JClass LiveTable™

Programmer's Guide & Reference Manual

Version 3.6
JDK 1.0.2, JDK 1.1, JDK 1.1+Swing, and JDK 1.2

The Essential Java Grid/Table Component

Software Development Productivity

260 King Street East
Toronto, Ontario, Canada M5A 1K3
(416) 594-1026
www.klg.com

December 1998

RefNo: PRGDE-JCTBL/J/360-12/98

LIMITED END-USER LICENSE AGREEMENT FOR KL GROUP JCLASS PRODUCTS

The following is the limited end user license agreement ("LEULA") for limited use on all of KL Group Inc.'s JClass products, other than JClass JarMaster and JClass JarHelper.

IMPORTANT — READ CAREFULLY: This KL Group Inc. ("KL Group") Limited End-User License Agreement ("LEULA") is a legal agreement between you (either an individual or a single entity) and KL Group for the KL Group software product identified above, which computer software includes class libraries, Sun Microsystems, Inc.'s Java© Project X Technology and may include associated media, printed materials, and "online" or electronic documentation ("SOFTWARE"). By installing, copying, or otherwise using the SOFTWARE, you agree to be bound by the terms of this LEULA. If you do not agree to the terms of this LEULA, do not install or use the SOFTWARE; you may, however, return it to your place of purchase for a full refund.

SOFTWARE LICENSE

The SOFTWARE is protected by copyright laws and international copyright treaties, as well as other intellectual property laws and treaties. The SOFTWARE is licensed, not sold.

1. GRANT OF LICENSE. This LEULA grants you the following rights:

(a) If You Have Any Version Of A JClass Product. This license permits a single developer to use the SOFTWARE on a single computer, subject to the restrictions in Section 3:
   i. To Build Applets. Provided that applets you build are used only as an internal component in end-user oriented user-interfaces, you may copy them to additional computers (e.g. Web Servers), from which you may allow end-users to download, royalty-free, the applets in the course of browsing or interacting with Web pages you create. You are not permitted to distribute the applets in any fashion which would promote, encourage or allow reuse or redistribution of the applet, other than as permitted above; and
   ii. To Build Stand-Alone Java Applications. You have a royalty-free right to reproduce and distribute the class libraries as an integral part of your application(s). You are not permitted to expose, either directly or indirectly, any API that allows programmatic access to the class libraries.

(b) Definition Of Use. The SOFTWARE is "in use" on a computer when it is loaded into temporary memory (i.e. RAM) or installed into permanent memory (e.g. hard disk, CD-ROM, or other storage device) of that computer, except that a copy installed on a network server for the sole purpose of distribution to other computers is not "in use".

2. LIMITED DISTRIBUTION RIGHTS. Your royalty-free distribution rights described in Section 1 above are granted provided that you:

(a) distribute the Applet(s) you build only in conjunction with and as an integral part of your Web pages, and distribute the class libraries only as an integral part of your end-user, stand-alone application;
(b) your Web pages or software product(s) are targeted at end-users, and are not a development tool;
(c) you do not use KL Group's name, logo or trademark to market your Web pages or application;
(d) you include a valid copyright notice on your Web pages and software products; and
(e) you agree to indemnify, hold harmless, and defend KL Group and its suppliers from and against any claims or lawsuits, including attorney's fees, that arise or result from the use or distribution of your Web pages and/or applications.

3. DESCRIPTION OF OTHER RIGHTS AND LIMITATIONS.

(a) Rental. You may not rent, lease, or lend the SOFTWARE, but you may transfer the SOFTWARE and accompanying written materials on a permanent basis provided you retain no copies and the recipient agrees to the terms of this License Agreement. If the SOFTWARE is an upgrade, any transfer must include the most recent upgrade and all prior versions.
(b) Support Services. KL Group may provide you with support services related to the SOFTWARE ("Support Services"). Use of Support Services is governed by the KL Group policies and programs described in the user manual, "online" documentation, and/or other KL Group-provided materials. Any supplemental software code provided to you as part of the Support Services shall be considered part of the SOFTWARE and subject to the terms and conditions of this LEULA. With respect to technical information you provide to KL Group as part of the Support Services, KL Group may use such information for its business purposes, including for product support and development. KL Group will not utilize such technical information in a form that personally identifies you. This LEULA does not entitle you to purchase KL Group's Gold Support service offerings. Only a non-limited EULA entitles you to purchase such support services.
(c) Termination. Without prejudice to any other rights, KL Group may terminate this LEULA if you fail to comply with the terms and conditions of this LEULA. In such event, you must destroy all copies of the SOFTWARE and all of its component parts.

4. UPGRADES.

This LEULA does not entitle you to Upgrades for the SOFTWARE. Only a non-limited EULA entitles you to such Upgrades

5. COPYRIGHT.

All title and copyrights in and to the SOFTWARE (including but not limited to any images, photographs, animations, video, audio, music, text, and "applets" incorporated into the SOFTWARE), the accompanying printed materials, and any copies of the SOFTWARE are owned by KL Group or its suppliers. Specifically, all title and copyrights in and to the Java© Project X Technology are owned and licensed by Sun Microsystems, Inc., Copyright © Sun Microsystems, Inc. All rights reserved.
The SOFTWARE is protected by copyright laws and international treaty provisions. Therefore, you must treat the SOFTWARE like any other copyrighted material except that you may install the SOFTWARE on a single computer provided you keep the original solely for backup or archival purposes. You may not copy the printed materials accompanying the SOFTWARE.

6. DUAL-MEDIA SOFTWARE.

You may receive the SOFTWARE in more than one medium. Regardless of the type or size of medium you receive, you may use only one medium that is appropriate for your single computer. You may not use or install the other medium on another computer. You may not loan, rent, lease, or otherwise transfer the other medium to another user.

7. U.S. GOVERNMENT RESTRICTED RIGHTS.

The SOFTWARE and documentation are provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph(c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable. Manufacturer is KL Group Inc., 260 King Street East, Toronto, Ontario, Canada, M5A 4L5.

8. EXPORT RESTRICTIONS.

You agree that you do not intend to or will, directly or indirectly, export or transmit the SOFTWARE or related documentation and technical data, or process, or service that is the direct product of the SOFTWARE, to any country to which such export or transmission is restricted by any applicable U.S., Canadian or other State regulation or statute, without the prior written consent, if required, of the Bureau of Export Administration of the U.S. Department of Commerce, or such other governmental entity as may have jurisdiction over such export or transmission.

9. MISCELLANEOUS.

If you acquired this product in the United States this LEULA is governed by the laws of New York State, and the parties agree to resolve any dispute exclusively in the courts at New York City. If you acquired this product in Canada, this LEULA is governed by the laws of the Province of Ontario, and the parties agree to resolve any dispute exclusively in the courts at Toronto.
If this product was acquired outside the United States or Canada, then local law may apply.
Should you have any questions concerning this LEULA, or if you desire to contact KL Group for any reason, please contact the KL Group subsidiary serving your country, or write: KL Group Sales Information, 260 King Street East, Toronto, Ontario, Canada, M5A 4L5.

10. LIMITED WARRANTY.

LIMITED WARRANTY. KL Group warrants that (a) the SOFTWARE will perform substantially in accordance with the accompanying written materials for a period of ninety (90) days from the date of receipt, and (b) any Support Services provided by KL Group shall be substantially as described in applicable written materials provided to you by KL Group, and KL Group support engineers will make commercially reasonable efforts to solve any problem issues. Some states and jurisdictions do not allow limitations on duration of an implied warranty, so the above limitation may not apply to you. To the extent allowed by applicable law, implied warranties on the SOFTWARE, if any, are limited to ninety (90) days.
CUSTOMER REMEDIES. KL Group's and its suppliers' entire liability and your exclusive remedy shall be, at KL Group's option, either (a) return of the price paid, if any, or (b) repair or replacement of the SOFTWARE that does not meet KL Group's Limited Warranty and that is returned to KL Group with a copy of your receipt. This Limited Warranty is void if failure of the SOFTWARE has resulted from accident, abuse, or misapplication. Any replacement SOFTWARE will be warranted for the remainder of the original warranty period or thirty (30) days, whichever is longer. Outside the United States and Canada, neither these remedies nor any product support services offered by KL Group are available without proof of purchase from an authorized international source.
SPECIFIC DISCLAIMER FOR HIGH-RISK ACTIVITIES. The SOFTWARE is not designed or intended for use in high-risk activities including, without restricting the generality of the foregoing, on-line control of aircraft, air traffic, aircraft navigation or aircraft communications; or in the design, construction, operation or maintenance of any nuclear facility. KL Group and its suppliers specifically disclaim any express or implied warranty of fitness for such purposes or any other purposes.
NO OTHER WARRANTIES. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, KL GROUP AND ITS SUPPLIERS DISCLAIM ALL OTHER WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, WITH REGARD TO THE SOFTWARE AND THE ACCOMPANYING PRINTED MATERIALS. THIS LIMITED WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY HAVE OTHERS WHICH VARY FROM STATE/JURISDICTION TO STATE/JURISDICTION.

11. LIMITATION OF LIABILITY.

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL KL GROUP OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE OR THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES, EVEN IF KL GROUP HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN ANY CASE, KL GROUP'S ENTIRE LIABILITY UNDER ANY PROVISION OF THIS LEULA SHALL BE LIMITED TO THE GREATER OF THE AMOUNT ACTUALLY PAID BY YOU FOR THE SOFTWARE OR US$5.00; PROVIDED, HOWEVER, IF YOU HAVE ENTERED INTO A KL GROUP SUPPORT SERVICES AGREEMENT, KL GROUP'S ENTIRE LIABILITY REGARDING SUPPORT SERVICES SHALL BE GOVERNED BY THE TERMS OF THAT AGREEMENT. BECAUSE SOME STATES AND JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY, THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

JCL-BINLIC-LTD-9904

Table of Contents

Preface

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 KL Group's JClass BWT or JClass Field. JClass BWT provides additional Java components that complement or replace their equivalent AWT components. A JClass BWT 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 Licence Agreement.

Feature Overview

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

- the data source for the table
- preset and custom cell editing and display behavior for all types of data
- labels for columns and rows
- colors, fonts, borders (including custom borders), alignment, and spacing for cells and labels
- row and column dragging
- column sorting
- adding, deleting, moving, and dragging rows and columns
- scrolling and attaching default or custom scrollbars
- cell selection and traversal

Assumptions

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

Typographical Conventions in this Manual

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

Overview of the Manual

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

Chapter 1 , "Getting Started", provides help with common configuration problems, including CLASSPATH and IDE setup.

Chapter 2 , "'Hello Table' – A Simple JClass LiveTable Program", provides a tutorial exercise to familiarize new users with the basics of writing a JClass LiveTable program.

Chapter 3 , "Building a Table", explains how to set most JClass LiveTable properties to customize the appearance and display of JClass LiveTable applications.

Chapter 4 , "Working with Table Data", gives details on getting data into and out of tables using the new Model View Controller data handling in JClass LiveTable 3.

Chapter 5 , "Displaying and Editing Cells", describes how to configure JClass LiveTable so users can edit cells of any data type.

Chapter 6 , "Programming User Interactivity", explains how to control how users interact with your table application, including cell traversal, selection, sorting, etc.

Chapter 7 , "Events and Listeners", explains how to send events and register event listeners in your JClass LiveTable programs.

Chapter 8 , "Table Printing", describes the enhanced printing features of JClass LiveTable.

Chapter 9 , "JClass LiveTable Beans and IDEs", describes the JClass LiveTable JavaBeans and how to use them within a Java Development Environment.

Part II – Reference Appendices are provided for quick access to detailed information of JClass LiveTable features and implementation.

Appendix A, Event Summary, lists events and corresponding event listeners.

Appendix B, JClass LiveTable Property Listing, is a quick reference to properties, their functions, and settable values.

Appendix C, Moving from JClass LiveTable 2.x to 3.x, explains how to use the `JCTable` transitional class in JClass LiveTable, and includes an explanation of the differences between the two versions.

Appendix D, JCString Properties, describes types of JCString properties available for adding hypertext and text within programs using JClass LiveTable components.

Appendix E, Colors and Fonts, lists all of the color names and RGB values available to JClass LiveTable applications. It also lists all of the available fonts and font style constants.

Related Documents

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

- "*Writing Java Programs*" at *http://www.javasoft.com/docs/programmer.html* and the "*Java Tutorial*" at *http://www.javasoft.com/docs/books/tutorial/index.html* from Sun Microsystems.
- *Java in a Nutshell, 2nd Edition* from O'Reilly & Associates Inc.
- Resources for using JavaBeans are at *http://www.javasoft.com/beans/resources.html*.

However, these documents are not required to develop applications using JClass LiveTable and Java.

Technical Support

Many of the initial questions you may have are basic installation or configuration problems. Consult this product's *readme* file and Chapter 1, "Getting Started", for help with these types of problems.

KL Group's **Standard Support** plan is included with your purchase and entitles registered users with a valid JClass software license to the following support:

- 30 days of direct technical support via telephone, email or fax.
- FAQ Documents on our Web site.
- JClass Knowledge Base, a searchable collection of information including program samples and problem/resolution documents.
- JClass Forum Newsgroup, where you can communicate with other developers using JClass products around the world.
- Minor bug-fix update releases downloadable from our Web site.

Upgrading to KL Group's **Gold Support with Subscription** plan entitles you to the following *additional* support:

- Unlimited direct technical support for one full year.
- Web-based Express Case Submission form for quickly logging problems; a Customer Support Engineer will contact and assist you directly.
- All product upgrade releases; download from Web site or shipped to you on CD-ROM.

For information on obtaining **Gold Support** for your JClass product, please visit our online store or your JClass reseller. You can also email `sales@klg.com`.

To Contact JClass Support

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

- The type and version of the operating system you are using
- Your development environment and its version
- A full description of the problem including the steps required to duplicate it.

Telephone:	800-663-4723 (toll free in North America) or 416-594-1026 Available Monday – Friday, 9:00 a.m. to 8:00 p.m. Eastern time
Fax:	416-594-1919
Standard Support Email:	jclass_support@klg.com
Express Case Submission Form (Gold Support only)	http://www.klg.com/cgi-bin/webcase.cgi

Other Support Resources

JClass Technical Support (links to Knowledge Base):	http://www.klg.com/cs/tech/jclass/
JClass FAQs:	http://www.klg.com/cs/tech/jclass/faq/
Using JClass in IDEs:	http://www.klg.com/jclass/ides.html

Product Feedback and Annoucements

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

Please send your comments to:

KL Group Inc.
260 King Street East
Toronto, Ontario, M5A 1K3 Canada

Phone: (416) 594-1026
Fax: (416) 594-1919
Email: dev_jclass@klg.com
Internet: news://news.klg.com/klg.forum.jclass

While we appreciate your feedback, we cannot guarantee a response. Please do not use the `dev_jclass` email address for technical support questions.

Occasionally, we send JClass-related product announcments to our customers using an email list. To add yourself to this mailing list, send email with the word "`subscribe`" in the body of the message to jclass_announce-request@klg.com. Visit the KL Group web site at http://www.klg.com for more details.

Part I

Using JClass LiveTable

1

Getting Started

1.1 Introduction

This chapter covers common configuration issues so you can start using JClass LiveTable as quickly as possible. Because of the wide variety of Java platforms and development environments, JClass LiveTable may not be configured correctly for your environment after installation.

Please see the *readme-table.txt* file included with this release for details on installing JClass LiveTable and for information on supported Java environments.

1.2 Matching JClass and JDK Versions

Separate versions of JClass LiveTable are available for specific versions of the Java Platform. The version you use should match the JDK version needed by your application/applet. For example, if you are creating an applet to run in Microsoft Internet Explorer 4.0 (JDK 1.1 platform), use the JClass LiveTable version for JDK 1.1. Use the following table to determine which version of JClass LiveTable to use for your application:

Version	Java Platform	Description
JClass LiveTable 3.6**T**	JDK 1.0.2	■ "Transitional Beans" that provide JDK 1.1-level event APIs for easy migration to JDK 1.1.
JClass LiveTable 3.6	JDK 1.1	■ Standard "AWT-style" JavaBeans.
JClass LiveTable 3.6**S**	JDK 1.1 + Swing	■ JavaBeans for JDK 1.1 applications using Swing 1.0.3 components.

Version	Java Platform	Description
JClass LiveTable 3.6**J**	JDK 1.2	■ JavaBeans for JDK 1.2 applications. ■ Also for JDK 1.1 applications using Swing 1.1.

Each version has the same API and virtually the same features to make it easy for existing applications to migrate to new versions of the Java platform. For clarity, distribution filenames and JAR/ZIP archives contain the full version number in the name, for example, *jctable360.jar* and *jctable360S.jar*.

This documentation covers all versions of JClass LiveTable, noting any differences between versions where they occur.

Determining the JDK and JClass LiveTable Version

To determine the version of the JDK you are using, enter the following at a command prompt:

```
java -version
```

To determine the version of JClass LiveTable you are using on your system, run the version program provided:

```
java jclass.table3.JCVersion
```

This program will only run if the CLASSPATH has been set correctly as described in the following section.

1.3 Setting the CLASSPATH Environment Variable

The Java Virtual Machine (JVM) and other applications use the CLASSPATH environment variable to locate user-defined classes. You should ensure that the CLASSPATH points to the location of the JClass LiveTable classes (and classes you develop). The installation program does this automatically for Windows users; Unix users need to add JClass LiveTable to the CLASSPATH manually.

Two entries should be part of the CLASSPATH – one specifying the JClass product classes (a JAR or ZIP file located in the product's *\lib* directory), and one specifying the installation directory (necessary to run JClass LiveTable example and demo programs). You should not need to unzip the JAR/ZIP archive to develop with JClass LiveTable.

For example, if you installed JClass LiveTable on a Windows machine in *C:\JClass36*, the CLASSPATH would include the following ([*xxx*] is the product version number):

```
C:\JClass36\lib\jctable[xxx].jar;C:\JClass36\
```

To determine the current CLASSPATH, enter the following at a command prompt:

Windows – `echo %CLASSPATH%`
Unix – `echo $CLASSPATH`

Some CLASSPATH specification tips:

- Each entry is separated by a semicolon (Windows) or a colon (Unix).
- An entry is typically a root directory to search through for *.class* files (if a class is part of a package, each level in the package is treated as a subdirectory from here), for example, `C:\JClass36`.
- Entries can also specify a JAR or ZIP file containing archived classes, for example, `C:\JClass36\lib\jctable[`*xxx*`].jar`.
- Add a period (.) to the CLASSPATH to include the current directory.
- Setting the CLASSPATH in a startup file causes it to be used when running web browsers and other applications for your entire session.

1.3.1 Setting the CLASSPATH in Windows

The Windows-based setup program automatically adds JClass LiveTable to the CLASSPATH during installation. The following instructions are provided in case you need to configure the CLASSPATH manually for some reason.

Windows 95 and Windows 98

Add the following statement to your *autoexec.bat* file to include JClass LiveTable in the CLASSPATH ([*xxx*] is the product version number):

```
set CLASSPATH=%CLASSPATH%;C:\JClass36\lib\jctable[xxx].jar;
  C:\JClass36;
```

JDK 1.0.2 users: Replace *jctable[xxx].jar* above with *jctable[xxx]-classes.zip*.

Restart Windows to make the change take effect.

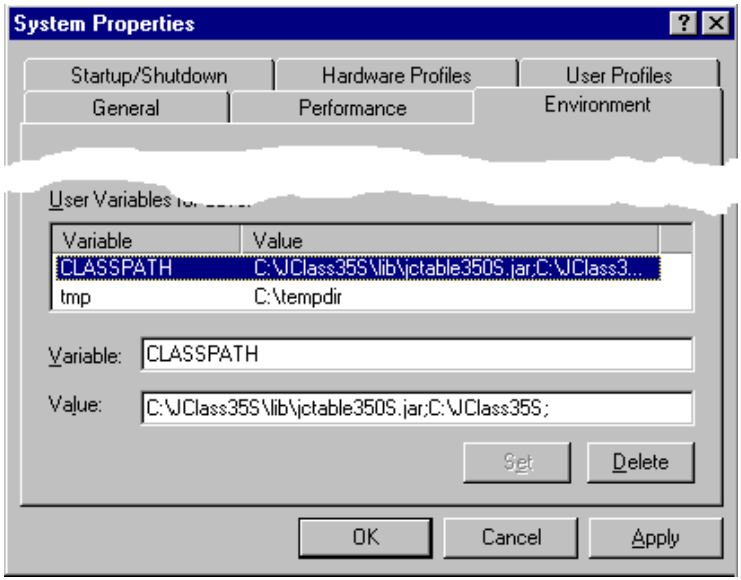
Windows NT (3.51 and higher)

The best way to set environment variables is using the **Control Panel**. Start **Control Panel** and select **System**. Locate the CLASSPATH environment variable (if it doesn't exist, create it). Add the following value to the variable to include JClass LiveTable in the CLASSPATH ([*xxx*] is the product version number):

```
[existing-classes];C:\JClass36\lib\jctable[xxx].jar;C:\JClass36;
```

JDK 1.0.2 users: Replace *jctable[xxx].jar* above with *jctable[xxx]-classes.zip*.

The following illustrates setting the CLASSPATH on Windows NT; your actual setting may vary or have additional directories/JAR files.



1.3.2 Setting the CLASSPATH in Unix

You must manually configure the CLASSPATH environment variable before you can start using JClass LiveTable. The CLASSPATH must point to the location of the JClass LiveTable classes and installation directory (for example */usr/local*).

Add a *setenv* command to your startup file (such as *.cshrc*) to set CLASSPATH to point to the JClass LiveTable classes, for example ([*xxx*] is the JClass LiveTable version number):

```
setenv CLASSPATH .:/usr/local/JClass36/lib/jctable[xxx].jar:
    /usr/local/JClass36
```

JDK 1.0.2 users: Replace *jctable[xxx].jar* above with *jctable[xxx]-classes.zip*.

1.3.3 Testing the Installation

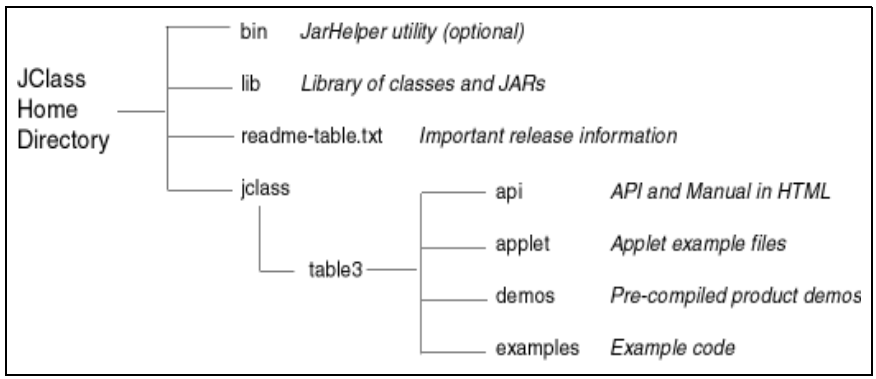
After setting the CLASSPATH environment variable you should verify that it has been configured correctly. The easiest way to test whether you can start programming with JClass LiveTable is to execute the `JCVersion` class. Enter the following at a command prompt:

```
java jclass.table3.JCVersion
```

If the version number does not match the version just installed, there is probably an older version of JClass LiveTable listed earlier in the CLASSPATH.

1.4 Installed Files Overview

JClass products install into a single root directory. The directory hierarchy is designed to make it easy to work with multiple JClass products in one location. The following diagram provides an overview of the directory hierarchy created for JClass LiveTable.



Class Library Archives

The *\lib* directory contains the JClass LiveTable class library archives in JAR or ZIP format. JClass LiveTable developers can add these files to an IDE, or simply work with them through the JDK. You usually do not need to unzip the archives when programming with JClass LiveTable.

Your release of JClass LiveTable may include the following archives ([*xxx*] is the product version number):

jctable[*xxx*].jar	Standard JClass LiveTable components.
jctable[*xxx*]jb.jar	The standard components plus a Bean that data binds to Borland JBuilder data source components.
jctable[*xxx*]vc.jar	The standard components plus a Bean that data binds to Visual Café data source components.
jctable[*xxx*]ds.jar	JClass LiveTable components that data bind with JClass DataSource data Beans. JClass DataSource is available separately or as part of a JClass product suite from KL Group.
jctable[*xxx*]-classes.zip	Standard JClass LiveTable components for development environments that cannot use JAR files.

See the *readme-table.txt* file for details on the archives that ship with each version of JClass LiveTable.

Sample Code

The *jclass\table3\examples* and *jclass\table3\demos* directories contain sample Java programs that use JClass LiveTable. The programs can be executed as either applets or applications. To run as applications, use the Java interpreter, specifying the application class's full package path, for example:

```
java jclass.table3.examples.simple
```

To run as applets, either open *index.html* in a compatible browser (you may need to unset the CLASSPATH environment variable first) or use the JDK *appletviewer* program.

JDK 1.2 Note: To run JClass LiveTable sample programs using *appletviewer*, you may need to extract the product JAR file into your *JCLASS_HOME* directory. This is because *appletviewer* in JDK 1.2 does not use the CLASSPATH environment variable. You may also need to use the `-nosecurity` switch, for example:

```
appletviewer -nosecurity index.html
```

Product Documentation

The *jclass\table3\api* directory contains JClass LiveTable programming and reference documentation in HTML format. Open *index.html* in a frames-capable web browser to read the documentation.

Version Notes, Compatibility, Known Problems

The *readme-table350.txt* file contains details on version-specific files installed with the JClass LiveTable version for each JDK platform, compatibility with JDK and browser environments, and changes and known problems with this release.

1.5 Adding JClass LiveTable to Your IDE

JClass LiveTable works well with any JavaBeans-compliant Integrated Development Environment (IDE), including Symantec Visual Café, Inprise Borland JBuilder, IBM VisualAge for Java, Sybase PowerJ, and SuperCede for Java.

Once added to the development environment's component palette you can use JClass LiveTable the same way you use standard AWT or Swing components – adding them to forms, setting initial property values, specifying event-handling, and so on.

All environments provide a way to add components contained in a JAR or ZIP file to their component palette. The exact steps are unique to each environment so the best source for details is the documentation for your development environment. The JClass LiveTable JAR and ZIP files are located in the *\lib* subdirectory of where you installed JClass LiveTable.

1.5.1 Using Visual Café with JClass LiveTable

We recommend installing JClass LiveTable *after* installing Visual Café; this way, JClass LiveTable can be added to the Component Library automatically. The setup program copies the JClass LiveTable JAR file to Visual Café's *\bin\components* directory. (If you install Visual Café after installing JClass LiveTable, you can add the JAR to the Component Library manually as described in the Visual Café help.)

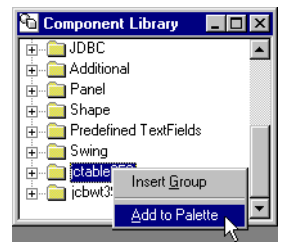
Replacing the Bundled JClass Components

It is important to note that installing this release **does not** automatically replace the older JClass BWT, JClass Chart, and JClass LiveTable components that are included with Visual Café (located in *\KLGroup\klg.jar*). If the bundled JClass components have been added to the Component Library, a newer version will not be shown in the Component Library.

To force Visual Café to replace the old JClass components in the Component Library, you must explicitly add *jctable[xxx]vc.jar* to the Component Library (**Insert | Component into Library...**).

Adding JClass LiveTable to the Component Palette

When JClass LiveTable is in the Component Library, you can add its components to the Component Palette to make them convenient to use. The following steps describe one easy way to add a new palette tab containing all of the JClass LiveTable components:



1. Display the Component Library window if it is not already visible (**View | Component Library**)
2. Right-click the "jctable[*xxx*]vc" folder and select **Add to Palette** from the popup menu. Visual Café creates a new tab on the Component Palette and adds all of the JClass LiveTable components to it.
3. You can rename the tab to make it easier to read. To do this, right-click the Component Palette, select **Customize Palette...** from the popup menu, and change the name of the "jctable[*xxx*]vc" folder to "JClass LiveTable".

Upgrading to a Newer Version of JClass LiveTable

Visual Café only allows one version of a component to be listed in the Component Library, so when you install a newer version of JClass LiveTable, it automatically replaces the older version in the Component Library (except for the version included with Visual Café; see Replacing the Bundled JClass Components for details).

When you reopen your project, it seamlessly uses the latest version of JClass LiveTable. There should generally be no problem using a newer version of JClass

LiveTable with an existing application. However, if you do experience problems, you can revert back to the previous version by moving the new version's JAR file out of the *\bin\components* directory (previous versions' JARs are not deleted).

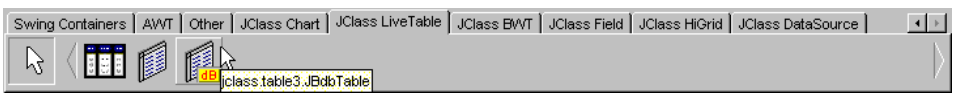
Note: You must add JClass LiveTable to the Component Palette again manually when you install a newer version.

