functions, and settable values.

[Appendix C, Colors and Fonts](#_bookmark879), 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 D, JClass LiveTable Inheritance Hierarchy](#_bookmark893), summarizes the

com.klg.jclass.table package.

[Appendix E, Distributing Applets and Applications](#_bookmark899), is a quick tutorial that demonstrates how to take a completed Java applet and deploy it on a Web page and Web server.

[Appendix F, Overview of Examples and Demos](#_bookmark905), 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 Documentation (Javadoc)

The *JClass DesktopViews API 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 *JClass DesktopViews 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 Platform Documentation*” at [*http://java.sun.com/docs/index.html*](http://java.sun.com/docs/index.html) and the “*Java Tutorial*” at [*http://java.sun.com/docs/books/tutorial/index.html*](http://java.sun.com/docs/books/tutorial/index.html) from Sun Microsystems
* For an introduction to creating enhanced user interfaces, see “*Creating a GUI with JFC/Swing*” at [*http://java.sun.com/docs/books/tutorial/uiswing/index.html*](http://java.sun.com/docs/books/tutorial/uiswing/index.html)
* *Java in a Nutshell, 2nd Edition* from O’Reilly & Associates Inc. See the O’Reilly Java Resource Center at [*http://java.oreilly.com*](http://java.oreilly.com/).
* Resources for using JavaBeans are at [*http://java.sun.com/beans/resources.html*](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 Software, Inc.

Quest Software, Inc. delivers innovative products that help organizations get more performance and productivity from their applications, databases and Windows infrastructure. Through a deep expertise in IT operations and a continued focus on what works best, Quest helps more than 18,000 customers worldwide meet higher expectations for enterprise IT. Quest Software can be found in offices around the globe and at [*www.quest.com*](http://www.quest.com/).

Contacting Quest Software

|  |  |
| --- | --- |
| **Email** | [*info@quest.com*](mailto:info@quest.com) |
| **Mail** | Quest Software, Inc. World Headquarters 5 Polaris Way  Aliso Viejo, CA 92656 USA |
| **Web site** | [*www.quest.com*](http://www.quest.com/) |

Refer to our web site for regional and international office information.

Contacting Quest Support

Quest Support is available to customers who have a trial version of a Quest product or who have purchased a commercial version and have a valid maintenance contract. Quest Support provides around the clock coverage with SupportLink, our web self-service. Visit SupportLink at: [*http://support.quest.com*](http://support.quest.com/)

From SupportLink, you can do the following:

* + Quickly find thousands of solutions (Knowledgebase articles/documents).
  + Download patches and upgrades.
  + Seek help from a Support engineer.
  + Log and update your case, and check its status.

View the *Global Support Guide* for a detailed explanation of support programs, online services, contact information, and policy and procedures. The guide is available at: [*http://support.quest.com/pdfs/Global Support Guide.pdf*](http://support.quest.com/pdfs/Global%20Support%20Guide.pdf)

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

**JClass Community**

For the latest product information, helpful resources, and discussions with theJClass Quest team and other community members, join the JClass community at [http://jclass.inside.quest.com/.](http://jclass.inside.quest.com/)

## Part I Using JClass

*LiveTable*

***1***

#### ‘Hello Table’ – JClass LiveTable Tutorial

[*The Basic Table*](#_bookmark22)■ [*Overview of Table Changes*](#_bookmark29)■ [*Improving the Table’s Appearance*](#_bookmark31)[*Adding Interactivity*](#_bookmark59)■ [*Proceeding from Here*](#_bookmark71)■ [*Internationalization*](#_bookmark72)

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.

17

* 1. **The Basic T****able**

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](#_bookmark276) 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);

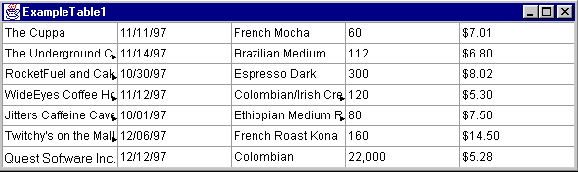
Once the data source is set to the JCVectorDataSource (ds) object, that object handles the data, including setting the number of rows and columns, and accessing the cell values.

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:

* + 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



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. **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 | *examples/table/intro/ExampleTable2.java* |
| label colors | *examples/table/intro/ExampleTable3.java* |
| label text alignment  label font | *examples/table/intro/ExampleTable4.java* |
| color of an individual non-label cell  cell and frame borders and spacing | *examples/table/intro/ExampleTable5.java* |

|  |  |
| --- | --- |
| **Change to ExampleTable1.java** | **Example File that Encompasses Changes Made** |
| cell height and width  enabling cell editing | *examples/table/intro/ExampleTable6.java* |
| enabling cell selection  resize only with labels | *examples/table/intro/ExampleTable7.java* |
| enable column sorting | *examples/table/intro/ExampleTable8.java* |

###### 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. 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:

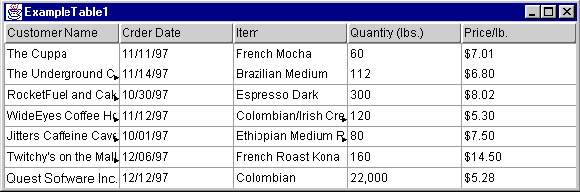
table.setColumnLabelDisplay(true);

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:



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](#_bookmark59)). However, in this case, the labels do not perform any interactive function.

* + 1. 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](#_bookmark74) 2.

* + 1. 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;

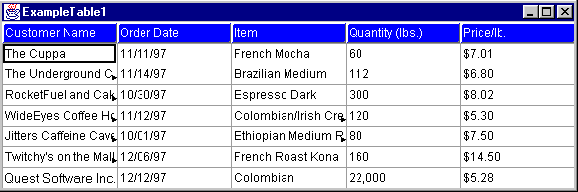
Since you are using default styles in the examples, you need to import the interface with which you implement the style. To do this, import the CellStyleModel class by adding this line to your list of import statements:

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:



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

* + 1. 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. 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:

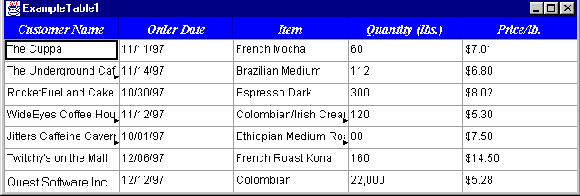
import java.awt.Font;

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:



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. 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:

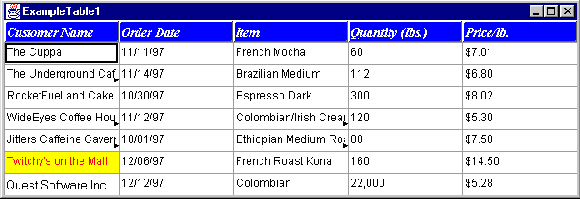
CellStyleModel specialStyle = new JCCellStyle((JCCellStyle)cellStyle);

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:



* + 1. 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](#_bookmark74) 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;

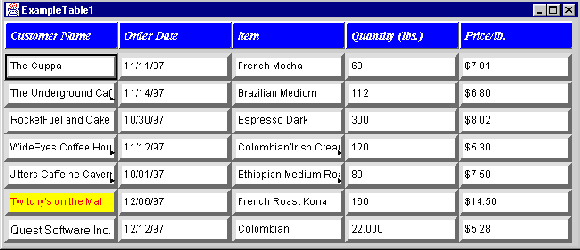
Next, add these lines to the ExampleTable1 class:

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:



* + 1. 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,

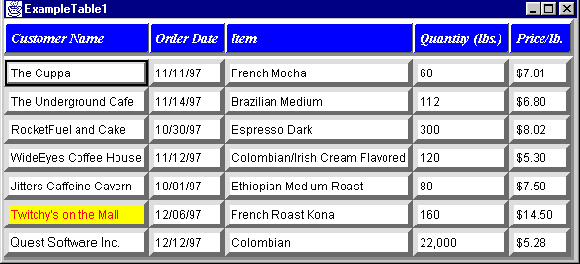
JClass LiveTable sets all of the cells to a width of 10 characters and a height of one character. You could specify the height and width of the cells in rows and columns in terms of lines and characters using the CharHeight and CharWidth properties. However, in this program we want the cells to size 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:



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](#_bookmark175) 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.

###### 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](#_bookmark490) 6.

* + 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

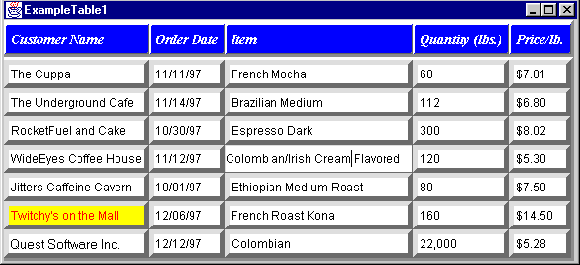
The modified *ExampleTable1.java* currently contains the lines:

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:



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

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](#_bookmark340) 4.

* + 1. 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](#_bookmark556) 6, for more information.

* + 1. 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.

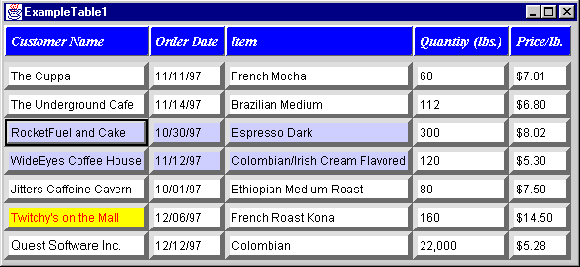
JClass LiveTable provides the AllowResizeBy property to enable this feature.

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.



*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. 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*:

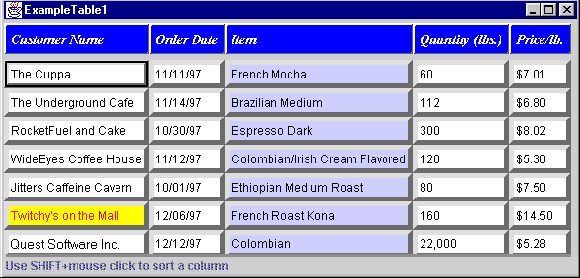
import com.klg.jclass.table.MouseActionInitiator; import java.awt.event.InputEvent;

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.





*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.

###### 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](#_bookmark74) 2. [Appendix B](#_bookmark739), [JClass LiveTable Property Listing](#_bookmark738), 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](#_bookmark250) 3, and [Displaying and Editing Cells, in Chapter](#_bookmark340) 4.
* To learn about the formulae package in *com.klg.jclass.util*, which has special capabilities for working with mathematical objects, see [Adding Formulas to JClass LiveTable, in Chapter](#_bookmark387) 5.
  + To learn about user-interaction with JClass LiveTable, see [Programming User Interactivity, in Chapter](#_bookmark490) 6.
  + To try this same tutorial in a JavaBeans development environment, see [JClass LiveTable Beans and IDEs, in Chapter](#_bookmark653) 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.

###### Internationalization

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

*Localization* is the process of making internationalized software run appropriately in a particular environment.

In JClass DesktopViews, all Strings that may be seen by a typical user have been internationalized and are ready for localization. These Strings are in resource bundles in every package that requires them. You need to create additional resource bundles for each of the locales that you want to support.

Note: Localizations that are built into the Java platform – such as number and date formatting – are handled by JClass LiveTable, without the need for you to do any extra work.

To localize your JClass LiveTable, you need the JClass LiveTable source code (requires a source code license). The packages that require localization have a *resources* subdirectory that contains the resource bundles, called *LocaleInfo* (or some similar variation, such as *LocaleBeanInfo*). You may want to perform an automated search of the package structure to find all the resource bundles.

To create a new resource bundle, copy the *LocaleInfo.java* file (staying within the same *resources* directory) and change its name to include standard language and country identifiers for the locale that you want to support. For example, if you want to support French as spoken in France, rename the copy of *LocaleInfo.java* to *LocaleInfo\_fr\_FR.java.* You can then replace the Strings in the copied file with the French translations.

To use a localized resource bundle, you pass the language and country identifiers to the setLocale() method. For example, setLocale(new Locale(fr, FR)) means that the Strings will be read from *LocaleInfo\_fr\_FR.java*.

For more information, including standard language and country identifiers, see

[*http://java.sun.com/j2se/1.4.2/docs/guide/intl/index.html*](http://java.sun.com/j2se/1.4.2/docs/guide/intl/index.html).

# 2

#### Building a Table

[*Table Anatomy 101*](#_bookmark78)■ [*Setting and Getting Properties*](#_bookmark84)■ [*Preset Table Styles*](#_bookmark103)■ [*Global Table Properties*](#_bookmark110)[*Column Width and Row Height Properties*](#_bookmark174)■ [*Cell Styles*](#_bookmark197)■ [*Cell and Label Spanning*](#_bookmark243)

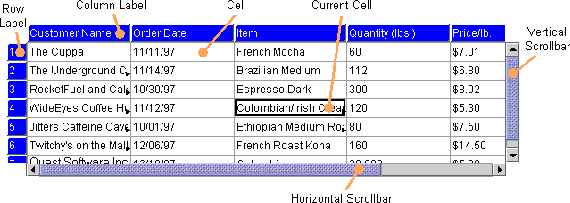
Using the JClass LiveTable API, you can customize the appearance of your tables with colors, borders, custom scrollbars, and other display properties. This chapter describes the properties you can set to define the structure and appearance of your tables. The properties are set for rows, columns, and cells. See [Appendix B, JClass LiveTable](#_bookmark738) [Property Listing](#_bookmark738), 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](#_bookmark250) 3, and [Displaying](#_bookmark340) [and Editing Cells, in Chapter](#_bookmark340) 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.

###### Table Anatomy 101

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



*Figure 4 The components of a Table.*

The following list defines the terminology used with JClass LiveTable:

35

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](#_bookmark135), 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](#_bookmark490) 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](#_bookmark340) [Chapter](#_bookmark340) 4, for more information on cell editors and renderers.

###### 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.

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

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

* + 1. 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](#_bookmark197), 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](#_bookmark197), later in this chapter.

* + 1. 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](#_bookmark653) 9, for complete details on using JClass LiveTable’s JavaBeans in IDEs.

###### 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.

###### 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.

* + 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

* + 1. 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.

* + 1. 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](#_bookmark524) 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](#_bookmark529) 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.

* + 1. 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](#_bookmark550) 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.
    1. 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](#_bookmark250) [Chapter](#_bookmark250) 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** |
| JCTableEnum.PLACE\_TOP JCTableEnum.PLACE\_RIGHT  The labels are displayed at the top and to the right of the table (default). |  |

|  |  |
| --- | --- |
| **Column and Row Placement** | **Example** |
| JCTableEnum.PLACE\_TOP JCTableEnum.PLACE\_LEFT  The labels are displayed at top and to the left of the table. |  |
| JCTableEnum.PLACE\_BOTTOM JCTableEnum.PLACE\_RIGHT  The labels are displayed at the bottom and to the right of table. |  |
| JCTableEnum.PLACE\_BOTTOM JCTableEnum.PLACE\_LEFT  The labels are displayed at the bottom and to the left of the table (reversed default). |  |

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); |  |

**Offset value examples**

|  |  |
| --- | --- |
| ColumnLabelOffset(15); RowLabelOffset(15); |  |
| ColumnLabelOffset(-10); RowLabelOffset(-10); |  |

* + 1. **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

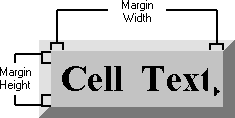
|  |  |
| --- | --- |
| table.setCellBorderWidth(2);  sets the bordercell width for all cells and labels to a value of two pixels |  |
| table.setCellBorderWidth(5);  sets the bordercell width for all cells and labels to a value of five pixels |  |

* + 1. **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

|  |  |
| --- | --- |
| table.setMarginHeight(2); table.setMarginWidth(5);  sets the margin height to 2 and the width to 5 |  |
| table.setMarginHeight(10); table.setMarginWidth(10);  sets the margin height and width to 10 |  |

* + 1. **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.

* + 1. 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 JCCellBorder(value)).

FrameBorder Attribute Examples

|  |  |
| --- | --- |
| JCTableEnum.BORDER\_ETCHED\_IN  creates a border that appears set in | JCTableEnum.BORDER\_ETCHED\_OUT  creates a raised border |
| JCTableEnum.BORDER\_FRAME\_IN  creates a frame border whose enclosed cells and labels appear set in | JCTableEnum.BORDER\_FRAME\_OUT creates a frame border whose enclosed cells and labels appear raised |
| JCTableEnum.BORDER\_IN  creates a border whose enclosed cells and labels appear set in | JCTableEnum.BORDER\_OUT  creates a border whose enclosed cells and labels appear raised |
| JCTableEnum.BORDER\_PLAIN  creates a plain frame border | JCTableEnum.BORDER\_THIN  creates a thin frame border |

**FrameBorder Attribute Examples**

|  |  |
| --- | --- |
| JCTableEnum.BORDER\_NONE  creates no frame border (default) |  |

* + 1. **Row and Column Definitio****n**

**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](#_bookmark295) [Properties, in Chapter](#_bookmark295) 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](#_bookmark166).

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

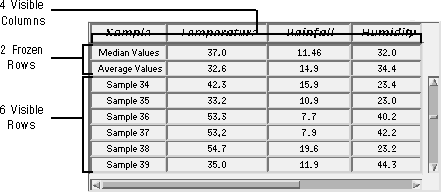
resetSwappedRows() or resetSwappedColumns() methods.

‘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](#_bookmark247) 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.



*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](#_bookmark276) 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 JCTableEnum.PLACE\_TOP or JCTableEnum.PLACE\_BOTTOM.

To change the placement of all frozen columns, set the FrozenColumnPlacement property to either JCTableEnum.PLACE\_LEFT or JCTableEnum.PLACE\_RIGHT.

* + 1. 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:

* + - * JCTableEnum.EDIT\_SIZE\_TO\_CELL: resize the component to fit the Table’s cell size.
      * JCTableEnum.EDIT\_ENSURE\_MINIMUM\_SIZE: resize the component to its minimum size.
      * JCTableEnum.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](#_bookmark340) 4.

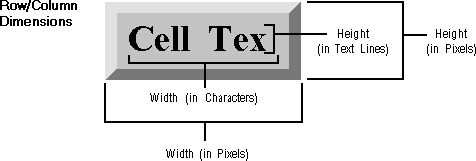
###### Column Width and Row Height Properties

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.*

* + 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

JCTableEnum.NOVALUE.

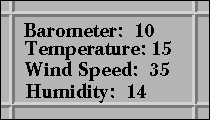
To determine the pixel dimensions of a row or column whose height/width was set by CharWidth or CharHeight, use the getColumnPixelWidth() or getColumnPixelHeight() methods.

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.

* + 1. 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 |  |

* + 1. **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 are height are set to zero, the row/column becomes hidden.
     1. 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.

* + 1. 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.