Removing JClass LiveTable from the Component Library

Using the **Add/Remove Programs** dialog in the **Control Panel** does not remove JClass LiveTable from Visual Café. You must manually delete the JClass LiveTable JAR file from Visual Café's *\bin\components* directory and manually remove the JClass LiveTable tab from the Component Palette.

1.5.2 Using JBuilder with JClass LiveTable

We recommend installing JClass LiveTable *after* installing Borland JBuilder; this way, JClass LiveTable is added to the Component Palette automatically. If you install JBuilder after installing JClass LiveTable, you can add the JAR file to the Palette manually (**Tools | Configure Palette...**) as described in the JBuilder help.



Upgrading to a Newer Version of JClass LiveTable

When you install a newer version of JClass LiveTable, the Component Palette is automatically updated to use the new version. However, existing JClass LiveTable projects need to be reconfigured to use the new version, as outlined below:

1. With your project open, display the Project Properties dialog (**File | Project Properties...**).
2. Edit the Java Libraries list on the Paths tab to use the new version of the JClass LiveTable JAR file.
3. Save your project files.
4. Similarly, edit the default Java Libraries list (**Tools | Default Project Properties...**) to use the new version of the JClass LiveTable JAR file for new projects.

See the JBuilder help for complete details. There should generally be no problem using a newer version of JClass LiveTable with an existing application. However, if you do experience problems, you can revert back to the previous version in the Project Properties dialog.

Removing JClass LiveTable from JBuilder

Using the **Add/Remove Programs** dialog in the **Control Panel** does not remove JClass LiveTable from JBuilder. You must manually configure JBuilder to remove all references to JClass LiveTable:

- Remove JClass LiveTable from the default Java libraries list using the Default Project Properties... dialog (**Tools | Default Project Properties...**).
- To remove the JClass LiveTable tab from the Palette, right-click the palette, select **Properties...**, and click the **Remove** button.

1.6 Java and JavaBeans Basics

Java is both a compiled and an interpreted language. After writing a Java program using a text editor, save it as a source file with the extension *.java*. When this source file is run through the Java compiler, it compiles the file into a *.class* file. Unlike *.exe* files, these compiled *.class* files are not directly executable under any operating system, because they do not contain machine-language code that can be understood directly by the microprocessor. Instead, they are compiled into a byte-code format consisting of machine-language instructions designed for a virtual microprocessor. This virtual microprocessor is the Java Virtual Machine, which interprets the byte-code into a machine-language code that can be understood by your system's microprocessor. As long as the Java Virtual Machine software exists for a computing platform, any Java programs you create will be able to run on that platform.

If the Java compiler and CLASSPATH are properly configured, you can compile a Java program by running the Java compiler at the command prompt, for example:

```
javac MyJavaProgram.java
```

Java Applications and Applets

Java programs are usually one of two types: stand-alone applications and applets. Stand-alone applications can be run directly on a system containing the Java interpreter or Java runtime environment, while applets can be added to web pages for execution by a Java-compatible browser. JClass components can be used to create both types of Java programs.

JClass LiveTable and JavaBeans

JavaBeans(TM) is the software component model for Java. Introduced in JDK 1.1, the JavaBeans specification enables developers to create and use platform-independent, reusable software components on a wide variety of platforms and development environments. JClass LiveTable components are JavaBeans; they follow standard API naming conventions, the JavaBeans event model, and can easily be integrated with Java IDEs.

A good source of general information on Java and JavaBeans is the Frequently Asked Questions (FAQ) list that can be found at the JavaSoft Web site at *http://www.javasoft.com/products/jdk/faq.html* and *http://www.javasoft.com/beans/FAQ.html* respectively.

1.7 Moving from JClass LiveTable 2.x to JClass LiveTable 3.x

JClass LiveTable 3.0 and 3.6 include significant changes from the previous version. Nevertheless, it should not be difficult for you to use your JClass LiveTable 2.x applications in the new version because we have provided a transitional class called `JCTable`.

In most cases, your JClass LiveTable 2.x applications should run using LiveTable 3.x by simply calling `JCTable` instead of `Table` as the core class name. You will also need to change `import` statements to use the new package name, `jclass.table3`, instead of `jclass.table`.

For specific differences between JClass LiveTable 2.x and 3.x, and for further details on how to use `JCTable` please see Appendix C, Moving from JClass LiveTable 2.x to 3.x.

2

‘Hello Table’ — A Simple JClass LiveTable Program

You can immediately learn about some fundamental JClass LiveTable programming concepts by compiling and running an example program[1]. 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/Irish 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
KL Group	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.

2.1 The Basic Table

The following code is from *ExampleTable1.java*, found in the *examples\chapter2* directory of your JClass LiveTable installation directory. The code creates a very plain-looking table, without column labels or any other of JClass LiveTable's features to improve usability and appearance.

```
package jclass.table3.examples.chapter2;

// import the necessary java classes, including the Table package
import jclass.table3.*;
import jclass.contrib.ContribFrame;

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

// 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"},
    {"KL Group Inc.","12/12/97", "Colombian","22,000","$5.28"}
};

// initialize the Table object
    Table table;

// Build the table, point to the data source and define the table properties

    public ExampleTable1() {
        table = new Table();
        VectorDataSource ds = new VectorDataSource();
        table.setColumnLabelDisplay(false);
        table.setRowLabelDisplay(false);
        table.setDataSource(ds);
        ds.setNumRows(7);
        ds.setNumColumns(5);
        ds.setCells(cells);
        this.add(table);           // JDK 1.0.2 and 1.1 code
//      this.getContentPane().add(table);    // or code for Swing and JDK 1.2
        pack();
        show();
        }

    public static void main(String args[]) {
        new ExampleTable1();
    }
    public void addTableDataListener(TableDataListener l) {
    // the data source isn't going to change so this can do nothing
    }
    public void removeTableDataListener(TableDataListener l) {
    // the data source isn't going to change so this can do nothing
    }

}
```

The *ExampleTable1.java* program should be straightforward to Java programmers. The only difference between Java environments is the line that adds the table to its container – in JDK 1.0.2 and 1.1, we use "`this.add(table);`", but in JDK 1.2 and Swing, we use "`this.getContentPane().add(table);`"

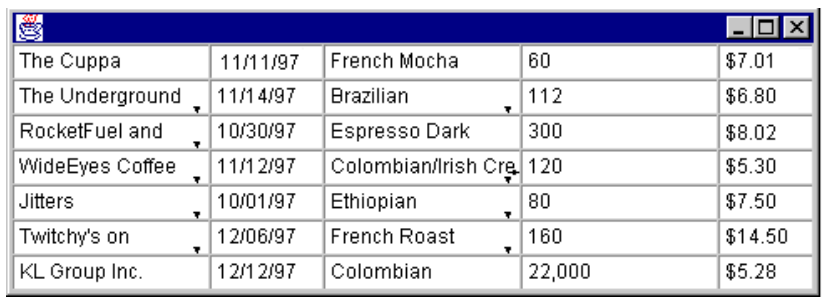
The table uses a Model-View-Controller (MVC) data mechanism (unlike JClass LiveTable 2); the table data is now stored in a separate object. In this case, we've used `VectorDataSource`, a class provided with JClass LiveTable that retrieves data from the data source and stores it in memory (see Using Stock Data Sources in Chapter 4 for more information).

The data source is set using the `table.setDataSource()` method. Once the data source is set to the `VectorDataSource` (ds) object, then that object handles the data, including setting the number of rows and columns, and accessing the cell values:

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

The data in the cells is of type String. The data source retrieves the data as Strings, but the data is invisibly translated for the table object to a specific cell data type called `TextCellData`. For small programs, this automatic translation is convenient. If you have large tables with many cells, you should specify the `CellData` data type for the cells. See Chapter 5, Displaying and Editing Cells for more information about `CellData` data types.

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



The Cuppa	11/11/97	French Mocha	60	$7.01
The Underground	11/14/97	Brazilian	112	$6.80
RocketFuel and	10/30/97	Espresso Dark	300	$8.02
WideEyes Coffee	11/12/97	Colombian/Irish Cre	120	$5.30
Jitters	10/01/97	Ethiopian	80	$7.50
Twitchy's on	12/06/97	French Roast	160	$14.50
KL Group Inc.	12/12/97	Colombian	22,000	$5.28

Figure 1 ExampleTable1.java

The clip arrows illustrate 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

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 jclass.table3.examples.chapter2.ExampleTable1`

cell. Notice that as you click in a cell, a *focus rectangle* appears, showing you the current cell.

If you resize any of the rows or columns to make them larger, the area required to display the cells is greater than the area contained in the table frame. The Table then attaches scrollbars so you can scroll the table display.

2.2 Improving the Table's Appearance

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()`.

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's easier to understand, easier to use, and more interesting to look at. The following sections explain how to set properties in a JClass LiveTable program. A sample program, called *ExampleTable2.java*, contains all of the changes outlined below.

2.2.1 Adding and Formatting Labels

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 can't tell what kinds of information the various cells contain because there are no column labels. In the original data outline for the table, 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, (Strings, images, Integers, etc.). You can apply labels to rows or columns. The label values, like cell values, are set in the data source object.

For the example program, set the labels as a String:

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

Once you have defined the values for the column labels, you have to tell the Table object to display labels. The *ExampleTable1.java* program currently contains the line:

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

Change this to:

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

Once the `ColumnLabelDisplay` property is set to `true`, you can set the column labels in the data source. Add the line:

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

After the line

```
ds.setCells(cells);
```

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

If you compile and run your amended *ExampleTable1.java* (or the sample *ExampleTable2.java*) you'll see that the column labels are now displayed. Notice that if you click on a label, you don't get the focus rectangle the way you do if you click on a cell: labels cannot be edited or traversed to. In certain circumstances, clicking on a label will perform an action (see Adding Interactivity below), but in this case the labels don't perform any interactive function.

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

Figure 2 Labels Displayed in ExampleTable2.java

Other than the absence of a focus rectangle, it's hard to see a difference between the labels and the cells. To visually differentiate label cells from the other cells, you can specify different background and foreground colors for the labels.

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
```

In order to make changes with AWT colors, you need to include the java.awt package if it has not already there. Insert the following line near the beginning of your code to declare the use of AWT packages:

```
import java.awt.*
```

To set the background color of the labels to blue, and the foreground color (text) to white, insert the following lines:

```
table.setBackground(JCTblEnum.LABEL, JCTblEnum.ALL, Color.blue);
table.setForeground(JCTblEnum.LABEL, JCTblEnum.ALL, Color.white);
```

The parameters `JCTblEnum.LABEL`, and `JCTblEnum.ALL` represent the *contexts*, of the property. In this case, the property is set for all of the labels. For more information on table contexts, see Setting and Getting Properties on page 49.

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

Figure 3 ExampleTable3.java, displaying labels with different colors

2.2.2 Changing Alignment

Another way to visually differentiate the text that appears within a table is to change its alignment within a cell relative to the default 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. If you want to set the labels in the sample program to appear horizontally centered and at the top of the cell, insert the following line:

```
table.setAlignment(JCTblEnum.LABEL, JCTblEnum.ALL,
                   JCTblEnum.TOPCENTER);
```

2.2.3 Changing the Fonts

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

In order to make changes with AWT fonts, you need to include the java.awt package if it has not already there. Insert the following line near the beginning of your code to declare the use of AWT packages:

```
import java.awt.*
```

To change the text column labels in the example program from their default to display in 14 point bold-italic Times Roman, insert the following line:

```
table.setFont(JCTblEnum.LABEL, JCTblEnum.ALL,
    new Font("TimesRoman", Font.BOLD + Font.ITALIC, 14));
```

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).

Having changed the alignment and font, your table should now look something like the following illustration:

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

Figure 4 ExampleTable4.java with new fonts and alignment for the labels

You can compile and run *ExampleTable4.java* to see the example with these additions.

2.2.4 Adding Color to an Individual Cell

In some cases, you will want the information in a certain cell or range of cells to stand out from the rest. The following code, when added to the improved example program, highlights premium coffee orders using different foreground and background colors (in this case, row 6, column 1 – 'Twitchy's on The Mall' will be highlighted):

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

Notice that row 6, column 1 displayed in the table are designated as row 5, column 0 (zero) 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).

2.2.5 Changing the Cell Borders and Spacing

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 in Border Types and Sides, in Chapter 3. For the example program, we're going to thicken the cell borders and change the border style for the cells (not labels) by adding the following lines of code:

```
table.setCellBorderWidth(5);
table.setCellBorderType(JCTblEnum.ALLCELLS, JCTblEnum.ALLCELLS,
    JCTblEnum.BORDER_ETCHED_OUT);
table.setFrameBorderWidth(3);
```

The example program should now resemble *ExampleTable5.java* when compiled and run:

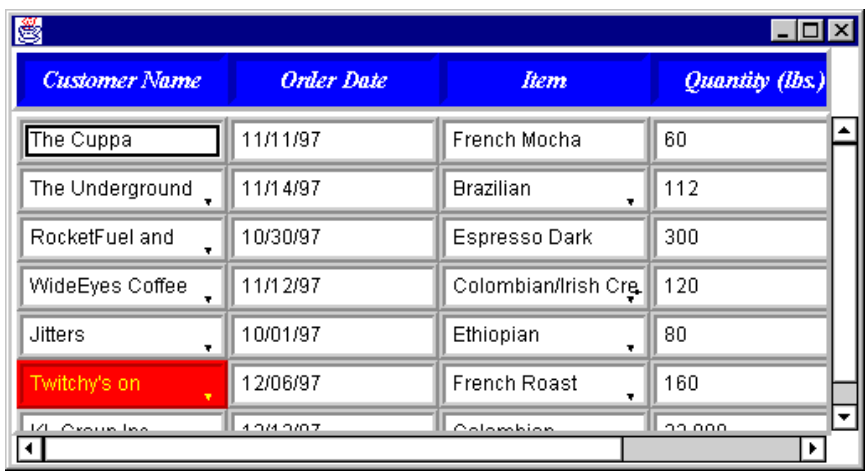


Figure 5 Example table with highlighted cell and new borders

2.2.6 Displaying More of the Cells

The example table has certainly come a long way with very few properties set, but there is still a small problem: the table only ever displays a single row of the cell's contents when you run the program. 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 10 characters wide and one row high. 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. In this program, however, we want the cells to size themselves to display the entire contents (if possible). You can do this using the following lines of code:

```
table.setPixelHeight(JCTblEnum.ALLCELLS, JCTblEnum.VARIABLE);
table.setPixelWidth(JCTblEnum.ALLCELLS, JCTblEnum.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. You can also set these properties to specific pixel values for rows and columns; see the section on how to set Row Height and Column Width in Chapter 3 for more details.

2.3 Adding Interactivity

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

We'll add some basic user-interactivity to our example table to give you a sense of some of the things JClass LiveTable can do. You can explore user-interactivity further in Chapter 6, Programming User Interactivity.

2.3.1 Making the Cells Editable

One problem with the table we have created is that it is not editable. When a user clicks on 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 `VectorDataSource` class we used as our data source has an editable counterpart called, appropriately enough, `EditableVectorDataSource`. We have to change our code to use this class instead. Simply change the line:

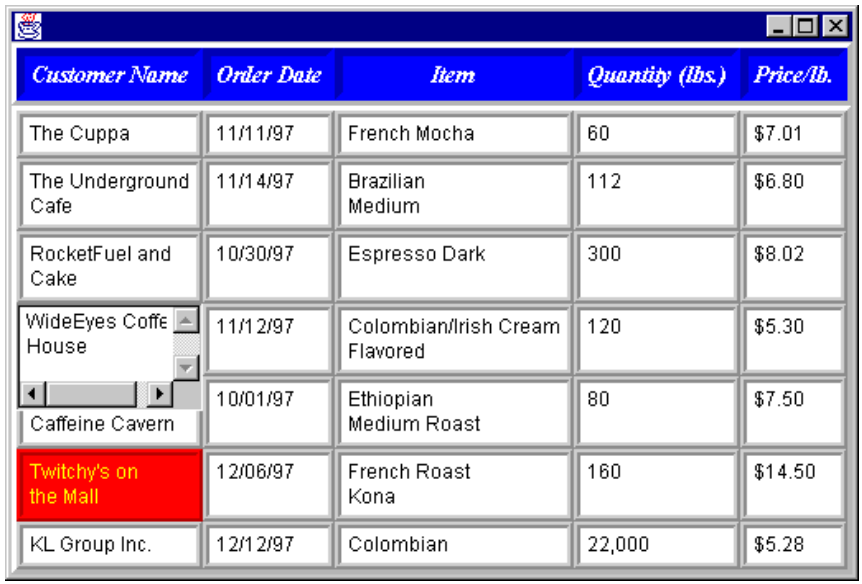
```
VectorDataSource ds = new VectorDataSource();
```

to use the `EditableVectorDataSource` interface instead:

```
EditableVectorDataSource ds = new EditableVectorDataSource();
```

Once you add this and compile the program, clicking in a cell will bring up the editing component for that type of `CellData`. Since all of the cells contain strings, the editing component is a text editor (see Chapter 5, Displaying and Editing Cells for more information).

Having added variable cell spacing and set your data source as editable, your table should now look much like the one compiled using *ExampleTable6.java*:



Customer Name	Order Date	Item	Quantity (lbs.)	Price/lb.
The Cuppa	11/11/97	French Mocha	60	$7.01
The Underground Cafe	11/14/97	Brazilian Medium	112	$6.80
RocketFuel and Cake	10/30/97	Espresso Dark	300	$8.02
WideEyes Coffe House	11/12/97	Colombian/Irish Cream Flavored	120	$5.30
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
KL Group Inc.	12/12/97	Colombian	22,000	$5.28

Figure 6 Editable cells sized to fit their contents. Note the editing component displayed over the cell.

2.3.2 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 adding the following code to the example program:

```
table.setSelectionPolicy(JCTblEnum.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. 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 Cell Selection in Chapter 6 for more information.

2.3.3 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. JClass LiveTable provides the `ResizeByLabelsOnly` property to enable this feature. In your example program, add the line:

```
table.setResizeByLabelsOnly(true);
```

When you compile and run the program, (or use *ExampleTable7.java*), you'll see that the mouse cursor becomes a "resize" cursor only over the borders of the column labels, and not over the cell borders.

2.3.4 Enabling Column Sorting

It might be easier for your users to find certain information if they can sort the table based on cell values in a column. That way they can find a customer name alphabetically, or determine large orders by sorting on the order amounts 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 (but not the label) with a mouse-click, you need a way to differentiate between a selection and a call to sort the column.

Using the ColumnTrigger and RowTrigger properties, you can allow users to sort the column by using a **Shift-click** combination:

```
table.setColumnTrigger(Event.SHIFT_MASK, LabelTrigger.SORT);
```

When you compile and run the program, (or use *ExampleTable8.java*) you will see that holding down the **Shift** key and clicking on a column label sorts the rows in ascending alphabetical/numerical order according to the contents of the column.

2.4 Distributing Applets and Applications on a Web Server

Once you have finished programming your Java applet or application, you will undoubtedly want to distribute it to your users. A common method of applet and application distribution is with your Web server. Here is a brief overview of how to deploy applets and applications, as well as reduce the size and customize the contents of the deployment archive[1].

1. Although the term "archive" has a somewhat ambiguous and flexible definition, for the purpose of this section, it refers to the JClass product JAR files.

2.4.1 Publishing an Applet on a Web Server

You can distribute your applet by putting the Web pages that contain it onto your Web server. Distributing your applet this way involves:

- creating directories for your JClass archive, HTML and class files
- copying the required JClass archive files to the Web server
- setting a CLASSPATH on the Web server
- copying the HTML and class files to the Web server
- ensuring that the HTML files properly reference the JClass archive and class files

Install the JClass Archives on the Server

First, you need to make sure that your CLASSPATH is not set. Although you will need to set it later when adding applets to the server, keep it undefined for now.

Create a JClass directory on your Web server (e.g. *\JClassLib*, just below the root document directory). This directory holds all of the archives that came with your JClass products.

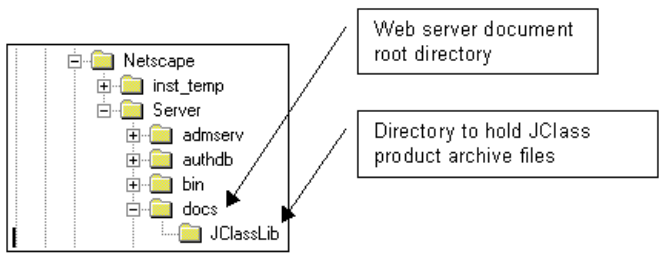


Figure 7 Example: suggested JClass archives folder name and location

Copy the JClass archive files to the newly created *\JClassLib* directory. The number and version of archives copied over, depend on which JClass products you own. These JAR files are found in the *\lib* directory of your JClass installation. Please refer to the Installed Files Overview section in the Getting Started chapter for more information about these files.

Preparing the directory for the applet

Create a directory for the applet classes and their HTML pages. It is important to keep the directory structure identical to the one found in the original location of the classes.

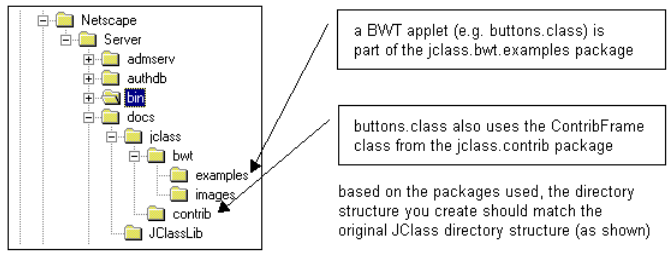


Figure 8 Example: proper applet class directory structure (using BWT)

Set a CLASSPATH on Your Web Server

If the applet reads local files from the Web server, the CLASSPATH needs to include the directory in which these files are located. As an example, if your applet uses images, the CLASSPATH needs to point to that images directory.

Install Your Applet Classes and HTML files on Your Web Server

Now that the directories have been created with the correct structure, you can copy over all of the required class and HTML files. The directories in which the class files are copied must be the same as the ones from where they are being copied. The HTML files can be placed together in a different location from the associated HTML files (as a suggestion, either the *\JClassLib* or *\JClass* will work fine), since they can point to class files in different locations.

Since your HTML files contain a JClass applet, and they might be located in a different directory from the associated class files, there are certain attributes that must be used to ensure that the file points to the proper JClass archive, class and location.

- `<ARCHIVE>`: The value given for this attribute is the path or URL of the JClass product archive (ZIP or JAR) that the applet requires to run.
- `<CODEBASE>`: You will need this if your applet is in a package or uses classes that are in other packages. The value of this attribute points to the 'top' of the directory structure that contains these classes and packages.
- `<CODE>`: The value of this attribute points to your applet class file.

Any printed or online HTML reference can provide more in–depth information about these attributes. Please refer to it if you need to.

For troubleshooting information about the above procedures, please refer to the JClass Knowledge Base on KL's Web site support area, and perform an online search for *Publishing JClass products on a Web Server*.

2.4.2 Using JarHelper to Customize the Deployment Archive

Deploying your applet or application does not end with copying the required class and HTML files to your Web server; the size of the archive should also be a consideration. The size of the archive, and its related download time are important factors to consider when deploying your applet or application on a Web server.

JarHelper is a utility that allows you to customize and reduce the size of the deployment archive. Using JarHelper, you can combine different JClass product JARs. As well, you can also choose which of the components found within one or more JClass product JARs will be included in the deployment archive. JarHelper takes the selected components of the JClass JAR(s), and creates a new, smaller file, which results in faster download times.

As an example, you can use JarHelper to exclude the enhanced editors and renderers from your new deployment JAR. Doing so would reduce your file size by approximately 200K, which would result in less server space being used, and faster download times.

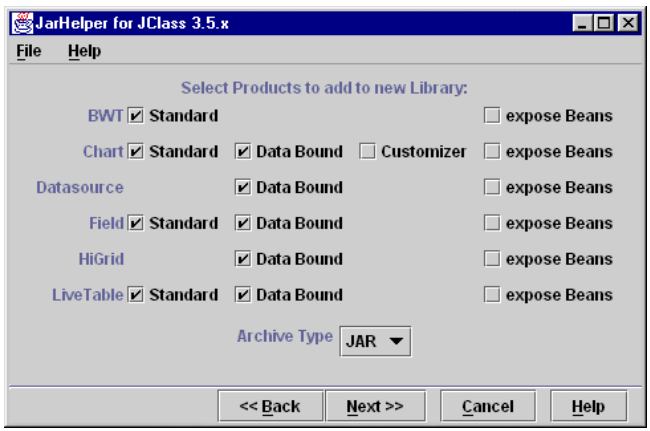


Figure 9 JarHelper's JClass product and component selection screen

JClass JarHelper comes with the JClass Enterprise Suite, and is installed automatically with the rest of the bundle's products. It is also available for download from KL Group's Web site for licensees of any JClass product.

Please refer to the *readme-jarhelper.txt* file for JDK and Swing requirements and installation.

Running JarHelper

Windows 95 and NT: Using the **Start** menu, navigate to the JClass JarHelper program group and select the **JarHelper.bat** (**DOS**) icon. You can also run JarHelper from a command line; the batch file is located in *JCLASS_HOME\bin\JarHelper.bat*.

Unix: Execute the shell script located in the *$JCLASS_HOME/bin* directory from a command prompt.

Using JarHelper

For more information about using JarHelper to create new JARs, please consult its online documentation.

2.5 Proceeding from Here

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

- For detailed information on the design elements of JClass LiveTable, see Chapter 3, Building a Table. Appendix B, JClass LiveTable Property Listing, contains the JClass LiveTable Properties in table format.
- To learn about using the new JClass LiveTable data model, see Chapter 4, Working with Table Data and Chapter 5, Displaying and Editing Cells.
- To learn about user-interaction with JClass LiveTable, see Chapter 6, Programming User Interactivity.
- To try this same tutorial in a JavaBeans development environment, see Chapter 9, JClass LiveTable Beans and IDEs.

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

3

Building a Table

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 JClass LiveTable Property Listing for a quick-reference summary of the properties.

Many of the table's properties are set using methods of the `Table` class. However, some properties are set in the data source. For more information setting properties using methods in the data source, see Chapter 4, Working with Table Data, and Chapter 5, Displaying and Editing Cells. 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.

3.1 Table Anatomy 101

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

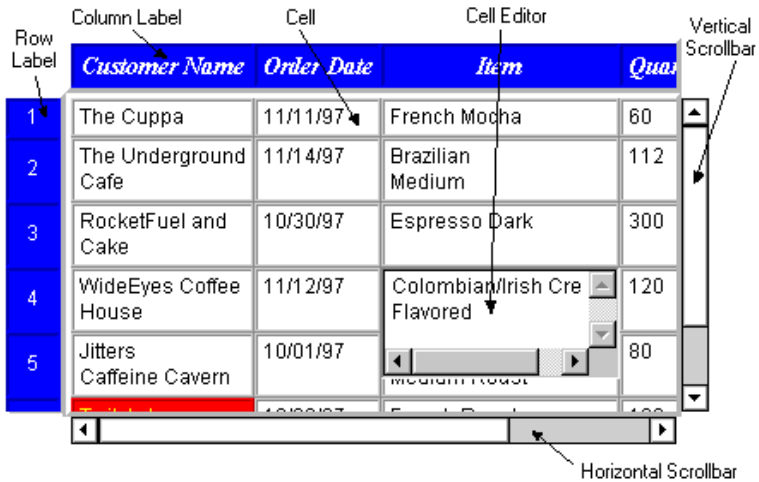


Figure 10 Components of a Table

The following list defines the terminology used throughout JClass LiveTable:

Cell
A cell is an individual container of table data. Cell drawing and editing is handled by the `jclass.cell` package (see Chapter 5, Displaying and Editing Cells). A cell is *visible* if it is currently scrolled into view. The entire collection of displayed cells is called the *cell area*.

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 and components, and can display an image or URL.

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).

3.2 JClass LiveTable Inheritance Hierarchy

The following figure provides an overview of class inheritance of JClass LiveTable.

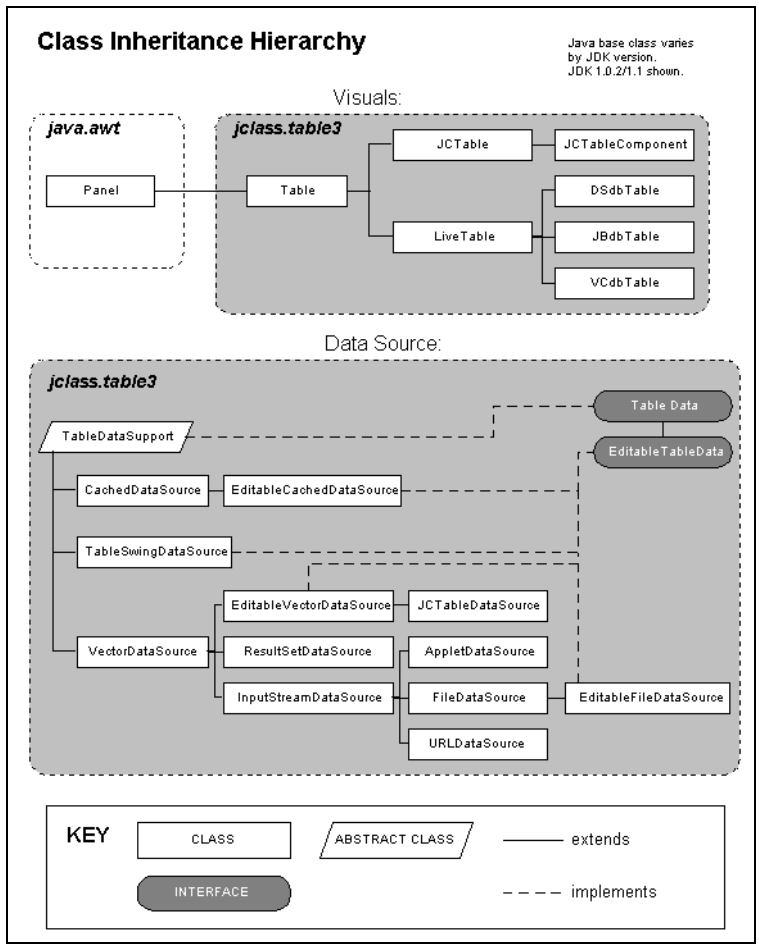


Figure 11 The jclass.table3 package

`TableData` is the core data source interface, and `EditableTableData` extends this interface to allow editing of the data. `VectorDataSource` stores table data in a series of vectors. `InputStreamDataSource` extends `VectorDataSource` to read from any stream, and `AppletDataSource`, `FileDataSource` and `URLDataSource` further extend it to read from specific stream types. The other stock data sources exist for other, more specific, situations.

The following figure describes the top–level inheritence variations for the different Java development platforms.

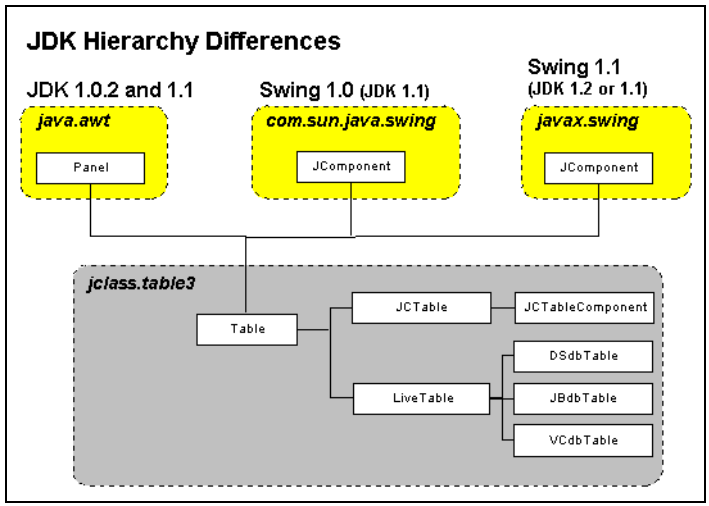


Figure 12 Hierarchy differences by JDK platform

3.3 Cell Management

JClass LiveTable displays and edits cells through the `jclass.cell.CellRenderer` and `jclass.cell.CellEditor` interfaces. It manages the display of the cell area by using two scrollbars.

3.3.1 CellRenderer and CellEditor

Cells are drawn into the cell area by a `CellRenderer` object that understands how to draw that specific type of data. If the user types or clicks in a cell, and there is a `CellEditor` for the data type of the cell, the editor component is drawn is displayed over the cell. See Chapter 5, Displaying and Editing Cells for more information on cell editors.

3.3.2 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 Chapter 6, Programming User Interactivity.

3.4 Setting and Getting Properties

There are three ways to set (and retrieve) JClass LiveTable properties:

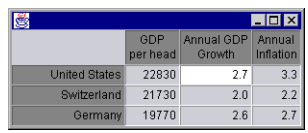
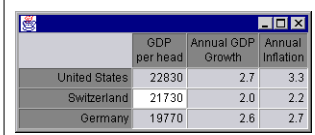
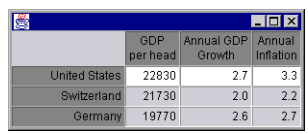
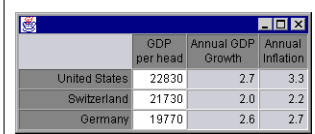
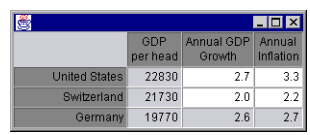
1. By calling property `set` and `get` methods in a Java program
2. By specifying applet properties in an HTML file
3. By using a Java IDE at design-time (JavaBeans)

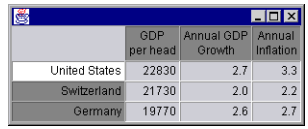
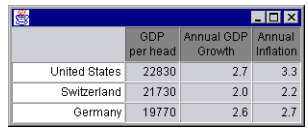
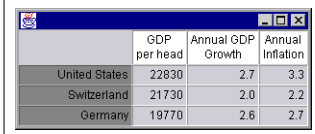
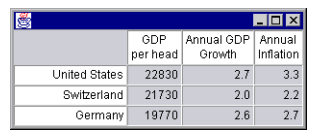
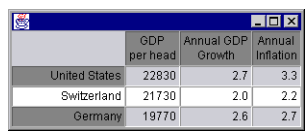
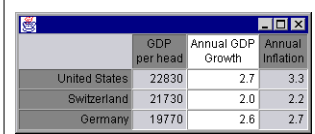
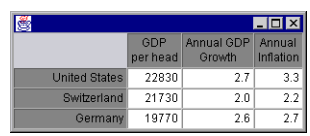
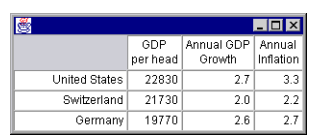
Each method changes the same table property. This manual therefore uses *properties* to discuss how features work, rather than using the method, Property Editor, or HTML parameter you might use to set that property.

3.4.1 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. `Context` defines the current context. Specifying a table context is part of any method that sets table properties.

The following outlines the valid table contexts available, using the `Background` property, with `Color.white` changes, as a working example (`table.setBackground(`context definition`,Color.white);`).

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 `JCTblEnum.ALLCELLS` in conjunction with a row or column index. This does not include labels.	`(0,JCTblEnum.ALLCELLS)` 	`(JCTblEnum.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) */` 	

Context selection	Examples	
a row or column label Referenced by the constant `JCTblEnum.LABEL` in conjunction with a row or column index.	`(0,JCTblEnum.LABEL)` 	`(JCTblEnum.LABEL,0)` 
all row or column labels Referenced by both `JCTblEnum.ALL` and `JCTblEnum.LABEL`, the order dependent on which set of labels is being referenced.	`(JCTblEnum.ALL,JCTblEnum.LABEL)` 	`(JCTblEnum.LABEL,JCTblEnum.ALL)` 
all labels Referenced using `(JCTblEnum.LABEL, JCTblEnum.LABEL)`.	`(JCTblEnum.LABEL,JCTblEnum.LABEL)` 	
an entire row or column Referenced by the constant `JCTblEnum.ALL` in conjunction with a row or column index. The context includes labels.	`(1,JCTblEnum.ALL)` 	`(JCTblEnum.ALL,1)` 
all table cells Referenced by `(JCTblEnum.ALLCELLS, JCTblEnum.ALLCELLS)`. The context does not include labels.	`(JCTblEnum.ALLCELLS,JCTblEnum.ALLCELLS)` 	
an entire table Referenced by `(JCTblEnum.ALL, JCTblEnum.ALL)`. The context includes labels.	`(JCTblEnum.ALL,JCTblEnum.ALL)` 	

3.4.2 Setting Cell/Label Properties with Java Code

Every JClass LiveTable property has a `set` and `get` method associated with it. For example, to retrieve the value of the `Background` property, the `getBackground()` method is called:

```
col = table.getBackground(0, 3);
```

The following code sets `Background` property:

```
table.setBackground(0, 3, Color.blue);
```

The following code shows setting properties for several contexts:

```
table.setBackground(JCTblEnum.ALL, JCTblEnum.ALL, Color.yellow);
table.setForeground(JCTblEnum.ALL, JCTblEnum.ALL, Color.blue);
table.setBackground(0, 3, Color.red);
table.setForeground(0, 3, Color.yellow);
```

In this case, all of the backgrounds in the table are set to yellow, and their foregrounds (text) are set to blue. The subsequent `Background` and `Foreground` members override this general condition by setting the background to red and the foreground to yellow for the cell located at `0, 3`.

You can set some properties for a range of cells defined by a `JCCellRange`. If this is possible for a specific property, it will be noted in this manual.

Generally, to set a property for a `JCCellRange`, the code would resemble the following:

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

3.4.3 Setting Applet Properties in an HTML File

Another way to set table properties, particularly appropriate for applets, is in an HTML file. Applets built with JClass LiveTable can parse applet `<PARAM>` tags and set the table properties defined in the file. (A pre-built applet called *JCTableApplet.class* is provided with JClass LiveTable). Even standalone Java applications can make use of HTML parameters as a debugging tool.

Using HTML to set properties has the following benefits:

- Speed – see the effect of different property values quickly without recompiling.
- Flexibility – use a single applet class to create many different kinds of tables simply by varying HTML properties; end-users can modify HTML properties to suit their own needs.

Table properties are coded in HTML as applet `<PARAM>` tags. The `NAME` element of the `<PARAM>` tag specifies the property name; the `VALUE` element specifies the property value to set.

Most JClass LiveTable properties can be set in an HTML file. The HTML property parameters match the API property names. Properties that cannot be set in HTML are generally too complex or too obscure for end-users to want to modify them. The

following example HTML file specifies a complete table entirely through HTML parameters:

```
<HTML>
<HEAD>
<TITLE>Quote.Com Example</TITLE>
</HEAD>
<BODY BGCOLOR=#FFFFFF>
<h1>Quote.Com Portfolio Example</h1>
Initial Prototype based on the data and configuration at
<A HREF="http://www.javaworld.com/javaworld/quote.com/
jw-quote.html">http://www.javaworld.com/javaworld/quote.com/
jw-quote.html</A>.
<P>
<APPLET CODE="jclass/table3/JCTableApplet.class" codebase="../../.."
height=250 width=550>
<!-->
<! Stock sample similar to -->
<! http://www.javaworld.com/javaworld/quote.com/jw-quote.html  -->
<! o bordersides top and bottom -->
<! o border width = 1 -->
<PARAM NAME=columnLabelSort VALUE=true>
<PARAM NAME=ColumnLabels
  VALUE="*|symbol|price|change|volume|open|high|low|time">
<PARAM NAME=VisibleRows VALUE=7>
<PARAM NAME=VertScrollbarOffset VALUE=0>
<PARAM NAME=HorizScrollbarOffset VALUE=0>
<PARAM NAME=BorderTypeList VALUE="(ALL ALL BORDER_FRAME_OUT)
  (LABEL ALL BORDER_OUT)">
<PARAM NAME=BorderSidesList VALUE="
     (ALLCELLS ALLCELLS BORDERSIDE_top +  BORDERSIDE_bottom)">
<PARAM NAME=shadowThickness VALUE=1>
<PARAM NAME=frameShadowThickness VALUE=1>
<PARAM NAME=frameBorderType VALUE=border_out>
<PARAM NAME=ForegroundList VALUE="(ALL ALL black)(label all white)
(0 3 red)(1 3 #006600)(2 3 #006600)(3 3 red)(4 3 red)(5 3 red)(6 3
red)">
<PARAM NAME=BackgroundList VALUE="(ALL ALL lightgray)
  (LABEL ALL #803366)">
<PARAM NAME=PixelHeightList VALUE="(ALL variable)">
<PARAM NAME=PixelWidthList VALUE="(ALL variable)">
<PARAM NAME=FontList VALUE="(label all dialog-bold-14)">
<PARAM NAME=AlignmentList VALUE="(ALL ALL center)(all 1 left)
  (ALL 4 right)">
<PARAM NAME=datatypeList VALUE="(all 2-3 type_double)
  (all 4 type_integer)(all 5-7 type_double)">
<PARAM NAME=columnLabelOffset VALUE=3>
<PARAM NAME=cells VALUE="
([IMAGE=down.gif]|BORL|7.125|-0.125|103400|7.375|7.375|7.125|17:28)
([IMAGE=up.gif]|SYMC|10.5625|0.3125|321200|10.5|10.75|10.25|17:25)
([IMAGE=up.gif]|ORCL|40.625|0.75|4472900|40.125|40.875|40.875|17:32)
([IMAGE=down.gif]|NSCP|46.875|-1.125|1281300|48.5|48.75|45.5|17:32)
([IMAGE=down.gif]|ADBE|32.25|-0.875|558700|33.25|33.625|31.625|17:27)
([IMAGE=down.gif]|USRX|57|-2.125|2086800|60.125|60.5|56.75|17:31)
([IMAGE=down.gif]|DIS|58.5|-0.5|784100|59|59|58.25|17:31)
">
</APPLET>
<P>
</BODY>
</HTML>
```

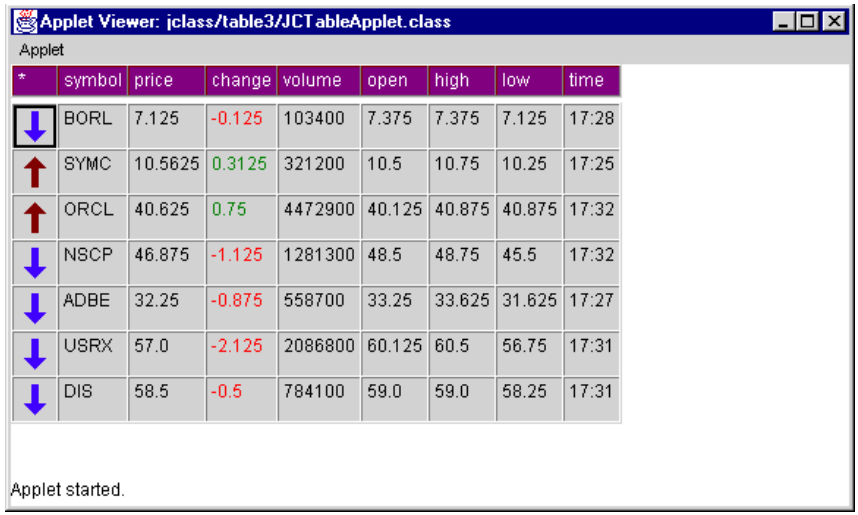


Figure 13 The applet displayed in JCQuote.html using AppletViewer

Several example HTML files are located in the *jclass/table3/applet/* directory.

3.4.4 Setting Properties with a Java IDE at Design-Time

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

See JClass LiveTable Beans and IDEs, in Chapter 9 for complete details on using JClass LiveTable's JavaBeans in IDEs.

3.5 Preset Table Styles

You can quickly build a standard table with a number of default settings by setting the table to a specific mode, which is overridden by any properties you specifically set later in your program. These styles are set using the `setMode()` method using one of the following parameters:

- `JCTblEnum.MODE_TABLE`
- `JCTblEnum.MODE_LIST`

The `setMode()` method sets the initial state of the `Table`. Any properties you set after calling `setMode()` will override the properties set by the above parameters. Because of this, you cannot call `setMode()` to 'reset' the table to default properties.

Table Mode

This is the default mode for JClass LiveTable programs. By default, calling `setMode(JCTblEnum.MODE_TABLE)` produces a table with the following default properties:

- The horizontal and vertical scrollbars are both attached to the table cells and end at the edge of the visible cells (see Attaching Scrollbars, in Chapter 6).
- Cell and cell range selection is not enabled.
- The `CellBorderWidth` property of the cells is set to 1 pixel.
- Cells are traversable (see Controlling Interactive Traversal, in Chapter 6).

List Mode

List mode essentially displays a table as a multicolumn list. The following are default settings in List Mode:

- The horizontal and vertical scrollbars are both attached to the side of the table and end at the edge of the table frame (see Attaching Scrollbars, in Chapter 6).
- Users can select single cells; when a cell is clicked on, it selects the entire row (see Cell Selection, in Chapter 6).
- The `CellBorderWidth` property of the cell borders is set to 0; so borders are not displayed (see Border Types and Sides).
- Cells are not traversable (see Cell Traversal, in Chapter 6).
- The `ResizeByLabelsOnly` property is set to `true`, since there are no cell borders to use for resizing the individual cells (see Controlling Resizing, in Chapter 6 for more details).

3.6 Defining Rows and Columns

3.6.1 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 `VectorDataSource.getNumRows()` and `VectorDataSource.getNumColumns()` methods. Please see Setting Stock Data Source Properties, in Chapter 4 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 Section 3.6.4, Specifying 'Frozen' Rows and Columns.

3.6.2 Setting Visible Rows/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()`.

If you set the number of visible rows or columns to greater than the corresponding `NumRows` and `NumColumns` properties, a theoretical value is calculated based on 10-character columns and one-line rows.

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

Displaying the Entire Table

To display the entire table, set `VisibleRows` and `VisibleColumns` to `JCTblEnum.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. Any change to the number of rows/columns in the table will cause the table to attempt to resize when this flag has been set.

3.6.3 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.

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

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

3.6.4 Specifying 'Frozen' Rows and Columns

An application can make rows and columns non-scrollable 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 Using Spanning to Create Multi-line Headers below 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. They are still editable and traversable unless set otherwise. The following figure shows an example of frozen rows.

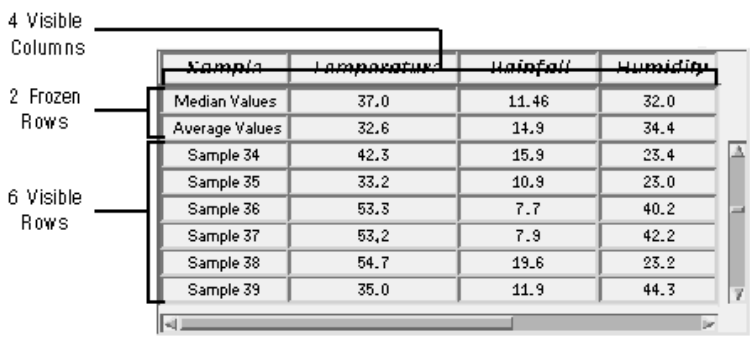


Figure 14 Visible and Frozen Rows and Columns-note absence of scrollbar to right of frozen rows

Freezing Rows and Columns

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

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

Freezes the first two rows of the table.

```
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 4) 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 `JCTblEnum.PLACE_TOP` or `JCTblEnum.PLACE_BOTTOM`.

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

3.7 Adding Row and Column Labels

A row or column label is a non-editable cell that identifies the row or label to the user. Row and column label values are set in the data source (see Chapter 4, Working with Table Data). 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 the methods:

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

3.7.1 Label Placement and Spacing

Label Placement

You can specify the positioning of row/column labels on the screen using the `setRowLabelPlacement()` and `setColumnLabelPlacement()` methods. Valid values include:

- `JCTblEnum.PLACE_TOP` - label displayed at top of table
- `JCTblEnum.PLACE_BOTTOM` - label displayed at bottom of table
- `JCTblEnum.PLACE_LEFT` - label displayed to left of table
- `JCTblEnum.PLACE_RIGHT` - label displayed to right of table

The next figure displays the effect of reversing the default row/column label placement by using the following lines:

```
table.setColumnLabelPlacement(JCTblEnum.PLACE_BOTTOM);
table.setRowLabelPlacement(JCTblEnum.PLACE_RIGHT);
```

Item Description	SKU #	Qty	
Magnetic Porcelain Whiteboard 48"x72"	940-874	100	1
Elbattide Adjustable Footrest	940-538	50	2
Premium Leather Business Card Holder (holds 120)	940-249	250	3
BIC-mate Premium Ballpoint Pens (pkg. 10)	940-859	1000	4
Foldback Binder Clips (pkg. 20)	940-903	2500	5

Figure 15 Reversing the default row/column label placement

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 labels overlap the cell area.

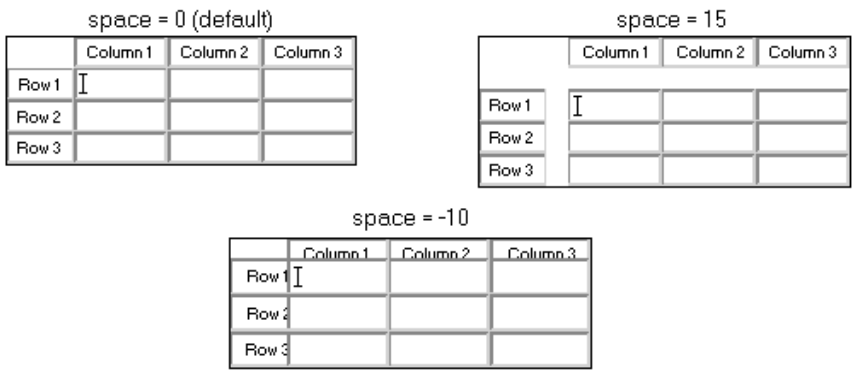


Figure 16 Row and column label spacing

3.8 Row Height and Column Width

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.

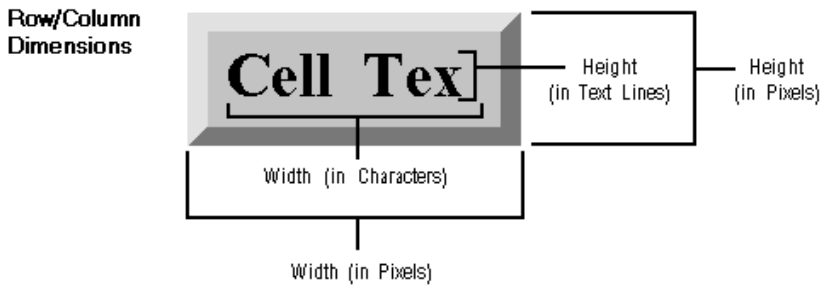


Figure 17 Difference between Character and Pixel Row/Column specification

3.8.1 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 `JCTblEnum.NOVALUE`.

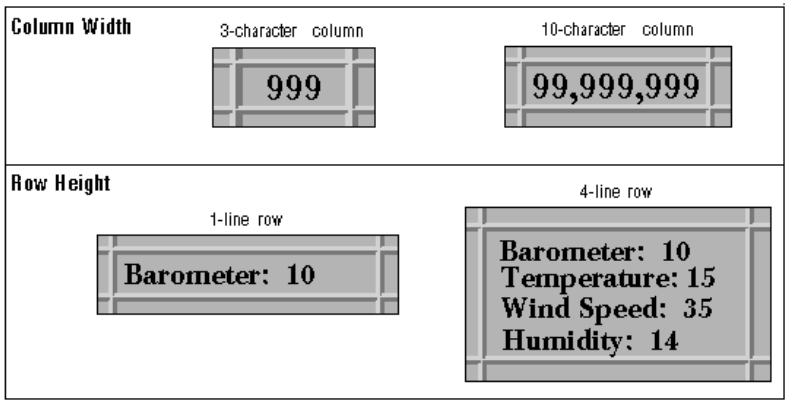


Figure 18 Character specification of row height

Character specification is convenient when you know how many characters you want a row/column to display. It works best with non-proportional[1] 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.

The following example sets the width of the third column to 15 characters:

```
table.setCharWidth(2, 15);
```

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

3.8.2 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, `JCTblEnum.NOVALUE`, `JCTblEnum.VARIABLE`. (This value is discussed in detail in the following section).

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

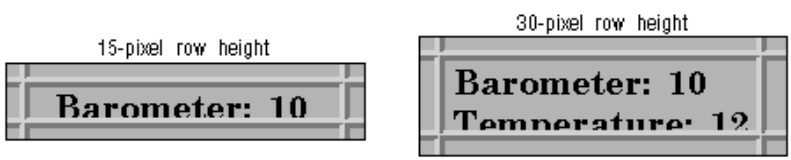


Figure 19 Pixel specification of row height

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

The following code sets the width of the second column to 120 pixels:

```
table.setPixelWidth(1, 120);
// Wraps the text
table.setMultiline(JCTblEnum.ALLCELLS,1,true);
```

3.8.3 Variable 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 `JCTblEnum.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 `getColumPixelWidth()` methods.

3.8.4 Multiple Lines in Cells

When you set the height and width of your cells, you necessarily 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. This is accomplished by setting the `Multiline` property with a `boolean` value (default: false).

To set the Multiline property, call `setMultiline()` for a particular context:

```
table.setMultiline(JCTblEnum.ALLCELLS,JCTblEnum.ALLCELLS,true);
```

You can also set this property for a JCCellRange range of cells, as in:

```
JCCellRange range = new JCCellRange(0,3,2,4);
table.setMultiline(range,true);
```

If the data displayed in the cells contains a newline character (\n), the cell is automatically displayed as a multiline cell regardless of the value of the `Multiline` property.

For rendering, the data determines whether multiple lines are displayed (because of the newline character). For editing, the boolean value determines whether multiple lines are used.

3.8.5 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.[2] Though the row/column appears to have vanished, the application can set attributes or change cell values. The next figure illustrates the effect.

---

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

2. Users cannot interactively hide or reveal rows/columns.

The current cell should not be in the hidden row/column.

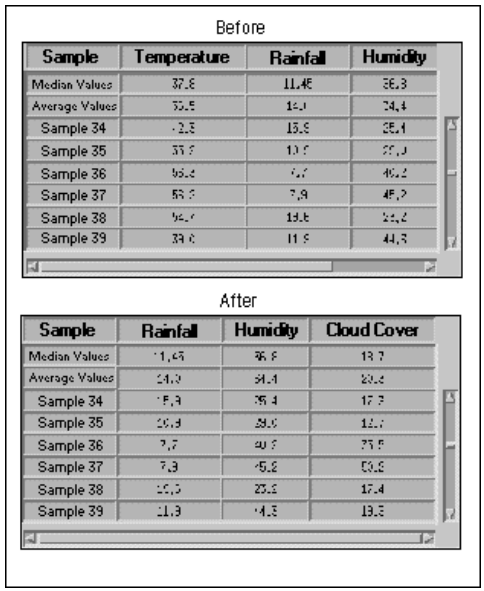


Figure 20 Hiding the "Temperature" column

To reveal a hidden row/column, set the pixel height or width to a pixel value or `JCTblEnum.NOVALUE` (to use the character specification defined for the row/column).

3.8.6 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:

- `JCTblEnum.EDIT_SIZE_TO_CELL`: resize the component to fit the Table's cell size
- `JCTblEnum.EDIT_ENSURE_MINIMUM_SIZE`: resize the component to its minimum size
- `JCTblEnum.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 `jclass.cell.CellEditor` interface. For more information about cell editors, see Chapter 5, Displaying and Editing Cells.