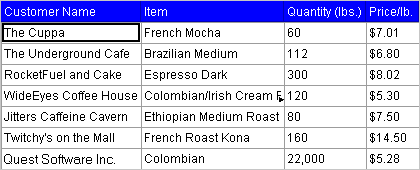
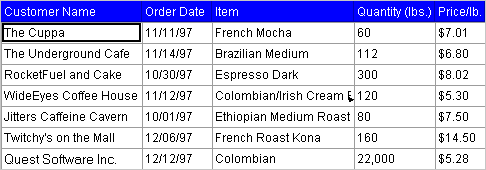
For cell editing, by default, text is edited on a single line. For multiline editing, you must set the multiline editor. To set a multiline editor, you need to set the Cell Style’s editor properties. Create a Cell Style and set the editor for it, then call setCellStyle() with a row, column, or range. Please refer to [Section 2.6, Cell Styles](#_bookmark197), for more information about setting editor properties.

* + 1. 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.



*Figure 7 Hiding the “Order Date” column.*

###### 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.

* + 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);

* + 1. 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.getCellStyle(); 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.

* + 1. 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

JCCellStyle 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](#_bookmark340) 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.getDefaultLabelStlye(); csm = table.getDefaultCellStyle();

JClass LiveTable handles PLAF through a “parenting” mechanism. When the PLAF changes, JCTableUI 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 JCCellStyle(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 JCTableUI 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 JCTableUI 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 JCTableUI assumes the PLAF logic set it. If it is a regular object (in this case, Color), the value will not be updated by JCTableUI.

* + 1. 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** |
| JCTableEnum.REPEAT\_COLUMN  sets repeating background or foreground colors by rows  use as value for setRepeatBackground or  setRepeatForeground methods |  |
| JCTableEnum.REPEAT\_ROW  sets repeating background or foreground colors by columns  use as value for setRepeatBackground or  setRepeatForeground methods |  |

* + 1. 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** |
| setVerticalAlignment(JCTableEnum.TOP) setHorizontalAlignment(JCTableEnum.LEFT) |  |
| setVerticalAlignment(JCTableEnum.TOP) setHorizontalAlignment(JCTableEnum.CENTER) |  |
| setVerticalAlignment(JCTableEnum.TOP) setHorizontalAlignment(JCTableEnum.RIGHT) |  |
| setVerticalAlignment(JCTableEnum.CENTER) setHorizontalAlignment(JCTableEnum.LEFT) |  |

|  |  |
| --- | --- |
| **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) |  |

* + 1. 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:

JCCellStyle labelfont = new JCCellStyle(); 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 C](#_bookmark880).

* + 1. 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](#_bookmark110) [Properties](#_bookmark110), 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.BORDER\_ETCHED\_IN  sets an etched border whose enclosed cells appear raised. | JCTableEnum.BORDER\_ETCHED\_OUT  sets an etched border whose enclosed cells appear set in. |
| JCTableEnum.BORDER\_FRAME\_IN  sets a framed border whose enclosed cells appear set in. | JCTableEnum.BORDER\_FRAME\_OUT  sets a framed border whose enclosed cells appear raised. |
| JCTableEnum.BORDER\_IN  sets a plain border whose enclosed cell appears set in. | JCTableEnum.BORDER\_OUT  sets a plain border whose enclosed cell appears raised. |
| JCTableEnum.BORDER\_PLAIN  set a plain border. | JCTableEnum.BORDER\_THIN  sets a plain border that appears thin. |
| JCTableEnum.BORDER\_NONE  sets no border. |  |

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

JCCellStyle border = new JCCellStyle();

border.setCellBorder(new JCCellBorder(JCTableEnum.BORDER\_NONE)); table.setCellStyle(0, JCTableEnum.ALLCELLS, border);

To retrieve the border style for a cell, use the getCellBorderType() 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:

JCCellStyle cellborder = new JCCellStyle(); 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.

* + 1. 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.



*Figure 8 Customized Cell Borders.*

The valid values for CellBorderSides are:

* + - * JCTableEnum.BORDERSIDE\_LEFT ■ JCTableEnum.BORDERSIDE\_BOTTOM
      * JCTableEnum.BORDERSIDE\_RIGHT ■ JCTableEnum.BORDERSIDE\_ALL
      * JCTableEnum.BORDERSIDE\_TOP ■ 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);

* + 1. 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

|  |  |
| --- | --- |
| JCTableEnum.SHOW\_HORIZTONAL | JCTableEnum.SHOW\_VERTICAL |
| JCTableEnum.SHOW\_NONE | JCTableEnum.SHOW\_ALL (default) |

* + 1. **Displaying Images in Table Cell****s**

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](#_bookmark340) 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.

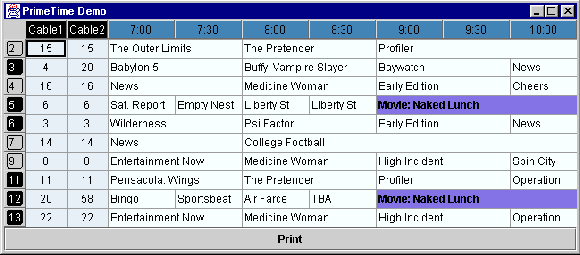
The position of the image within the cell is specified in the same way as Strings, using HorizontalAlignment and VerticalAlignment. This aspect of displaying images is handled by the Styles property. This is covered in [Section 2.6.5, Text and Image](#_bookmark218) [Alignment](#_bookmark218).

###### 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.



*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:

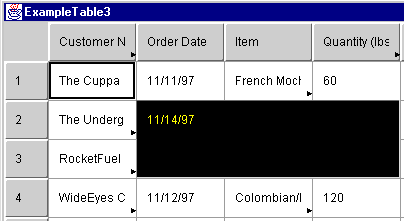
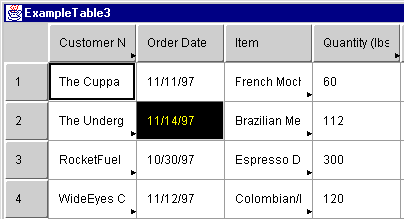
spanlist.overlap: Range R1C2:R1C4 overlaps R1C1:R1C2

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);



*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).*

* + 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.



*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.

# 3

#### Working with Table Data

[*Overview: Data Handling in JClass LiveTable*](#_bookmark252)■ [*Getting Data into your Table*](#_bookmark260)[*Using Stock Data Sources*](#_bookmark275)■ [*Setting Stock Data Source Properties*](#_bookmark294)

[*Loading Data from an XML Source*](#_bookmark312)■ [*Creating your own Data Sources*](#_bookmark325)■ [*Dynamically Updating Data*](#_bookmark329)

###### 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 that you create, rather than the JCTable object itself. 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.

69

**3.1.1 How the Table and Data Source Communi****cate**

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.

###### 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](#_bookmark275)) or one of your own data sources (see [Section 3.6,](#_bookmark325) [Creating your own Data Sources](#_bookmark325)).

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](#_bookmark340) 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 Ed****itable**

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

public interface EditableTableDataModel extends TableDataModel { 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](#_bookmark340) 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.

###### 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** |
| JCAppletDataSource | Reads in data from the DATA tag of an applet. |
| JCCachedDataSource | Caches previously read data from the data source. |

|  |  |
| --- | --- |
| **Data Source** | **Description** |
| JCEditableCachedDataSource | Allows users to edit cell values in tables with the above data source. |
| 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 TableModel data objects in JClass LiveTable. Swing TableModel objects are typically used by the Swing JTable 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 D, JClass LiveTable Inheritance Hierarchy](#_bookmark893), for a complete hierarchy diagram that outlines the relationship between the stock data source classes.

* + 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 JCTable 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 EdtitableTableDataModel interface model.

* + 1. 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](#_bookmark294) [Source Properties](#_bookmark294)). 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.
    1. 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.

* + 1. 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

TableDataModel’s getTableItem, getTableRowLabel, and/or getTableColumnLabel 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.

* + 1. Using Swing TableModel Data Objects

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

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

When you create a JCTableModelDataSource, you need to pass the constructor a valid Swing TableModel object.

###### 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 TableDataModel 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.

* + 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.
    1. 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](#_bookmark329) [Updating Data](#_bookmark329).

###### Loading Data from an XML Source

* + 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/*](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*](http://www.javasoft.com/xml/tutorial_intro.html) – Sun’s XML site
      * [*http://www.oasis-open.org/cover/xml.html*](http://www.oasis-open.org/cover/xml.html) – thorough list of links to XML papers and ongoing work
    1. 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());

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:

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. 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>

* + 1. 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.

* + 1. 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.

###### 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 TableDataModel interface, as in the following example from *examples/table/datasource/StaticDataSource.java*:

import com.klg.jclass.table.TableDataModel; import com.klg.jclass.table.JCTableDataListener;

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(JCTableDataListener l) {

}

public void removeTableDataListener(JCTableDataListener 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 EditableTableDataModel interface, as in *examples/table/datasource/StaticEditableDataSource.java*:

import com.klg.jclass.table.EditableTableDataModel; import com.klg.jclass.table.JCTableDataListener;

public class StaticEditableDataSource implements EditableTableDataModel { 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.

###### 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 JCTable 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 JCTableDataEvent class:

* + - * CHANGE\_VALUE ■ NUM\_ROWS
      * CHANGE\_ROW ■ NUM\_COLUMNS
      * CHANGE\_COLUMN ■ ADD\_COLUMN
      * CHANGE\_ROW\_LABEL ■ REMOVE\_COLUMN
      * CHANGE\_COLUMN\_LABEL ■ MOVE\_ROW
      * ADD\_ROW ■ MOVE\_COLUMN
      * REMOVE\_ROW ■ RESET

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 JCTable that implements TableDataModel, and has methods for adding and removing JCTableDataListeners. 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.JCTableDataEvent;

import com.klg.jclass.table.JCTableDataListener; 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; 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); 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.

* + 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 JCTable. 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 JCTableDataEvent.

The MOVE\_ROW and MOVE\_COLUMN commands are the only commands that make use of the *num\_affected* and *destination* parameters in the JCTableDataEvent. 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 JCTable 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.

# 4

#### Displaying and Editing Cells

[*Overview*](#_bookmark343)■ [*Default Cell Rendering and Editing*](#_bookmark348)[*Rendering Cells*](#_bookmark351)■ [*Editing Cells*](#_bookmark368)■ [*The JCCellInfo Interface*](#_bookmark384)

###### 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** |
| com.klg.jclass.cell | Contains editor/renderer interfaces and support classes, including these interfaces:  JCCellEditor: used to define an editor JCCellRenderer: the common and basic interface for renderers JCComponentCellRenderer: allows the creation of renderers that are based on JComponent JCLightCellRenderer: allows the creation of renderers based on direct drawing |
| com.klg.jclass.cell.editors | Contains editors for common data types.  Please see [Section 4.4.1, Default Cell Editors](#_bookmark370), for details. |

87

|  |  |
| --- | --- |
| **JClass Cell Package** | **Contents** |
| com.klg.jclass.cell.renderers | Contains renderers for common data types, including expressions.  Please see [Section 4.3.1, JClass Cell Renderers](#_bookmark353), for details. |
| com.klg.jclass.cell.validate | 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, the JClass field component can work as a renderer and editor 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.

###### 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.

###### 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](#_bookmark362).

* + 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,](#_bookmark354) [Setting a Cell Renderer for a Series](#_bookmark354)) or as a mapping (see [Section 4.3.3, Mapping a Data](#_bookmark356)

[Type to a Cell Renderer](#_bookmark356)). 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** |
| JCCheckBoxCellRenderer | boolean | Defines a JCComponentCellRenderer object that paints boolean objects in a table cell using Swing’s JCheckBox. |
| JCComboBoxCellRenderer | integer | Defines a JCComponentCellRenderer that paints integer objects in a table using Swing’s JComboBox. |
| JCImageCellRenderer | image | Defines a JCLightCellRenderer object that paints Image objects in a table cell. |
| JCExpressionCellRenderer | expression | Defines the result of a formula (*com.klg.jclass.util.formulae.Expression*). |
| JCLabelCellRenderer | String and/or image | Defines a JCLabelCellRenderer object that uses Swing’s JLabel to render cell contents. |
| JCRawImageCellRenderer | image | Defines a JCLightCellRenderer object that paints unconverted Image objects in a table cell (extends JCScaledImageCellRenderer). |
| JCScaledImageCellRenderer | image | Defines a JCLightCellRenderer object that paints scaled Image objects in a table cell. |
| JCStringCellRenderer | String, boolean, double, float, integer, object | Defines a JCLightCellRenderer 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](#_bookmark362) [Renderers](#_bookmark362).

* + 1. 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](#_bookmark348), 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](#_bookmark198) 2). Inserting these lines of code into your program will do this:

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

The JCCheckBoxCellRenderer 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.

* + 1. 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.

* + 1. 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 JCCel- lInfo Interface](#_bookmark384), 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, JCCellInfocellInfo, Object o boolean selected) {

Polygon p = makePolygon(o);

gc.getColor(selected ? cellInfo.getSelectedForeground(): cellInfo.getForeground());

gc.fillPolygon(p);

}

The draw() method renders the object o by making it into a polygon and drawing the polygon using the gc provided. 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](#_bookmark384)).

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

public Dimension getPreferredSize(Grahpics gc, JCCellInfo cellInfo,

Object o) {

Polygon p = makePolygon(o); if (p == null) {

return new Dimension(0,0);

}

Rectangle r = p.getBoundingBox();

return new Dimension(r.x+r.width,r.y+r.height);

}

Here, the object is used to create a polygon (using a local method called makePolygon()). If it doesn’t create a polygon from the object, the object is deemed to have no size (0,0) and will not be displayed by the renderer. If a polygon was created from the object, the polygon’s bounds determine the size of the rectangle in the drawing area of the cell. The size returned is only a suggestion; control of the cell size can be overridden by the 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](#_bookmark384), for more details).
* o is the object to be rendered.
* selected is a boolean indicating whether the cell is selected. Many implementors use this information to modify the component appearance.

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

import com.klg.jclass.cell.JCComponentCellRenderer; import com.klg.jclass.cell.JCCellInfo;

import javax.swing.JLabel; import java.awt.Component;

public class JCLabelCellRenderer extends JLabel implements JCComponentCellRenderer {

public JCLabelCellRenderer() { 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 JCLabelCellRenderer 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.

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

###### Editing Cells

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

A typical cell edit works through the following process:

* + - The container listens for events that come from the editor by implementing

JCCellEditorListener.

* + - When a user initiates a cell edit with either a mouse click or a key press, the container calls JCCellEditor.initialize() and passes a JCCellInfo object with information about the cell, and the object (data) that will be edited.
    - The JCCellEditor displays the data and changes it according to user input.
    - If the user traverses out of the cell, then the *container* calls the stopCellEditing() method, which asks the JCCellEditor to validate the edit. If the edit is not valid — that is, stopCellEditing() returns false – the container then retrieves the original cell value from the data source. If the edit is valid, then the container calls getCellEditorValue() on the editor to retrieve the new value of the cell and send it to the data source.
    - If the user types a key that the editor interprets as “done” (for example, Enter), the editor will inform the 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.

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

While these classes provide editing capability for most data types, many real-world situations require greater control over cell editing, editing components, and their

relationships to specific data types. The following sections explore how you can more minutely control the cell editing mechanism in your programs.

* + 1. 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.

* + 1. 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.

* + 1. 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.

Method and Description

public boolean isModified();

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

public void cancelCellEditing();

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

public JCKeyModifier[] getReservedKeys();

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

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

Method and Description

public abstract void addCellEditorListener(JCCellEditorListener l);

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

public abstract void removeCellEditorListener(JCCellEditorListener l);

Removes the listener.

In addition to implementing the methods of JCCellEditor, an editor is responsible for monitoring events and sending editingStopped and editingCanceled events to the table. This functionality is further explained in [Creating Your Own Cell Editors](#_bookmark377).

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.cellTriangeTable.

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

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

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

}

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

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

Editor Key Control

Sometimes, you may want your cell editor to be able to accept keystrokes that have already been reserved for a specific purpose in the container (a Tab key in LiveTable, for example). To do this, you need to use the JCKeyModifier class to reserve a key/modifier combination:

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

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

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

* + - * new JCKeyModifier(KeyEvent.VK\_TAB, KeyModifier.ALL);
      * new JCKeyModifier(KeyEvent.VK\_TAB);

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

###### The JCCellInfo Interface

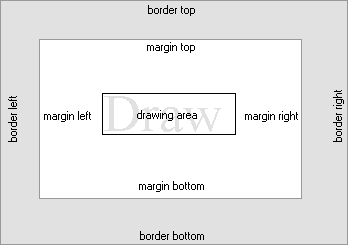
You can see that JCComponentCellRenderer, JCLightCellRenderer and JCCellEditor use the JCCellInfo interface to get information about the cell. The JCCellInfo interface provides information about how the container wants to show the cell. The renderer and editor determine whether or not to honor the container’s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

The JCCellInfo interface gives the renderer and editor access to cell formatting information from the 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 JCCellInfo 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 JCCellInfo.*

For more information, please see the JCCellInfo API documentation.

# 5

#### Adding Formulas to JClass LiveTable

[*Introduction*](#_bookmark389)■ [*com.klg.jclass.util.formulae’s Hierarchy*](#_bookmark394)■ [*Expressions and Results*](#_bookmark397)[*Math Values*](#_bookmark400)■ [*Operations*](#_bookmark438)■ [*Expression Lists*](#_bookmark470)■ [*Events and Listeners*](#_bookmark475)

[*Exceptions*](#_bookmark478)■ [*Using Formulae in JClass LiveTable*](#_bookmark482)

###### Introduction

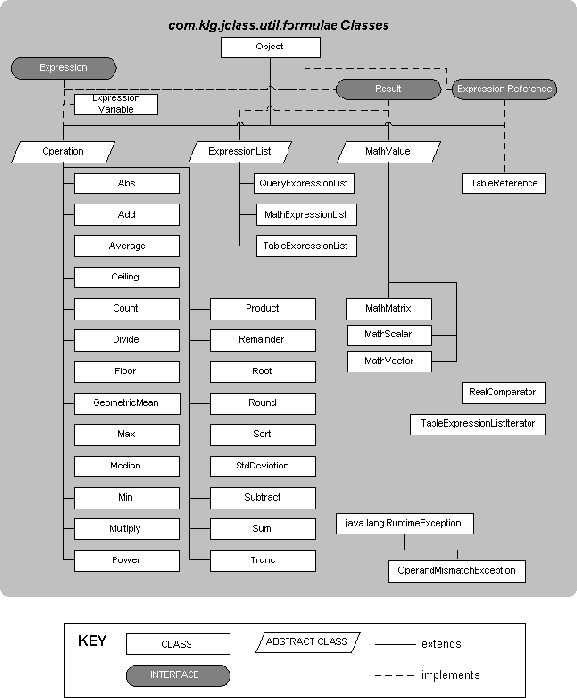
The formulae package in com.klg.jclass.util has special capabilities for evaluating mathematical objects. Similar to the way that objects such as java.lang.Double wrap a primitive type, those in com.klg.jclass.util.formulae encapsulate mathematical expressions (operators) whose operands may be scalars, 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.

###### com.klg.jclass.util.formulae’s Hierarchy

The interfaces, abstract classes, and derived classes, including possible exception classes, are shown in [Figure 13](#_bookmark396).

109



*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.