3.9 Colors

3.9.1 Foreground and Background Colors

The foreground and background colors used for cells are specified by the `Foreground` and `Background` properties. The following example displays the effect of setting the background color of column 2 to blue, and the foreground color for cell (1, 3) to white:

```
table.setBackground(JCTblEnum.ALL, 1, Color.blue);
table.setForeground(0, 3, Color.white);
```

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);
table.setBackground(range, Color.red);
```

3.9.2 Color of Selected Cells

The foreground and background colors used for selected cells are specified by `SelectedBackground` and `SelectedForeground`. By default, selected cells are displayed with the table's reversed default colors. The background color of the selected cell is the table's default foreground color (black), and the foreground color of the selection is the default background color (gray). The current cell displays the selection colors in its border.

When `SelectedBackground` or `SelectedForeground` is set to `null`, selected cells look identical to unselected cells–the background and foreground colors are the same as the colors defined for the cells. Other cell properties, such as cell fonts, can be changed for selected cells.

3.9.3 Focus Rectangle Color

You can change the color of the focus rectangle (the rectangle drawn inside a cell when it is traversed to) using the `setFocusRectColor()` method:

```
setFocusRectColor(Color.blue);
```

3.9.4 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` property as shown by the following example:

```
Color[] c1 = {Color.orange, Color.green, Color.magenta};
table.setRepeatBackgroundColors(c1);
table.setBackground(JCTblEnum.ALLCELLS, JCTblEnum.ALLCELLS,
    JCTblEnum.REPEAT_COLUMN);
```

A list of repeating foreground colors can be created by setting the `RepeatForegroundColors` property.

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

`JCTblEnum.REPEAT_ROW` repeats colors in row order, whereas `JCTblEnum.REPEAT_COLUMN` repeats colors in column order. The following illustration displays the effect of each setting.

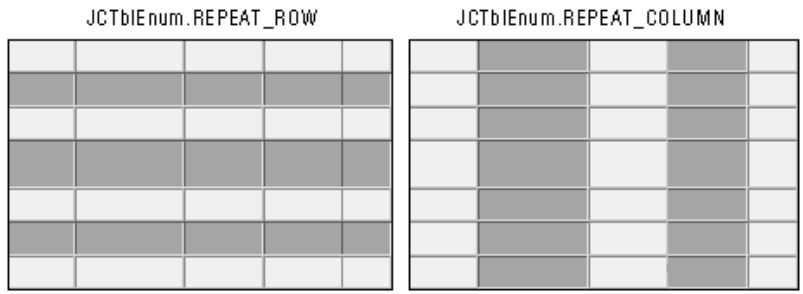


Figure 21 Repeating background colors

To set the foreground colors for the current context to a repeating color list defined by `RepeatForegroundColors`, set `ForegroundContext` to `JCTblEnum.REPEAT_ROW` or `JCTblEnum.REPEAT_COLUMN`.

3.10 Cell and Label Text Alignment

The horizontal and vertical alignment of text (or images) within cells and labels is specified by the `Alignment` property. Cell/label values can be centered or positioned along any side of the cell/label. Valid values for `Alignment` are:

- `JCTblEnum.TOPLEFT` (default)
- `JCTblEnum.TOPCENTER`
- `JCTblEnum.TOPRIGHT`
- `JCTblEnum.MIDDLELEFT`
- `JCTblEnum.MIDDLECENTER`
- `JCTblEnum.MIDDLERIGHT`
- `JCTblEnum.BOTTOMLEFT`
- `JCTblEnum.BOTTOMCENTER`
- `JCTblEnum.BOTTOMRIGHT`

The following figure shows how the values of this property affect the text display.

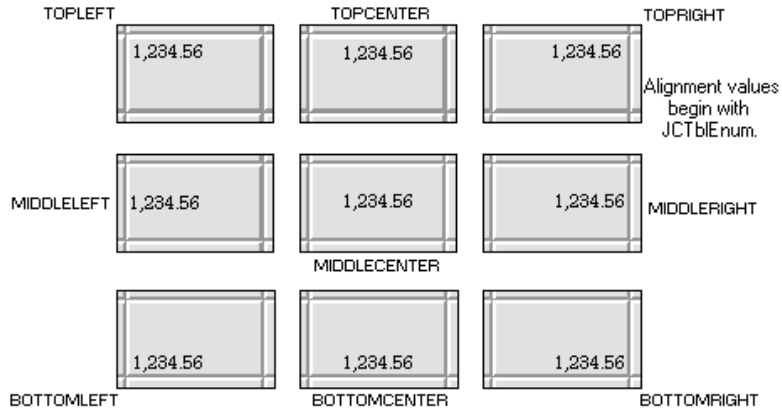


Figure 22 Text alignment positions all position values begin with JCTblEnum.

In addition to being able to set alignment on row or column–indexed contexts, you can set the `Alignment` property for a range of cells specified by a `JCCellRange` object:

```
JCCellRange range = new JCCellRange(0,2,1,2);
table.setAlignment(range, JCTblEnum.BOTTOMCENTER);
```

3.11 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 separate font for all column labels:

```
table.setFont(JCTblEnum.LABEL, JCTblEnum.ALL,
new Font("TimesRoman", Font.ITALIC, 20));
```

You can also set the `Font` property for a range of cells specified by a `JCCellRange` object:

```
JCCellRange range = new JCCellRange(1, 3, 1, 7);
table.setFont(range, new Font("TimesRoman", Font.BOLD, 10));
```

JClass LiveTable can use any of the fonts available to Java. See your Java documentation for details on finding and setting fonts.

3.12 Border Types and Sides

All cells and labels have a border around them. The visual look of the border, and the sides on which it appears, can be customized for individual cells and labels. The border width is specified for the entire table.

In addition, the table frame, which encloses the cells and labels, can have the visual look of its border customized.

3.12.1 Cell and Label Border Types

The `CellBorderType` property specifies the type of border drawn around cells or labels. The `CellBorderType` is defined a number of ways, the simplest being the `setCellBorderType()` method, which takes a row and column value and one of the following `JCTblEnum` border type parameters:

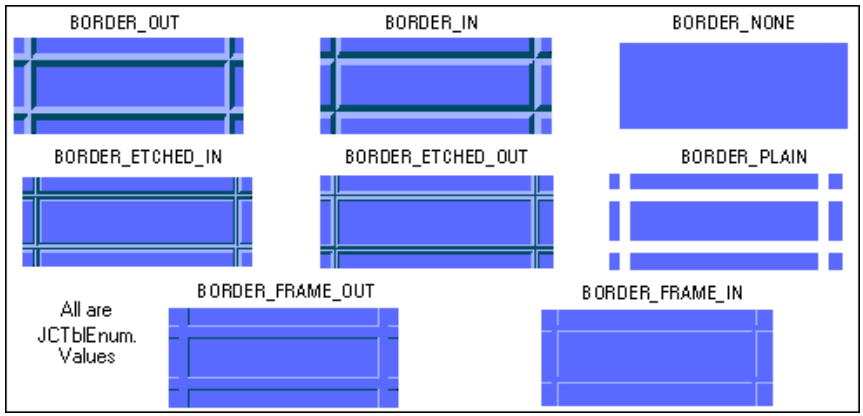


Figure 23 Border Types

The following example sets a blank border for a row:

```
table.setCellBorderType(2, JCTblEnum.ALLCELLS,
  JCTblEnum.BORDER_NONE);
```

The above method is actually a convenience method that uses a class called `StandardCellBorder`. The `StandardCellBorder` class implements the `CellBorder` interface and defines the above border types for you.

You can also set `CellBorderType` for a range of cells specified by a `JCCellRange` object. The following lines set a standard border type for a range of cells, note that here we use a `StandardCellBorder` object:

```
JCCellRange range = new JCCellRange(2, 3, 2, 8);
table.setCellBorderType(range, new
  StandardCellBorder(JCTblEnum.BORDER_FRAME_OUT));
```

Note: To see the effect of the `ETCHED` and `FRAME` borders, `CellBorderWidth` must be set to a value greater than 5 pixels.

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

3.12.2 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 `CellBorder` 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 `CellBorder` interface. The following (from the *BorderTypes.java* example in *examples\misc*) defines a single-line border object called `LiteBorder`:

```
class LiteBorder implements CellBorder {

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);
}

}
```

To apply this custom border to a cell, use:

```
table.setCellBorderType(3, 4, new LiteBorder(Color.gray));
```

You can also apply it to a range of cells defined by a `JCCellRange`:

```
JCCellRange range = new JCCellRange(1, 3, 1, 7);
table.setCellBorderType(range, new LiteBorder(Color.black));
```

The *examples\misc* 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.

3.12.3 Cell and Label Bordercells Width

The width of the borders around the cells and labels is specified by `CellBorderWidth`. This property applies to the entire table. By default, the borders are 1 pixel wide. The following image shows the visual effect of different border widths.

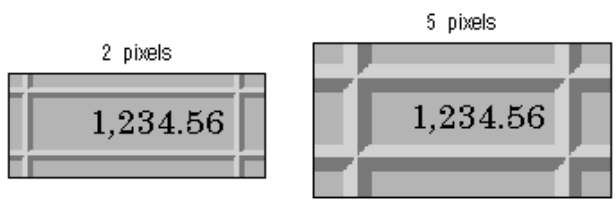


Figure 24 Setting CellBorderWidth

Use the `setCellBorderWidth()` method to set the pixel value:

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

3.12.4 Cell and Label Border Sides

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

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

Figure 25 Customized Cell Borders

The valid values for `CellBorderSides` are:

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

Specifying border sides is accomplished by OR-ing together all of the sides to have borders, for example:

```
table.setCellBorderSides(JCTblEnum.ALL, 0,
  JCTblEnum.BORDERSIDE_LEFT | JCTblEnum.BORDERSIDE_TOP |
  JCTblEnum.BORDERSIDE_BOTTOM);
```

You can also set `CellBorderSides` for a range of cells specified by a `JCCellRange` object:

```
JCCellRange range = new JCCellRange(2, 3, 2, 8);
table.setCellBorderSides(range, JCTblEnum.BORDERSIDE_LEFT |
  JCTblEnum.BORDERSIDE_TOP);
```

3.12.5 Frame Border Attributes

The `FrameBorderType` property specifies the border type for the frame enclosing the cell and label areas. Its possible values are:

- `JCTblEnum.BORDER_NONE (default)`
- `JCTblEnum.BORDER_PLAIN`
- `JCTblEnum.BORDER_OUT`
- `JCTblEnum.BORDER_ETCHED_OUT`
- `JCTblEnum.BORDER_IN`
- `JCTblEnum.BORDER_ETCHED_IN`

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.

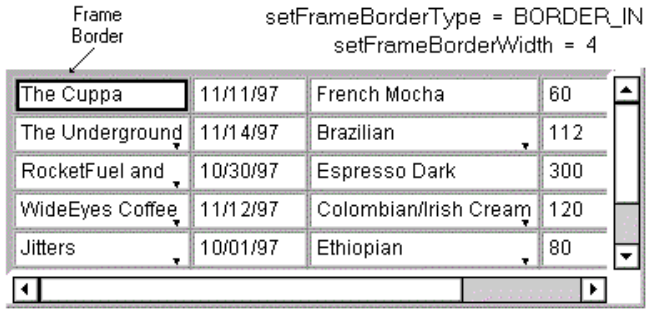


Figure 26 A table with frame border attributes set

3.13 Cell and Label Margins

You can 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 top and bottom 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 left/right edge of the cell's contents (default: 3). The next illustration shows the appearance of different margins. These properties affect all cells/labels in the table–margins cannot be set for individual cells.

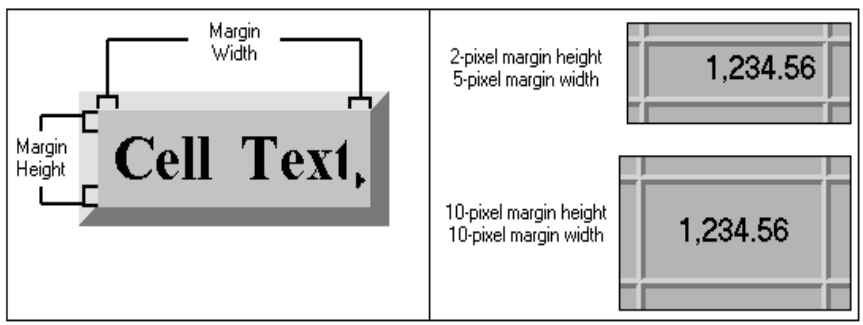


Figure 27 Margin Height and Margin Width

3.14 Displaying Images in Table Cells

JClass LiveTable can display an image in each cell or label in the table. The image appears inside the margin of the cell. Images are displayed using the `ImageCellRenderer` class in the `jclass.cell` package. For more information, please see Chapter 5, Displaying and Editing Cells.

3.14.1 Image Format

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

3.14.2 Image Layout

The position of the image within the cell is specified in the same way as Strings, using `Alignment`. The next figure shows how the values of this property change the positioning of the image.

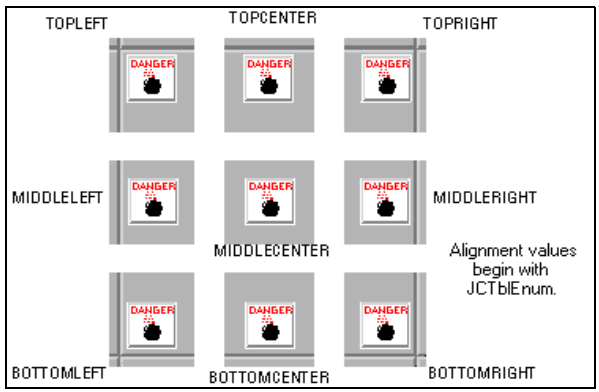


Figure 28 Image layouts

In addition to being able to set alignment of images on row or column–indexed contexts, you can set the `Alignment` property for a range of cells specified by a `JCCellRange` object:

```
JCCellRange range = new JCCellRange(0,2,1,2);
table.setAlignment(range, JCTblEnum.BOTTOMCENTER);
```

3.15 Text and Image Clipping

When cell and label values don't fit in their cells, JClass LiveTable can clip the display of the cell value. The `ClipArrows` property determines which method is used. The `setClipArrows()` method can take the following values:

- `JCTblEnum.CLIP_ARROWS_RIGHT`
- `JCTblEnum.CLIP_ARROWS_DOWN`
- `JCTblEnum.CLIP_ARROWS_BOTH` (default)
- `JCTblEnum.CLIP_ARROWS_NONE`

3.16 Cell and Label Spanning

Spanning is a way to join a range of cells or labels together and treat them as one 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 multi-line 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, covering 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.

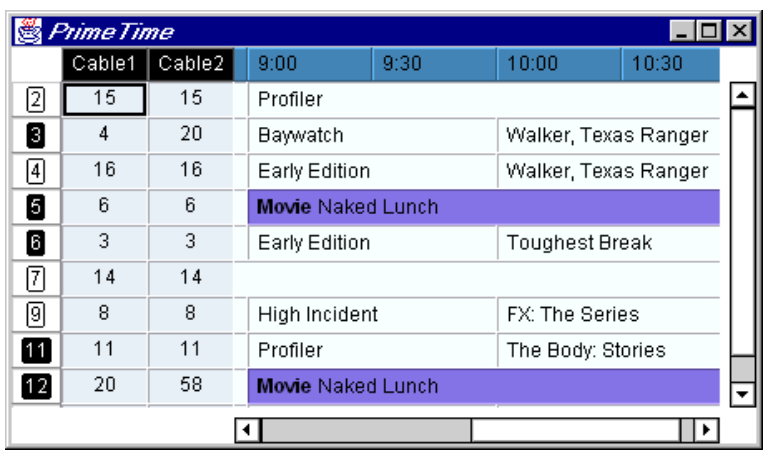


Figure 29 Table design using spanned cells

The `setSpans()` method is used to set a `Vector` of ranges of cells or labels. 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.

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 `setSpans()` method with a `null` value.

Each item in a span list is an instance of a `JCCellRange`. A `JCCellRange` object defines the start and end columns/rows for the specified range.

The following example defines a cell that spans three columns and four rows (columns 2 through 4, and rows 2 through 5):

```
Vector spans = new Vector();
spans.addElement(new JCCellRange(1, 1, 4, 3));
table.setSpans(spans);
```

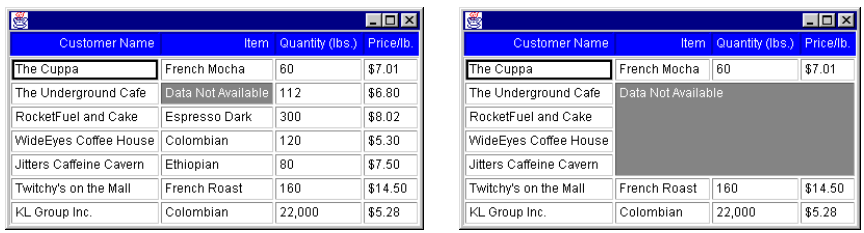


Figure 30 Color properties of source cell (1,1) in the original table (left) are retained over the spanned cells in the table after the code has been added (right)

3.16.1 Using Spanning to Create Multi-line Headers

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

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

Figure 31 Multi-line headers

While JClass LiveTable does not support multi-row column labels, this effect can be achieved by using a frozen row at the top of the table to mimic the appearance of the column labels as follows:

- The rightmost 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, the `CellBorderSides`, `Background`, and `Foreground` properties for the column labels and row zero are all set to blend the two together.

4

Working with Table Data

4.1 Overview: Data Handling in JClass LiveTable

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

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

With LiveTable's MVC architecture, the data source object is the Model, which manages the underlying data being displayed and manipulated. The `Table` 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).

This method of handling data is different from the method used by JClass LiveTable 2.x. For information about using JClass LiveTable 2.x programs and data in this release, please see Appendix C, Moving from JClass LiveTable 2.x to 3.x.

Because the `Table` object and the data source are separated, you are free to use whatever data storage mechanism you want; the `Table` 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 `Table` object. The data source is able to

copy only the data that is currently displayed by the `Table` object. An external data source can also manage large sets of data more efficiently than the `Table` object can.

4.1.1 How the Table and Data Source Communicate

Between the `Table` object and the data source lies another object called the `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 `Table` 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.

4.2 Getting Data into your Table

To display data in a JClass LiveTable application or applet, you need to create a data source object. Any object that implements the `TableData` interface can be a data source. This can either by one of the stock data sources included with LiveTable (see the Using Stock Data Sources section in this chapter), or one of your own data sources (see Creating your own Data Sources section in this chapter).

The `TableData` interface is as follows:

```
public interface TableData {
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 `TableData` 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 Chapter 5, Displaying and Editing Cells. In short, you can have any type of object (usually one of Integer, Double, String, 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.

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()`.

4.2.1 Making the Data Source Editable

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

```
public interface EditableTableData extends TableData {
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 Chapter 5, Displaying and Editing Cells), and passes the new data to the data source using the `setTableDataItem()` method. If the data source doesn't accept the value of the object (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 to turn editing on and off for specific cells and ranges of cells. For `setEditable(true)` to have any effect, the data source *must be editable.*

4.3 Using Stock Data Sources

While it isn't hard to create a data source for a table, JClass LiveTable includes several stock data sources to save you the work of writing data sources for the most common data types:

Data Source	Description
VectorDataSource	General purpose data source: extended by almost all stock data sources.
EditableVectorDataSource	Allows users to edit cell values in tables with the above data source.
InputStreamDataSource	Base class for any data source that relies on streamed input.
AppletDataSource	Reads in data from the DATA tag of an applet.
URLDataSource	Uses URLs to create a data source object.
FileDataSource	Creates an input data stream from a file.
EditableFileDataSource	Allows users to edit cell values in tables with the above data source.
ResultSetDataSource	Simple read–only JDBC database source.

Data Source	Description
JCTableDataSource	For use with JCTable (backwards compatibility with LiveTable 2.0).
CachedDataSource	Caches previously read data from the data source.
EditableCachedDataSource	Allows users to edit cell values in tables with the above data source.
TableSwingDataSource	Enables users to display and edit Swing `TableModel` data objects in JClass LiveTable. Swing TableModel objects are typically used by the Swing JTable component.

Most of the stock data sources extend the `VectorDataSource` class. Please see Chapter 3, JClass LiveTable Inheritance Hierarchy, for a complete hierachy diagram that outlines the relationship between the stock data source classes.

4.3.1 VectorDataSource: the Data Source Workhorse

A `VectorDataSource` simply stores all of its data in memory using Vectors. The `VectorDataSource` 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 `VectorDataSource` implements `TableData`, it can't be edited by the `Table` object. If you want users to be able to edit the cell values through the table, you should use `EditableVectorDataSource`. The `EditableVectorDataSource` class is just a subclass of `VectorDataSource` that implements the `EdtitableTableData` interface.

4.3.2 Getting Data from an Input Stream

JClass LiveTable provides the `InputStreamDataSource` class to read data in through a standard `java.io.InputStream`.. Since `InputStreamDataSource` is derived from `VectorDataSource`, it has all of the same capabilities as a `VectorDataSource` (see Setting Stock Data Source Properties in this chapter). 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 `InputStreamDataSource` class has the following subclasses that provide convenient constructors to create an `InputStream` from various sources:

- `FileDataSource`, for reading data from a file
- `URLDataSource`, for reading data from a URL

- `AppletDataSource`, for reading data from the `DATA <PARAM>` tag associated with the specified applet.

4.3.3 Getting Data from a Database

The `ResultSetDataSource` uses a JDBC database connection and an SQL query to create a data source. This feature is available only in the JDK 1.1. The `ResultSetDataSource` 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 `ResultSetDataSource` is not an updatable data source; that is, it will not write to the database.

4.3.4 Using a Data Source with JCTable

`JCTable` is a subclass of `Table` that provides backwards compatibility with LiveTable 2.x. If you are using this class for your application, `JCTableDataSource` is automatically created for the table.

4.3.5 Caching Data with CachedDataSource

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

This method saves time by creating a second instance of previously–retrieved data, outside of the actual data source. `CachedDataSource` should only be used when the `TableData`'s `getTableItem`, `getTableRowLabel`, `getTableColumnLabel` are calulation–intensive or expensive to retrieve.

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

> **Note:** A non–editable data source bound to `EditableCachedDataSource` will display an editor but reject all changes

4.3.6 Using Swing TableModel Data Objects

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

`TableSwingDataSource` 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 TableSwingDataSource, you need to pass the constructor a valid Swing TableModel object.

4.4 Setting Stock Data Source Properties

The following properties are set using methods of the VectorDataSource class. Since the stock data sources are derived from the VectorDataSource 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 VectorDataSource class contains properties that are not inherent to the TableData 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.

4.4.1 Working with Rows and Columns

Setting the Number of Rows/Columns

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

Specifying Row and Column Labels

Set row and column labels by calling:

- setRowLabel() and setColumnLabel() for individual labels, and
- setRowLabels() and setColumnLabels() methods 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, JCString, Image, Component, or other object. To clear column or row labels, call the method with a null argument.

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

To retrieve the values, use:

- getTableRowLabel() and getTableColumnLabel() for individual labels, and
- 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

row and column labels will also be shifted unless you have registered a `JCLabelValueListener` in your program.

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 above 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 `JCTblEnum.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 comprise 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_rows)
```

In the above methods,

- The *position* parameter specifies the first row or column number to delete from the data source.
- The *num_rows* or *num_columns* parameters specify 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 above 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 is invalid.

4.4.2 Working with Other Properties

Setting Cell Values

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

Adding and Removing TableDataListeners

The `VectorDataSource` 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 Dynamically Updating Data on page 86.

4.5 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 `TableData` interface, as in the following example from *examples\chapter4\MyDataSource.java*:

```
import jclass.table3.TableData;
import jclass.table3.TableDataListener;
public class MyDataSource implements TableData {
String data[];
public MyDataSource(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(TableDataListener l) {
}
public void removeTableDataListener(TableDataListener l) {
}
}
```

The `MyDataSource` class takes a one-dimensional array of Strings and turns it into a read-only data source. The constructors takes the array of strings, the `getTableDataItem()` method supply 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, thus does not need to keep track of its listeners. You can attach this data source to a table quite easily, as in *examples\chapter4\MyTable.java*:

```
import java.awt.*;
import jclass.table3.*;
public class MyTable extends Frame {
Table table;
TableData dataSource;
String names[] = {"James",
                  "Daniel",
                  "Don",
                  "Brian",
                  "Geoff",
                  "Worf",
                  "Ethan",
                 };
public MyTable() {
  setLayout(new GridLayout(1,1));
  table = new Table();
  dataSource = new MyDataSource(names);
  table.setDataSource(dataSource);
  table.setCharWidth(JCTblEnum.LABEL,3);
  add(table);
  pack();
  show();
}
public static void main(String args[]) {
  new MyTable();
}
}
```

To make the items in the table editable, you must implement the `EditableTableData` interface, as in *examples\chapter4\MyEditableDataSource.java*:

```
import jclass.table3.EditableTableData;
import jclass.table3.TableDataListener;
public class MyEditableDataSource implements EditableTableData {
String data[];
public MyEditableDataSource(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(TableDataListener l) {
}
public void removeTableDataListener(TableDataListener l) {
}
}
```

The `MyEditableDataSource` class could have been a subclass of `MyDataSource`, 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`.

The `MyEditableDataSource` class is used in the the program *examples\chapter4\MyTable2.java*:

```
import java.awt.*;
import jclass.table3.*;
public class MyTable2 extends Frame {
Table table;
TableData dataSource;
String names[] = {"James",
                  "Daniel",
                  "Don",
                  "Brian",
                  "Geoff",
                  "Worf",
                  "Ethan",
                 };
public MyTable2() {
  setLayout(new GridLayout(1,1));
  table = new Table();
  dataSource = new MyEditableDataSource(names);
  table.setDataSource(dataSource);
  table.setCharWidth(JCTblEnum.LABEL,3);
  add(table);
  pack();
  show();
}
public static void main(String args[]) {
  new MyTable2();
}
}
```

4.6 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 `TableDataEvent` to all of the `TableDataListener` 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\chapter4\DynamicDataSource.java*:

```
import java.util.Enumeration;
import java.util.Random;
import jclass.table3.TableData;
import jclass.table3.TableDataEvent;
import jclass.table3.TableDataListener;
import jclass.table3.JCListenerList;
public class DynamicDataSource implements TableData, Runnable {
int data[];
JCListenerList listeners;
Thread kicker;
public DynamicDataSource() {
  data = new int[10];
  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 addTableDataListener(TableDataListener l) {
  listeners = JCListenerList.add(listeners,l);
}
public void removeTableDataListener(TableDataListener l) {
  listeners = JCListenerList.remove(listeners,l);
}
public void run() {
  Random random = new Random();
  Enumeration e;
  TableDataListener l;
  TableDataEvent event;
```

```
    int i;
    for(;;) {
      i = random.nextInt() % data.length;
      if(i < 0)
        i = -i;
      data[i] += (int)(random.nextGaussian()*10);
      event = new
       TableDataEvent(this,i,0,0,0,TableDataEvent.CHANGE_VALUE);
      for(e = JCListenerList.elements(listeners);e.hasMoreElements();)
{
        l = (TableDataListener)e.nextElement();
        l.dataChanged(event);
      }
      try {
        Thread.sleep(400);
      }
      catch(Exception ex) {
      }
    }
  }
  }
```

The `DynamicDataSource` class sends `CHANGE_VALUE` messages to all of its listeners whenever a value changes. When the `Table` 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 `TableDataEvent` class:

- `CHANGE_VALUE`
- `CHANGE_ROW`
- `CHANGE_COLUMN`
- `CHANGE_ROW_LABEL`
- `CHANGE_COLUMN_LABEL`
- `ADD_ROW`
- `REMOVE_ROW`
- `NUM_ROWS`
- `NUM_COLUMNS`
- `ADD_COLUMN`
- `REMOVE_COLUMN`
- `MOVE_ROW`
- `MOVE_COLUMN`
- `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's being shown on screen. For example, if you send a `CHANGE_ROW` message for row 55 and row 55 isn't currently on screen, the table won't do anything.

The file *examples\chapter4\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 `TableDataSupport`. `TableDataSupport` is an object provided by `Table` that has methods for adding and removing `TableDataListeners`. In addition, it provides a simple way to send data events using the `fireTableDataEvent()` method.

As an example, the *DynamicDataSource.java* program could be reimplemented to use the `TableDataSupport` object as follows (*examples\chapter4\DynamicDataSource2.java*):

```
import java.util.Enumeration;
import java.util.Random;
import jclass.table3.TableDataSupport;
import jclass.table3.TableDataEvent;
import jclass.table3.TableDataListener;
import jclass.util.JCListenerList;

public class DynamicDataSource2 extends TableDataSupport implements
  Runnable {

int data[];
Thread kicker;

public DynamicDataSource2() {
  data = new int[10];

  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;
  TableDataListener l;
  TableDataEvent event;
  int i;

  for(;;) {
    i = random.nextInt() % data.length;
    if(i < 0)
    i = -i;
    data[i] += (int)(random.nextGaussian()*10);

    event = new
```

```
            TableDataEvent(this,i,0,0,0,TableDataEvent.CHANGE_VALUE);
        fireTableDataEvent(event);

        try {
          Thread.sleep(400);
        }
        catch(Exception ex) {
        }
      }
    }
    }
```

Running *examples\chapter4\DynamicTest2.java* demonstrates that the same results can be achieved more easily by using `TableDataSupport`.

4.6.1 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 `DELETE_ROW` message to the `Table`. 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 `TableDataEvent`.

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

The `RESET` message causes the `Table` 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 like fonts, colors, etc 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.

5

Displaying and Editing Cells

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.

All of the example code is available in the *examples\chapter5* directory of the JClass LiveTable distribution[1].

5.1 Overview

In order to display a cell, JClass LiveTable has to know what type of data renderer the cell will contain 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. This package includes two interfaces that control displaying and editing cells: `CellRenderer` and `CellEditor`. The `jclass.cell` package is a generic package; renderers and editors written for JClass LiveTable will work with other JClass products. In addition, JClass BWT and JClass Field components can work as renderers and editors within LiveTable, allowing very lightweight operation.

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. 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 `java.awt TextArea` as the editor, rather than rendering and editing single line Strings.

1. Note that the example programs contain package names, and must be run using the full package name.

If you want to have ultimate control over the way cells are rendered and edited, you can create a custom `CellData` object for a particular data type, which acts as a container for the data, its renderer, and its editor.

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. Finally, you can explore the advanced procedure for creating your own `CellData` objects as containers.

5.2 Default Cell Rendering and Editing

Basic Editors and Renderers

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

1. The table first looks for a `CellData` object that will provide the data, and a renderer and editor for that data.
2. If the data is not a `CellData` object, the table checks to see if a renderer has been assigned to the cell or a series of cells by the `table.setCellRenderer()` method.
3. 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. When going through the above steps, LiveTable uses these default mappings if there is no `CellData` object or editor/renderer set.

Data Type	Renderer	Editor
String	StringCellRenderer	TextCellEditor
Boolean	StringCellRenderer	BooleanCellEditor
Date	StringCellRenderer	DateCellEditor
Double	StringCellRenderer	DoubleCellEditor
Float	StringCellRenderer	FloatCellEditor
Integer	StringCellRenderer	IntegerCellEditor
JCString	JCStringCellRenderer	JCStringCellEditor
Image	ImageCellRenderer	
Object	StringCellRenderer	

Advanced Editors and Renderers

The following table lists the advanced editor/renderers, which are the ones used by default. As with the the basic list above, these are included with JClass LiveTable. These advanced properties are "lighter" versions of those found in JClass Field, and

so are missing some functionality and customizability, such as the use of customizable masked edits and validation control (full featured cell editing and rendering can be used if you own JClass Field). These properties are part of the `jclass.field.cell` package.

Data Type	Editor/Renderer	Limitation in relation to JClass Field
Float	DoubleRendererEditor	only allows numbers and a decimal point
Double	DoubleRendererEditor	only allows numbers and a decimal point
Integer	IntegerRendererEditor	numeric characters only
Long	IntegerRendererEditor	numeric characters only
Byte	IntegerRendererEditor	numeric characters only
java.util.Date	DateRendererEditor	no freeform typing allowed
java.util.Calendar	DateRendererEditor	no freeform typing allowed
java.sql.Date	DateRendererEditor	no freeform typing allowed
java.sql.Timestamp	DateRendererEditor	no freeform typing allowed
java.sql.Time	DateRendererEditor	no freeform typing allowed

Although these basic and advanced editors and renderers are included with 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.

5.3 Rendering Cells

Cell rendering is simply the way in which data is drawn into a cell. JClass LiveTable uses a model where a method is passed all of the information it needs to render the data, including the data itself. It does not require a `java.awt.Component` to render the data, and it allows you to reuse a single instance of a renderer.

Rendering is done by objects that implement the `jclass.cell.CellRenderer` interface. The `CellRenderer` interface allows the implementor to:

- Draw the cell, and
- Determine the preferred size of the cell

For both these operations, the Table passes information to the renderer, including:

- The color, font, alignment, and other visual attributes of the cell (these are retrieved using the `jclass.cell.CellInfo` interface).
- The `Graphics` component; the Table clips the `Graphics` component to the cell location and size.

Other operations, such as initiating the renderer, providing and positioning a `Graphics` object for drawing the cell, and context for the rendering operation. The

`CellRenderer` and `CellInfo` interfaces are described more fully in Creating your own Cell Renderers, below.

5.3.1 JClass Cell Renderers

As shown in the table above, JClass LiveTable maps standard data types to specific renderers when the program does not specify a renderer for that data type (either by setting for a series or mapping). This means that most tables are easily rendered without doing 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 5.3.2, Setting a Cell Renderer for a Series) or as a mapping (see Section 5.3.3, Mapping a Data Type to a Cell Renderer). These cell renderers are described in the following table:

Name	Package	Data Type	Description
ButtonCellRenderer	jclass.cell.renderers	String	Defines a `CellRenderer` object that paints a table cell as a button.
CheckboxCellRenderer	jclass.cell.renderers	boolean	Defines a `CellRenderer` object that paints `boolean` objects in a table cell as checks.
ChoiceCellRenderer	jclass.cell.renderers	boolean	Renders an integer by mapping it to a String in a String array.
EllipsisCellRenderer	jclass.cell.renderers	String	Uses an ellipsis to draws a single line text if the text is larger than the width of the cell.
ImageCellRenderer	jclass.cell.renderers	image	Defines a `CellRenderer` object that paints `Image` objects in a table cell.
JCStringCellRenderer	jclass.table3	JCString objects	Renders an integer by mapping it to a String in a String array.
RawImageCellRenderer	jclass.cell.renderers	image	Defines a `CellRenderer` object that paints unconverted Image objects in a table cell (extends scaled)
ScaledImageCellRenderer	jclass.cell.renderers	image	Defines a `CellRenderer` object that paints scaled `Image` objects in a table cell.
StringCellRenderer	jclass.cell.renderers	String, boolean, double, float, integer, object.	A simple renderer that can draw strings.
WordWrapCellRenderer	jclass.cell.renderers	String	Defines word-wrapping logic for multi-line display of strings in cells.

The default mappings and these special renderer classes should provide for rendering most data types. Few programmers work under ideal conditions, however,

and you will probably need to extend the capability of these renderers. JClass LiveTable includes ways for you to customize cell rendering as described below.

5.3.2 Setting a Cell Renderer for a Series

Often, the rows and columns that comprise a table are grouped by the type of data they contain. You may be creating an order form that has a product name (a String) in one column, a part number (an Integer) in another, and a checkbox (a special type of object) in the final column to indicate that you want that product:

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 the previous section, Default Cell Rendering and Editing, for a list of default renderers). However, you can use your own renderer if the default does not suit your needs.

In the case of the Order Checkbox, the default renderer for its Boolean data type will be the `StringCellRenderer`. 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 LiveTable's default renderer. Inserting this line of code into your program will do this:

```
table.setCellRenderer(JCTblEnum.ALL, 3,
  new jclass.cell.renderers.CheckboxCellRenderer());
```

The `CheckBoxCellRenderer` 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.

5.3.3 Mapping a Data Type to a Cell Renderer

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

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

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

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

For example, in the following code fragment (from *TriangleTest.java* in the *examples\chapter5* directory of the JClass LiveTable 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
   ("jclass.table3.examples.chapter5.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.

5.3.4 Creating your own Cell Renderers

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

Subclassing the Default Renderers

A simple way to create your own `CellRenderer` objects is to subclass the `CellRenderer`s provided with JClass LiveTable. '`CalendarCellRenderer`, found in the *demos\customCells\CustomCells.*`java` file, is an example of subclassing from the StringCellRenderer in the `jclass.cell.renderers` package.

Writing your own Cell Renderer

To create a `CellRenderer` object of your own, you must implement the `jclass.cell.CellRenderer` interface.

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

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

1. A `draw()` method, which is passed a `CellInfo` object (see The CellInfo Interface, below 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 jclass.cell.CellRenderer;
import jclass.cell.CellInfo;

public class TriangleCellRenderer implements CellRenderer {

public void draw(Graphics gc, CellInfo cellInfo, Object o, boolean
      selected) {
  Polygon p = makePolygon(o);
  gc.fillPolygon(p);
}

public Dimension getPreferredSize(CellInfo 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 `CellRenderer`, the program contains a `draw()` method in the lines:

```
public void draw(Graphics gc, CellInfo cellInfo, Object o, boolean
    selected) {
  Polygon p = makePolygon(o);
  gc.fillPolygon(p);
}
```

The `draw()` method renders the object `o` by making it into a polygon and drawing the polygon using the `gc` provided. The 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` will retrieve other cell property information through the `CellInfo` interface (see Section 5.5, The CellInfo Interface).

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

```
public Dimension getPreferredSize(CellInfo 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.

The renderer also contains two methods (`makePolygon(Object o)`) and `makePolygon(int s)`) that create a polygon, the first a local method for either Number or Polygon instances, and the second a generic method for creating a polygon of size `s`.

You can also add clipping arrows to the Cell Renderer. When a table is displayed, these indicate that the cell is not large enough to display all of its contents. Use the `setClipArrows()` method to manually set the value of the `ClipArrows` property:

- `JCTblEnum.CLIP_ARROWS_RIGHT`
- `JCTblEnum.CLIP_ARROWS_DOWN`
- `JCTblEnum.CLIP_ARROWS_BOTH` (default)
- `JCTblEnum.CLIP_ARROWS_NONE`

The *examples\chapter5* directory and the *demos* 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.

5.4 Editing Cells

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

A typical cell edit works through the following process:

- When a user initiates a cell edit with either a mouse click or a key press, the `Table` object calls `CellEditor.initialize()` and passes a `CellInfo` object with information about the cell, and the object (data) that will be edited.
- The `CellEditor` 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 `CellEditor` 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.

5.4.1 Default Cell Editors

Cell editors are typically AWT or Swing components with extended functionality provided by the `jclass.cell.CellEditor` 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 `jclass.cell.editors` package:

Editor	Description
BaseCellEditor	Provides a base editing component for other editors.
BigDecimalCellEditor	An editor using a simple text field for `BigDecimal` objects.

Editor	Description
BooleanCellEditor	Provides a simple text editing component that allows the user to set the boolean value as either 'true', 'false', 't' or 'f'.
ButtonCellEditor	A featherweight editor that generates action events.
ByteCellEditor	An editor using a simple text field for `Byte` objects.
CheckboxCellEditor	A featherweight editor for `CheckboxCellData` that automatically changes the checked state.
ChoiceCellEditor	An editor using a simple `Choice` component for `ChoiceCellData`.
DateCellEditor	An editor using a simple text field for `Date` objects
DoubleCellEditor	An editor using a simple text field for `Double` objects.
FeatherweightCellEditor	A convenience class for featherweight editors. A featherweight editor is an editor that has no component, does no event processing, and typically completes the edit operation during `initialize()`. An example is `CheckboxCellEditor`
FloatCellEditor	An editor using a simple text field for `Float` objects.
ImageCellEditor	An editor using a simple text field for `Image` objects.
IntegerCellEditor	An editor using a simple text field for `Integer` objects.
LongCellEditor	An editor using a simple text field for `Long` objects.
MultilineCellEditor	A simple text editing component for multiline data.
ShortCellEditor	An editor using a simple text field for `Short` objects.
SQLDateCellEditor	An editor using a simple text field for SQL `Date` objects.
SQLTimeCellEditor	An editor using a simple text field for SQL `Time` objects.
SQLTimestampCellEditor	An editor using a simple text field for SQL `Timestamp` objects.
StringCellEdtitor	Provides a simple text editing component.
WordWrapCellEditor	Provides a simple text editing component that wraps text.

In addition, the `jclass.table3` package contains the following cell editors:

Editor	Description
TextCellEditor	An editor that extends the basic functionality of `StringCellEditor` by converting keys based on `table.getStringCase` and handles the `table.MaxLength` property.
JCStringCellEditor	Provides an editor using a simple text field for `JCString` objects (see Appendix D, JCString Properties for more information about `JCString` objects).

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.

5.4.2 Setting a Cell Editor for a Series

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

```
table.setCellEditor(int row,
                    int col,
                    CellEditor editor);
```

Therefore, the following method:

```
table.setCellEditor(JCTblEnum.ALL, 3,
  new jclass.cell.editors.StringCellEditor());
```

would use the same `CellEditor` (the default String editor in the `jclass.cell.editors` package) for all of the cells in the fourth column in the table.

5.4.3 Mapping a Data Type to a Cell Editor

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

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

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

Normally, you would use these mappings in a construction that would test for the presence of the editor you specify, and throw an exception if the class was not found as in the following sample from *TriangleTest.java* in the *examples\chapter5* directory of the JClass LiveTable distribution:

```
try {
  table.setCellEditor(Class.forName("java.awt.Polygon"),
  Class.forName
   ("jclass.table3.examples.chapter5.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.

5.4.4 Creating Your Own Cell Editors

To create a `CellEditor` object, you must implement the `jclass.cell.CellEditor` interface. The following code comprises the `CellEditor` interface:

```
import java.awt.Component;
import java.awt.Dimension;

public interface CellEditor extends CellEditorEventSource {
public void initialize(InitialEvent ev, CellInfo info, Object o);
public Component getComponent();
public Object getCellEditorValue();
public boolean stopCellEditing();
public boolean isModified();
public void cancelCellEditing();
public Dimension getPreferredSize(CellInfo cellInfo, Object o);
public KeyModifier[] getReservedKeys();
}
```

Look at each of the methods in `CellEditor`:

Method	Description
`public void initialize(InitialEvent ev, CellInfo info, Object o);`	The table calls `initialize()` before the edit starts to let the editor know what kind of event started the edit, using the `jclass.cell.InitialEvent` object. The size of the cell comes from the `CellInfo` interface (detailed below). The `initialize()` method also provides the data object.
`public Component getComponent();`	Returns the AWT component that does the editing.
`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 Table that editing is not complete.
`public boolean isModified();`	Table 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.

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

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

Method	Description
`public abstract void removeCellEditorListener(CellEditorListener l);`	Removes the above listener.
`public KeyModifier[] 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.

In addition to implementing the methods of `CellEditor`, an editor is responsible for monitoring events and sending `editingStopped` and `editingCanceled` events to the table. This functionality is further explained in Section 5.4.4, Creating Your Own Cell Editors.

Subclassing the Default Editors

One easy way to create your own editor is to subclass one of the editors provided in the `jclass.cell.editors package`. The following code is from *examples\chapter5\MoneyCellEditor.java*. It creates a simple editor that extends the `jclass.cell.editors.StringCellEditor` class. The `MoneyCellEditor` class formats the data as money (two digits to the right of the decimal point) instead of a raw string; but `StringCellEditor` 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.TextField;
import java.awt.Dimension;
import jclass.cell.editors.StringCellEditor;
import jclass.cell.CellInfo;
import jclass.cell.InitialEvent;
import java.awt.Component;
import java.awt.Event;

public class MoneyCellEditor extends StringCellEditor {

Money initial = null;

public void initialize(InitialEvent ev, CellInfo 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 `CellEditor`s provided with the `jclass.cell.editors` package can save you a lot of work coding entire editors on your own.

Writing your own Editors

Of course, you may be a bit of a maverick, and not want to subclass any of the editors provided with the `jclass.cell.editors` package. The following is from an editor that was written without subclassing an existing editor. By implementing the `CellEditor` interface, we have written an editor that will edit triangles. The code is in *examples\chapter5\TriangleCellEditor.java*. You can see it work by running `jclass.table3.examples.chapter5.TriangleTest`.

Note that the example code in this manual is for JDK 1.1.x versions of the program. The example code in the *examples* directory will be specific to the version of the JDK that your distribution supports.

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

  new_poly = null;

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

The editor also needs to retrieve the AWT component that will be associated with it. In this case the editor is an `awt.Canvas` object.

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

The next `CellEditor` methods called are `isModified()`, which 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 `CellEditor` 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, and its own preferred size by looking at the object it's editing.

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

public Dimension getPreferredSize(CellInfo cellInfo, Object o) {
  if (o != null && o instanceof Polygon) {
    Polygon p = (Polygon) o;
    Rectangle r = p.getBoundingBox();
    return new Dimension(r.x+r.width,r.y+r.height);
  }
  return minimumSize();

}
```

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

```
....
public void paint(Graphics gc) {
  int x, y;

  Polygon local_poly = getDrawPoly();
  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);
}


int dragging_point = -1;
boolean dragging_whole = false;
int last_x, last_y;

int dist(int x1, int y1, int x2, int y2) {
  return (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1);
}
```

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.

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

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

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

The `TriangleCellEditor` is an example of a fairly complex implementation of the `CellEditor` 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 `CellEditor` classes that handle any type of data you care to display and edit.

Handling Editor Events

The `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 `CellEditor` events is to use the `CellEditorSupport` convenience class. `CellEditorSupport` makes it easy for cell editors to implement standard editor event handling by registering event listeners and providing easy methods for sending events.

`CellEditorSupport` methods include:

Method	Description
`addCellEditorListener()`	Adds a new `CellEditorListener` to the listener list
`removeCellEditorListener()`	Removes a `CellEditorListener` 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 CellEditorEvent(e));
}
```

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

Method	Description
`InitialEvent`	Tells the `CellEditor` what event started the edit (usually a mouse click or key press). This is useful if the event affects the edit. For example, `StringCellEditor` will pass the key into the `TextField`.
`CellEditorEvent`	Sent when the `CellEditor` finishes an operation. The `CellEditorEvent` contains the event that originated the operation in the editor.
`CellEditorListener`	The container registers a `CellEditorListener` to let the `CellEditor` inform it when editing has stopped or been canceled.
`CellEditorEventSource`	This class defines the add and remove methods for an object that posts `CellEditorEvents`.

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 `KeyModifier` class to reserve a key/modifier combination.

```
import java.awt.Event;
public class KeyModifier {
public int key;
public int modifier;
public static final int ALL = 1 << 4;
public KeyModifier(int key, int modifier) {
  this.key = key;
  this.modifier = modifier;
}

public KeyModifier(int key) {
  this.key = key;
  this.modifier = ALL;
}

public boolean match(int key, int modifier) {
  // Keys don't match
  if (this.key != key) return false;

  // Keys don't match, all modifiers accepted
  if (this.modifier == ALL) return true;

  // Modifiers match exactly
  if (this.modifier == modifier) return true;
```

```
        // Modifiers don't match exactly, and ALL not specified.
        return false;
    }
    }
```

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 `KeyModifier` 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 KeyModifier(KeyEvent.VK_TAB, KeyModifier.ALL);

new KeyModifier(KeyEvent.VK_TAB);
```

When you call `getReservedKeys()` in your `CellEditor`, the editor will return the values set in the `KeyModifier` object. Note that the container can still choose to ignore reserved keys.

5.5 The CellInfo Interface

You can see that `CellRenderer` and `CellEditor` use the `CellInfo` interface to get information about the cell. The `CellInfo` 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 `CellInfo` 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 `jclass.table3` package also contains an object called `TableCellInfo`, which extends `CellInfo` to include more detailed information from the Table. `TableCellInfo` is useful for retrieving Table-specific information for use in the editor or renderer. For example, `TextCellRenderer` uses `TableCellInfo` to access the `MaxLength` and `StringCase` series.

Note that editors and renderers that rely on `jclass.table3.TableCellInfo` can only be used with JClass LiveTable.

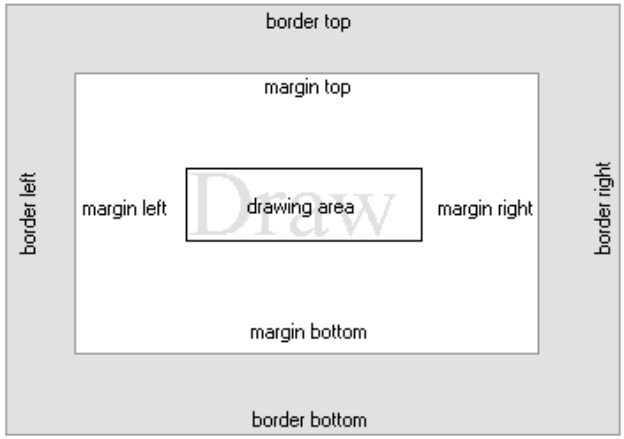


Figure 32 The relationship of border sides, margins, and drawing area provided by CellInfo

The following code comprises the `jclass.cell.CellInfo` interface:

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Insets;

public interface CellInfo {
public Color getBackground();
public Color getForeground();
public Color getSelectedBackground();
public Color getSelectedForeground();
public Font getFont();
public FontMetrics getFontMetrics();
public int getHorizontalAlignment();
public int getVerticalAlignment();
public Insets getMarginInsets();
public Insets getBorderInsets();
public int getBorderStyle();
public Rectangle getDrawingArea();
public boolean isEditable();
public boolean isEnabled();
public static final int LEFT = 0;
public static final int CENTER = 1;
public static final int RIGHT= 2;
public static final int TOP= 0;
public static final int BOTTOM = 2;
public boolean getSelectAll();
public int getClipHints();
public class getDataType();
}
```

6

Programming User Interactivity

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 Chapter 5, Displaying and Editing Cells.

6.1 Cell Traversal

Traversal is the act of moving the current cell from one location to another. The `Traversable` property performs interactive cell traversal. A traversal passes through three stages: validating the edited current cell, determining the new current cell location, and entering that cell.

6.1.1 Default Cell Traversal

Users can traverse cells by clicking the primary mouse button when the mouse pointer is over a cell. This changes the focus to that cell (a focus rectangle appears around the inside of the cell borders). Users can traverse cells from the keyboard by using the cursor keys (up, down, left, and right) and the **Tab** key to traverse right and **Shift+Tab** key to traverse left.

6.1.2 Focus Rectangle Color

You can change the color of the focus rectangle using the `setFocusRectColor()` method:

```
setFocusRectColor(Color.blue);
```

6.1.3 Customizing Cell Traversal

By default, all cells are traversable. To prevent users from traversing to a cell, set `Traversable` to `false`. Making a cell non-traversable also prevents it from being traversed to programmatically.

Disabling traversal also disables cell editability regardless of whether the cell's data source is editable.

The following code fragment sets all cells in row 3 to be non-traversable:

```
table.setTraversable(3, JCTblEnum.ALLCELLS, false);
table.addTraverseCellListener(this);
```

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);
table.setTraversable(range, false);
```

Use the `setTraverseCycle()` method 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. The default is `true`.

6.1.4 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 traversed to. 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 6.4.5, Managing Table Scrolling.

6.1.5 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`.

6.1.6 Controlling Interactive Traversal

You can use `TRAVERSE_CELL`, the ID variable of `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 the `Text` component to the next cell. Each event listener is passed an object of type `JCTraverseCellEvent`.

`JCTraverseCellEvent` uses the `getParam()` method to retrieve information on the direction of the traversal. `getParam()` retrieves one of the following strings indicating the direction of traversal:

- `POINTER` - Traverse to the cell user clicked on
- `LEFT` - Traverse left to first traversable cell
- `RIGHT` - Traverse right to first traversable cell
- `UP` - Traverse up to first traversable cell
- `DOWN` - Traverse down to first traversable cell
- `null` - Initiated programmatically by call to `table.traverse`.

In addition to `Param`, there are several other properties available to `JCResizeCellEvent`. The `getColumn()` and `getRow()` methods get the column and row of the current cell respectively. Finally, the `NextColumn` and `NextRow` properties respectively set or retrieve the column and row of the cell to traverse to.

The `TRAVERSE_CELL` action attempts to traverse to the cell specified by these members. Note that if `NextColumn` and `NextRow` reference a non-traversable cell, the traversal attempt will be unsuccessful. The following example code prevents the user from traversing outside of column 0:

```
public void traverseCell(JCTraverseCellEvent ev) {
  if (ev.getNextColumn() > 0) {
  if (ev.getRow() >= table.getNumRows())
    ev.setNextRow(0);
  else
    ev.setNextRow(ev.getRow() + 1);
    ev.setNextColumn(0);
}
}
```

6.2 Cell Selection

6.2.1 Default Cell Selection

Cell selection is not enabled by default. When cell selection is enabled (see Customizing Cell Selection), the default selection behavior is as follows:

- Clicking on a cell, holding the mouse button down, and dragging selects those cells.
- Clicking over 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 in 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:

- `JCTblEnum.SELECT_MULTIRANGE`: multirange selection (selecting multiple ranges of cells)
- `JCTblEnum.SELECT_RANGE`: single range
- `JCTblEnum.SELECT_SINGLE`: single cell
- `JCTblEnum.SELECT_NONE`: no selection.

6.2.2 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.

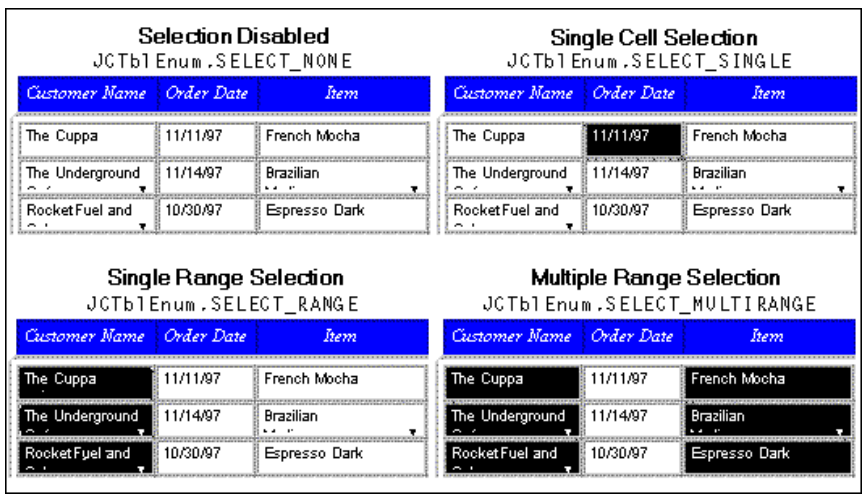


Figure 33 Selection policies

When `SelectionPolicy` is set to `JCTblEnum.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.

Setting the Selection Mode

In some cases, you may want to further control selection by specifying that selection occurs only for rows or columns. To do this, set the `SelectionMode` property. The `setSelectionMode()` method can take one of the following values:

- `JCTblEnum.SELECT_BY_CELL` (default)
- `JCTblEnum.SELECT_BY_ROW`
- `JCTblEnum.SELECT_BY_COLUMN`

Selecting Row/Column Labels

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

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

6.2.3 Selected Cell List

The `SelectedCells` property specifies the list of all currently selected ranges in the table, where each element of the vector 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` variable is the column of the first cell in range (top-left corner), while `start_row` is the first cell in range (top-left corner). The `end_column` variable is the column of the last cell in a range (top-left corner), and `end_row` is the row of the last cell in 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 any of the four corners of a range.

The following example sets two selected ranges:

```
table.setSelectionPolicy(JCTblEnum.SELECT_MULTIRANGE);
JCCellRange ranges[] = new JCCellRange[2];
ranges[0] = new JCCellRange(0, 0, 4, 2);
ranges[1] = new JCCellRange(7, 1, 7, 4);
Vector v = new Vector(ranges);
table.setSelectedCells(v);
```

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

6.2.4 Selection Colors

The background/foreground colors used for selected cells are specified by `SelectedBackground` and `SelectedForeground`. 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.

To reset the `SelectedForeground` and `SelectedBackground` properties to their default, reverse video mode, set them to a `null` value.

6.2.5 Working with Selected Ranges

To get a selected range, allocate a `JCCellRange` structure and call `getSelectedRange()`. This method has the following prototype:

```
public boolean getSelectedRange(int pos, JCCellRange range)
```

`SelectedRange` retrieves a currently-selected range from the `SelectedCellList`, and it can take the following parameters:

- `pos` - the position within the selected cell list
- `range` - the returned range

`range` is rationalized to read from top to bottom and from left to right, and special values such as `MAXINT` are converted to valid range values. `SelectedRange` returns `false` if no cells are selected, or if any argument is invalid.

The following example gets each selected range (assuming that selection policy is `JCTblEnum.SELECT_MULTIRANGE`):

```
int i = 0;
JCCellRange r = new JCCellRange();
while (table.getSelectedRange(i++, r)) {
  System.out.println("Range " + i + " is
  (" + r.start_row + "," + r.start_column + ") to
  (" + r.end_row + "," + r.end_column + ")");
}
```

To determine whether a particular cell is selected, and retrieve the range if it is, call `getSelectedCells()`. This method has the following prototype:

```
public Vector getSelectedCells()
```

Each element of the Vector 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. Users can select in any direction, so `start_row` and/or `start_column` may be greater than `end_row` and/or `end_column`. When a user clicks a row/column label to select an entire row or column, `end_row` or `end_column` is set to `MAXINT`.

6.2.6 Forcing Selection

An application can add a selection to the selected cell list by adding the new range to the SelectedCells Vector, as shown by the following code fragment:

```
JCCellRange nr = new JCCellRange(8, 1, 8, 4);
Vector ve = table.getSelectedCells();
ve.addElement(nr);
table.setSelectedCells(ve);
```

6.2.7 Removing Selections

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

6.2.8 Selection in List Mode

To select an entire row or column, set the Row or Column properties of JCSelectEvent to JCTblEnum.ALL. Note that you cannot set a RowLabel or ColumnLabel in this manner. JClass LiveTable allows you to specify that a table displays the entire row as selected when a user clicks on a cell. To do this, set the setMode() property to JCTblEnum.MODE_LIST (see Preset Table Styles, in Chapter 3). When a table is in list display mode, an entire row is selected when a cell is selected. The SelectionPolicy value controls whether users are allowed to select single rows, a range of rows, or multiple ranges of rows.

The default setting for the Mode property is JCTblEnum.MODE_TABLE.

6.2.9 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. One of these, getParam(), is used to retrieve information on the initiating action. The getParam() method retrieves one of the following strings to determine how the cell was selected (if no selection is possible on a label):

- START - select the cell if SelectionPolicy is not SELECT_NONE
- CURRENT - select the current cell if SelectionPolicy is not SELECT_NONE
- EXTEND - extend the selected region to include cell if SelectionPolicy is SELECT_RANGE or SELECT_MULTIRANGE
- ADD - select the cell if SelectionPolicy is to SELECT_MULTIRANGE
- CANCEL - cancels the current selection
- END - finish a selection.

The getAllowSelection() method determines whether the selection (or unselection) should be allowed (default: true). The Row and Column properties set or retrieve the respective value of the row or column being selected or unselected. The getEvent() method gets the event that initiated the action. The getStage() method retrieves the

selection stage (either INITIAL or EXTEND), while getStateChange() returns the state change type generated by the event (either SELECTED or DESELECTED).

JCSelectListener is called before selection begins (selectBegin(JCSelectEvent)), and after a selection has finished (selectEnd(JCSelectEvent)).

The following event listener routine constrains selection to the column where it started:

```
public void selectBegin(JCSelectEvent ev) {
  if (ev.getStage() == JCSelectEvent.INITIAL) {
     save_column = ev.getColumn();
  }
     else if (ev.getStage() == JCSelectEvent.EXTEND) {
  if (ev.getColumn() != save_column)
       ev.setAllowSelection(false);
  }
}
public void selectEnd(JCSelectEvent ev) {}
```

6.3 Resizing Rows and Columns

6.3.1 Default Resizing Behavior

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.

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 cannot resize rows or columns to smaller than 5 pixels.

6.3.2 Disallowing Cell Resizing

Use the setAllowCellResize() method to control interactive row/column resizing over the entire table. The valid parameters of the AllowCellResize property are:

- JCTblEnum.RESIZE_ALL: user resizing of cell permitted (default)
- JCTblEnum.RESIZE_NONE: no row/column resizing is allowed
- JCTblEnum.RESIZE_COLUMN: only columns may be resized
- JCTblEnum.RESIZE_ROW: only rows may be resized

6.3.3 Controlling Resizing

You can use a JCResizeCellListener (registered with addResizeCellListener(JCResizeCellListener)) to control interactive row/column resizing on a case-by-case basis. JCResizeCellEvent is the event posted as a user resizes a row and/or column.

JCResizeCellEvent uses the method getParam(), which retrieves the interactive resize stage. The getParam() method retrieves one of the following string values:

- START - begins interactive resize
- MOVE - select new height/width
- END - finish resize selection
- CANCEL - cancels the current resize.

In addition to getParam(), there are other methods/properties available to JCResizeCellEvent. The AllowResize property determines whether an interactive resize is allowed (default: true).

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, resizeCellBegin(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 resizeCellEnd(JCResizeCellEvent) is sent to setNewColumnWidth() and setNewRowHeight().

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 resizeCellBegin(JCResizeCellEvent ev) {}
public void resizeCellEnd(JCResizeCellEvent ev) {
  ev.setNewColumnWidth(ev.getNewColumnWidth() / 10 * 10);
}
```

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

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 `setResizeByLabelsOnly()` method requires a boolean value (default is `false`). If set to `true`, users can resize rows and columns only by dragging on the borders between row and column labels. The mouse pointer will not change to a resize arrow over cell borders in the body of the table.