###### 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.

###### 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** |
| evaluate() | Satisfies the Expression interface by returning the stored value. No evaluation is required because no operation is implied. |
| getDataFormat() | Retrieves the NumberFormat associated with this data. |
| matrixValue() | Gets the contents of this MathValue as a matrix of Numbers. |
| numberValue() | Gets the contents of this MathValue as a Number. |
| setDataFormat() | Sets a NumberFormat to use on the contents of this MathValue. |
| vectorValue() | Gets the contents of this MathValue as a vector of Numbers. |

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.

* + 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. |

* + 1. **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. |

* + 1. **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** |
| getValueAt() | Retrieves the value at a particular row, column pair of index values in the matrix. |
| matrixValue() | Gets the contents of this MathValue as an array of Numbers. |
| numberValue() | Throws an UnsupportedOperationException. |

|  |  |
| --- | --- |
| **MathMatrix Method** | **Description** |
| setValueAt() | Sets the value at a particular row, column pair of index values in the matrix. |
| toString() | Outputs the value of this vector as a String. |
| vectorValue() | Throws an UnsupportedOperationException. |

###### 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.

* + 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: trunc(-3.5) = -3. |

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 = root( (sum(1 to n)(element - 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. |

* + 1. 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.

###### 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 ExpressionLists.

// 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);

sss1 = new Sum(explist1, explist2); ssss1 = sss1.evaluate(); System.out.println(

"Summing ExpressionLists with evaluate(): " + ssss1);

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](#_bookmark482)

[JClass LiveTable](#_bookmark482).

###### 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.

###### 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.

###### Using Formulae in JClass LiveTable

* + 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.

* + 1. 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.

((EditableTableDataModel)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.

# 6

#### Programming User Interactivity

[*Cell Traversal*](#_bookmark493)■ [*Cell Selection*](#_bookmark549)■ [*Resizing Rows and Columns*](#_bookmark506)■ [*Table Scrolling*](#_bookmark523)[*Dragging Rows and Columns*](#_bookmark568)■ [*Sorting Columns*](#_bookmark570)■ [*Custom Mouse Pointers*](#_bookmark580)

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](#_bookmark340) [Editing Cells, in Chapter](#_bookmark340) 4.

###### 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](#_bookmark198) 2).

* + 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.

* + 1. 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.

123

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.

* + 1. 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](#_bookmark528).

* + 1. 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.

* + 1. 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);

}

}

###### Resizing Rows and Columns

* + 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.

* + 1. 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.
    1. 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.

###### Table Scrolling

* + 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](#_bookmark118) 2.

* + 1. 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 JCTableEnum.TRACK\_LIVE, the table redisplays 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 JCTableEnum.TRACK\_COLUMN\_NUMBER for horizontal scrollbar tracking, and JCTableEnum.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 JCTableEnum.TRACK\_ROW, JCTableEnum.TRACK\_COLUMN, or JCTableEnum.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);

* + 1. 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);

###### Cell Selection

* + 1. Default Cell Selection

Cell selection is not enabled by default. When cell selection is enabled (see [Section 6.4.3,](#_bookmark555) [Customizing Cell Selection](#_bookmark555)), 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
      * JCTableEnum.SELECT\_NONE: no selection.
    1. 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](#_bookmark130) 2.

* + 1. 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  JCTableEnum.SELECT\_NONE |  |
| single cell selection  JCTableEnum.SELECT\_SINGLE |  |
| single range selection  JCTableEnum.SELECT\_RANGE |  |
| multiple range selection  JCTableEnum.SELECT\_MULTIRANGE |  |

When SelectionPolicy is set to JCTableEnum.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);

* + 1. 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.

* + 1. 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.

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:

1. Clicking a label selects all of the cells in the row or column, including the label.

Collection col = table.getSelectedCells(); col.add(new JCCellRange(1, 1, 3, 3));

* + 1. Removing Selections

To remove all selections from the table, call clearSelection().

* + 1. 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)).

###### 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.

###### 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](#_bookmark167) [Chapter](#_bookmark167) 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.DESCENDING):

sortByColumn(2, Sort.DESCENDING);

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.DESCENDING, 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 |
| **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);

* + 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.

* + 1. 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.

###### 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.

# 7

#### Events and Listeners

[*Displaying Cells*](#_bookmark589)■ [*Editing Cells*](#_bookmark596)■ [*Painting Tables*](#_bookmark599)■ [*Printing Tables*](#_bookmark603)■ [*Resizing Cells*](#_bookmark607)[*Scrolling in Tables*](#_bookmark613)■ [*Selecting Cells*](#_bookmark619)■ [*Sorting Table Data*](#_bookmark622)■ [*Table Data Changes*](#_bookmark626)■ [*Traversing Cells*](#_bookmark630)

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.

###### Displaying Cells

JCCellDisplayEvent

This event is posted for every cell that is displayed in the table. When you receive a

JCCellDisplayEvent 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 JCTable generates a JCCellDisplayEvent and notifies any JCCellDisplayListeners that they can customize the display object by calling setDisplayData() on the event. JCTable does not generate the event if there are no listeners registered with the table.

JCCellDisplayListener

To register the above event listener routine, use the following call (where (this) refers to the class, which implements the JCCellDisplayListener interface):

table.addCellDisplayListener(this);

**139**

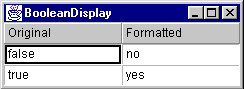
JCCellDisplayListener requires the following method to be implemented:

public void cellDisplay(JCCellDisplayEvent e)

**Calling JCCellDisplayEvent and JCCellDisplayListener methods** JCCellDisplayListener’s method is called before each cell is rendered, and all JCCellDisplayEvent methods are available at all times during the display process. For more information, please refer to [Appendix A](#_bookmark724), which provides a complete event summary.

Using JCCellDisplay Events and Listeners

JCCellDisplayListener 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, JCCellDisplayEvent 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 JCCellDisplayEvent to display BooleanCellData objects as yes/no Strings.*

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

import com.klg.jclass.table.data.JCEditableVectorDataSource; import com.klg.jclass.table.JCCellDisplayListener;

import com.klg.jclass.table.JCCellDisplayEvent; 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 JCCellDisplayListener

{

// Table instance protected JCTable table;

// Editable table data source

protected JCEditableVectorDataSource evds;

public BooleanDisplay() { setBackground(Color.lightGray);

// Create table instance table = new JCTable();

// 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);

}

}

###### 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](#_bookmark724), 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 JCEditableVectorDataSource 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 JCTable();

// Create and set up data source

evds = new JCEditableVectorDataSource(); 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.addEditCellListener(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);

}

}

###### 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](#_bookmark724), 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.

###### 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](#_bookmark635) 8, has details and examples for using the JCPrintListener.

###### 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

JCTblEnum.NOVALUE if only a column is being resized.

* + - getCurrentColumnWidth() – retrieves the current column width. Returns

JCTblEnum.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

JCTblEnum.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](#_bookmark724), 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);

}

}

###### 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:

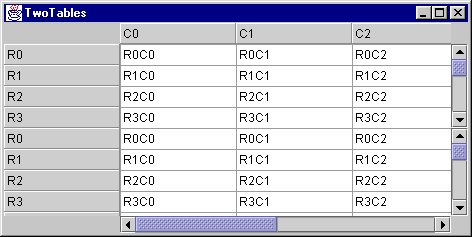
public void scroll(JCScrollEvent e) public void afterScroll(JCScrollEvent e)

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](#_bookmark724), which provides a complete event summary.

Using JCScroll Events and Listeners

JCScrollListener 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 JCScrollListener to synchronize scrolling between two tables.*

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

import com.klg.jclass.table.data.JCVectorDataSource;

import com.klg.jclass.table.data.JCEditableVectorDataSource; import com.klg.jclass.table.JCScrollListener;

import com.klg.jclass.table.JCScrollEvent; 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 JCScrollListener {

// First table

protected JCTable table1;

// Second table

protected JCTable 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 JCTable();

// Create and set up data source for first table evds1 = new JCEditableVectorDataSource(); 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(JCTableEnum.RESIZE\_NONE); table1.setHorizSBDisplay(JCTableEnum.SBDISPLAY\_NEVER); table1.getDefaultCellStyle().setTraversable(false); table1.setVisibleRows(2);

table1.setVisibleColumns(3);

// Create second table table2 = new JCTable();

// Connect second table to same data source as first table. table2.setDataSource(evds1);

// Set up visuals and interactions for table 2. table2.setAllowCellResize(JCTableEnum.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);

}

}

###### 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](#_bookmark724), 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);

}

}

###### 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):

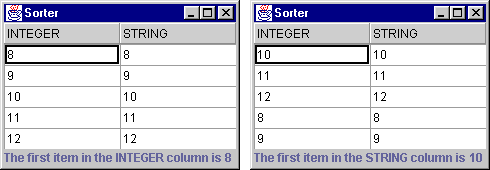
table.addSortListener(this);

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

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 JCTable table;

// Table data source

protected JCEditableVectorDataSource ds;

// Label that will display column top value protected JLabel topItem;

public Sorter() { setBackground(Color.lightGray);

// Create table instance table = new JCTable();

// 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),

JCTableEnum.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);

}

}

###### 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 TableDataModel object changes.
    - getDestination() – indicates the destination location for MOVE events.
    - getCommand() – returns the command that initiated the event. Valid commands include:

CHANGE\_VALUE CHANGE\_ROW ADD\_ROW REMOVE\_ROW CHANGE\_COLUMN ADD\_COLUMN REMOVE\_COLUMN CHANGE\_ROW\_LABEL

CHANGE\_COLUMN\_LABEL MOVE\_ROW MOVE\_COLUMN NUM\_ROWS NUM\_COLUMNS

RESET

Single cell value changed. Single row changed.

New row added. Row removed.

Single column changed. New column added.

Column removed.

Single row label changed. Single column label changed.

Row moved, new location in getDestination. Column moved, new location in getDestination. Number of rows changed.

Number of columns changed.

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);

JCDataTableListener 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 TableDataModel

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](#_bookmark326) 3, to see descriptions of the included sample code.

###### 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](#_bookmark724), 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.JCTable; import com.klg.jclass.table.JCTableEnum;

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 JCTable table;

// Table data source

protected JCVectorDataSource ds;

public SkipNavigation() { 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, "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);

}

}

# 8

#### Table Printing

[*Printing*](#_bookmark637)■ [*Print Preview*](#_bookmark648)

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.

###### 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();

163

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 top, left, bottom, and right margins in pixels:

printtable.setPageMargins(new Insets(54,36,36,54));

You can choose to specify margins in inches instead of pixels. Call the getMarginUnits()

method with JCPrintTable.MARGIN\_IN\_INCHES:

setMarginUnits(JCPrintTable.MARGIN\_IN\_INCHES); printtable.setPageMargins(new Insets(1,0.5,1,0.5));

Note: By default, 72 pixels on screen converts to one inch in print (72:1). 72 pixels is the standard Java display value; you should not need to change this value. If you find that you do need to change it, use the setPageResolution() method and specify the number of pixels that you want to be converted to one inch in print. The getPageResolution() method returns this value.

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 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, intnumPages, 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();

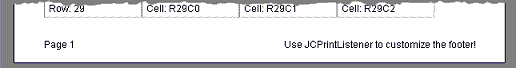
String page = "Page " + e.getPage();

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)

Chapter 8 ■ Table Printing **165**

For example, the following provides a preview beginning at the first page of the job:

JCPrintPreview pf = new JCPrintPreview("Table Print Preview",printtable);

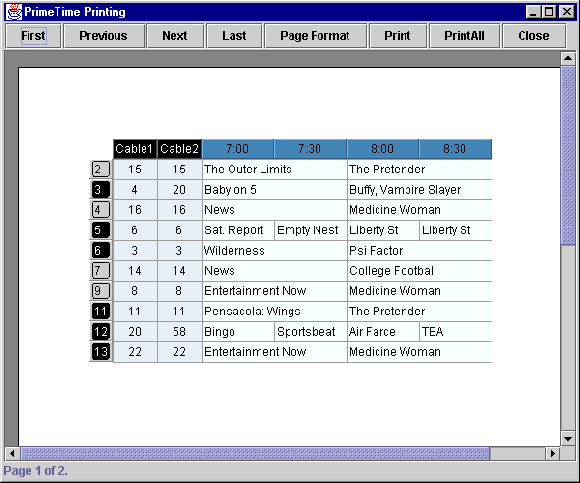
pf.showPage(0);

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.



*Figure 18 The JClass LiveTable Print Preview Window.*

# 9

#### JClass LiveTable Beans and IDEs

[*An Introduction to JavaBeans*](#_bookmark658)■ [*JClass LiveTable JavaBean*](#_bookmark668)[*Setting Properties for the LiveTable Bean*](#_bookmark669)■ [*Tutorial: Building a Table in an IDE*](#_bookmark708)

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).

###### 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*](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
    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

167

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.

* + 1. 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.

* + 1. 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);

###### JClass LiveTable JavaBean

The LiveTable JavaBean included with JClass LiveTable makes it easy to create applications and applets in an Integrated Development Environment.

###### Setting Properties for the LiveTable Bean

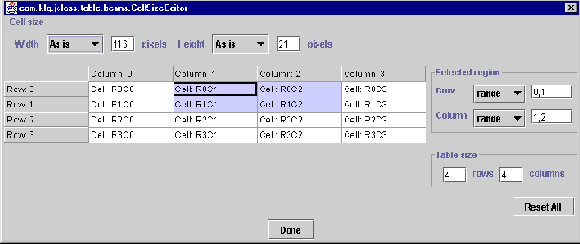
At design-time, most LiveTable properties are set using simple menu choices or text entry boxes on the property sheet. Some properties that are set for individual cells or

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.

* + 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.

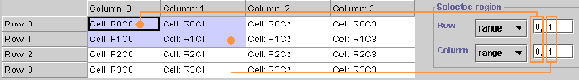
Selecting a Cell or Cell Range

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.

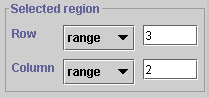
To select cells using the mouse:

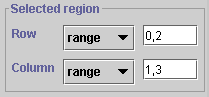
* + - * Click an individual cell with the mouse to select that cell.
      * Click and drag the mouse to select a range of cells.
      * Click on a row or column label to select that row or column.
      * Click on a cell, hold down the Shift key and click another cell to select a range of cells between 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:



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).

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.

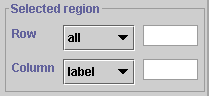
Selecting Labels

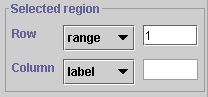
To select labels using the mouse:

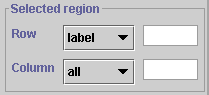
* Row labels: click the row label or select a range of row labels and choose Label from the Column pull-down menu in the Selected region controls.
* Column labels: click the column label or select a range of column labels and choose Label from the Row pull-down menu in the Selected region controls.

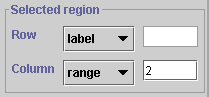
It may seem odd to be choosing in the Column box for *row* labels and in the Row box for *column* labels, but it is easier to understand if you consider that you are really specifying the row of column labels or the column of row labels.

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.

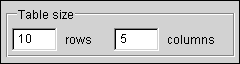
To select a row label, choose label from the Column pull-down menu. Then choose range in the Row pull down menu, and type the desired row number in the text field.

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.

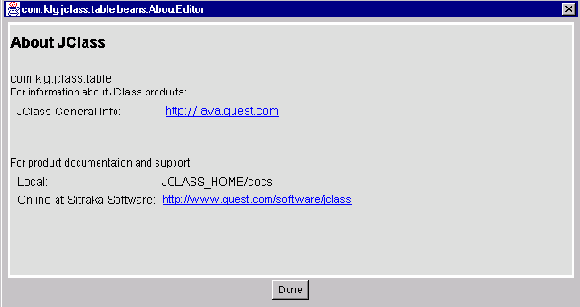
You can undo any of your changes and reset the properties to the values they had when you opened the editors by clicking the Reset All button.

* + 1. LiveTable Properties

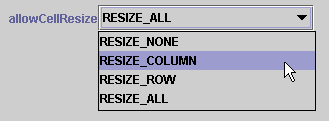
The LiveTable Bean exists because the current generation of Java IDEs do not support properties in contained objects. While current Java IDEs allow properties in contained objects to be modified, they cannot yet properly generate code for the property change. LiveTable works around this problem by exposing 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](#_bookmark671), 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.

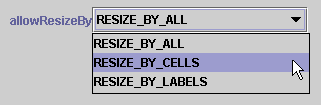
about

about displays the component version and points to other sources of JClass information

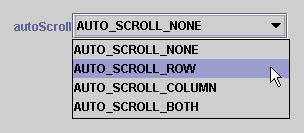
allowCellResize

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

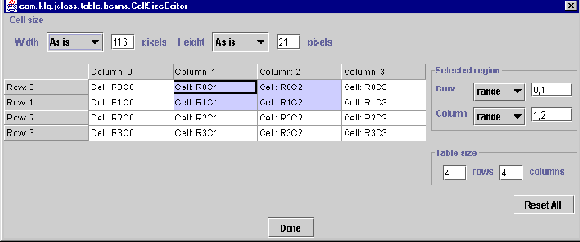
The autoScroll property determines if the table automatically scrolls when the user drags the mouse or traverses past the borders of the table.

cellBorderWidth

Entering a number into the cellBorderWidth field specifies the thickness of the border around each cell and label.

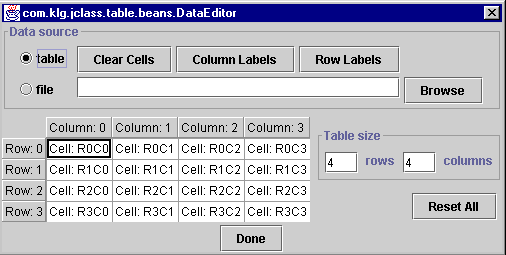
cellSize

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 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'

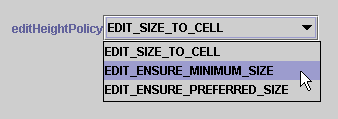
'Row 2' '2,0' '2,1' '2,2'

'Row 3' '3,0' '3,1' '3,2'

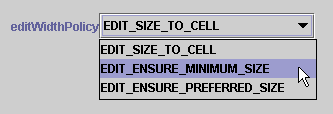
editHeightPolicy

The table can control the height of a cell editing component using the editHeightPolicy

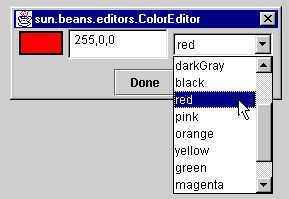
property. This property can take one of three values:

* + - * 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.

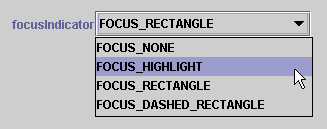
editWidthPolicy

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

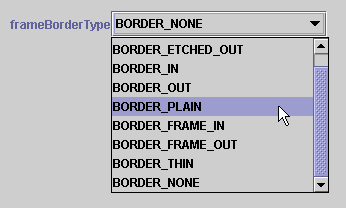
Choosing a color for the focusColor property determines the color of the focus rectangle. The focus rectangle is the line drawn around the inside of the cell that currently has focus.

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

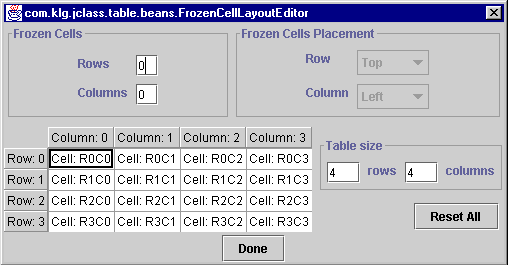
Setting the frameBorderType property determines what type of border is used for the frame enclosing the cell and label areas. Choose a border type from the pull- down menu. Note that the FrameBorderWidth property must be set to greater than 5 in order for the etched border types to be visible.

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.

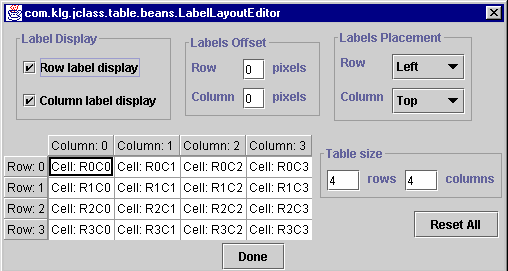
frozenCellLayout

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.

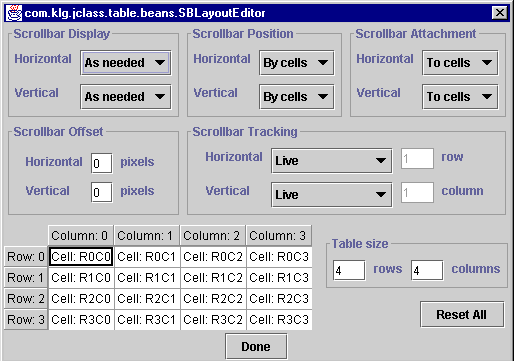
The Scrollbar Position settings determine whether scrollbars are positioned by cells or at sides. The former option places the scrollbar beside the cell/label viewport, while the latter places the scrollbar beside the edge of the table area. Note that there is no visual difference between the two options 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.

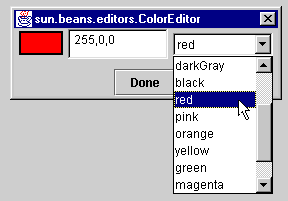
Scrollbar Offset sets the amount of space, in pixels, between the scrollbar and the table.

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.

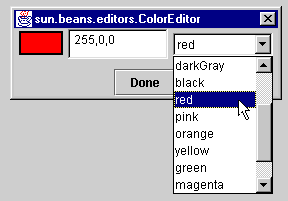
However, to offer the user feedback, the Column number/Row number option displays a box with the current cell number, while the Row/Column option displays the actual contents of the current cell. The row or column from which the displayed contents are taken is determined by the number entered in the Row or Column fields.



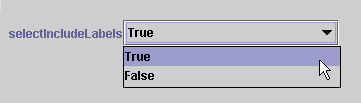
selectedBackground

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.

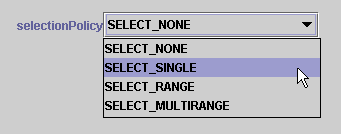
selectedForeground

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

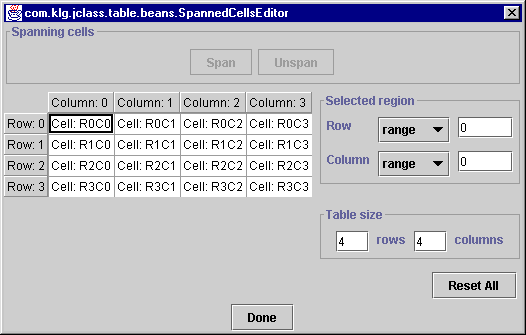
By default, when a user clicks a label, the entire row or column, including the label, is highlighted. To change it so that the label is not highlighted with the rest of the cells, set this property to false.

selectionPolicy

The SelectionPolicy property controls the amount of selection allowed on the table, both by end- users and by the application.

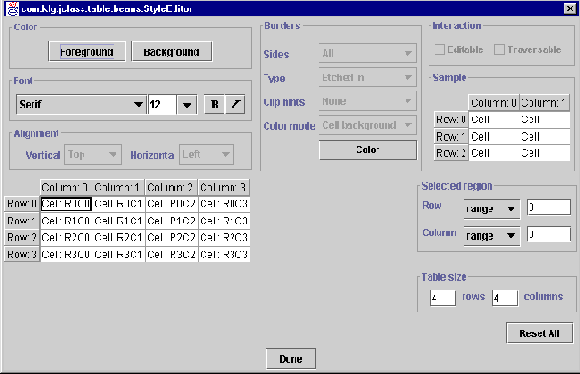
spannedCells

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.

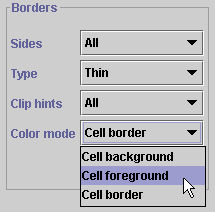


styles

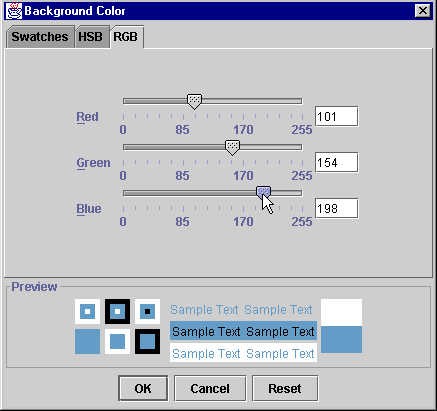
The styles property editor gives you control over cell style options, including: cell colors, font attributes, editable and traversable states, and alignment in cells.



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.

###### 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’ –](#_bookmark20)

[JClass LiveTable Tutorial, in Chapter 1](#_bookmark20), 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:

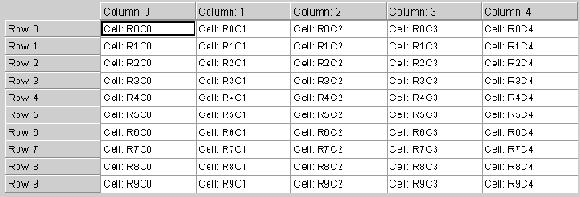
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Customer Name** | **Order Date** | **Item** | **Quantity (lbs.)** | **Price/lb.** |
| The Cuppa | 11/11/97 | French Mocha | 60 | $7.01 |
| The Underground Cafe | 11/14/97 | Brazilian Medium | 112 | $6.80 |
| RocketFuel and Cake Cafe | 10/30/97 | Espresso 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/Ir ish 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 |

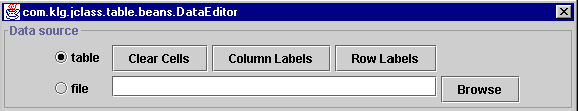
* + 1. The Basic Table

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:



Supplying the Data

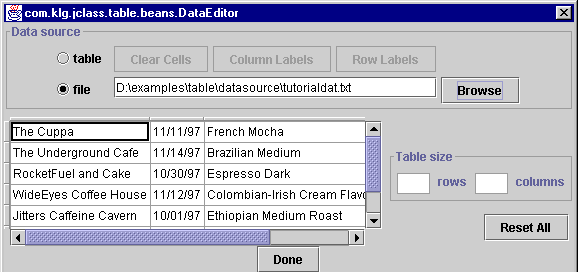
The data for the table is contained in the data source. You can provide the data by entering it into the table using the DataEditor, or by specifying that a file is the data source. Start the DataEditor by clicking the data property on the property sheet. Notice that the editor defaults to using the table as the data source.



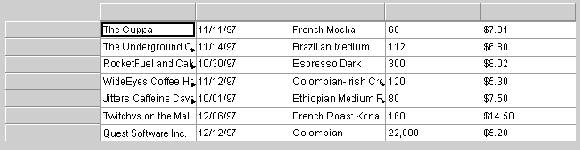
The data we want to display is stored in a file. To use this file as the data source:

* + - 1. Click Browse.
      2. Navigate to the *examples/table/datasource* directory of your JClass LiveTable distribu- tion.
      3. 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:



* + - 1. To close the editor, click Done. The table displayed in the BeanBox now shows the values from the data source:



* + 1. 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

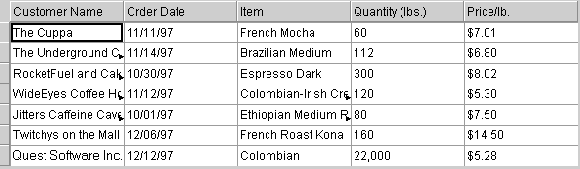
The table currently displayed in the BeanBox is not very useful to an end-user. Not only is it not interesting to look at, but you cannot tell what kinds of information the various cells contain because there are no column labels. In the original data outline for the table, we indicated that we wanted 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.

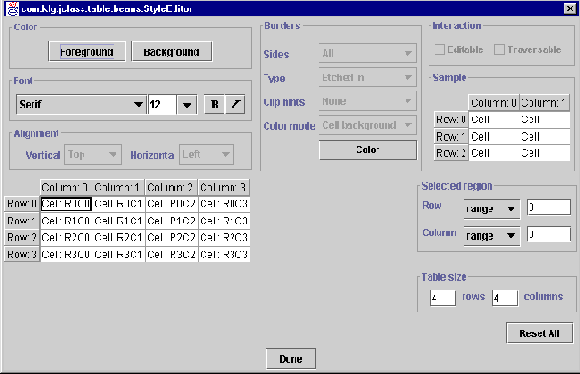
If you had entered the table data directly into the Bean, you could add the labels using the DataEditor. But since the file is the data source, adding the labels must be done by editing the file. For convenience we have included the labels in another data file. Load this data file using the DataEditor 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:



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](#_bookmark716) [Interactivity](#_bookmark716)), 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:

* + - 1. Choose Times New Roman from the font pull-down menu.
      2. Choose 14 from the point size pull-down menu.

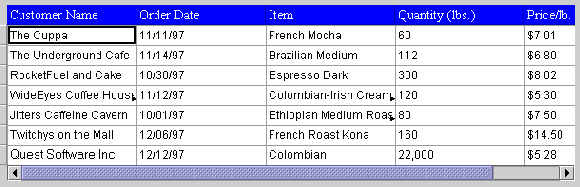
Note: The type of font displayed on a user’s system depends entirely on the fonts that are local to that user’s computer. If a font name specified in a Java program is not found on a user’s system, the closest possible match is used (as determined by the Java AWT).

Finally, change the color of the label text. In the Color box:

* + - 1. Choose Foreground, then select a white color from the Swatches tab.
      2. Choose Background, then select a blue color from the Swatches tab.
      3. You should be able to see these changes in the sample table in the StyleEditor. Click

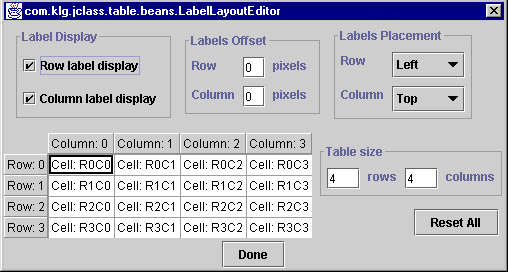
Done to commit the changes and return to the BeanBox.

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.



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:



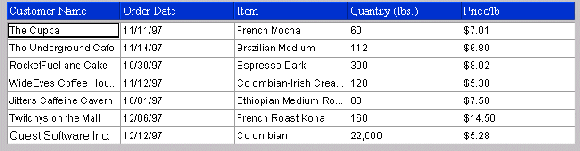
For this exercise, you are going to add some space between the column labels and the top of the table and take out the row labels. In the LabelLayout editor:

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.

1. 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:



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).

1. 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:

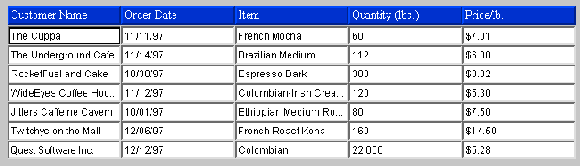
1. Click the Types drop-down list in the Borders box and select border in.

To change the border type for the column labels, you now need to select all column labels. In the Selected Region box:

1. 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:

1. Click the Types drop-down list in the Borders box and select border out. The table should now resemble the following in the BeanBox:
   * 1. 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](#_bookmark490) 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.

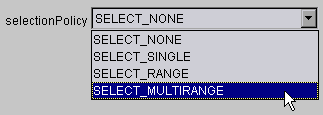
Note that each cell in the editor’s table reflects its current traversable and editable state. A cell that is editable must also be traversable, but a cell that is traversable does not necessarily have to be editable. For this particular cell, leave traversability on, and simply unset its editability:

* + - 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.

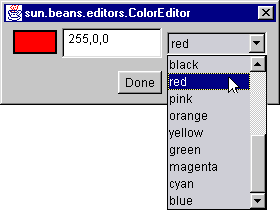
By default, setting the SelectionPolicy property enables selection of entire rows or columns by clicking on the row or column label. When the user clicks on the column label, the column display, including the label, is reversed to highlight the selection. You can configure the table not to highlight 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:



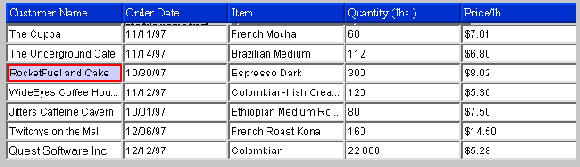
When you click on the FocusRectColor property, the default color chooser appears.

Choose red from the color chooser. Now your users should be able to see the focus rectangle clearly.

* + 1. 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.



*PartII* *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](#_bookmark586) 7.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Event and Description** | **Event Methods** | **Action** | | | **Listener Interface** |
| **before** | **on** | **after** |
| JCCellDisplayEvent | getRow |  |  |  | JCCellDisplayListener |
|  | getColumn |  |  |
| Posted when a cell’s contents are to be displayed in a table. | getCellData getDisplayData setDisplayData |      |  |
| JCEditCellEvent | getRow |  |  |  | JCEditCellListener |
|  | getColumn |  |  |  |  |
| Posted when a cell’s contents are to be edited. | getEditingComponent getType isCancelled setCancelled |        |       — |       — |  |
| JCPaintEvent | getType |  |  |  | JCPaintListener |
|  | getStartRow |  |  |  |
| Posted when a portion of the table is painted. | getStartColumn getEndRow  getEndColumn |      |      |  |
|  | getCellRange |  |  |  |

193

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Event and Description** | **Event Methods** | **Action** | | | **Listener Interface** |
| **before** | **on** | **after** |
| JCPrintEventa | getGraphics |  |  |  | JCPrintListener |
|  | getMarginUnits |  |  |  |  |
| The event posted for each page during the printing process. | getNumHorizontalPages getNumPages getNumVerticalPages  getPage |        |        |        |  |
|  | getPageDimensions |  |  |  |  |
|  | getPageMargins |  |  |  |  |
|  | getPageResolution |  |  |  |  |
|  | getTableDimensions getType | —   |    | —   |  |
| JCResizeCellEvent | getRow |         —      —     |  |         —      —    — | JCResizeCellListener |
|  | getColumn |  |  |
| Posted when a cell in the table is resized. | getCurrentRowHeight getNewRowHeight  setNewRowHeight |      |  |
|  | getCurrentColumnWidth |  |  |
|  | getNewColumnWidth |  |  |
|  | setNewColumnWidth |  |  |
|  | isCancelled |  |  |
|  | setCancelled |  |  |
| JCResizeCellEvent | getRow |  |  |  | JCResizeCellMotionListenerb |
|  | getColumn |  |  |
| Posted while a cell is being resized. | getCurrentRowHeight  getNewRowHeight setNewRowHeight |      |  |
|  | getCurrentColumnWidth |  |  |
|  | getNewColumnWidth |  |  |
|  | setNewColumnWidth |  |  |
|  | isCancelled |  |  |
|  | setCancelled |  |  |
| JCScrollEvent | getAdjustable |  |  |  | JCScrollListener |
|  | getDirection |  |  |  |
| Posted when a user resizes a row and/or column. | getEvent getType getValue setValue |        |       — |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Event and Description** | **Event Methods** | **Action** | | | **Listener Interface** |
| **before** | **on** | **after** |
| JCSelectEvent | getType |  |  |  | JCSelectListener |
|  | getStartRow |  |  |  |  |
| Posted when a user selects one or more cells. | getStartColumn getEndRow getEndColumn  isCancelled |        |        |        |  |
|  | setCancelled  getAction |    |    | —   |  |
|  | getActionString |  |  |  |  |
| JCSortEvent | getColumns |  |  |  | JCSortListener |
|  | getNewRows |  |  |
| Posted after a |  |  |  |
| sortByColumn call. |  |  |  |
| JCTableDataEvent | getRow getColumn |  |    |  | JCTableDataListenerc |
| Posted when the TableDataModel object is modified. | getNumAffected getDestination getCommand |      |  |
| JCTraverseCellEvent | getRow |  |  |         —  —      — | JCTraverseCellListener |
|  | getColumn |  |  |
| Posted when a user traverses from one cell to another. | getNextRow getNextColumn setNextRow  setNextColumn |        |  |
|  | getTraverseType |  |  |
|  | isCancelled |  |  |
|  | setCancelled |  |  |

1. JCPrintEvent actions are not before<Event>, on<Event> and after<Event> as they are with other events; the action events are printPageHeader(), printPageBody() and printPageFooter().
2. JCResizeCellMotionListener has one method: resizeCellDragged(). It is called repeatedly during cell resizing.
3. JCTableDataEvents are posted by the table’s data source, and not by the table itself.

Appendix A ■ Event Summary **195**

### Appendix B

#### JClass LiveTable Property Listing

[*Properties of com.klg.jclass.table.JCTable*](#_bookmark743)■ [*Properties of com.klg.jclass.table.CellStyleModel*](#_bookmark823)

[*Properties of com.klg.jclass.table.beans.LiveTable*](#_bookmark841)

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 (\*).