6.4 Table Scrolling

6.4.1 Default Scrolling Behavior

By default, users can scroll through the table using scrollbars and the mouse, or by using arrow/page keys.

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 this control in detail.

6.4.2 Specifying your own Scrollbars

Using a Different 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.

Any scrollbar component you specify must implement `JCAdjustable`, as in the following:

```
class MyScrollbar extends java.awt.Scrollbar implements JCAdjustable
{
public MyScrollbar() {
  super();
}
public MyScrollbar(int orientation) {
  super(orientation);
}
```

```
public MyScrollbar(int orientation, int value, int visible, int
minimum, int maximum) {
  super(orientation, value, visible, minimum, maximum);
}
}
```

Scrollbar Attributes

You can set or retrieve any of the properties of either scrollbar. The following code gets the vertical scrollbar and sets its background color to red:

```
table.getVertSB().getComponent().setBackground(Color.red);
```

6.4.3 Attaching 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. List-style tables attach the scrollbars to the table component itself and do not move when the size of the visible area changes–only changes in the component size cause the scrollbars to move and change size (see Preset Table Styles, in Chapter 3 for more information about table styles).

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 `JCTblEnum.SBPOSITION_CELLS` (default), the scrollbar is attached to the cell/label viewport (that is, the cells that are visible).
- When set to `JCTblEnum.SBPOSITION_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 `JCTblEnum.ATTACH_CELLS` (default), the scrollbar ends at the edge of the visible cells. When set to `JCTblEnum.ATTACH_SIDE`, 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.

To specify List-style table scrollbars:

- set the horizontal and vertical position properties to `JCTblEnum.SBPOSITION_SIDE`
- set the left, right, top, and bottom attachment properties to `JCTblEnum.ATTACH_SIDE`
- set `HorizSBDisplay` and `VertSBDisplay` to `JCTblEnum.SBDISPLAY_ALWAYS`
- set `HorizSBOffset` and `VertSBOffset` to zero.

`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.

6.4.4 Setting Scrollbar Display Options

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 `JCTblEnum.SBDISPLAY_ALWAYS`. Set them to `JCTblEnum.SBDISPLAY_NEVER` to completely disable the scrollbar display[1].

6.4.5 Managing Table Scrolling

Jump Scrolling

By using the `JumpScroll` property, you can control the scrolling behavior of each scrollbar. Scrollbars can either scroll smoothly or "jump" scroll in whole row/column increments. To enable jump scrolling, call the `setJumpScroll()` method with one of the following parameters:

- `JCTblEnum.JUMP_VERTICAL` (default), only the vertical scrollbar jump scrolls
- `JCTblEnum.JUMP_NONE`, both scrollbars will scroll smoothly
- `JCTblEnum.JUMP_ALL`, both scrollbars jump scroll
- `JCTblEnum.JUMP_HORIZONTAL`, only the horizontal scrollbar jump scrolls

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:

- `JCTblEnum.AUTO_SCROLL_NONE` (default)
- `JCTblEnum.AUTO_SCROLL_ROW`
- `JCTblEnum.AUTO_SCROLL_COLUMN`
- `JCTblEnum.AUTO_SCROLL_BOTH`

Note that automatic scrolling is disabled when no scrollbars are visible, and also 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 set `HorizSBDisplay` and/or `VertSBDisplay` to `JCTblEnum.SBDISPLAY_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.

---

1. Although scrollbars are removed, a user can still scroll with the keyboard. See "Disabling Interactive Scrolling" for complete information on disabling interactive scrolling.

Forcing Scrolling

An application can force the table to scroll in the following ways:

- 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)`
- To scroll a particular column to the left side of the display, set the `LeftColumn` property to the column number you want to display; for example to display the thirteenth column at the left of the table:
  `setLeftColumn(12)`
- To determine whether a row or column is visible, call the `Table.isRowVisible()` or `Table.isColumnVisible()` methods. To check if a particular cell is visible, use `Table.isVisible()`.
- 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.

6.4.6 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 `scrollBegin(JCScrollEvent)` and `scrollEnd(JCScrollEvent)` scrollbar interface methods to store an internal state:

```
public MyClass extends Frame implements JCScrollListener {
....
  public void scrollBegin(JCScrollEvent ev) {
  if (ev.getDirection() == TableScrollbar.HORIZONTAL)
  hScrollingActive = true;
  else if (ev.getDirection() == TableScrollbar.VERTICAL)
  vScrollingActive = true;
  }
public void scrollEnd(JCScrollEvent ev) {
  if (ev.getDirection() == TableScrollbar.HORIZONTAL)
  hScrollingActive = false;
  else if (ev.getDirection() == TableScrollbar.VERTICAL)
  vScrollingActive = false;
}
```

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

```
table.addScrollListener(this);
```

6.5 Dragging Rows and Columns

You can configure your JClass LiveTable program to allow users to drag rows and columns to a new position in the table. This feature is implemented using the RowTrigger and ColumnTrigger properties to specify a key-mouse-click combination for dragging a row or column by its label. For example, you can specify that when a user holds the **Shift** key and clicks on a row label, the user can drag that row to another location in the table. When dragging is enabled, the mouse pointer turns into a hand to indicate that the row or column can be dragged.

To enable users to drag rows and columns by holding down the Shift key and clicking on the row or column label:

```
table.setColumnTrigger(Event.SHIFT_MASK,LabelTrigger.DRAG);
table.setRowTrigger(Event.SHIFT_MASK,LabelTrigger.DRAG);
```

Other key-click combinations are available:

- `CTRL_MASK`
- `ALT_MASK`
- `META_MASK`

Under the JDK 1.1, you can also specify which mouse button to use in the trigger method, using the following parameters:

- `InputEvent.BUTTON1_MASK`
- `InputEvent.BUTTON2_MASK`
- `InputEvent.BUTTON3_MASK`

The following would enable row dragging on an **Alt**-right (secondary) mouse button click combination under JDK 1.1:

```
table.setRowTrigger(InputEvent.BUTTON3_MASK,
  LabelTrigger.DRAG);
```

Dragging a row or column affects only the data view. It does not change the data source.

6.6 Sorting Columns

You can easily program your JClass LiveTable applications and applets to allow users to sort columns in the table. Sorting columns rearranges the rows in the table display, but *does not* affect the data source of the table (this is true for the JClass LiveTable transitional layer also). By default, sort behavior does not sort frozen rows set with the `setFrozenRows()` method (see Specifying 'Frozen' Rows and Columns, in Chapter 3).

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(2, Sort.DESCENDING, 2, 18);
```

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);
```

6.6.1 Sort by Clicking on a Column Label

With JClass LiveTable you can easily configure your table to sort columns based on a key-mouse-click combination on the column's label. For example, you can specify that when a user holds the **Ctrl** key and clicks on the column label, that column gets sorted in ascending order. This is done using the `setColumnTrigger()` method.

```
table.setColumnTrigger(Event.CTRL_MASK,LabelTrigger.SORT);
```

Other key-click combinations are available:

- `SHIFT_MASK`
- `ALT_MASK`
- `META_MASK`

Under the JDK 1.1, you can also specify which mouse button to use in the trigger method, using the following parameters:

- `InputEvent.BUTTON1_MASK`
- `InputEvent.BUTTON2_MASK`
- `InputEvent.BUTTON3_MASK`

The following enables column sorting on an **Alt**-right (secondary) mouse button click combination under JDK 1.1:

```
table.setColumnTrigger(InputEvent.BUTTON3_MASK, LabelTrigger.SORT);
```

Note: If the selection policy (see Customizing Cell Selection) is set to `JCTblEnum.SELECT_SINGLE` or `JCTblEnum.SELECT_MULTIRANGE`, selected cells will remain selected once the column is sorted.

6.6.2 Resetting the Table after Sorting

To clear all of the changes to the display resulting from column sorting, call the `resetSortedRows()` method, which resets the display to match the data source.

6.7 Custom Mouse Pointers

When tracking the mouse pointer, JClass LiveTable considers the current settings of the `Traversable` and `AllowCellResize` properties.

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`.

Disabling JCString URL Tracking

JClass LiveTable detects JCString URLs when both the `TrackCursor` and `TrackJCStringURL` properties are set to `true`. To disable this behavior, set `TrackJCStringURL` to `false`. This can improve performance when the table is using a data source that takes a long time to access, such as a database.

7

Events and Listeners

Displaying Cells ■ Creating Components ■ Displaying Components
Entering Cells ■ Painting ■ Printing ■ Resizing
Scrolling ■ Sorting ■ Traversing

The following sections explain how to generate and receive events in your JClass LiveTable programs.

The descriptions are listed in sets of events and event listeners, with examples of when you would use the event and listener, and sample code.

In order to register an event listener in your program, it must implement the listener's interface.

7.1 Displaying Cells

JCCellDisplayEvent

This event will be posted when your program displays cell values in the table. When you create a `JCCellDisplayEvent` object, you can call the following methods:

- `getCellData()` retrieves the `CellData` object displayed by the table.
- `getRow()` retrieves the row number of the cell or label displayed.
- `getColumn()` retrieves the column number of the cell or label displayed.
- `setDisplayString()` specifies a string to be displayed for the `CellData` object.
- `getDisplayString()` retrieves the string displayed by the `CellData` object.

A request for display from `Table` generates a `JCCellDisplayEvent` and notifies any `JCCellDisplayListeners` that they can customize the display string by calling `setDisplayString()` on the event. The `getDisplayString()` method returns the String as it will be displayed if `setDisplayString()` is not called.

JCCellDisplayListener

To register table to receive `JCCellDisplayEvent` objects, use the following call (where `(this)` refers to the class `MyClass`, which implements the `JCCellDisplayListener` interface):

```
table.addCellDisplayListener(this);
```

`JCCellDisplayListener` requires the following method to be implemented:

```
public void cellDisplay(JCCellDisplayEvent e)
```

Using JCCellDisplay Events and Listeners

`JCCellDisplayListener` can be used to format the display string. Unlike the `JCLabelValueEvent` and `JCCellValueEvent` of JClass LiveTable 2.x, `JCCellDisplayEvent` does not provide any mechanism to store the displayed data in the data source. The following example (see *examples/chapter7/BooleanDisplay.java*) displays objects as yes/no. Setting the display string does not have any effect during edit.

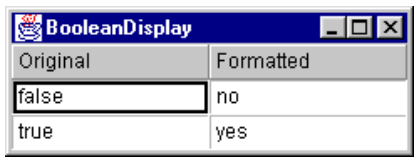


Figure 34 Using JCCellDisplayEvent to display BooleanCellData objects as yes/no strings

```
import jclass.table3.*;
import java.awt.*;
import java.awt.event.*;
import jclass.util.JCVector;

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

    table = new Table();

    evds = new EditableVectorDataSource();
    evds.setNumRows(2);
    evds.setNumColumns(2);

    evds.setColumnLabel(0, "Original");
    evds.setColumnLabel(1, "Formatted");

    // 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));

     table.setDataSource(evds);
     table.setMode(JCTblEnum.MODE_STYLEDTABLE);
     table.setRowLabelDisplay(false);

     add("Center", table);
```

```
    table.addCellDisplayListener(this);
  }

  public void cellDisplay(JCCellDisplayEvent e) {
    if(e.getColumn() == 1 && e.getRow() != JCTblEnum.LABEL) {
      if(e.getDisplayString().equalsIgnoreCase("true"))
        e.setDisplayString("yes");
      else
        e.setDisplayString("no");
    }
  }

  public boolean handleEvent(Event event) {
    if(event.id == Event.WINDOW_DESTROY) {
      hide();
      dispose();
      System.exit(0);
    }
    return super.handleEvent(event);
  }

  public static void main(String args[]) {
    BooleanDisplay f = new BooleanDisplay();
    f.pack();
    f.show();
  }

}
```

7.2 Creating Components

JCCreateComponentEvent

When components are added to a range of cells, table creates as many components as required to fill the cells displayed on the screen. A `JCCreateComponentEvent` will be generated by table whenever table requires more components than are currently created. When you receive a `JCCreateComponentEvent` object, you can call the following methods:

- `getRow()` retrieves the row of the cell or label which will contain the component.
- `getColumn()` retrieves the column of the cell or label which will contain the component.
- `getSourceComponent()` retrieves the component being cloned.
- `setComponent()` sets the component to be placed in the cell.

JCCreateComponentListener

To register table to receive `JCCreateComponentEvent` objects, use the following call (where `(this)` refers to the class `MyClass`, which implements the `JCCreateComponentListener` interface):

```
table.addCreateComponentListener(this);
```

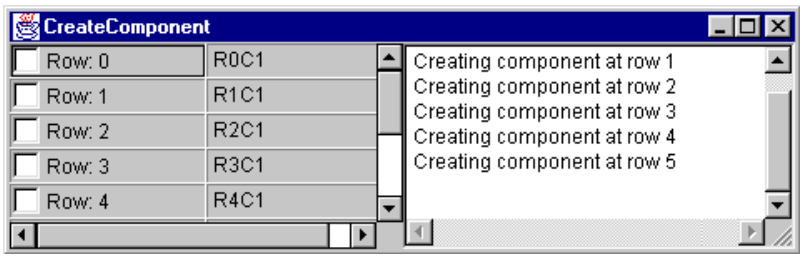
JCCreateComponentListener requires the following method to be implemented:

```
public void createComponent(JCCreateComponentEvent e)
```

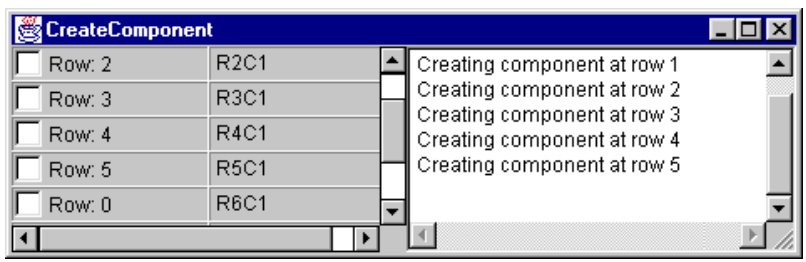
Using JCCreateComponent Events and Listeners

When components are added to single cells, there is no need to register a JCCreateComponentListener. However, when components are added to a range of cells, table makes "clones" of the component on an as-needed basis. The JCCreateComponentListener gives the developer the ability to control how those components are created. If you add components to a range and do not specify a JCCreateComponentListener, the table will attempt to generate “clones” by calling newInstance() on the class of the original component.

The following example (see *examples/chapter7/CreateComponent.java*) creates new Checkbox components in the createComponent() method. Because a JCDisplayComponentListener is not registered in this example, the checkbox titles reflect how table is conserving component creation.



As you scroll the display down, notice that the Checkbox components are reused by the program:



JClass LiveTable will only generate a new JCCreateComponentEvent when the table is resized and more rows are required. To synchronize the display of the component with the actual cell in which the component appears, a JCDisplayComponentListener must be registered.

The following code comprises the above example program.

```
import jclass.table3.*;
import java.awt.*;
import java.awt.event.*;
import jclass.util.JCVector;
```

```
public class CreateComponent extends Frame
implements JCCreateComponentListener {

  Table table;
  EditableVectorDataSource evds;
  TextArea ta;

  public CreateComponent() {
    super("CreateComponent");
    setBackground(Color.lightGray);

    table = new Table();
    evds = new EditableVectorDataSource();
    evds.setNumRows(10);
    evds.setNumColumns(2);

    for(int r = 0; r < evds.getNumRows(); r++) {
      evds.setCell(r, 0, new BooleanCellData(false));
      evds.setCell(r, 1, "R"+r+"C1");
    }
    table.setComponent(JCTblEnum.ALLCELLS, 0, new Checkbox());
    table.setDataSource(evds);
    table.setMode(JCTblEnum.MODE_STYLEDTABLE);
    table.setBackground(JCTblEnum.ALLCELLS, JCTblEnum.ALLCELLS,
                  Color.lightGray);
    table.setRowLabelDisplay(false);

    add("West", table);
    ta = new TextArea();
    add(ta, "Center");
    table.addCreateComponentListener(this);
  }

  public void createComponent(JCCreateComponentEvent e) {
    if(e.getColumn() == 0) {
      ta.appendText("Creating component at row " + e.getRow() +
"\n");
      e.setComponent(new Checkbox("Row: " + e.getRow()));
    }
  }
  public boolean handleEvent(Event event) {
    if(event.id == Event.WINDOW_DESTROY) {
      hide();
      dispose();
      System.exit(0);
    }
    return super.handleEvent(event);
  }
  public static void main(String args[]) {
    CreateComponent f = new CreateComponent();
    f.resize(600,150);
    f.show();
  }
}
```

7.3 Displaying Components

JCDisplayComponentEvent

When components are added to a range of cells, the table creates as many components as required to fill the cells displayed on the screen. A `JCDisplayComponentEvent` will be generated by table whenever the component is displayed. This allows you to tie component creation (using a `JCCreateComponentListener`) to the actual cell and underlying data source. When you receive a `JCDisplayComponentEvent` object, you can call the following methods:

- `getRow()` retrieves the row of the cell or label which will display the component
- `getColumn()` retrieves the column of the cell or label which will display the component
- `getComponent()` retrieves the component being displayed

JCDisplayComponentListener

To register table to receive `JCDisplayComponentEvent` objects, use the following call (where `(this)` refers to the class `MyClass`, which implements the `JCDisplayComponentListener` interface):

```
table.addDisplayComponentListener(this);
```

`JCDisplayComponentListener` requires the following method to be implemented:

```
public void displayComponent(JCDisplayComponentEvent e)
```

Using JCDisplayComponent Events and Listeners

Used in conjunction with a `JCCreateComponentListener`, adding a `JCDisplayComponentListener` to table allows you to customize how the component is displayed when scrolled into view.

Caution: The row and column returned in the event are the display row and column numbers. If you have sorted your data or dragged rows or columns, you must request the data row and column from the `TableDataView` object if you wish to customize the display relative to data source values.

The following example (see *examples/chapter7/DisplayComponent.java*) expands on the *CreateComponent.java* example above.

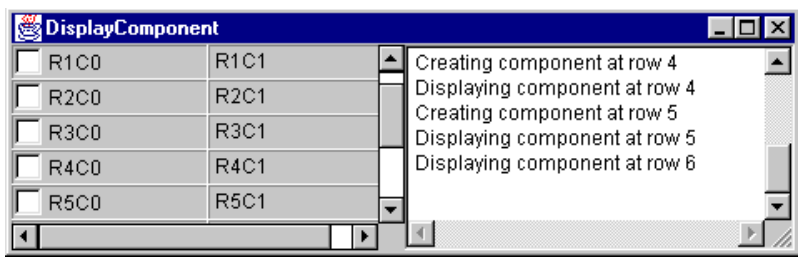


Figure 35 Notifying an application when component is displayed

Only portions of the source code that differ from the `CreateComponent` example are shown.

```
public class DisplayComponent extends Frame
implements JCCreateComponentListener, JCDisplayComponentListener {
...
...
public DisplayComponent() {
...
...
 table.addCreateComponentListener(this);
 table.addDisplayComponentListener(this);
}
public void createComponent(JCCreateComponentEvent e) {
  if(e.getColumn() == 0) {
   ta.appendText("Creating component at row " + e.getRow() + "\n");
    e.setComponent(new Checkbox("Row: " + e.getRow()));
 }
}

  public void displayComponent(JCDisplayComponentEvent e) {
   if(e.getColumn() == 0) {
    Checkbox cb = (Checkbox)e.getComponent();
    // You can use the event row/column values directly if your
    // table does not support sorting or dragging.
 int datarow = table.getDataView().getDataRow(e.getRow());
 int datacolumn = table.getDataView().getDataColumn(e.getColumn());
    ta.appendText("Creating component at row " + e.getRow() + "\n");
    cb.setLabel("R"+datarow+"C"+datacolumn);

  // Get the value in the data source
   BooleanCellData cd =
   (BooleanCellData)evds.getTableDataItem(datarow, datacolumn);
    cb.setState((cd.getData()).booleanValue());
   }
  }
```

How do I get my program to update my data when I click on a component?

The `Table` does not know how to update the data source when components are in place. You must intercept events on your components and update the data source as required. The `Table` method `getPosition()` can be used to determine the cell location for a component placed on the table. The following example (see *examples\chapter7\ComponentSave.java*) saves the checkbox state from the example above. The example in *examples\chapter7\ComponentSave102.java* shows how to process using the 1.0.2 event methods.

```
public class ComponentSave extends Frame
implements JCCreateComponentListener, JCDisplayComponentListener,
  ItemListener {
...
...
public void createComponent(JCCreateComponentEvent e) {
  if(e.getColumn() == 0) {
    ta.appendText("Creating component at row " + e.getRow() + "\n");
    Checkbox cb = new Checkbox("Row: " + e.getRow());
    cb.addItemListener(this);
    e.setComponent(cb);
  }
}
...
...
public void itemStateChanged(ItemEvent event) {
  if(event.getSource() instanceof Checkbox) {
    Checkbox cb = (Checkbox)event.getSource();
    JCCellPosition cp = table.getPosition(cb, null);

    int datarow = table.getDataView().getDataRow(cp.row);
    int datacolumn = table.getDataView().getDataColumn(cp.column);

    evds.setTableDataItem(new BooleanCellData(cb.getState()),
            datarow, datacolumn);
  }
}
...
...
```

7.4 Entering Cells

JCEnterCellEvent

This event will be posted whenever a user traverses into a cell. When you receive a `JCEnterCellEvent` object, you can call the following methods:

- `getRow()` retrieves the row number of the cell being entered.
- `getColumn()` retrieves the column number of the cell being entered.
- `getType()` retrieves the type where type is `JCEnterCellEvent.BEGIN` or `JCEnterCellEvent.END`
- `getParam()` retrieves a string indicating the initiating action:

  `POINTER` – Traverse to the cell the user clicked on

  `LEFT` – Traverse left to the first traversable cell

  `RIGHT` – Traverse right to the first traversable cell

  `UP` – Traverse up to the first traversable cell

  `DOWN` – Traverse down to the first traversable cell

  `null` – Initiated programatically by call to `JCTable.traverse`

JCEnterCellListener

To register table to receive `JCEnterCellEvent` objects, use the following call (where `(this)` refers to the class `MyClass`, which implements the `JCEnterCellListener` interface):

```
table.addEnterCellListener(this);
```

`JCEnterCellListener` requires the following methods to be implemented:

```
public void enterCellBegin(JCEnterCellEvent e)
public void enterCellEnd(JCEnterCellEvent e)
```

Using JCEnterCell Events and Listeners

The following example (see *examples\chapter7\EnterCell.java*) displays a status comment whenever a user enters a cell.

```
import jclass.table3.*;
import java.awt.*;

public class EnterCell extends Frame
implements JCEnterCellListener {

  Table table;
  EditableVectorDataSource evds;
  TextField message;

  String messages[] = {
    "This is the first column",
    "This is the second column",
    "This is the third column",
    "This is the forth column" };

  public EnterCell() {
    super("EnterCell");
    setBackground(Color.lightGray);

    table = new Table();

    evds = new EditableVectorDataSource();
    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);

    table.setDataSource(evds);
    table.setMode(JCTblEnum.MODE_STYLEDTABLE);
    table.setRowLabelDisplay(false);

    add("Center", table);
    add("South", message = new TextField());
    table.addEnterCellListener(this);
  }
```

```
  public void enterCellBegin(JCEnterCellEvent event) {
    message.setText(messages[event.getColumn()]);
  }

  public void enterCellEnd(JCEnterCellEvent event) {
  }

  public boolean handleEvent(Event event) {
    if(event.id == Event.WINDOW_DESTROY) {
      hide();
      dispose();
      System.exit(0);
    }
    return super.handleEvent(event);
  }

  public static void main(String args[]) {
    EnterCell f = new EnterCell();
    f.setSize(400,250);
    f.show();
  }
}
```

7.5 Painting

JCPaintEvent

This event will be posted when a portion of the table is painted. When you receive a `JCPaintEvent` object, you can call the following methods:

- `getRect()` retrieves the rectangle that is repainted.
- `getStartRow()` retrieves the start row of the repainted region.
- `getStartColumn()` retrieves the start end of the repainted region.
- `getEndRow()` retrieves the end row of the repainted region.
- `getEndColumn()` retrieves the end row of the repainted region.
- `getType()` retrieves the type where type is `JCPaintEvent.BEGIN` or `JCPaintEvent.END`

JCPaintListener

To register table to receive `JCPaintEvent` objects, use the following call (where `(this)` refers to the class `MyClass`, which implements the `JCPaintListener` interface):

```
table.addPaintListener(this);
```

`JCPaintListener` requires the following methods to be implemented:

```
public void paintBegin(JCPaintEvent e)
public void paintEnd(JCPaintEvent e)
```

Using JCPaint Events and Listeners

`JCPaintListener` allows you to monitor the repainting of table cells. Labels, frozen cells and scrollable cells are painted independently.