197

###### Properties of com.klg.jclass.table.JCTable

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Values/Examples** |
| AllowCellResize | The AllowCellResize property specifies whether and how an end user is able to resize rows and columns. | JCTableEnum.RESIZE\_ALL\* JCTableEnum.RESIZE\_NONE JCTableEnum.RESIZE\_COLUMN JCTableEnum.RESIZE\_ROW |
| AllowResizeBy | This property determines whether row heights and column widths can be resized by labels. | JCTableEnum.RESIZE\_BY\_LABELS\* JCTableEnum.RESIZE\_BY\_CELLS JCTableEnum.RESIZE\_BY\_ALL |
| AutoEdit | Determines whether an editor is automatically displayed in a cell when it is entered. | boolean value: default false |
| AutoScroll | This property sets how the table scrolls when the user moves out of the bounds of the displayed table. | JCTableEnum.AUTO\_SCROLL\_NONE\* JCTableEnum.AUTO\_SCROLL\_ROW JCTableEnum.AUTO\_SCROLL\_COLUMN JCTableEnum.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, JCTableEnum.LABEL, or JCTableEnum.ALL, plus the number of characters |
| CharWidth | Width of column in characters. | specific column number, JCTableEnum.LABEL, or JCTableEnum.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) |

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Values/Examples** |
| ColumnLabel Placement | Location of the column labels (top or bottom of the table). | JCTableEnum.PLACE\_TOP\* JCTableEnum.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 |
| ComponentBorder Width | This property determines the spacing between the border of a cell and the cell component. | integer value (pixels) |
| 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. | JCTableEnum.EDIT\_SIZE\_TO\_CELL\* JCTableEnum.EDIT\_ENSURE\_  MINIMUM\_SIZE  JCTableEnum.EDIT\_ENSURE\_  PREFERRED\_SIZE |
| EditWidthPolicy | Horizontal sizing policy for cell editors. | JCTableEnum.EDIT\_SIZE\_TO\_CELL\* JCTableEnum.EDIT\_ENSURE\_  MINIMUM\_SIZE  JCTableEnum.EDIT\_ENSURE\_  PREFERRED\_SIZE |
| FocusColor | Determines the color of the focus indicator. | Any java.awt.Color object. |

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Values/Examples** |
| FocusIndicator | This property sets the focus indicator. | JCTableEnum.FOCUS\_RECTANGLE\* JCTableEnum.FOCUS\_DASHED\_  RECTANGLE  JCTableEnum.FOCUS\_NONE JCTableEnum.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 |
| 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. | JCTableEnum.PLACE\_LEFT\* JCTableEnum.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. | JCTableEnum.PLACE\_TOP\* JCTableEnum.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); |

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Values/Examples** |
| 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. | JCTableEnum.SIZE\_TO\_CELLS\* JCTableEnum.SIZE\_TO\_TABLE |
| HorizSBDisplay | Determines when to display horizontal scrollbar. | JCTableEnum.SBDISPLAY\_ALWAYS JCTableEnum.SBDISPLAY\_NEVER  JCTableEnum.SBDISPLAY\_AS\_  NEEDED\* |
| HorizSBOffset | Distance between the table and horizontal scrollbar in pixels. | integer: number of pixels. For example: setHorizSBOffset(3); |
| 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. | JCTableEnum.POSITION\_BY\_CELLS\* JCTableEnum.POSITION AT\_SIDE |
| HorizSBTrack | Determines how the horizontal scrollbar acts during scroll tracking. | JCTableEnum.TRACK\_LIVE\* JCTableEnum.TRACK\_COLUMN\_  NUMBER  JCTableEnum.TRACK\_ROW |
| HorizSBTrackRow | Determines the row number whose text is displayed when  JCTableEnum.TRACK\_ ROW is used with setHorizSBTrack() | JCTableEnum.LABEL  or  integer value (row number’s cell data) |

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Values/Examples** |
| JumpScroll | Determines whether the table will visually scroll smoothly or whether the display will ‘jump’ to display the cells scrolled to. | JCTableEnum.JUMP\_NONE\* JCTableEnum.JUMP\_HORIZONTAL  JCTableEnum.JUMP\_VERTICAL JCTableEnum.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); |
| 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 |

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Values/Examples** |
| 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 JBdbTable 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) |
| 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 |

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Values/Examples** |
| 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 |
| SelectedForeground | Foreground color for cells that have been selected. The default is the cells’ background color. | Color value. For example:  setSelectedForeground  (Color.blue); |
| SelectedForeground Mode | Sets the table’s foreground color for selected cells. | JCTableEnum.USE\_SELECTED\_  FOREGROUND  JCTableEnum.USE\_CELL\_  BACKGROUND  JCTableEnum.USE\_CELL\_  FOREGROUND |
| SelectionModel | Specifies the mode for selecting, based on cells, rows, or columns. | SelectionModel |
| SelectionPolicy | Sets the type of allowable selection. | JCTableEnum.SELECT\_NONE JCTableEnum.SELECT\_SINGLE JCTableEnum.SELECT\_RANGE JCTableEnum.SELECT\_MULTIRANGE |

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Values/Examples** |
| SeriesDataSorted | Specifies if series are sorted when the table is sorted. If set as true, the series information sorts with the table data. | boolean (default: true) |
| StoreImageEnabled | Determines whether an image of the table is maintained off-screen. | boolean (default: false) |
| TopRow | Indicates the non-frozen row at least partially visible at the top of the table. | integer: row number |
| TrackBackground | Determines the background color of the track component. | Color value |
| TrackCursor | Determines whether the cursor changes. | boolean (default: true) |
| TrackForeground | Determines the foreground color of the track component. | Color value |
| 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) |
| VariableEstimate Count | Sets the number of cells to use in estimating variable pixel calculations. | default: JCTableEnum.ALL |

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Values/Examples** |
| 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. | JCTableEnum.SIZE\_TO\_CELLS JCTableEnum.SIZE\_TO\_TABLE |
| VertSBDisplay | Determines when to display vertical scrollbar. | JCTableEnum.SBDISPLAY\_AS\_  NEEDED\* JCTableEnum.SBDISPLAY\_ALWAYS JCTableEnum.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. | JCTableEnum.POSITION\_BY\_CELLS JCTableEnum.POSITION\_AT\_SIDE |
| VertSBTrack | This property determines how the vertical scrollbar acts during scroll tracking. | JCTableEnum.TRACK\_LIVE\* JCTableEnum.TRACK\_ROW\_NUMBER JCTableEnum.TRACK\_COLUMN |
| VertSBTrackColumn | Determines the column number whose text is displayed when  JCTableEnum.TRACK\_ ROW is used with setVertSBTrack(). | JCTableEnum.LABEL  or  integer value (row number’s cell data) |

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Values/Examples** |
| 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 |

###### 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. | JCTableEnum.USE\_CELL\_  BORDER\_COLOR  JCTableEnum.BASE\_ON\_  BACKGROUND  JCTableEnum.BASE\_ON\_  FOREGROUND |
| CellBorderSides | Visible border sides (defined by CellBorderType) for individual cells. | JCTableEnum.BORDERSIDE\_NONE  JCTableEnum.BORDERSIDE\_ALL\* JCTableEnum.BORDERSIDE\_LEFT  JCTableEnum.BORDERSIDE\_  RIGHT  JCTableEnum.BORDERSIDE\_TOP JCTableEnum.BORDERSIDE\_  BOTTOM |
| ClipHints | Determines whether clip arrows are shown, and where, when the contents of the cell do not fit in the cell frame. | JCTableEnum.SHOW\_NONE JCTableEnum.SHOW\_HORIZONTAL  JCTableEnum.SHOW\_VERTICAL JCTableEnum.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. | JCTableEnum.LEFT\* JCTableEnum.CENTER JCTableEnum.RIGHT |
| RepeatBackground | Determines if the background color repeats for rows or columns. | JCTableEnum.REPEAT\_NONE\* JCTableEnum.REPEAT\_ROW JCTableEnum.REPEAT\_COLUMN |

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Values/Examples** |
| RepeatBackground Colors | Repeats pattern for background colors. | array of colors |
| RepeatForeground | Determines if the foreground color repeats for rows or columns. | JCTableEnum.REPEAT\_NONE\* JCTableEnum.REPEAT\_ROW JCTableEnum.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. | JCTableEnum.TOP\* JCTableEnum.CENTER JCTableEnum.RIGHT |

###### Properties of com.klg.jclass.table.beans.LiveTable

|  |  |
| --- | --- |
| **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. |
| data | Specifies table data, data source, and row/column labels. |
| 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. |
| jumpScroll | Determines whether jump scrolling is turned on. |
| 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. |

|  |  |
| --- | --- |
| **Name** | **Description** |
| 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. |
| swingDataModel | Sets the table’s data source to use a specified Swing  TableModel object, instead of using the data 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. |

**Appendix C**

**Colors and Fonts**

[*Colorname Values*](#_bookmark882)■ [*RGB Color Values*](#_bookmark884)■ [*Fonts*](#_bookmark887)

This section provides information on common colorname values, specific RGB color values, and fonts applicable to all Java programs.

###### 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 ■ lightGray
    - blue ■ lightBlue
    - cyan ■ magenta
    - darkGray ■ orange
    - darkGrey ■ pink
    - gray ■ red
    - grey ■ white
    - green ■ yellow
    - lightGray

###### 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.

213

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

###### 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.

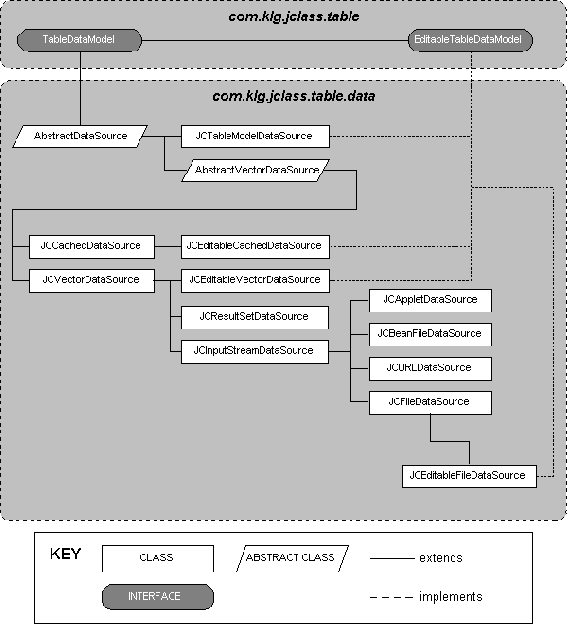
Note: Font display may vary from system to system. If a font does not exist on a system, the default font is displayed instead.

### Appendix D

#### JClass LiveTable Inheritance Hierarchy

General JClass LiveTable Classes

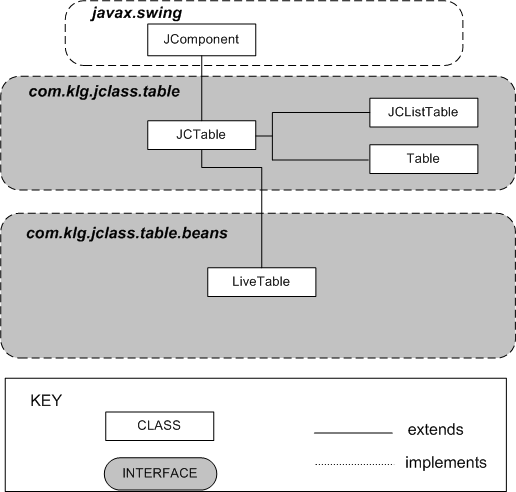
The following figure gives an overview of class inheritance for table creation in JClass LiveTable.



JCTable is the core JClass LiveTable class, with which most table programming is performed, and from which all LiveTable Beans are extended. The JCListTable class extends JCTable and provides a quick way of formatting a table to look and act like a list.

JClass LiveTable Classes

The following figure provides an overview of class inheritance 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 E

#### Distributing Applets and Applications

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.

Using JarMaster to Customize the Deployment Archive

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](http://www.quest.com/).

### Appendix F

#### Overview of Examples and Demos

[*JClass LiveTable Examples*](#_bookmark908)■ [*JClass LiveTable Demos*](#_bookmark918)

This appendix summarizes the examples and demos that ship with JClass LiveTable.

###### 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.

* + 1. Introductory Examples

The introductory JClass LiveTable examples are part of the tutorial found in [‘Hello Table’](#_bookmark20)

[– JClass LiveTable Tutorial, in Chapter](#_bookmark20) 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:

*ExampleTable1.java* A table with basic visual and interactive properties.

*ExampleTable2.java* A table based on the previous example, but with labels.

*ExampleTable3.java* The previous example table’s label colors have been

changed.

*ExampleTable4.java* The label fonts and text alignment have been modified.

*ExampleTable5.java* Cell and frame borders and spacing have been

changed. One table cell’s colors have been changed.

*ExampleTable6.java* Cell editing has been enabled. Cell width and height have been changed.

*ExampleTable7.java* Table resizing only with labels has been set.

*ExampleTable8.java* Column sorting has been enabled.

* + 1. 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:

*Cars.java* A table that contains tables. This example also demonstrates cell spanning and visual property settings.

*Flexible.java* Columns resize dynamically, depending on how the whole table is resized.

* + 1. 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](#_bookmark198) 2. These examples are part of the examples.table.styles package, and are found in the *examples/table/styles* directory:

*Animated.java* Displays animated cells.

*BorderTypes.java* Showcases the cell border options available, including

some customized cell borders.

*CellBorders.java* Demonstrates various cell border appearance attributes,

including border type, border width, and cell side coverage. The table updates with each selection.

*RepeatColor.java* Uses alternating background colors for rows or columns, to improve readability.

*TextureTable.java* Uses a customized border type that tiles a background image onto table cells.

*UpdateStyle.java* Continuously updates the background color property

for a particular table row’s style.

* + 1. 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](#_bookmark340) 4. These examples are part of the examples.table.cell package, and are found in the *examples/table/cell* directory:

*CurrencyTable.java* Uses CurrencyRenderer to take all data, regardless of its

original type, and render it as dollar values in the table.

*Histogram.java* Uses a custom cell renderer to take randomized integer data, and render them as horizontal bars in a table.

*MoneyTable.java* Uses a custom cell editor and a custom data type.

*TriangleTable.java* A demonstration of a non-text based editor and

renderer for Integer and Polygon types.

*WordWrap.java* Uses a word wrapping renderer to handle long String

data in cells.

* + 1. 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](#_bookmark586) 7. These examples are part of the examples.table.listeners package, and are found in the *examples/table/listeners* directory:

*BooleanDisplay.java* Uses the JCCellDisplay listener to intercept the display

event and change it before rendering (changes the boolean from true/false to yes/no).

*CancelEdit.java* Shows how to use JCTableDataEdit and JCTableDataListener to cancel active edits when the data source changes.

*DoubleClickEdit.java* Demonstrates the use of JCEditCell 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.

*EditCell.java* Demonstrates the use of JCEditCell events by placing a message in the table’s pane, telling you which column’s cell is being edited.

*ResizeCell.java* Uses JCResizeCell 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.

*SelectListener.java* Uses JCSelectListener 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.

*SkipNavigation.java* 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.

*Sorter.java* Demonstrates column sorting by performing a String and numerical sort on a column of integers.

*TwoTables.java* JCScroll events are used to sync the horizontal scrolling for two separate tables.

* + 1. 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](#_bookmark490) 6. These examples are part of the examples.table.interactions package, and are found in the *examples/table/interactions* directory:

*ColumnLabelPopUp.java* Shows how to use column labels as tool tips.

*DragDrop.java* 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.

*ExcelTableExample.java* Demonstrates how to copy and paste selected cells from

a JCTable to a Microsoft Excel spreadsheet and vice- versa.

*TableAutoColumnResize.java* Depicts how to emulate JTable auto column resizing in

JCTable.

*TraverseOnEnter.java* Displays how cell traversal (across rows) can optionally

be handled by using the Enter key.

* + 1. 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:

*DynamicTest.java* This example table continually updates the data it

displays, as its data source’s values randomly change.

*DynamicTest2.java* This example is the same as *DynamicTest.java*, except

that it uses AbstractDataSource.

*FileData.java* Demonstrates how to load data from an external file by using JCFileDataSource. Both a CSV and table format file are loaded and displayed side by side.

*FontList.java* Uses the getFont() 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.

*Pivot.java* Shows how to create a data source instance that can transpose itself. You can use a customized data source or create your own.

*StaticEditableTest.java* A basic, custom data source that is built on editable

String values.

*StaticTest.java* This example is the same as the previous example, except that the table data is not editable.

*XMLFileData.java* Demonstrates how to load XML-formatted data from

an external file by using JCFileDataSource.

Please note that in order to run this example, you will need to have the *jaxp.jar* and *crimson.jar* files in your CLASSPATH. For more information, please see [Loading Data from an XML Source, in Chapter](#_bookmark313) 3.

*XMLTableModelData.java* Demonstrates how to load XML-formatted data into a Swing TableModel class.

Please note that in order to run this example, you will need to have the *jaxp.jar* and *crimson.jar* files in your CLASSPATH. For more information, please see [Loading Data from an XML Source, in Chapter](#_bookmark313) 3.

* + 1. 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.

###### JClass LiveTable Demos

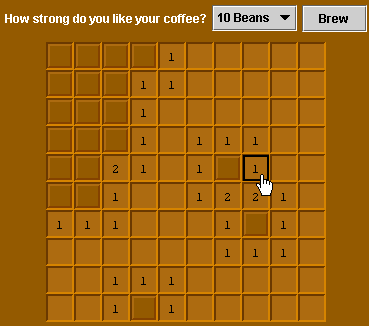
The JClass LiveTable demos showcase different types of complete table solutions. They combine several table concepts that are explained in this manual, including cell spanning, user interaction, and editors and renderers.

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*

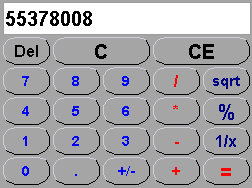
demos.table.beanSweeper



*Calculator.java*

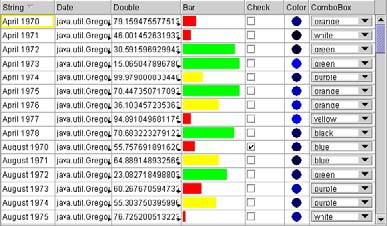
demos.table.calculator

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.

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*

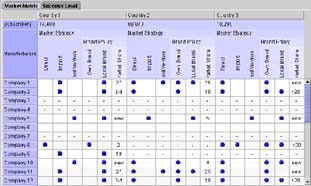
demos.table.customCells



*Matrix.java*

demos.table.matrix

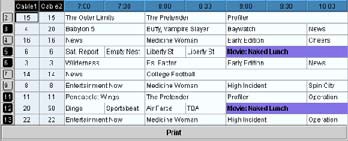
An effective demonstration of using custom cell renderers, which take various data types and converts them to the standard desired formats. Also, column sorting and the JClass field component integration are showcased.



*PrimeTime.java*

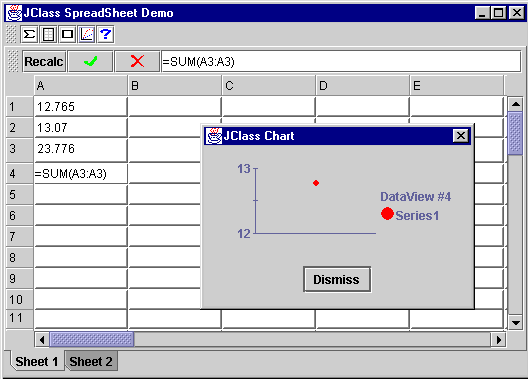
demos.table.primetime

This information matrix shows how a custom JCCellRenderer 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.

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

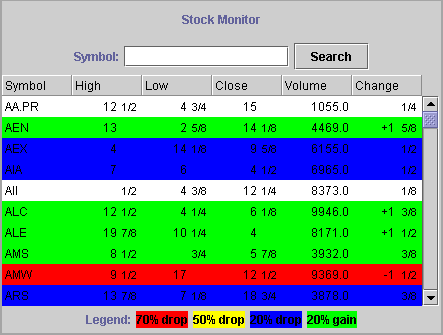


*Stocks.java*

demos.table.stocks

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.

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.

**A**

about property [172](#_bookmark679), [210](#_bookmark843)

Abstract Windowing Toolkit, see AWT [23](#_bookmark43)

AbstractDataSource [84](#_bookmark335)

adding labels [43](#_bookmark136)

alignment cells [61](#_bookmark220)

changing, tutorial [24](#_bookmark46)

AllowCellResize property [126](#_bookmark510), [172](#_bookmark680), [198](#_bookmark745), [210](#_bookmark844)

effect on mouse pointers [137](#_bookmark582)

AllowResizeBy property [172](#_bookmark681), [198](#_bookmark746), [210](#_bookmark845)

API [11](#_bookmark9)

programming [35](#_bookmark76)

setting properties [168](#_bookmark667)

applets [221](#_bookmark900)

JarMaster [221](#_bookmark903)

applications distributing [221](#_bookmark900)

editWidthPolicy property [174](#_bookmark687)

focusColor property [175](#_bookmark688)

focusIndicator property [175](#_bookmark689)

frameBorderType property [175](#_bookmark690)

frameBorderWidth property [175](#_bookmark692)

frozenCellLayout property [175](#_bookmark693)

LabelLayout property [176](#_bookmark694)

leftColumn property [176](#_bookmark695)

marginHeight property [176](#_bookmark696)

marginWidth property [176](#_bookmark697)

minCellVisibility property [177](#_bookmark698)

sBLayout property [177](#_bookmark699)

selectedBackground property [178](#_bookmark700)

selectedForeground property [178](#_bookmark701)

selectIncludeLabels property [179](#_bookmark702)

selectionPolicy property [179](#_bookmark703)

spannedCells property [179](#_bookmark705)

styles property [180](#_bookmark706)

topRow property [181](#_bookmark707)

#### Index

JarMaster [221](#_bookmark903)

attaching scrollbars [41](#_bookmark120)

autoEdit property [198](#_bookmark747), [210](#_bookmark846)

automatic scrolling [128](#_bookmark531)

autoScroll property [173](#_bookmark682), [198](#_bookmark748), [210](#_bookmark847)

AWT

color constants [23](#_bookmark44)

font styles, tutorial [25](#_bookmark49)

image file formats supported [66](#_bookmark235)

##### B

background colors [23](#_bookmark42), [60](#_bookmark215)

repeating [60](#_bookmark217)

property [60](#_bookmark215), [208](#_bookmark825)

basic table [18](#_bookmark23)

Bean [167](#_bookmark654)

LiveTable [168](#_bookmark670), [171](#_bookmark678)

about property [172](#_bookmark679)

allowCellResize property [172](#_bookmark680)

allowResizeBy property [172](#_bookmark681)

autoScroll property [173](#_bookmark682)

CellBorderWidth property [173](#_bookmark683)

CellSize property [173](#_bookmark684)

data property [173](#_bookmark685)

editHeightPolicy property [174](#_bookmark686)

LiveTable, changing property editor table size [171](#_bookmark676)

LiveTable, property

editors [169](#_bookmark672)

LiveTable, selecting cell [169](#_bookmark673) LiveTable, selecting labels [170](#_bookmark675) setting properties [168](#_bookmark670)

BeanBox [167](#_bookmark656)

Beans Development Kit [167](#_bookmark657)

borders

colors [46](#_bookmark156)

component [46](#_bookmark151)

custom [64](#_bookmark226)

frame attributes [170](#_bookmark674), [179](#_bookmark704)

sides

specifying [65](#_bookmark229)

table frame [46](#_bookmark154)

type [62](#_bookmark224)

width [45](#_bookmark144)

built-in styles, using and modifying [58](#_bookmark209)

##### C

CellBorder property [208](#_bookmark826)

CellBorderColor property [208](#_bookmark827)

CellBorderColorMode property [208](#_bookmark828)

CellBorderSides property [65](#_bookmark229), [208](#_bookmark829)

CellBorderType property [27](#_bookmark55)

231

CellBorderWidth property [27](#_bookmark55), [45](#_bookmark145), [198](#_bookmark749)

cellBorderWidth property [173](#_bookmark683), [210](#_bookmark848)

CellInfo interface [107](#_bookmark385)

cells

adding color to one cell, tutorial [26](#_bookmark53)

alignment [61](#_bookmark220)

alignment, tutorial [24](#_bookmark46)

area

spacing from labels [44](#_bookmark140)

border sides [65](#_bookmark229)

border types [62](#_bookmark224)

border width [45](#_bookmark144)

border, IDE tutorial [186](#_bookmark715)

borders, tutorial [27](#_bookmark55)

CellEditor interface [87](#_bookmark344)

CellInfo interface [107](#_bookmark385)

CellRenderer interface [87](#_bookmark344)

clipping arrows [51](#_bookmark177)

controlling editor size [50](#_bookmark173) controlling selection at runtime [134](#_bookmark567) current [36](#_bookmark82)

custom borders [64](#_bookmark226)

customizing traversal [123](#_bookmark498)

default editors [97](#_bookmark371)

default selection [131](#_bookmark552)

default traversal [123](#_bookmark496)

definition [36](#_bookmark81)

determining visibility [129](#_bookmark539)

dimensions [51](#_bookmark178)

displaying [87](#_bookmark341), [139](#_bookmark590)

displaying images [66](#_bookmark234) displaying multiple lines [54](#_bookmark191) editable, tutorial [29](#_bookmark62)

editing [36](#_bookmark83), [71](#_bookmark274), [87](#_bookmark341), [97](#_bookmark369), [142](#_bookmark597)

default [88](#_bookmark349)

editors

and CellInfo interface [107](#_bookmark385)

controlling size [50](#_bookmark173)

creating [100](#_bookmark378)

default [97](#_bookmark371)

defined [98](#_bookmark372)

getting reserved keys [101](#_bookmark379)

handling events [105](#_bookmark382)

key control [106](#_bookmark383) mapping a data type [99](#_bookmark376) registering [119](#_bookmark485)

reserving keys [101](#_bookmark379), [106](#_bookmark383)

setting for a series [99](#_bookmark374)

subclassing [101](#_bookmark380)

writing [102](#_bookmark381)

fonts [62](#_bookmark222)

forcing traversal [124](#_bookmark502)

image alignment [61](#_bookmark219)

image layout [66](#_bookmark239)

interactive traversal [125](#_bookmark504)

making visible [129](#_bookmark540)

margins [45](#_bookmark147)

mathematical operation [120](#_bookmark487)

maximum height/width [54](#_bookmark189)

minimum height/width [54](#_bookmark189)

minimum visibility [124](#_bookmark500)

multiline [54](#_bookmark191)

newline characters [54](#_bookmark192) preset styles, selection [40](#_bookmark106) range

referencing [37](#_bookmark90)

referencing, all [38](#_bookmark95)

referencing, one [37](#_bookmark88)

removing a selection range [134](#_bookmark565)

renderers [92](#_bookmark363)

component based [95](#_bookmark367)

creating [92](#_bookmark363)

data type [91](#_bookmark357)

mapping [91](#_bookmark357)

mapping a data type [91](#_bookmark357)

registering [119](#_bookmark485)

setting [91](#_bookmark355)

subclassing [93](#_bookmark364)

writing [93](#_bookmark365)

rendering [36](#_bookmark83), [89](#_bookmark352)

default [88](#_bookmark349)

reserving keys for editors [101](#_bookmark379)

resizing [126](#_bookmark516) selected cell list [133](#_bookmark561) selecting [153](#_bookmark620)

selecting ranges [133](#_bookmark563)

selection

customizing [132](#_bookmark557)

selection colors [42](#_bookmark131)

selection, row and column labels [133](#_bookmark559)

selection, tutorial [30](#_bookmark64)

setting dimensions [51](#_bookmark181)

setting properties [39](#_bookmark98)

setting renderers [91](#_bookmark355) setting selection colors [42](#_bookmark132) setting values [77](#_bookmark310)

size

absolute [52](#_bookmark183)

character height/width [51](#_bookmark181)

character width/height [51](#_bookmark179)

pixel width/height [52](#_bookmark183)

variable [53](#_bookmark185)

changing to fixed values [53](#_bookmark187)

size, tutorial [28](#_bookmark57)

spacing [27](#_bookmark55)

spanning [66](#_bookmark244)

specific data types [88](#_bookmark350)

styles [55](#_bookmark199)

built-in, modifying [58](#_bookmark209)

built-in, using [58](#_bookmark209)

changing default [56](#_bookmark204)

defining [56](#_bookmark204)

getting and setting [56](#_bookmark202)

parent styles [57](#_bookmark207)

pluggable look and feel [59](#_bookmark212)

properties [55](#_bookmark201)

retrieving from table [57](#_bookmark206)

setting properties [39](#_bookmark99)

tutorial [22](#_bookmark38)

text alignment [61](#_bookmark219) thickness, IDE tutorial [186](#_bookmark715) traversal [123](#_bookmark494)

traversing [159](#_bookmark632)

values

setting [77](#_bookmark310)

variable dimensions [53](#_bookmark185)

changing to fixed values [53](#_bookmark187)

cellSize property [173](#_bookmark684), [210](#_bookmark849)

CellStyleModel [22](#_bookmark39), [58](#_bookmark210)

central registry [119](#_bookmark485)

change values [84](#_bookmark333)

changing default cells styles [56](#_bookmark204)

character determining cells

size [51](#_bookmark179)

height [51](#_bookmark181)

width [51](#_bookmark181)

CharHeight property [51](#_bookmark181), [198](#_bookmark750)

tutorial [28](#_bookmark58)

CharWidth property [51](#_bookmark181), [198](#_bookmark751)

tutorial [28](#_bookmark58)

ClassCastException [119](#_bookmark481)

clip arrows tutorial [20](#_bookmark28)

clip hints displaying [65](#_bookmark231)

ClipHints property [65](#_bookmark231), [208](#_bookmark830)

clipping arrows [51](#_bookmark177)

image [65](#_bookmark232)

text [65](#_bookmark232)

colors [60](#_bookmark214)

AWT constants [23](#_bookmark44)

background [60](#_bookmark215)

colorname values [213](#_bookmark883)

focus rectangle [41](#_bookmark113)

foreground [60](#_bookmark215)

repeating [60](#_bookmark216)

RGB [213](#_bookmark885)

RGB color value list [213](#_bookmark886)

selection [132](#_bookmark554)

setting [60](#_bookmark215)

setting, tutorial [22](#_bookmark38)

colors, selection [42](#_bookmark131)

setting [42](#_bookmark132)

ColumnHidden property [54](#_bookmark196), [198](#_bookmark752)

ColumnLabelDisplay property [198](#_bookmark753)

tutorial [22](#_bookmark36)

ColumnLabelOffset property [44](#_bookmark141), [198](#_bookmark754)

ColumnLabelPlacement property [43](#_bookmark138), [199](#_bookmark755)

columns

adding [75](#_bookmark299), [86](#_bookmark337)

adding labels, IDE tutorial [183](#_bookmark713) controlling resizing [126](#_bookmark518) default resizing behavior [126](#_bookmark513) deleting [76](#_bookmark303)

determining number [48](#_bookmark159)

determining visibility [129](#_bookmark541)

disallowing resizing [126](#_bookmark516)

displaying [49](#_bookmark164)

dragging [134](#_bookmark569)

freezing [49](#_bookmark168)–[50](#_bookmark170)

hiding [54](#_bookmark194)

labels [43](#_bookmark136), [70](#_bookmark265), [137](#_bookmark576)

displaying [22](#_bookmark36)

labels, placement [43](#_bookmark137)

making visible [129](#_bookmark540)

moving [76](#_bookmark306)

placement of frozen [50](#_bookmark171)

referencing, all [37](#_bookmark89)–[38](#_bookmark92)

referencing, entire [38](#_bookmark94)

referencing, one [37](#_bookmark91)

removing [86](#_bookmark337)

resizing [126](#_bookmark507)

resizing all at once [127](#_bookmark520) resizing with labels [127](#_bookmark521) selecting labels [133](#_bookmark559)

set as left [42](#_bookmark123)

setting the number [74](#_bookmark297)

sorting [135](#_bookmark571), [137](#_bookmark576)

sorting frozen [135](#_bookmark572)

sorting multiple [135](#_bookmark574)

sorting, tutorial [31](#_bookmark69)

specifying labels [75](#_bookmark298)

swapping [49](#_bookmark165)

visible, getting [48](#_bookmark161)

visible, setting [48](#_bookmark161)

width

pixel value [51](#_bookmark179)

width property [51](#_bookmark176)

width, setting [51](#_bookmark178)

ColumnSelection property [199](#_bookmark756)

ColumnTrigger property and dragging [134](#_bookmark569)

and sorting [137](#_bookmark576)

com.klg.jclass.table.beans.LiveTable [210](#_bookmark842)

com.klg.jclass.table.CellStyleModel [208](#_bookmark824)

com.klg.jclass.table.JCTable [198](#_bookmark744)

com.klg.jclass.util.formulae [109](#_bookmark390)

component borders [46](#_bookmark151)

Component property [199](#_bookmark757)

ComponentBorderWidth property [46](#_bookmark152), [199](#_bookmark758)

context [37](#_bookmark87)

creating a cell editor [100](#_bookmark378) creating cell renderers [92](#_bookmark363) CSV [71](#_bookmark268)

current cell [36](#_bookmark82)

definition [36](#_bookmark82)

current context [37](#_bookmark87)

Cursor property [199](#_bookmark759)

cursor type [41](#_bookmark116)

tracking [41](#_bookmark116)

##### D

data

caching [73](#_bookmark288)

cell editor [88](#_bookmark346)

cell renderer [88](#_bookmark346)

data source [70](#_bookmark261)

data storage [69](#_bookmark254)

editing [71](#_bookmark272)

format, detection [71](#_bookmark266) from an input stream [73](#_bookmark282) getting from database [73](#_bookmark286) getting into a table [70](#_bookmark261) handling [69](#_bookmark253)

property [173](#_bookmark685), [210](#_bookmark850)

storing [72](#_bookmark280)–[73](#_bookmark284)

Swing TableModel objects [74](#_bookmark291)

updating dynamically [82](#_bookmark330)

data source

adding and removing listeners [71](#_bookmark270), [77](#_bookmark311)

and table size [70](#_bookmark264)

communication with the table [70](#_bookmark257)

creating [80](#_bookmark327)

editable [71](#_bookmark272)

editable, tutorial [29](#_bookmark62)

event listeners [71](#_bookmark269)

examples [226](#_bookmark916)

IDE tutorial [182](#_bookmark711)

JCVectorDataSource [72](#_bookmark279)

model-view-controller [69](#_bookmark254)

object [69](#_bookmark254)

retrieving data [70](#_bookmark261) setting cell values [77](#_bookmark310) setting properties [74](#_bookmark295) stock data properties [74](#_bookmark295) stock data sources [71](#_bookmark277) tutorial [19](#_bookmark25)

data type

cell renderers [91](#_bookmark357)

mapping [91](#_bookmark359)

mapping to a cell editor [99](#_bookmark376)

database

getting data [73](#_bookmark286)

default

cell editors [97](#_bookmark371)

scrolling [128](#_bookmark527)

defaultCellStyle [59](#_bookmark212)

defaultLabelStyle [59](#_bookmark212)

deleting rows and columns [76](#_bookmark303)

demos [228](#_bookmark919)

BeanSweeper.java [228](#_bookmark920)

Calculator.java [228](#_bookmark921)

CustomCells.java [229](#_bookmark922)

Matrix.java [229](#_bookmark923)

overview [223](#_bookmark906)

PrimeTime.java [229](#_bookmark924)

SpreadSheet.java [230](#_bookmark925)

Stocks.java [230](#_bookmark926)

destination parameter [76](#_bookmark308)

dimensions cell [51](#_bookmark178) displaying

cells [139](#_bookmark591)

clip hints [65](#_bookmark231)

rows and columns [49](#_bookmark164)

distributing [221](#_bookmark900)

applets and applications [221](#_bookmark900)

JarMaster [221](#_bookmark903)

dragging rows and columns [134](#_bookmark569)

drawBackground [64](#_bookmark227)

drawing cells [88](#_bookmark350)

dynamically updating data [82](#_bookmark330)

##### E

editable cells [71](#_bookmark274)

Editable property [71](#_bookmark274), [208](#_bookmark831)

EditableTableDataModel [71](#_bookmark273)

EditHeightPolicy property [199](#_bookmark760)

editHeightPolicy property [50](#_bookmark173), [174](#_bookmark686), [210](#_bookmark851)

editing cells [71](#_bookmark274), [88](#_bookmark350), [97](#_bookmark369)

editors

LiveTable [169](#_bookmark672)

setting [99](#_bookmark374)

EditWidthPolicy property [50](#_bookmark173)

editWidthPolicy property [174](#_bookmark687), [199](#_bookmark761), [210](#_bookmark852)

evaluate

method in MathValue [111](#_bookmark404)

event listeners [139](#_bookmark593)

adding and removing [77](#_bookmark311)

cell display [140](#_bookmark595)

data source [71](#_bookmark269)

entering cells [142](#_bookmark598)

JCCellDisplayListener [139](#_bookmark594)

JCEnterCellListener [142](#_bookmark598)

JCPaintListener [144](#_bookmark600)

JCPrintListener [145](#_bookmark604)

JCResizeListener [146](#_bookmark610)

JCScrollListener [149](#_bookmark616)

painting [144](#_bookmark600)

printing [145](#_bookmark604)

resizing [146](#_bookmark610)

events [119](#_bookmark476), [139](#_bookmark587)

cell display events [140](#_bookmark595)

cell editors [105](#_bookmark382)

editing cells [142](#_bookmark597)

JCCellDisplayEvent [139](#_bookmark592)

JCEditCellEvent [142](#_bookmark597)

JCPaintEvent [144](#_bookmark600)

JCPrintEvent [145](#_bookmark604)

JCResizeEvent [146](#_bookmark608)

JCScrollEvent [149](#_bookmark617)

JCSortEvent [155](#_bookmark623)

JCTraverseCellEvent [158](#_bookmark627)–[159](#_bookmark631)

painting [144](#_bookmark600)

printing [145](#_bookmark604)

resizing [146](#_bookmark608)

scrolling [149](#_bookmark616)

sorting [155](#_bookmark623)

summary [193](#_bookmark725)

TableListenerPropagator [119](#_bookmark477)

traversal [158](#_bookmark627)–[159](#_bookmark631)

examples [223](#_bookmark909)

advanced [227](#_bookmark917)

cell [224](#_bookmark913)

cell style [224](#_bookmark912)

data source [226](#_bookmark916)

introductory [223](#_bookmark910)

overview [223](#_bookmark906)

table interaction [226](#_bookmark915)

table layout [224](#_bookmark911)

table listener examples [225](#_bookmark914)

exceptions [119](#_bookmark479)

ClassCastException [119](#_bookmark481)

OperandMismatchException [119](#_bookmark480)

expression [111](#_bookmark398)

interface [111](#_bookmark399)

lists [118](#_bookmark471), [120](#_bookmark488)

MathExpressionList [118](#_bookmark472)

QueryExpressionList [118](#_bookmark473)

TableExpressionList [118](#_bookmark474)

mathematical [109](#_bookmark391)

references [120](#_bookmark488)