7.6 Printing

JCPrintEvent

This event will be 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.
- `getNumPages()` retrieves the total number of pages (handy for *page x of x* footers).
- `getType()` retrieves the type where type is `JCPrintEvent.HEADER`, `JCPrintEvent.FOOTER`, `JCPrintEvent.BODY` and `JCPrintEvent.END`

JCPrintListener

To register table to receive `JCPrintEvent` objects, use the following call (where `(this)` refers to the class `MyClass`, 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)
public void printEnd(JCPrintEvent e)
```

Using JCPrint Events and Listeners

`JCPrintListener` allows you to customize the header and footer regions for each page of the printout. Chapter 8, Table Printing has details and examples for using the `JCPrintListener`.

7.7 Resizing

JCResizeCellEvent

This event will be 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 `JCTblEnum.NOVALUE` if only a column is being resized.
- `getColumn()` retrieves the column being resized. Returns `JCTblEnum.NOVALUE` if only a row is being resized.
- `getParam()` retrieves stage in the resize process where stage is `JCResizeCellEvent.START` or `JCResizeCellEvent.END`.
- `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.

- `getAllowResize()` returns true if a resize is allowed.
- `setAllowResize()` determines whether to allow an interactive resize (default: true).
- `getType()` retrieves the type where type is `JCResizeCellEvent.BEGIN` or `JCResizeCellEvent.END`

The following methods have meaning only in the `resizeCellEnd()` method.

- `getNewRowHeight()` retrieves the new row height. Returns `JCTblEnum.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.

Resize messages are not generated during the drag process.

JCResizeCellListener

To register table to receive `JCResizeCellEvent` objects, use the following call (where `(this)` refers to the class `MyClass`, which implements the `JCResizeCellListener` interface):

```
table.addResizeCellListener(this);
```

`JCResizeCellListener` requires the following methods to be implemented:

```
public void resizeCellBegin(JCResizeCellEvent e)
public void resizeCellEnd(JCResizeCellEvent e)
```

Using JCResizeCell Events and Listeners

`JCResizeCellListener` allows you to customize how table resizes on a per-cell basis. The following example (see *examples\chapter7\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 jclass.table3.*;
import java.awt.*;

public class ResizeCell extends Frame
implements JCResizeCellListener {

  Table table;
  EditableVectorDataSource evds;

  public ResizeCell() {
    super("ResizeCell");
    setBackground(Color.lightGray);

    table = new Table();
    evds = new EditableVectorDataSource();
    evds.setNumRows(100);
    evds.setNumColumns(4);

    for(int c = 0; c < evds.getNumColumns(); c++)
      evds.setColumnLabel(c,"Column: "+c);
```

```
    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);
    }

    table.setDataSource(evds);
    table.setMode(JCTblEnum.MODE_STYLEDTABLE);

    add("Center",table);
    table.addResizeCellListener(this);
  }

  public void resizeCellBegin(JCResizeCellEvent event) { }

  public void resizeCellEnd(JCResizeCellEvent event) {
    if(event.getColumn() == JCTblEnum.LABEL) {
      event.setAllowResize(false);
      return;
    }

    int width = event.getNewColumnWidth();
    if(width < 100)
      event.setNewColumnWidth(100);
    else if(width > 200)
      event.setNewColumnWidth(200);
  }

  public boolean handleEvent(Event event) {
    if(event.id == Event.WINDOW_DESTROY) {
      hide();
      dispose();
      System.exit(0);
    }
    return super.handleEvent(event);
  }

  public static void main(String args[]) {
    ResizeCell f = new ResizeCell();
    f.setSize(400,250);
    f.show();
  }

}
```

7.8 Scrolling

JCScrollEvent

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 `scrollBegin(JCScrollEvent)` and `scrollEnd(JCScrollEvent)` scrollbar interface methods to store an internal state:

```
public MyClass extends Frame implements JCScrollListener {
....
  public void scrollBegin(JCScrollEvent ev) {
  if (ev.getDirection() == TableScrollbar.HORIZONTAL)
    hScrollingActive = true;
  else if (ev.getDirection() == TableScrollbar.VERTICAL)
    vScrollingActive = true;
  }
public void scrollEnd(JCScrollEvent ev) {
  if (ev.getDirection() == TableScrollbar.HORIZONTAL)
    hScrollingActive = false;
  else if (ev.getDirection() == TableScrollbar.VERTICAL)
    vScrollingActive = false;
}
```

JCScrollListener

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);
```

`JCScrollListener` requires the following methods to be implemented:

```
public void scrollBegin(JCScrollEvent e)
public void scrollEnd(JCScrollEvent e)
```

Using JCScroll Events and Listeners

`JCScrollListener` allows you to synchronize table scrolling with another object. The following example (see *examples\chapter7\TwoTables.java*) links two tables together with one scrollbar. This example uses two tables inside another table to simulate a splitter window.

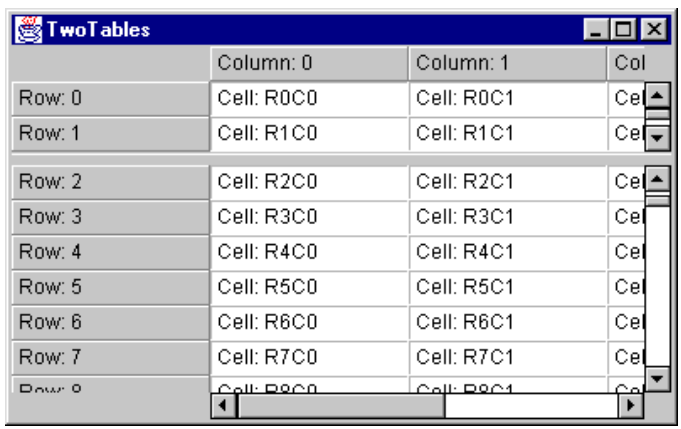


Figure 36 Example using JCScrollListener to synchronize scrolling between two tables

```
package jclass.table3.examples.chapter7;

import jclass.table3.*;
import jclass.contrib.ContribFrame;
import java.awt.Color;
import java.awt.Scrollbar;

public class TwoTables extends ContribFrame
    implements JCScrollListener {

Table table1;
EditableVectorDataSource evds1;
Table table2;
EditableVectorDataSource evds2;
Table parentTable;
VectorDataSource vdsParent;

boolean forcedScroll = false;

public TwoTables() {
    super("TwoTables");
    setBackground(Color.lightGray);

    //parent table
    parentTable = new Table();
    vdsParent = new VectorDataSource();
    vdsParent.setNumRows(2);
    vdsParent.setNumColumns(1);
    parentTable.setDataSource(vdsParent);
    parentTable.setPixelWidth(JCTblEnum.ALL, JCTblEnum.VARIABLE);
    parentTable.setPixelHeight(JCTblEnum.ALL, JCTblEnum.VARIABLE);
    parentTable.setHorizSBDisplay(JCTblEnum.SBDISPLAY_NEVER);
    this.add(parentTable);

    // table1
    table1 = new Table();

    evds1 = new EditableVectorDataSource();
    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);
    }

    table1.setAllowCellResize(JCTblEnum.RESIZE_NONE);
    table1.setDataSource(evds1);
    table1.setHorizSBDisplay(JCTblEnum.SBDISPLAY_NEVER);
    table1.setStyled();
    table1.setTraversable(JCTblEnum.ALLCELLS, JCTblEnum.ALLCELLS,
                          false);
    table1.setVisibleRows(2);
    table1.setVisibleColumns(3);

    parentTable.setComponent(0,0, table1);
```

```
        // table2
        table2 = new Table();

        table2.setAllowCellResize(JCTblEnum.RESIZE_NONE);
        table2.setColumnLabelDisplay(false);
        table2.setDataSource(evds1);
        table2.setStyled();
        table2.setTopRow(2);
        table2.setTraversable(JCTblEnum.ALLCELLS, JCTblEnum.ALLCELLS,
                              false);
        table2.setVisibleRows(5);
        table2.setVisibleColumns(3);

        parentTable.setComponent(1,0, table2);

        table1.addScrollListener(this);
        table2.addScrollListener(this);

    }

    public void scrollBegin(JCScrollEvent event) {
        // use forcedScroll to prevent a loop
        if(event.getDirection() == Scrollbar.HORIZONTAL) {
            if(forcedScroll == false) {
                if((event.getScrollbar().getComponent()).getParent() ==
                                                                    table2) {
                    forcedScroll = true;
                    table1.getHorizSB().setValue(event.getValue());
                }
                if((event.getScrollbar().getComponent()).getParent() ==
                                                                    table1) {
                    forcedScroll = true;
                    table2.getHorizSB().setValue(event.getValue());
                }
            } else {
                forcedScroll = false;
            }
        }
    }

    public void scrollEnd(JCScrollEvent event) {
    }

    public static void main(String args[]) {
        TwoTables f = new TwoTables();
        f.show();
        f.pack();
    }

    }
```

7.9 Sorting

JCSortEvent

This event will be 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 table to receive `JCSortEvent` objects, use the following call (where `(this)` refers to the class `MyClass`, 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\chapter7\Sorter.java*) uses the row sort array to pull out the top value.

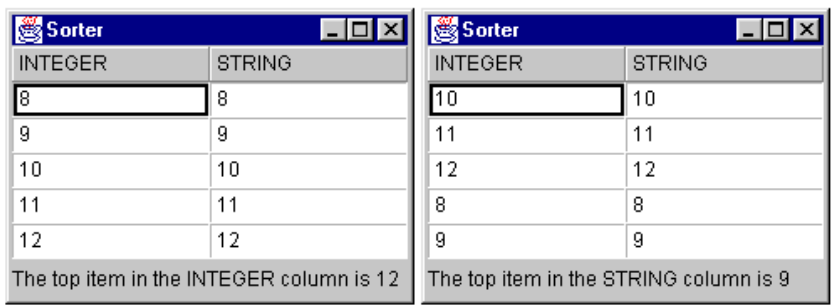


Figure 37 Sorter.java, illustrating how to use JCSort Events and Listeners

```
import jclass.table3.*;
import java.awt.*;

public class Sorter extends Frame
implements JCSortListener {
  Table table;
  VectorDataSource ds;
  Label topItem;

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

    table = new Table();

    ds = new EditableVectorDataSource();
```

```
    ds.setNumRows(5);
    ds.setNumColumns(2);
    ds.setColumnLabel(0, "INTEGER");
    ds.setColumnLabel(1, "STRING");

    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));
    }
    table.setDataSource(ds);
    table.setMode(JCTblEnum.MODE_STYLEDTABLE);
    table.setRowLabelDisplay(false);

    add("Center", table);
  // add a trigger to sort a column when it is clicked
    table.setColumnTrigger(0, LabelTrigger.SORT);
    table.addSortListener(this);

    add("South", topItem = new Label());
  }
  public void sort(JCSortEvent event) {
    int columns[] = event.getColumns();
    int rows[] = event.getNewRows();

    int maxrow = rows[rows.length - 1];
    topItem.setText("The top item in the " +
    ds.getTableColumnLabel(columns[0]) + " column is " +
     ds.getTableDataItem(maxrow,columns[0]));
  }
  public boolean handleEvent(Event event) {
    if(event.id == Event.WINDOW_DESTROY) {
      hide();
      dispose();
      System.exit(0);
    }
    return super.handleEvent(event);
  }
  public static void main(String args[]) {
    Sorter f = new Sorter();
    f.pack();
    f.show();
  }
}
```

7.10 Traversing

JCTraverseCellEvent

This event will be 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 proposed column of the cell to traverse to.
- `getRow()` retrieves the row of the current cell.
- `getNextRow()` retrieves the proposed row of the cell to traverse to.

- `getParam()` retrieves a string indicating the initiating action:
  `POINTER` Traverse to the cell the user clicked on
  `LEFT` Traverse left to the first traversable cell
  `RIGHT` Traverse right to the first traversable cell
  `UP` Traverse up to the first traversable cell
  `DOWN` Traverse down to the first traversable cell
  `null` Initiated programatically by call to `JCTable.traverse`
  `setNewColumn()` sets the column of the cell to traverse to.
  `setNewRow()` sets the row of the cell to traverse to.

JCTraverseCellListener

To register table to receive `JCTraverseCellEvent` objects, use the following call (where `(this)` refers to the class `MyClass`, which implements the `JCTraverseCellListener` interface):

```
table.addTraverseCellListener(this);
```

`JCTraverseCellListener` requires the following method to be implemented:

```
public void traverseCell(JCTraverseCellEvent e)
```

Using JCTraverse Events and Listeners

`JCTraverseCellListener` allows you to custom how cell traversal occurs in table. The following example (see *examples\chapter7\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 jclass.table3.*;
import java.awt.*;

public class RandomNavigation extends Frame
implements JCTraverseCellListener {

  Table table;
  VectorDataSource ds;

  public SkipNavigation() {
    super("SkipNavigation");
    setBackground(Color.lightGray);

    table = new Table();

    ds = new VectorDataSource();
    ds.setNumRows(10);
    ds.setNumColumns(4);

    for(int c = 0; c < ds.getNumColumns(); c++)
      ds.setColumnLabel(c, "Column: "+c);
      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);
      }
  table.setDataSource(ds);
  table.setMode(JCTblEnum.MODE_STYLEDTABLE);

  add("Center", table);
```

```
    table.addTraverseCellListener(this);
    }
    public void traverseCell(JCTraverseCellEvent event) {
      // for one-way:
      if(event.getColumn() == 0 && event.getNextColumn() == 1)
        event.setNextColumn(2);
      // for two-way:
      // if(event.getNextColumn() == 1)
      // event.setNextColumn(1 + event.getNextColumn() -
        event.getColumn());
    }
    public boolean handleEvent(Event event) {
      if(event.id == Event.WINDOW_DESTROY) {
        hide();
        dispose();
        System.exit(0);
      }
      return super.handleEvent(event);
    }
    public static void main(String args[]) {
      SkipNavigation f = new SkipNavigation();
      f.pack();
      f.show();
    }

  }
```

8

Table Printing

JClass LiveTable contains classes that enable end-users to print and print-preview table applications. Since printing is not available under JDK 1.0, these classes apply to JDK 1.1 (or higher) applications only.

8.1 Basic Printing

Basic print functionality is provided using the `table.print()` method, for example to print based on a java.awt.Button press:

```
public boolean handleEvent(Event event) {
    if (event.id == Event.ACTION_EVENT) {
    table.print();
```

Will print the table "as is" without any extra formatting.

8.2 Adding Enhanced Print Functionality

JClass LiveTable uses the `JCPrintTable` class for enhanced table printing. `JCPrintTable` creates a "copy" of the visible properties of the table and retrieves cell contents from the data source. The width and height of cells in the table are copied by actual pixel size. Printed tables do not display scrollbars or components.

8.2.1 Setting Page Layout Properties

The JCPrintTable class provides methods for more detailed control of print output from a JClass LiveTable application or applet.

Page Size

The following methods define printed page sizes:

```
getPageDimensions();
getPageWidth();
getPageHeight();
```

These methods retrieve page information from the printer.

Page Margins

Page margins are set using the setPageMargins() method. This method uses the awt.Insets class to set the margins as in the following example:

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

By default, JClass LiveTable sets the margins in pixels. To specify margin units in inches, use the variable MARGIN_IN_INCHES in the getMarginUnits() method:

```
setMarginUnits(JCPrintTable.MARGIN_IN_INCHES);
```

You can retrieve page margins based on the Insets of the page using the getPageMargins() method. Use the getDefaultPageMargins() to retrieve the default Insets.

Other Print Properties

Use getPageResolution() to get the printer page resolution. The default is 72 pixels per inch.

To control page numbering, use getHorizontalPages() and getNumVerticalPages() to determine the number of pages across and down.

8.2.2 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);
public GraphicsgetGraphics();
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 DimensiongetTableDimensions();
```

`getTableDimensions` can be used in the `printPageBody` method 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);
public void printEnd(JCPrintEvent e);
```

The `printPageBody` method is called after the body of the page is printed.

The following code would produce 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 38 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.

8.3 Adding Print Preview Capability

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, print individual pages, or print the whole job.

To add the print preview functionality, use `JCPrintPreview`:

```
JCPrintPreview(String title, JCPrintTable table)
showPage(int page)
```

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

```
JCPrintPreview pf = new JCPrintPreview("Table Print Preview",
    printtable);
pf.showPage(0);
```

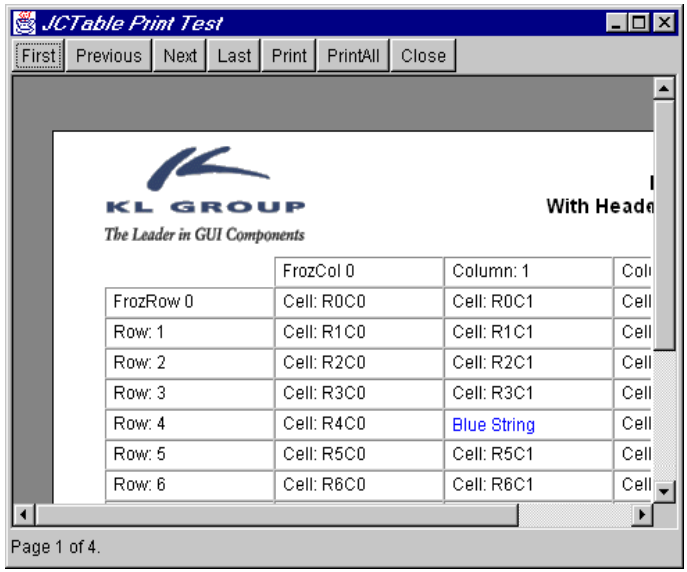


Figure 39 The JClass LiveTable Print Preview Window

9

JClass LiveTable Beans and IDEs

JClass LiveTable complies with the JavaBeans specification, and include several Beans that make it easy to create JClass LiveTable applications in an Integrated Development Environment (IDE). The following sections outline some principles of JavaBeans, and provide information about using JClass LiveTable in an IDE. All illustrations display the BeanBox, JavaSoft's test container for Beans included in the Beans Development Kit (BDK).

9.1 An Introduction to JavaBeans

Introduced in JDK 1.1, JavaBeans is a specification for reusable, pre-built Java software components. It is designed to be a fully platform-independent component model written for the Java programming language. The JavaBeans specification (available at *http://www.javasoft.com/beans/index.html*) enables developers to write components that can be combined in applications, reducing the total time needed to write entire applications.

The three main features of a Bean are:

- the set of properties it exposes
- the set of methods it allows other components to call
- and the set of events it fires.

9.1.1 Properties

Under the JavaBeans model, *properties* are public attributes that affect a Bean's appearance or behavior. Properties can be read only, read/write, or write only. Properties that are readable have a `get` method which enables you to retrieve the

property's value, and those properties which are writable have a `set` method which allows you to change their values.

For example, JClass LiveTable has a property called `FrameBorderType`. This property specifies the kind of border displayed around the table. To set the property value, use the `setFrameBorderType()` method. To obtain the property value, use the `getFrameBorderType()` method.

The main advantage of following the JavaBeans specification is that it makes it easy for a Java IDE to "discover" the set of properties belonging to an object. Developers can then manipulate the properties of the object easily through the graphical interface of the IDE when constructing a program.

There are three ways to set (and retrieve) JClass LiveTable Bean properties; use the method that applies best to your application:

1. By using a Java IDE at design-time
2. By calling property `set` and `get` methods in Java code
3. By specifying applet properties in an HTML file (see Setting Applet Properties in an HTML File, in Chapter 3 for details)

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

9.1.2 Setting Properties in a Java IDE at Design-Time

JClass LiveTable can be used with a Java Integrated Development Environment (IDE), and its properties can be manipulated at design time. If you install your IDE after you have installed JClass LiveTable, you will have to manually add LiveTable to the IDE's component manager. Refer to Adding JClass LiveTable to Your IDE, in Chapter 1 for more information. Also, consult your IDE documentation for information on working with third–party components.

Please see JClass LiveTable and JavaBeans for details on the LiveTable property editors.

Please see Property Differences Between the LiveTable and Data Binding Beans later in this chapter for information on the differences between the LiveTable Bean, and the data binding Beans.

9.1.3 Setting Properties using Methods in the API

As mentioned previously in this chapter, every property in JClass LiveTable has a `set` and `get` method associated with it. For example, to retrieve the value of the `FrameBorderType` property of a given cell and label area:

```
getFrameBorderType();
```

To set the `FrameBorderType` property in the same object:

```
setFrameBorderType(JCTbleEnum.BORDER_IN);
```

9.2 JClass LiveTable and JavaBeans

The JavaBeans included with JClass LiveTable make it easy to create applications and applets in an Integrated Development Environment. JClass LiveTable provides the following Beans:

- `LiveTable`: the core JClass LiveTable Bean
- `JCTableComponent`: provided for backward compatibility with LiveTable version 2.x (uses text editors to set properties).
- `Visual Café Bean (VCdbTable)`: the same as the `LiveTable` Bean, but can bind LiveTable to a database using Visual Café's QueryNavigator (version 2.5 or greater)
- `JBuilder Bean (JBdbTable)`: the same as the `LiveTable` Bean, but can bind LiveTable to a database using Borland JBuilder's DataSet (version 2.0 or greater)
- `LiveTable DataSource Bean (DSdbTable)`: the same as the `LiveTable` Bean, but can bind LiveTable to a database using KL Group's JClass DataSource

9.3 Setting Properties for the LiveTable Bean

At design-time, most `LiveTable` properties are set using simple menu choices or text entry boxes on the property sheet. Some properties that are set for individual cells or labels, or ranges of cells or labels are set using a property editor. The `LiveTable` property editors provide a visual interface for setting the properties using a model of the table you are creating, and a number of ways for selecting the cell(s) or ranges that you want to set the property for.

To make it easier to use, the `LiveTable` Bean combines some properties into special property groups that are set using a single editor. For example, the `Appearance` property combines the `Foreground`, `Background`, and `Font` properties and presents them in a single editor.

Note: Custom property editors are generated using introspection, a JDK 1.1 feature. Therefore, they are not available in the JDK 1.0.2 version of JClass LiveTable.

9.3.1 JClass LiveTable Property Editors

The following is a typical property editor:

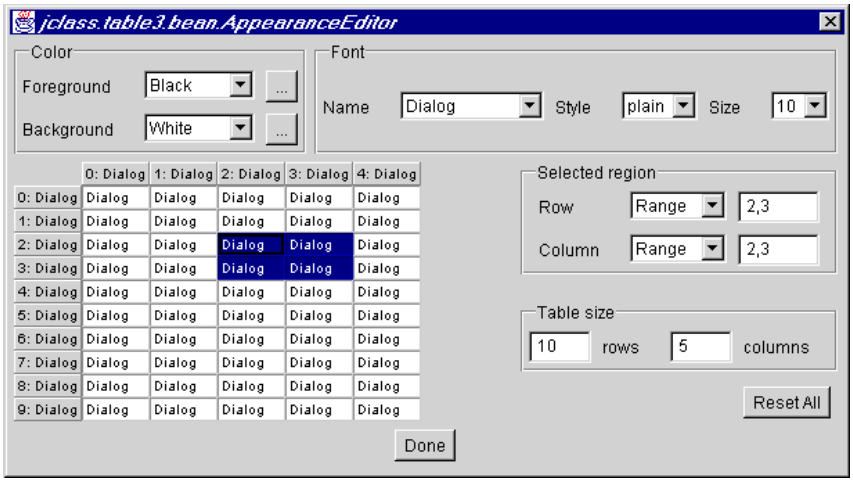


Figure 40 Property editor with elements common to other LiveTable property editors

Each property editor has the same interface for selecting the cells to which to apply a specific property. On the left is a view of the table (the data reflects the properties you're 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're building. For example, in the `AppearanceEditor` shown above, the table displays the name of the font set for each cell.

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 on 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 on another cell to select a range of cells between the two.

Note that when you make selections with the mouse, the ranges you have selected are displayed in the **Selected region** controls, as shown in the following diagram:

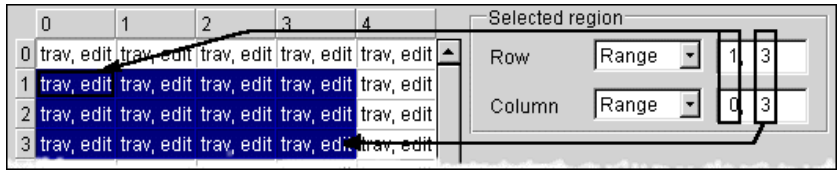
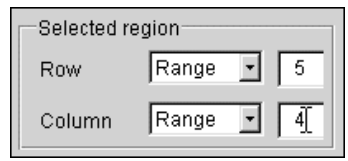


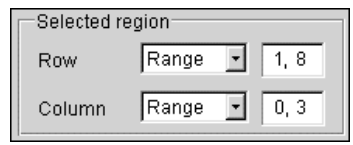
Figure 41 Mouse selection reflected as a range in the Selected Region box

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 5 for the **Row** range and 4 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 on 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 on 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. The row or column number for a label specified in a range is -1.

To select labels using the **Selected region** controls:

- To select a row label, choose Label from the **Column** pull-down menu; choose Range in the **Row** pull-down menu, and type the number of the row in the text field. To select all row labels, choose All in the **Row** pull-down menu.

- Likewise, to select a column label, choose Label from the **Row** pull-down menu, choose Range in the **Column** pull-down menu, and type the number of the column in the text field. To select all column labels, choose All in the **Column** pull-down menu.

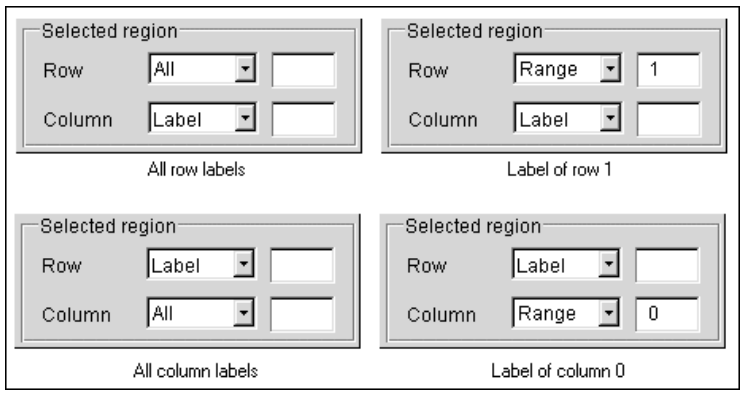
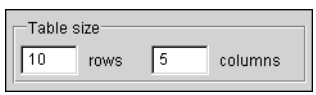


Figure 42 Selecting labels using the Selected region controls

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 on the Reset All button.

9.3.2 LiveTable Lite Features and Property Limitations

LiveTable Lite is a special version of LiveTable that is bundled with some popular IDEs. LiveTable Lite has some limited functionality, but imposes no run–time penalties. If you use LiveTable Lite and deploy an application or applet, there is no indication that the version of LiveTable that you are using is limited in any way: there are no special banners, dialogues or messages.

LiveTable Lite's feature differences are:

- When dropped into an IDE at design time, a dialog will appear that indicates that you are using the Lite version.
- All custom property editors will have a banner at the bottom that indicates you are using the Lite version and that some features are disabled.
- LiveTable's version number has "Lite" attached to the end of it.
- *All* Table Beans will have the design–time dialog. This means that even JCTableComponent will have the dialog, even though there is no limiting implemented for JCTableComponent.

LiveTable Lite's property limitations apply to `LiveTable`, `DSdbTable`, `JBdbTable` and `VCdbTable`. The limitations are:

Property	Lite Limitation
advancedEditorRenderers	property set to "off" (i.e. property not available)
appearance	**colors:** foreground and background colors limited to standard colors: black, blue, cyan, gray, green, light gray, magenta, pink, red, yellow and white dark gray, light blue, orange and custom colors not available **fonts:** access allowed to only 20% of available system fonts
cellBorderType columnLabelBorderType frameBorderType rowLabelBorderType	**settings available:** `BORDER_NONE`, `BORDER_IN`, `BORDER_OUT` and `BORDER_PLAIN`. **settings unavailable:** `BORDER_ETCHED_IN` and `BORDER_ETCHED_OUT`
focusRectColor	limited to black and white

jumpScroll	**settings available:**
	`JUMP_NONE`
	settings unavailable:
	`JUMP_HORIZONTAL`, `JUMP_VERTICAL` and `JUMP_ALL`

printSettings	no print preview, only wysiwyg printing enabled
selectedBackground selectedForeground	limited to black, white and null
spanningCells	cell span size is limited to one adjacent cell
textPositioning	positioning is limited only to top–left and bottom–right

9.3.3 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.

Since `LiveTable` is a subclass of `Table`, it is possible to switch and use the contained properties in `Table`.

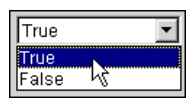
The following sections list the properties exposed in the `LiveTable` Bean. Many of the properties can be set for individual cells/labels or ranges; these properties are set using visual property editors (see JClass LiveTable Property Editors above for a description of a typical editor and how to select 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

This property displays the component version and where to find more information about JClass LiveTable.

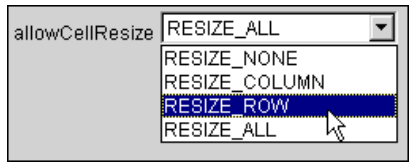
AdvancedEditorRenderers

Determines if the advanced editors and renderers are used for table cells. These are "light" versions of some of JClass Field's calendar date and numerical value entry features. This property takes a boolean value, selected from the pull–down menu.



AllowCellResize

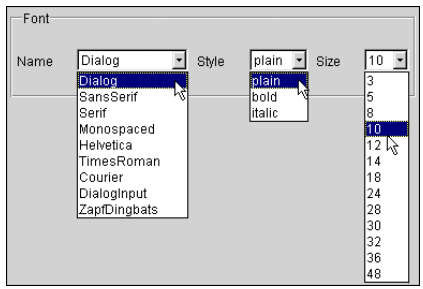
This property determines if the user can resize cells by dragging their borders. Set this property by choosing one of the values from the pull-down menu:



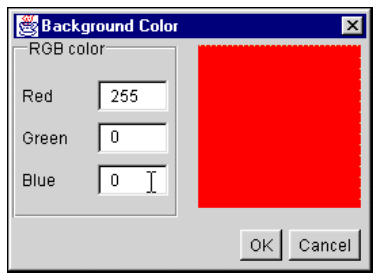
Appearance

This property editor sets the font, style, and size of cell and label text. From this editor, you can also set the foreground and background colors for specified cells and labels.

To set the font, style, and size of the text for a selection, make your selection and choose the options from the font panel of the `AppearanceEditor`.

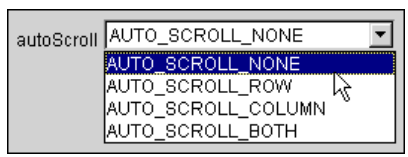


To set the foreground and background colors of selected cells, select from the standard AWT color constant by name in the pull-down menu, or click on the button marked ... to invoke a custom RGB color designer, as shown. Once you have made your choice, and removed the highlighting by moving the selection, your choice will appear in the table display of the editor.