##### F

feature overview [9](#_bookmark3)

fixed values [53](#_bookmark187)

focus rectangle color [41](#_bookmark113)

focusColor property [175](#_bookmark688), [199](#_bookmark762), [210](#_bookmark856)

focusIndicator property [175](#_bookmark689), [200](#_bookmark763), [210](#_bookmark857)

FocusRectColor property [41](#_bookmark114)

Font property [200](#_bookmark764), [208](#_bookmark832)

font styles

AWT [25](#_bookmark49)

fonts

cells [62](#_bookmark222)

labels [62](#_bookmark222)

matched by AWT [25](#_bookmark51)

names [218](#_bookmark888)

point size [218](#_bookmark890)

setting, tutorial [24](#_bookmark48)

size, tutorial [25](#_bookmark50)

style constants [218](#_bookmark889)

footers

printing [164](#_bookmark647)

foreground colors [23](#_bookmark42), [60](#_bookmark215)

repeating [60](#_bookmark217)

property [60](#_bookmark215)

Foreground property [200](#_bookmark765), [208](#_bookmark833)

format, RGB [213](#_bookmark886)

formulae hierarchy [109](#_bookmark395)

formulae package [109](#_bookmark392)

formulas

adding to JClass LiveTable [109](#_bookmark387)

using in JClass LiveTable [119](#_bookmark483)

frame

border [170](#_bookmark674), [179](#_bookmark704)

FrameBorder property [46](#_bookmark154), [200](#_bookmark766)

FrameBorderType property in IDEs [175](#_bookmark691)

frameBorderType property [175](#_bookmark690), [210](#_bookmark853)

frameBorderWidth property [46](#_bookmark155), [175](#_bookmark692), [200](#_bookmark767), [210](#_bookmark854)

freezing columns [50](#_bookmark170)

rows [50](#_bookmark170)

frozen

column placement [50](#_bookmark171)

columns [49](#_bookmark168)

and sorting [135](#_bookmark572)

row placement [50](#_bookmark171)

rows [49](#_bookmark168)

frozenCellLayout property [175](#_bookmark693), [210](#_bookmark855)

FrozenColumnPlacement property [200](#_bookmark768)

FrozenColumns property [200](#_bookmark769)

properties

FrozenColumns [49](#_bookmark168)

FrozenRowPlacement property [200](#_bookmark770)

FrozenRows property [49](#_bookmark169), [200](#_bookmark771)

##### G

get method [197](#_bookmark741)

getDataFormat

method in MathValue [111](#_bookmark405)

getValueAt

method in MathMatrix [113](#_bookmark432)

method in MathVector [113](#_bookmark422)

GIF

image file formats supported [66](#_bookmark236)

global table properties [40](#_bookmark111)

##### H

headers

printing [164](#_bookmark647)

hiding

columns [54](#_bookmark194)

rows [54](#_bookmark194)

HorizontalAlignment property [66](#_bookmark240), [208](#_bookmark834)

HorizSBAttachment property [41](#_bookmark122), [201](#_bookmark772)

HorizSBDisplay property [129](#_bookmark534), [201](#_bookmark773)

HorizSBOffset property [201](#_bookmark774)

HorizSBPosition property [201](#_bookmark775)

HorizSBTrack property [201](#_bookmark776)

HorizSBTrackRow property [201](#_bookmark777)

##### I

IDEs [167](#_bookmark654)

setting properties [40](#_bookmark102), [168](#_bookmark665)

tutorial [181](#_bookmark709)

image

alignment in cell [61](#_bookmark219) clipping [65](#_bookmark232) displaying in cells [66](#_bookmark234) formats supported [66](#_bookmark237)

layout [66](#_bookmark241)

layout in cell [66](#_bookmark239)

inch-to-pixel conversion factor [164](#_bookmark644)

inheritance hierarchy [219](#_bookmark894)

input stream

getting data from [73](#_bookmark282)

Integrated Development Environment (IDE) [168](#_bookmark663)

interactivity [123](#_bookmark491)

IDE tutorial [187](#_bookmark717)

tutorial [29](#_bookmark60)

introduction

JClass LiveTable [9](#_bookmark2)

##### J

JAR [221](#_bookmark902)

JarMaster [221](#_bookmark902)

JarMaster [221](#_bookmark903)

JAR [221](#_bookmark902)

JavaBeans features of [167](#_bookmark659)

JCCachedDataSource [73](#_bookmark289)

JCCellBorder class [62](#_bookmark225)

JCCellDisplayEvent [139](#_bookmark592), [193](#_bookmark727)

JCCellDisplayListeners [139](#_bookmark594)

JCCellRange

in cell selection [133](#_bookmark561)

JCCellRenderer [91](#_bookmark360)

JCCellStyle [22](#_bookmark40), [56](#_bookmark205), [58](#_bookmark211)

JCComponentCellRenderer [95](#_bookmark367)

JCEditCellEvent [193](#_bookmark728)

JCExpressionCellRenderer [88](#_bookmark347)

JCInputStream [71](#_bookmark267)

JCInputStreamDataSource [73](#_bookmark283)

JClass JarMaster [221](#_bookmark903)

JClass LiveTable, introduction [9](#_bookmark2)

jclass.cell package [87](#_bookmark345)

JCListTable [40](#_bookmark104)

JCPaintEvent [144](#_bookmark601), [193](#_bookmark729)

JCPaintListener [144](#_bookmark602)

JCPrintEvent [145](#_bookmark605), [194](#_bookmark730)

JCPrintListener [145](#_bookmark606)

JCPrintPreview [165](#_bookmark650)

JCPrintTable [163](#_bookmark638), [166](#_bookmark651)

JCResizeCellEvent [126](#_bookmark519), [146](#_bookmark609), [194](#_bookmark731)

JCResizeCellListener [146](#_bookmark611)

JCResizeCellMotionListener [147](#_bookmark612)

JCScrollEvent [130](#_bookmark547), [149](#_bookmark615), [194](#_bookmark732)

JCScrollListener [130](#_bookmark548), [149](#_bookmark618)

JCSelectEvent [195](#_bookmark733)

JCSelectListener [153](#_bookmark621)

JCSortEvent [155](#_bookmark624), [195](#_bookmark734)

JCSortListener [155](#_bookmark625)

JCTable [35](#_bookmark77)

JCTableDataEvent [158](#_bookmark628), [195](#_bookmark735)

JCTableDataListener [158](#_bookmark629)

JCTraversalCellEvent [159](#_bookmark631)

JCTraverseCellEvent [125](#_bookmark505), [195](#_bookmark736)

JCTraverseCellListener [159](#_bookmark633)

JCVectorDataSource [19](#_bookmark26)

editing [72](#_bookmark279)

JPEG

image file formats supported [66](#_bookmark238)

JumpScroll property [202](#_bookmark778), [210](#_bookmark858)

##### K

keys

control [106](#_bookmark383)

reserving for cell editors [101](#_bookmark379), [106](#_bookmark383)

##### L

label parameter [75](#_bookmark301)

labelLayout property [176](#_bookmark694), [210](#_bookmark859)

labels

adding [21](#_bookmark35), [43](#_bookmark136)

border sides [65](#_bookmark229)

border width [45](#_bookmark144)

column, IDE tutorial [183](#_bookmark713)

columns [70](#_bookmark265)

custom borders [64](#_bookmark226)

definition [36](#_bookmark79)

displaying, tutorial [22](#_bookmark36)

fonts [62](#_bookmark222)

formatting [21](#_bookmark35)

layout, IDE tutorial [185](#_bookmark714)

margins [45](#_bookmark147)

offset from table [44](#_bookmark142)

placement [43](#_bookmark137)

preset styles, selection [40](#_bookmark107) referencing, all [38](#_bookmark93) referencing, all columns [38](#_bookmark92) referencing, all rows [38](#_bookmark92) referencing, column [37](#_bookmark91)

referencing, row [37](#_bookmark91)

resize, tutorial [30](#_bookmark66)

rows [70](#_bookmark265)

selecting [133](#_bookmark559)

setting properties [39](#_bookmark98) spacing from cell area [44](#_bookmark140) spanning [66](#_bookmark244)

specifying row and column [75](#_bookmark298)

using for resizing [127](#_bookmark522)

leftColumn property [42](#_bookmark124), [129](#_bookmark538), [176](#_bookmark695), [202](#_bookmark779), [210](#_bookmark860)

list of cell editors [98](#_bookmark372)

listeners [119](#_bookmark476), [139](#_bookmark587)

examples [225](#_bookmark914)

management [84](#_bookmark334)

scroll [130](#_bookmark546)

TableListenerPropagator [119](#_bookmark477)

LiveTable

general classes, inheritance [219](#_bookmark896)

LiveTable Bean about property [172](#_bookmark679)

allowCellResize property [172](#_bookmark680)

allowResizeBy property [172](#_bookmark681)

autoScroll property [173](#_bookmark682)

CellBorderWidth property [173](#_bookmark683)

CellSize property [173](#_bookmark684)

data property [173](#_bookmark685)

editHeightPolicy property [174](#_bookmark686)

editWidthPolicy property [174](#_bookmark687)

focusColor property [175](#_bookmark688)

focusIndicator property [175](#_bookmark689)

frameBorderType property [175](#_bookmark690)

frameBorderWidth property [175](#_bookmark692)

frozenCellLayout property [175](#_bookmark693)

LabelLayout property [176](#_bookmark694)

leftColumn property [176](#_bookmark695)

marginHeight property [176](#_bookmark696)

marginWidth property [176](#_bookmark697)

minCellVisibility property [177](#_bookmark698)

properties [171](#_bookmark678)

sBLayout property [177](#_bookmark699)

selectedBackground property [178](#_bookmark700)

selectedForeground property [178](#_bookmark701)

selectIncludeLabels property [179](#_bookmark702)

selectionPolicy property [179](#_bookmark703)

setting properties [168](#_bookmark670)

spannedCells property [179](#_bookmark705)

styles property [180](#_bookmark706)

topRow property [181](#_bookmark707)

LiveTable Data

classes, inheritance [220](#_bookmark897)

loading data [77](#_bookmark313)

##### M

mapping [91](#_bookmark359)

a data type [91](#_bookmark357)

marginHeight property [45](#_bookmark148), [176](#_bookmark696), [202](#_bookmark780), [210](#_bookmark861)

margins

cell and label [45](#_bookmark147)

setting [45](#_bookmark149)

marginWidth property [45](#_bookmark149), [176](#_bookmark697), [202](#_bookmark781), [210](#_bookmark862)

math values [111](#_bookmark401)

mathematical expressions [109](#_bookmark391)

Mathematical operations [115](#_bookmark443)

binary [115](#_bookmark451)

range of cells [120](#_bookmark487)

unary [115](#_bookmark444)

MathExpressionList [118](#_bookmark472)

MathMatrix [113](#_bookmark429)

constructors [113](#_bookmark430)

getValueAt method [113](#_bookmark432)

matrixValue method [113](#_bookmark433)

methods [113](#_bookmark431)

numberValue method [113](#_bookmark434)

setValueAt method [114](#_bookmark435)

toString method [114](#_bookmark436)

VectorValue method [114](#_bookmark437)

MathScalar [112](#_bookmark411)

constructors [112](#_bookmark412)

matrixValue method [112](#_bookmark414)

methods [112](#_bookmark413)

numberValue method [112](#_bookmark415)

toString method [112](#_bookmark416)

vectorValue method [112](#_bookmark417)

MathValue [109](#_bookmark393)

class [111](#_bookmark402)

evaluate method [111](#_bookmark404)

getDataFormat method [111](#_bookmark405)

matrixValue method [111](#_bookmark406)

methods [111](#_bookmark403)

numberValue method [111](#_bookmark407)

setDataFormat method [111](#_bookmark408)

vectorValue method [111](#_bookmark409)

MathVector [112](#_bookmark419)

constructors [113](#_bookmark420)

getValueAt method [113](#_bookmark422)

matrixValue method [113](#_bookmark423)

methods [113](#_bookmark421)

numberValue method [113](#_bookmark424)

setValueAt method [113](#_bookmark425)

toString method [113](#_bookmark426)

vectorValue method [113](#_bookmark427)

matrixValue

method in MathMatrix [113](#_bookmark433) method in MathScalar [112](#_bookmark414) method in MathValue [111](#_bookmark406) method in MathVector [113](#_bookmark423)

MaxHeight property [202](#_bookmark782)

maximum pixel height [54](#_bookmark189)

width [54](#_bookmark189)

MaxWidth property [202](#_bookmark783)

methods accessor [197](#_bookmark741)

minCellVisibility property [177](#_bookmark698), [202](#_bookmark784), [210](#_bookmark863)

MinHeight property [202](#_bookmark785)

minimum

cell visibility [124](#_bookmark500)

pixel height [54](#_bookmark189)

pixel width [54](#_bookmark189)

MinWidth property [202](#_bookmark786)

model-view-controller [19](#_bookmark24), [69](#_bookmark255)

data source [69](#_bookmark254)

mouse pointers custom [137](#_bookmark581)

disabling tracking [137](#_bookmark583)

multiline [54](#_bookmark191)

headers, spanning [68](#_bookmark248)

multiple

columns, sorting [135](#_bookmark574)

lines in cells [54](#_bookmark191)

MVC, see model-view-controller [69](#_bookmark255)

##### N

newline character

and Multiline property [54](#_bookmark192)

num\_columns parameter [76](#_bookmark305)

num\_rows parameter [76](#_bookmark305)

numberValue

method in MathMatrix [113](#_bookmark434) method in MathScalar [112](#_bookmark415) method in MathValue [111](#_bookmark407) method in MathVector [113](#_bookmark424)

NumColumns property [48](#_bookmark160), [70](#_bookmark264)

NumRows property [48](#_bookmark160), [70](#_bookmark264)

**O**

offset of labels [44](#_bookmark142)

OperandMismatchException [119](#_bookmark480)

Operation class [114](#_bookmark439)

operations constructor [114](#_bookmark440)

mathematical [115](#_bookmark443)

binary [115](#_bookmark451)

unary [115](#_bookmark444)

methods [114](#_bookmark441)

reducing values [117](#_bookmark469)

operators Abs [115](#_bookmark445)

Add [115](#_bookmark453)

Average [115](#_bookmark454)

Ceiling [115](#_bookmark446)

Count [116](#_bookmark455)

Divide [116](#_bookmark456)

Floor [115](#_bookmark447)

GeometricMean [116](#_bookmark457)

in com.klg.jclass.util.formulae [115](#_bookmark452)

Max [116](#_bookmark458)

Median [116](#_bookmark459)

Min [116](#_bookmark460)

Multiply [116](#_bookmark461)

Power [116](#_bookmark462)

Product [116](#_bookmark463)

Root [115](#_bookmark448)

Round [115](#_bookmark449)

Sort [117](#_bookmark464)

StdDeviation [117](#_bookmark465)

Subtract [117](#_bookmark466)

Sum [117](#_bookmark467)

Trunc [115](#_bookmark450)

##### P

page layout page size [163](#_bookmark642)

setting, for printing [163](#_bookmark641)

page margins

setting, for printing [164](#_bookmark643)

page numbering

setting, for printing [164](#_bookmark645)

page resolution [164](#_bookmark644)

painting [144](#_bookmark600)

parent cell styles creating [57](#_bookmark207)

PixelHeight property [52](#_bookmark183), [202](#_bookmark787)

setting, tutorial [28](#_bookmark57) user row resizing [126](#_bookmark511) using to hide rows [54](#_bookmark195)

pixels

absolute height and width [52](#_bookmark183)

estimate [53](#_bookmark186)

maximum height/width [54](#_bookmark189)

minimum height/width [54](#_bookmark189)

variable height and width [53](#_bookmark185)

changing to fixed values [53](#_bookmark187) pixels-to-inch conversion factor [164](#_bookmark644) PixelWidth property [52](#_bookmark183), [203](#_bookmark788)

column resizing [126](#_bookmark512)

setting, tutorial [28](#_bookmark57)

using to hide columns [54](#_bookmark195)

pluggable look and feel (PLAF) [59](#_bookmark212)

popupMenuEnabled property [203](#_bookmark789), [210](#_bookmark864)

position parameter [75](#_bookmark300)–[76](#_bookmark304)

PreferredSize [95](#_bookmark366)

preset styles [40](#_bookmark105)

print preview [165](#_bookmark649)

printing [145](#_bookmark604), [163](#_bookmark639)

events [145](#_bookmark604)

headers and footers [164](#_bookmark647)

page layout [163](#_bookmark641)

page margins [164](#_bookmark643)

page numbering [164](#_bookmark645)

page resolution [164](#_bookmark644)

page size [163](#_bookmark642)

preview [165](#_bookmark649) programming the API [35](#_bookmark76) properties

about [210](#_bookmark843)

access in IDE [40](#_bookmark102)

accessor methods [197](#_bookmark741)

allowCellResize [198](#_bookmark745), [210](#_bookmark844)

effect on mouse pointers [137](#_bookmark582)

allowResizeBy [198](#_bookmark746), [210](#_bookmark845)

autoEdit [198](#_bookmark747), [210](#_bookmark846)

autoScroll [198](#_bookmark748), [210](#_bookmark847)

Background [208](#_bookmark825)

cell style [55](#_bookmark201)

CellBorder [208](#_bookmark826)

CellBorderColor [208](#_bookmark827)

CellBorderColorMode [208](#_bookmark828)

CellBorderSides [65](#_bookmark229), [208](#_bookmark829)

CellBorderWidth [45](#_bookmark145), [198](#_bookmark749), [210](#_bookmark848)

cellSize [210](#_bookmark849)

CharHeight [28](#_bookmark58), [51](#_bookmark181), [198](#_bookmark750)

CharWidth [28](#_bookmark58), [51](#_bookmark181), [198](#_bookmark751)

ClipHints [65](#_bookmark231), [208](#_bookmark830)

Color [213](#_bookmark885)

column width [51](#_bookmark176)

ColumnHidden [54](#_bookmark196), [198](#_bookmark752)

ColumnLabelDisplay [198](#_bookmark753)

ColumnLabelOffset [44](#_bookmark141), [198](#_bookmark754)

ColumnLabelPlacement [43](#_bookmark138), [199](#_bookmark755)

ColumnSelection [199](#_bookmark756)

ColumnTrigger [134](#_bookmark569), [137](#_bookmark576)

com.klg.jclass.table.beans.LiveTable [210](#_bookmark842)

com.klg.jclass.table.CellStyleModel [208](#_bookmark824)

com.klg.jclass.table.JCTable [198](#_bookmark744)

ComponentBorderWidth [46](#_bookmark152), [199](#_bookmark758)

Cursor [199](#_bookmark759)

data [210](#_bookmark850)

Editable [208](#_bookmark831)

EditHeightPolicy [50](#_bookmark173), [199](#_bookmark760), [210](#_bookmark851)

EditWidthPolicy [50](#_bookmark173), [199](#_bookmark761), [210](#_bookmark852)

focusColor [199](#_bookmark762), [210](#_bookmark856)

focusIndicator [200](#_bookmark763), [210](#_bookmark857)

Font [200](#_bookmark764), [208](#_bookmark832), [218](#_bookmark891)

Foreground [200](#_bookmark765), [208](#_bookmark833)

FrameBorder [46](#_bookmark154), [200](#_bookmark766)

frameBorderType [167](#_bookmark662), [210](#_bookmark853)

frameBorderWidth [46](#_bookmark155), [200](#_bookmark767), [210](#_bookmark854)

frozenCellLayout [210](#_bookmark855)

FrozenColumnPlacement [200](#_bookmark768)

FrozenColumns [200](#_bookmark769)

FrozenRowPlacement [200](#_bookmark770)

FrozenRows [49](#_bookmark169), [200](#_bookmark771)

getting [36](#_bookmark85)

global [40](#_bookmark111)

HorizontalAlignment [208](#_bookmark834)

HorizSBAttachment [201](#_bookmark772)

HorizSBDisplay [201](#_bookmark773)

HorizSBOffset [201](#_bookmark774)

HorizSBPosition [201](#_bookmark775)

HorizSBTrack [201](#_bookmark776)

HorizSBTrackRow [201](#_bookmark777)

JumpScroll [202](#_bookmark778), [210](#_bookmark858)

labelLayout [210](#_bookmark859)

leftColumn [42](#_bookmark124), [202](#_bookmark779), [210](#_bookmark860)

marginHeight [45](#_bookmark148), [202](#_bookmark780), [210](#_bookmark861)

marginWidth [45](#_bookmark149), [202](#_bookmark781), [210](#_bookmark862)

MaxHeight [202](#_bookmark782)

MaxWidth [202](#_bookmark783)

minCellVisibility [202](#_bookmark784), [210](#_bookmark863)

MinHeight [202](#_bookmark785)

MinWidth [202](#_bookmark786)

NumColumns [48](#_bookmark160), [70](#_bookmark264)

NumRows [48](#_bookmark160), [70](#_bookmark264)

PixelHeight [28](#_bookmark57), [52](#_bookmark183), [202](#_bookmark787) using to hide rows [54](#_bookmark195) PixelWidth [28](#_bookmark57), [52](#_bookmark183), [203](#_bookmark788)

using to hide columns [54](#_bookmark195)

popupMenuEnabled [203](#_bookmark789), [210](#_bookmark864)

RepaintEnabled [203](#_bookmark790)

RepeatBackground [208](#_bookmark835)

RepeatBackgroundColors [209](#_bookmark836)

RepeatForeground [209](#_bookmark837)

RepeatForegroundColors [209](#_bookmark838)

ResizeByLabelsOnly [30](#_bookmark67), [127](#_bookmark521)

resizeEven [127](#_bookmark520), [203](#_bookmark791), [210](#_bookmark865)

resizeInteractive [203](#_bookmark792), [211](#_bookmark866)

row height [51](#_bookmark176)

RowHidden [54](#_bookmark196), [203](#_bookmark793)

RowLabelDisplay [203](#_bookmark794)

RowLabelOffset [44](#_bookmark142), [203](#_bookmark795)

RowLabelPlacement [43](#_bookmark139), [203](#_bookmark796)

RowSelection [203](#_bookmark797)

RowTrigger [134](#_bookmark569)

sBLayout [211](#_bookmark867)

selectedBackground [42](#_bookmark133), [204](#_bookmark799), [211](#_bookmark868)

SelectedBackgroundMode [204](#_bookmark800)

SelectedCells [204](#_bookmark801)

selectedForeground [42](#_bookmark134), [204](#_bookmark802), [211](#_bookmark869)

SelectedForegroundMode [204](#_bookmark803)

selectIncludeLabels [204](#_bookmark798), [211](#_bookmark870)

SelectionModel [204](#_bookmark804)

selectionPolicy [30](#_bookmark64), [204](#_bookmark805), [211](#_bookmark871)

SeriesDataSorted [205](#_bookmark806)

setting [36](#_bookmark85)

setting cell styles [39](#_bookmark99) setting for a cell [37](#_bookmark88) setting for a cell range [40](#_bookmark100) setting for a range [38](#_bookmark95) setting for all cells [38](#_bookmark95)

setting for all columns [37](#_bookmark89)–[38](#_bookmark92)

setting for all labels [38](#_bookmark93)

setting for all rows [37](#_bookmark89)

setting for cells and labels [39](#_bookmark98)

setting for column [37](#_bookmark91) setting for entire column [38](#_bookmark94) setting for entire row [38](#_bookmark94) setting for entire table [38](#_bookmark96)

setting for labels [37](#_bookmark91)–[38](#_bookmark92)

setting for range of cells [37](#_bookmark90) setting in the API [21](#_bookmark32) SortBy Column [135](#_bookmark573)

spannedCells [211](#_bookmark872)

stock data source [74](#_bookmark295)

StoreImageEnabled [205](#_bookmark807)

styles [211](#_bookmark873)

summary of [197](#_bookmark742)

swingDataModel [211](#_bookmark874)

topRow [42](#_bookmark125), [205](#_bookmark808), [211](#_bookmark875)

TrackBackground [205](#_bookmark809)

trackCursor [205](#_bookmark810), [211](#_bookmark876)

TrackForeground [205](#_bookmark811)

TrackSize [205](#_bookmark812)

Traversable [209](#_bookmark839)

traverseCycle [205](#_bookmark813), [211](#_bookmark877)

VariableEstimateCount [205](#_bookmark814)

VerticalAlignment [209](#_bookmark840)

VertSBAttachment [206](#_bookmark815)

VertSBDisplay [206](#_bookmark816)

VertSBOffset [206](#_bookmark817)

VertSBPosition [206](#_bookmark818)

VertSBTrack [206](#_bookmark819)

VertSBTrackColumn [206](#_bookmark820)

VisibleColumns [48](#_bookmark162)–[49](#_bookmark164), [207](#_bookmark821)

VisibleRows [48](#_bookmark163)–[49](#_bookmark164), [207](#_bookmark822)

property

definition [167](#_bookmark661)

LiveTable properties [171](#_bookmark678) setting in Java IDE [168](#_bookmark665) setting using API [168](#_bookmark667)

property editors LiveTable [169](#_bookmark672)

##### Q

QueryExpressionList [118](#_bookmark473)

##### R

ranges

in cell selection [133](#_bookmark561)

referencing [38](#_bookmark95)

selected [133](#_bookmark563)

used in cell spanning [67](#_bookmark245)

references expression [120](#_bookmark488)

registry

JClass central [119](#_bookmark485) removing cell selections [134](#_bookmark565) renderers [91](#_bookmark358)–[92](#_bookmark363)

component based [95](#_bookmark367)

default [93](#_bookmark364)

editors [36](#_bookmark83)

subclassing [93](#_bookmark364)

unmap [92](#_bookmark361)

writing [93](#_bookmark365)

rendering cells [88](#_bookmark350)–[89](#_bookmark352)

RepaintEnabled property [203](#_bookmark790)

RepeatBackground property [208](#_bookmark835)

RepeatBackgroundColors property [209](#_bookmark836)

RepeatForeground property [209](#_bookmark837)

RepeatForegroundColors property [209](#_bookmark838)

repeating colors [60](#_bookmark217)

reset [86](#_bookmark338)

resetSortedRows() method [137](#_bookmark579)

ResizeByLabelsOnly property [127](#_bookmark521)

tutorial [30](#_bookmark67)

ResizeEven property [127](#_bookmark520), [203](#_bookmark791)

resizeEven property [210](#_bookmark865)

ResizeInteractive property [203](#_bookmark792)

resizeInteractive property [211](#_bookmark866)

resizing [146](#_bookmark608)

columns [126](#_bookmark507)

default behavior [126](#_bookmark509)

disabling [126](#_bookmark515)

events and listeners [146](#_bookmark610)

pixel width [126](#_bookmark512)

preset styles [40](#_bookmark108)

rows [126](#_bookmark507)

rows and columns [126](#_bookmark518)–[127](#_bookmark520)

using labels only [127](#_bookmark522)

using labels, tutorial [30](#_bookmark66)

results [111](#_bookmark398)

RGB color values [213](#_bookmark885)

RowHidden property [54](#_bookmark196), [203](#_bookmark793)

RowLabelDisplay property [203](#_bookmark794)

RowLabelOffset property [44](#_bookmark142), [203](#_bookmark795)

RowLabelPlacement property [43](#_bookmark139), [203](#_bookmark796)

rows

adding [75](#_bookmark299), [86](#_bookmark337)

controlling resizing [126](#_bookmark518) default resizing behavior [126](#_bookmark513) definition

columns definition [48](#_bookmark158)

deleting [76](#_bookmark303)

determining number [48](#_bookmark159)

disallowing resizing [126](#_bookmark516)

displaying [49](#_bookmark164)

dragging [134](#_bookmark569)

freezing [49](#_bookmark168)–[50](#_bookmark170)

height

pixel value [51](#_bookmark179)

height property [51](#_bookmark176)

height, setting [51](#_bookmark178)

hiding [54](#_bookmark194)

labels [43](#_bookmark136), [70](#_bookmark265)

labels, placement [43](#_bookmark137)

making visible [129](#_bookmark540)

moving [76](#_bookmark306)

placement of frozen [50](#_bookmark171)

referencing, all [37](#_bookmark89)–[38](#_bookmark92)

referencing, entire [38](#_bookmark94)

referencing, one [37](#_bookmark91)

removing [86](#_bookmark337)

resizing [126](#_bookmark507)

resizing all at once [127](#_bookmark520) resizing with labels [127](#_bookmark521) selecting labels [133](#_bookmark559)

set as top [42](#_bookmark123)

setting the number [74](#_bookmark297)

specifying labels [75](#_bookmark298)

swapping [49](#_bookmark165)

visible, getting [48](#_bookmark161)

visible, setting [48](#_bookmark161)

RowSelection property [203](#_bookmark797)

RowTrigger property and dragging [134](#_bookmark569)

runtime

cell selection [134](#_bookmark567)

##### S

sBLayout property [177](#_bookmark699), [211](#_bookmark867)

scroll listener methods [130](#_bookmark546)

scrollbars [41](#_bookmark119)

attaching [41](#_bookmark120)

component [42](#_bookmark128)

definition [36](#_bookmark80)

disabling interactive [129](#_bookmark533)

display [42](#_bookmark126)

force scrolling by an application [129](#_bookmark536)

jump scrolling [128](#_bookmark530)

options [42](#_bookmark127)

positioning [41](#_bookmark121)

programming [128](#_bookmark529)

tracking [130](#_bookmark544)

scrolling [128](#_bookmark525), [149](#_bookmark614)

automatic [128](#_bookmark531)

disabling interactive [129](#_bookmark533)

forcing [129](#_bookmark536)

jump scrolling [128](#_bookmark530)

listener [130](#_bookmark546)

managing [128](#_bookmark529)

mouse wheel support [129](#_bookmark543)

tracking scrollbars [130](#_bookmark544)

selected cells list [133](#_bookmark561)

selectedBackground property [42](#_bookmark133), [178](#_bookmark700), [204](#_bookmark799), [211](#_bookmark868)

SelectedBackgroundMode property [204](#_bookmark800)

SelectedCells property [204](#_bookmark801)

selectedForeground property [42](#_bookmark134), [178](#_bookmark701), [204](#_bookmark802), [211](#_bookmark869)

SelectedForegroundMode property [204](#_bookmark803)

selectIncludeLabels property [179](#_bookmark702), [204](#_bookmark798), [211](#_bookmark870)

selection cells [153](#_bookmark620)

cells, default [131](#_bookmark552)

colors [42](#_bookmark131), [132](#_bookmark554)

setting [42](#_bookmark132)

customizing [132](#_bookmark557)

enabling cell selection, tutorial [30](#_bookmark64)

SelectionModel property [204](#_bookmark804)

selectionPolicy property [132](#_bookmark558), [179](#_bookmark703), [204](#_bookmark805), [211](#_bookmark871)

tutorial [30](#_bookmark64)

SeriesDataSorted property [205](#_bookmark806)

set method [197](#_bookmark741)

setAutoScroll() method [128](#_bookmark532)

setDataFormat

method in MathValue [111](#_bookmark408) setEditable method [71](#_bookmark274) setting

cell renderers, for a series [91](#_bookmark355) properties in an IDE [40](#_bookmark102) scrollbar options [42](#_bookmark127)

setValueAt

method in MathMatrix [114](#_bookmark435)

method in MathVector [113](#_bookmark425)

SortableDataViewModel [70](#_bookmark258)

SortByColumn property [135](#_bookmark573)

sorting

and ColumnTrigger property [137](#_bookmark576)

columns [135](#_bookmark571)

columns, tutorial [31](#_bookmark69) events and listeners [155](#_bookmark623) frozen columns [135](#_bookmark572)

multiple columns [135](#_bookmark574)

resetting [137](#_bookmark578)

SortByColumn property [135](#_bookmark573)

source parameter [76](#_bookmark307)

SpanHandler [67](#_bookmark246)

spannedCells property [179](#_bookmark705), [211](#_bookmark872)

spanning cells [66](#_bookmark244)

create multiline headers [68](#_bookmark248)

using JCCellRange [67](#_bookmark245)

StaticDataSource [81](#_bookmark328)

stock data sources using [71](#_bookmark277)

StoreImageEnabled property [205](#_bookmark807)

storing data [72](#_bookmark280)–[73](#_bookmark284)

styles

preset [40](#_bookmark105)

property [180](#_bookmark706), [211](#_bookmark873)

subclassing

cell editors [101](#_bookmark380)

cells

renderers [93](#_bookmark364)

summary of properties [197](#_bookmark742)

support [12](#_bookmark13)

JClass Community [13](#_bookmark15)

swapColumns method [49](#_bookmark165) swapping rows and columns [49](#_bookmark165) swapRows method [49](#_bookmark165)

Swing TableModel class [80](#_bookmark324)

Swing, using TableModel data objects [74](#_bookmark292)

swingDataModel property [211](#_bookmark874)

##### T

table

basic [18](#_bookmark23)

frame border [46](#_bookmark154)

preset styles [40](#_bookmark105)

printing [145](#_bookmark604)

referencing, entire [38](#_bookmark96)

resize events [146](#_bookmark608)

resizing [146](#_bookmark608)

scrolling [149](#_bookmark614)

size defined by data source [70](#_bookmark264)

sorting [155](#_bookmark623)

table anatomy cell [36](#_bookmark81)

current cell [36](#_bookmark82)

current context [37](#_bookmark87)

label [36](#_bookmark79)

renderers [36](#_bookmark83)

scrollbars [36](#_bookmark80)

table context [37](#_bookmark87)

table frame

border [170](#_bookmark674), [179](#_bookmark704)

table layout examples [224](#_bookmark911)

table scrolling

attaching scrollbars [41](#_bookmark120)

default [128](#_bookmark527)

different component [42](#_bookmark128)

setting options [42](#_bookmark127)

Table.isRowVisible() [129](#_bookmark542)

TableDataEvent [82](#_bookmark331)

TableDataItem [70](#_bookmark263)

TableDataListener [82](#_bookmark332)

TableDataModel interface [70](#_bookmark262)

TableDataView [70](#_bookmark259)

TableExpressionList [118](#_bookmark474)

TableListenerPropagator [119](#_bookmark477) TableModel, using in table [74](#_bookmark292) TableSwingDataSource [74](#_bookmark293)

technical support [12](#_bookmark13)

text

alignment in cell [61](#_bookmark219)

clipping [65](#_bookmark232)

topRow property [42](#_bookmark125), [129](#_bookmark537), [181](#_bookmark707), [205](#_bookmark808), [211](#_bookmark875)

toString

method in MathMatrix [114](#_bookmark436) method in MathScalar [112](#_bookmark416) method in MathVector [113](#_bookmark426) TrackBackground property [205](#_bookmark809)

trackCursor property [137](#_bookmark584), [205](#_bookmark810), [211](#_bookmark876)

TrackForeground property [205](#_bookmark811)

tracking

cursor type [41](#_bookmark116)

mouse pointers, disabling [137](#_bookmark583)

scrollbars [130](#_bookmark544)

TrackSize property [205](#_bookmark812)

Traversable property [209](#_bookmark839)

traversal cell [123](#_bookmark494)

customizing cell [123](#_bookmark498)

default [123](#_bookmark496)

events [158](#_bookmark627)–[159](#_bookmark631)

forcing [124](#_bookmark502)

interactive [125](#_bookmark504)

preset styles [40](#_bookmark109)

traverseCycle property [205](#_bookmark813), [211](#_bookmark877)

trigger [31](#_bookmark70)

tutorial

adding color to an individual cell [26](#_bookmark53)

adding interactivity [29](#_bookmark60)

background colors [23](#_bookmark42)

basic table [18](#_bookmark23)

cell borders [27](#_bookmark55)

cell selection [30](#_bookmark64)

cell size [28](#_bookmark57)

cell spacing [27](#_bookmark55)

cell styles [22](#_bookmark38)

changing alignment [24](#_bookmark46)

clip arrows [20](#_bookmark28)

fonts, setting [24](#_bookmark48)

foreground color [23](#_bookmark42)

improving table appearance [21](#_bookmark33) making a table editable [29](#_bookmark62) PixelHeight property [28](#_bookmark57)

PixelWidth property [28](#_bookmark57)

resize using labels [30](#_bookmark66)

ResizeByLabelsOnly property [30](#_bookmark67)

resizing cells [20](#_bookmark28)

SelectionPolicy property [30](#_bookmark64) setting a data source [19](#_bookmark25) setting colors [22](#_bookmark38)

setting properties in the API [21](#_bookmark32)

sorting columns [31](#_bookmark69)

table appearance [19](#_bookmark27)

table changes [20](#_bookmark30)

typographical conventions [10](#_bookmark6)

##### U

user interactivity [123](#_bookmark491)

##### V

values

change [84](#_bookmark333)

parameter [75](#_bookmark302)

variable cell size [28](#_bookmark57) VariableEstimateCount property [205](#_bookmark814) vectorValue

method in MathMatrix [114](#_bookmark437) method in MathScalar [112](#_bookmark417) method in MathValue [111](#_bookmark409) method in MathVector [113](#_bookmark427)

VerticalAlignment [66](#_bookmark242)

VerticalAlignment property [209](#_bookmark840)

VertSBAttachment property [206](#_bookmark815)

VertSBDisplay [129](#_bookmark535)

VertSBDisplay property [206](#_bookmark816)

VertSBOffset property [206](#_bookmark817)

VertSBPosition property [206](#_bookmark818)

VertSBTrack property [206](#_bookmark819)

VertSBTrackColumn property [206](#_bookmark820)

visibility

cells [124](#_bookmark500), [129](#_bookmark539)

columns [129](#_bookmark541)

forcing [129](#_bookmark540)

VisibleColumns property [48](#_bookmark162)–[49](#_bookmark164), [207](#_bookmark821)

VisibleRows property [48](#_bookmark163)–[49](#_bookmark164), [207](#_bookmark822)

**W**

writing a cell renderer [93](#_bookmark365)

##### X

XML

examples [78](#_bookmark318)

in JClass [78](#_bookmark317)

interpreter [78](#_bookmark319)

loading data [77](#_bookmark313)

primer [77](#_bookmark315)

Swing TableModel class [80](#_bookmark324)

tags [79](#_bookmark322)