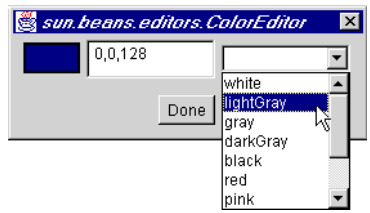
AutoScroll

The `AutoScroll` property determines if the table will automatically scroll if the user drags the mouse or traverses past the borders of the table. To set this property for rows, columns or both, choose a value from the pull-down menu:



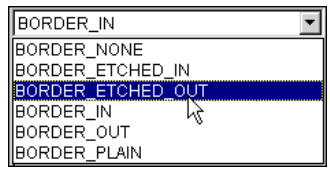
Background

The `Background` property is set for the whole table, that is, the area behind the table and the scrollbar components, and also sets the default cell background color. Click on the color box to bring up the color chooser:



CellBorderType

The `CellBorderType` property specifies the type of border drawn around cells or labels. To set this property, choose one of the border types from the pull-down menu. Note that etched and frame border types will not be visible if the `CellBorderWidth` is set to less than 5 pixels.

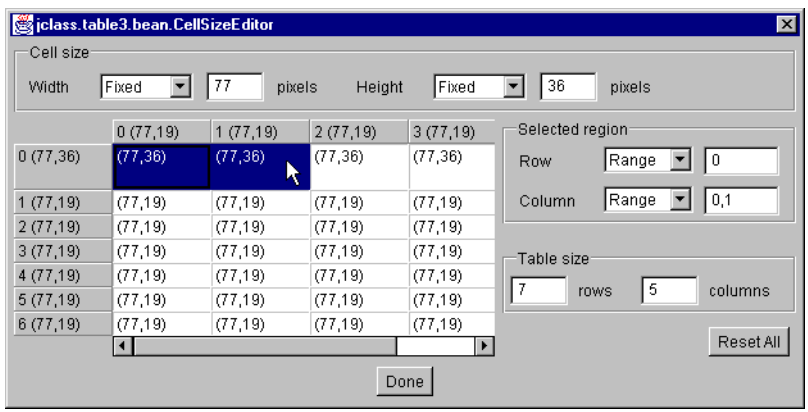


CellBorderWidth

This property specifies the thickness of the border around each cell and label. To change this property, enter a new value in the text field.

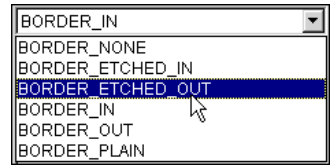
CellSize

The `CellSize` editor lets you set the dimensions of rows and columns as fixed pixel values or as variable values according to the cell contents. To set the value to a variable size, select `VARIABLE` in the **Width** or **Height** pulldown menus.



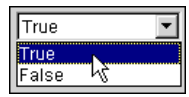
ColumnLabelBorderType

The `ColumnLabelBorderType` property has the same options as `CellBorderType`, but is set specifically for column labels. To set the `ColumnLabelBorderType` property, choose one of the border types from the pull-down menu. Note that etched and frame border types will not be visible if the `CellBorderWidth` is set to less than 5 pixels.



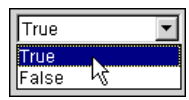
ColumnLabelDisplay

Specifies whether to display the column labels. In the `LiveTable` Bean, column labels are displayed by default even if they contain no data. To set this property, choose `true` or `false` from the pull-down menu.



ColumnLabelSort

Determines if the user can sort a column by clicking on its label. Choose a boolean value from the pull-down menu.



Data

This property exclusive to the LiveTable Bean, and not available in the data binding Beans, enables you add and customize table data and row/column labels. There are two ways to get data into the table – by entering data directly using the `DataEditor`, or by specifying a data file (can contain labels too).

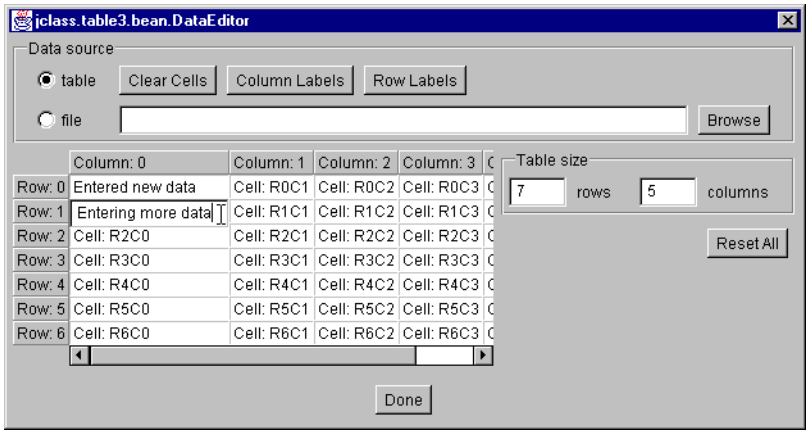
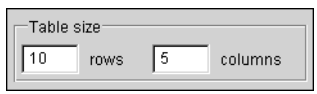


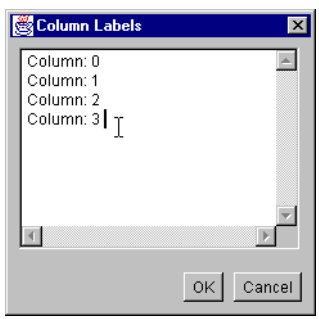
Table Data Source – To enter and edit data and labels directly, set the **Data source** radio button to **table**. You can then use the editor's table view to specify the data, as shown above.

To set the size of the table, use the **Table size** controls. You can increase or decrease the number of rows/columns. **Note:** These controls behave differently than in any other property editor because they change the actual size of the table you're building.



To delete all the data from the table, click the Clear Cells button.

To enter or edit row labels, click the Row Labels button. To edit column labels, click the Column Labels button. Type each label on its own line and press **Enter**:



File Data Source – To use data contained in a file, click the Browse button and select a valid table data file. The **Data source** radio button will switch to **file** when a valid file data source has been set. Once set, you cannot edit the data or labels directly, because the data source has been fixed to a file, so the **Table size** controls, and the **Clear Cells** and label editing buttons are disabled.

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:

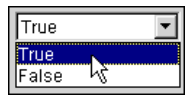
```
TABLE 4 4 NOLABEL
'0,0' '0,1' '0,2' '0,3'
'1,0' '1,1' '1,2' '1,3'
'2,0' '2,1' '2,2' '2,3'
'3,0' '3,1' '3,2' '3,3'
```

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'
```

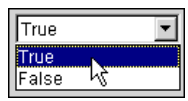
DisplayClipArrows

Determines whether text clip arrows are shown. This property takes a boolean value; choose `true` or `false` from the pull-down menu.



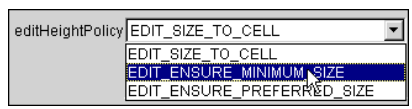
DoubleBuffer

Determines whether double buffering is used. This property takes a boolean value; choose `true` or `false` from the pull-down menu.



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.



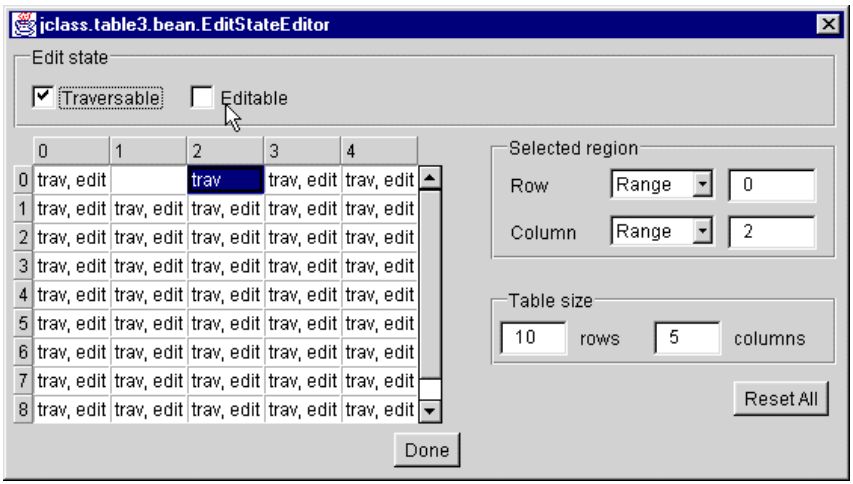
EditState

Sets the traversable and editable properties for specified cells.

Traversable indicates that users can traverse to the specified cells by clicking on 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 also 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.

If you set a cell to non-traversable, it cannot be edited.

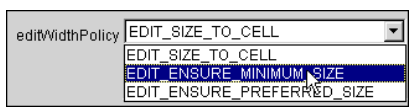
The `Editable` property is closely linked with cell traversal. While most cell editing functionality in JClass LiveTable relies on the `CellEditor` interface and the data source, you can still set individual cells or ranges to be `Editable` or not `Editable`, contingent on the existence of a cell editor and an editable data source.



To set the `Traversable` and `Editable` properties, select the cells you want to apply the property to in the EditState editor. The properties are set and unset using the checkboxes for each. The table display will reflect the current state of the selection, where possible. Note that if you uncheck the `Traversable` property for any cell, the `Editable` property is also unchecked.

Since labels are neither traversable nor editable, the selection options are limited to `ALLCELLS` and `RANGE`.

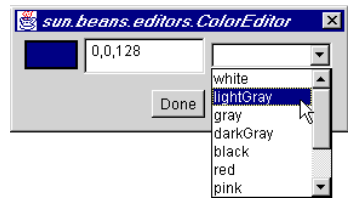
EditWidthPolicy



The table can control the width of a cell editing component using the `EditWidthPolicy` 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.

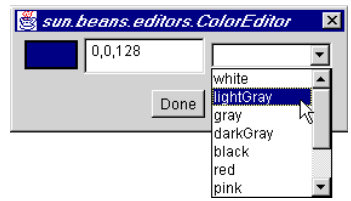
FocusRectColor

Sets the color of the focus rectangle. The focus rectangle is the line drawn around the inside of the cell that currently has focus. To set this property, choose the color from the color chooser dialog; when you have chosen the color, click the Done button.



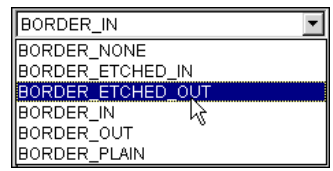
Foreground

This property sets the color for the foreground of the entire Table component, including the scrollbars, frame border, and cell borders. It also sets the default cell foreground color. When you have chosen the color, click the Done button.



FrameBorderType

The `FrameBorderType` property specifies the border type 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.

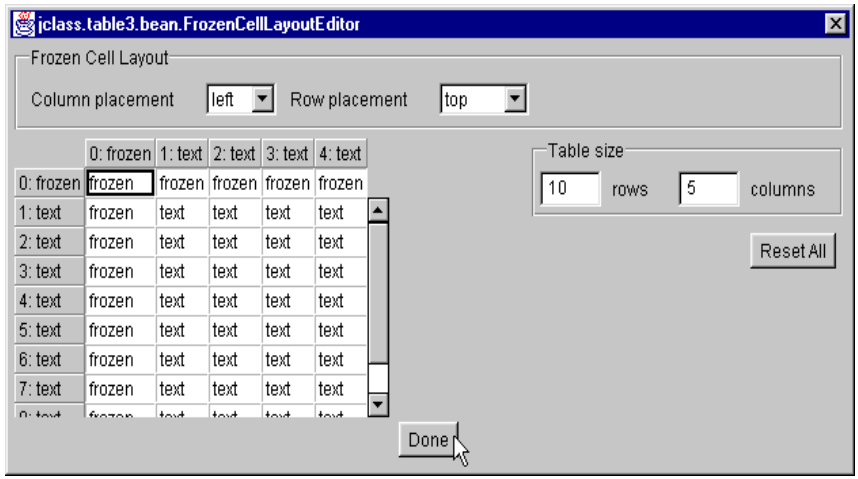


FrameBorderWidth

The `FrameBorderWidth` property is the thickness of the frame around the cell and label areas of the table in pixels. To change the value, type a new pixel value in the text field.

FrozenCellLayout

Sets the position of frozen rows and columns. 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.



FrozenColumns / FrozenRows

An application can make rows and columns non-scrollable using the `FrozenRows` and `FrozenColumns` properties. You can use frozen rows or columns to hold important information (such as totals at the bottom of the table) on the screen as a user scrolls through the table. You could also use frozen rows or columns as additional rows or columns that act like labels. They are still editable and traversable unless set otherwise.

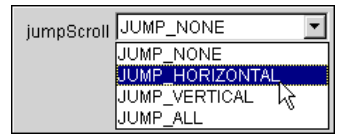
The number of frozen rows/columns always starts from the beginning of the table, although they can be repositioned using the FrozenCellLayout editor. Specify the number of frozen rows or columns in the text field.

HorizSBDisplay

This property specifies when the table should display its horizontal scrollbar: `DISPLAY_ALWAYS` keeps the horizontal scrollbar in place even if the table is big enough to display all of its columns; `DISPLAY_NEVER` hides the scrollbar all the time – scrolling is still available using the keyboard; `DISPLAY_AS_NEEDED` displays the horizontal scrollbar only when the table cannot display all of the columns.

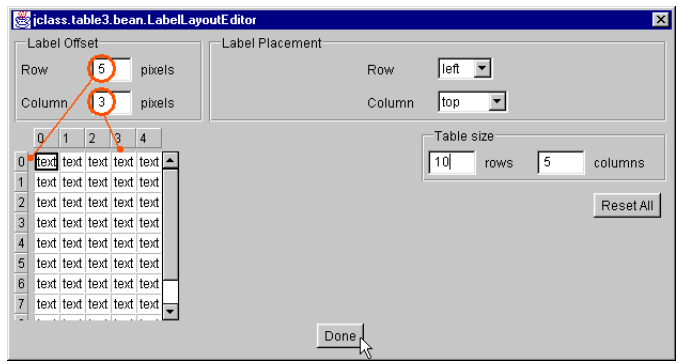
JumpScroll

By using the `JumpScroll` property, you can control the scrolling behavior of each scrollbar. Scrollbars can either scroll smoothly or "jump" scroll in whole row/column increments. To enable jump scrolling, choose one of the parameters from the pull-down menu.



LabelLayout

Sets the offset and placement of row and column labels. 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.

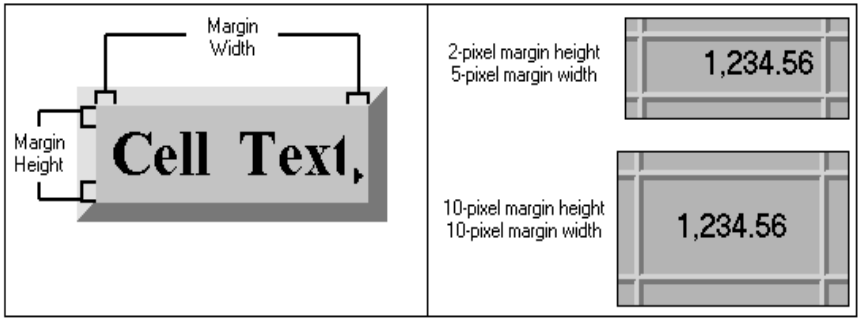


LeftColumn

This property specifies the leftmost column in the table as a referent for the table display. Enter the column number that you want displayed at the left of the table in the text field.

MarginHeight / MarginWidth

Height of the top and bottom margins of each cell, and width of the left and right margins of each cell. Type a pixel value in the text field to set this value.



MinCellVisibility

By default, when a user traverses to a cell that is not currently visible, `LiveTable` scrolls the table to display the entire cell.

The `MinCellVisibility` property sets the minimum amount of a cell made visible when it is traversed to. 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.

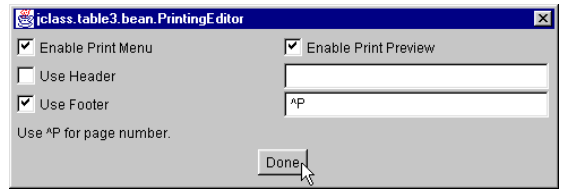
To change the value of `MinCellVisibility`, type the percentage in the text field.

PopupMenuEnabled

This property determines whether or not the popup menu is used. This property takes a boolean value. The availability of the Print and Print Preview in the menu is determined by the `printSettings` property (below). Additional menu options are available for the data binding Beans.

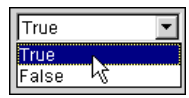
PrintSettings

This property determines if the print preview feature is available, and if the printing features are available as popup menu options. It also specifies if there are any headers or footers in the print out, and what information they will display (more extensive header and footer control is available through `JCPrintListener`).



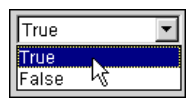
ResizeByLabelsOnly

This property specifies that rows and columns can be resized by moving the borders between their labels. Cells cannot be resized from the body of the table. This property takes a boolean value; choose `true` or `false` from the pull-down menu.



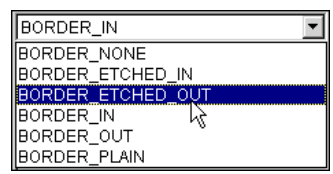
ResizeEven

When set to `true`, the `ResizeEven` property determines that when a row or column is resized, all of the other rows or columns in the table are set to the same dimensions. This property takes a boolean value; choose `true` or `false` from the pull-down menu.



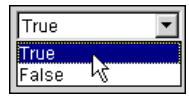
RowLabelBorderType

The `RowLabelBorderType` property has the same options as `CellBorderType`, but is set specifically for row labels. To set this property, choose one of the border types from the pull-down menu. Note that etched and frame border types will not be visible if the `CellBorderWidth` is set to less than 5 pixels.



RowLabelDisplay

By default, the `LiveTable` Bean displays row labels in the table. To specify that row labels are not displayed, set the boolean value to `false` in the pull-down menu.

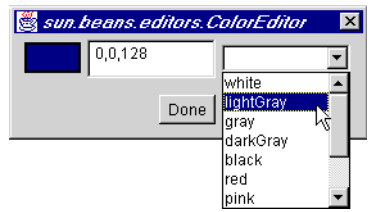


SBLayout

Sets the horizontal and vertical scrollbar offset.

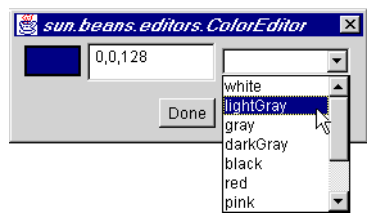
SelectedBackground

Background (highlight) color for cells that have been selected. The default is the foreground color for the cells. To choose a different background color, click on the color box to bring up the color chooser. When you have chosen the color, click the Done button.



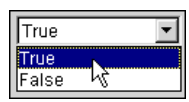
SelectedForeground

Foreground (highlight) color for cells that have been selected. The default is the background color for the cells. To choose a different background color, click on the color box to bring up the color chooser. When you have chosen the color, click the Done button.



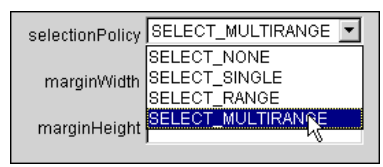
SelectIncludeLabels

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`.



SelectionPolicy

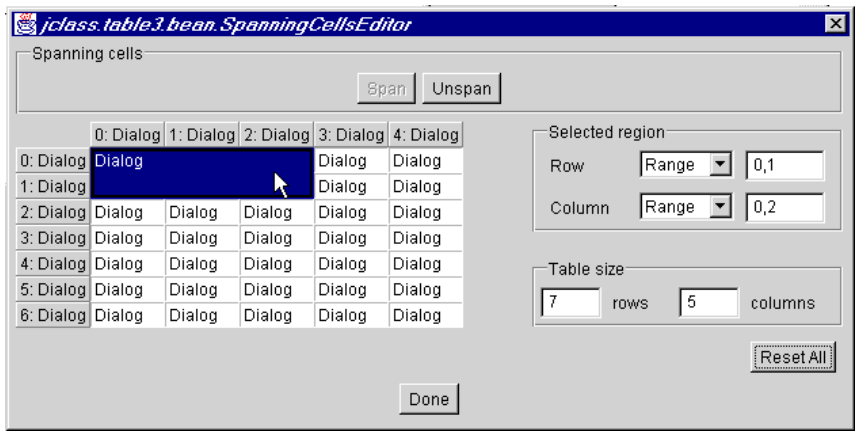
The `SelectionPolicy` property controls the amount of selection allowed on the table, both by end-users and by the application. Choose a value from the `SelectionPolicy` pull-down menu:



- SELECT_MULTIRANGE allows selection of discontinuous combinations of single cells and ranges of adjacent cells
- SELECT_SINGLE allows selection of only single cells
- SELECT_RANGE allows selection of only a single continuous range of adjacent cells
- SELECT_NONE does not allow any cell selection

SpanningCells

The spanningCells editor allows you to select a range of cells to span into a single cell area:



To span two or more cells:

- Select the cells using the mouse, or specify a cell range using the **Selected region** controls;
- Click the Span button. The cells are now displayed as a single cell, using the data from the top-left cell in the range.

To unspan the cells, select the spanned region and click the Unspan button.

Notes:

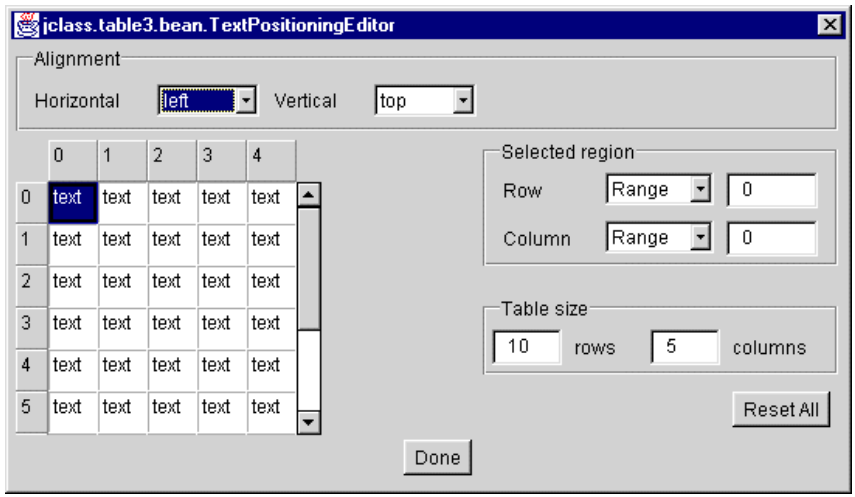
- You cannot combine cells and labels in a span, or frozen and non-frozen cells.
- The data hidden by spanned cells remains intact in the data source. When you unspan the cells, they are repopulated from the data source.

SwingDataModel

Sets the table's data source to use a specified Swing TableModel object, instead of using the Data property.

TextPositioning

Specifies the horizontal and vertical alignment of cell/label text.

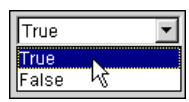


TopRow

This property specifies the topmost row in the table as a referent for the table display. Enter the row number that you want displayed at the top of the table in the text field.

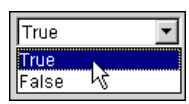
TrackCursor

Determines whether the cursor changes with mouse movements. This property takes a boolean value; choose `true` or `false` from the pull-down menu.



TraverseCycle

Determines if a traversal wraps to the opposite side of the table when the edge of the table is reached. This is a boolean value. Choose `true` or `false` from the pull-down menu.



VertSBDisplay

This property specifies when the table should display its vertical scrollbar: `DISPLAY_ALWAYS` keeps the vertical scrollbar in place even if the table is big enough to display all of its columns; `DISPLAY_NEVER` hides the scrollbar all the time – scrolling is still available using the keyboard; `DISPLAY_AS_NEEDED` displays the vertical scrollbar only when the table cannot display all of the columns.

9.4 Tutorial: Building a Table in an IDE

The following exercise will guide you through the steps to produce a JClass LiveTable program in an IDE. The exercise is the same as the one in Chapter 2, 'Hello Table' – A Simple JClass LiveTable Program, 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
WideEyes Coffee House	11/12/97	Colombian/Irish Cream Flavored	120	$5.30
Jitters Caffeine Cavern	10/01/97	Ethiopian Medium	80	$7.50
Twitchies on the Mall	12/06/97	French Roast Kona	160	$14.50
KL Group	12/12/97	Colombian	22,000	$5.28

1. We apologize for the addition of yet another coffee reference in an already crowded pantheon.

9.4.1 The Basic Table

The first step is to create a default table. In the BeanBox, click on the `LiveTable` component displayed in the Tool Box, then click in the BeanBox window. The BeanBox will display a default-sized (10 rows by 5 columns) table with row and column labels visible.

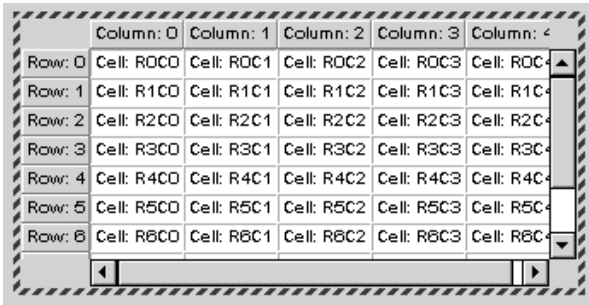
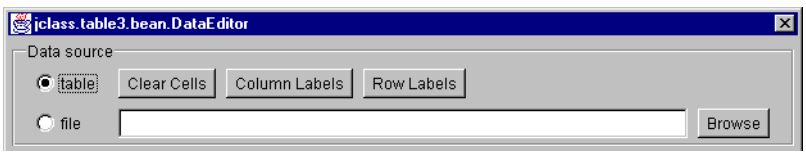


Figure 43 The default table in the BeanBox

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 on the `data` property value in 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:

- click the Browse button;
- navigate to the *examples\chapter9* directory of your JClass LiveTable distribution;
- choose the *tutorialdat.txt* file

Once you have chosen the file, the table display in the editor should be populated with the data from the data source, and the **Table size** and buttons at the top should be disabled, as shown in the following illustration:

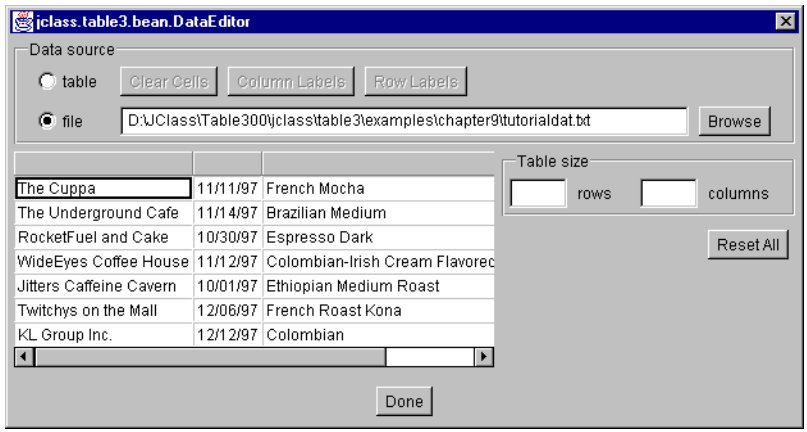


Figure 44 Using a file data source

To close the editor, click the Done button. The table displayed in the BeanBox will now show the values from the data source.

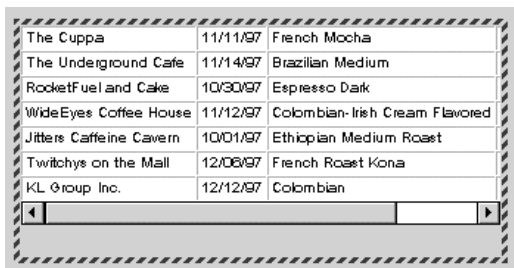


Figure 45 The Table after Importing the Data

9.4.2 Improving the Table's Appearance

Using some of the properties for modifying a table's appearance, you can easily move from the basic, drab table above, to a table that's easier to understand, easier to use, and more interesting to look at. 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 can't 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, etc.). 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\chapter9* directory of your JClass LiveTable distribution.

By default, the table displays row and column labels that have values. This is controlled by the `ColumnLabelDisplay` property, which takes a `boolean` value and has a default value of `true`. Now that you have column labels, the table in the IDE should update to look something like the following illustration:

Customer	Order Dat	Item	Quantity	Price/lb.
The Cupp	11/11/97	French M	60	$7.01
The Unde	11/14/97	Brazilian I	112	$6.80
RocketFue	10/30/97	Espresso	300	$8.02
WideEyes	11/12/97	Colombia	120	$5.30
Jitters Caf	10/01/97	Ethiopian	80	$7.50
Twitchys G	12/08/97	French Ro	160	$14.50
KL Group	12/12/97	Colombia	22,000	$5.28

Figure 46 Table after loading data file that includes labels

Notice that if you click on a label in your table, you don't get the focus rectangle the way you do if you click on a cell: labels cannot be edited or traversed to. In certain circumstances, clicking on a label will perform an action (see Adding Interactivity below), but in this case the labels don't 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'll take it one step further by changing the colors and fonts of the labels using the `AppearanceEditor` in the property sheet:

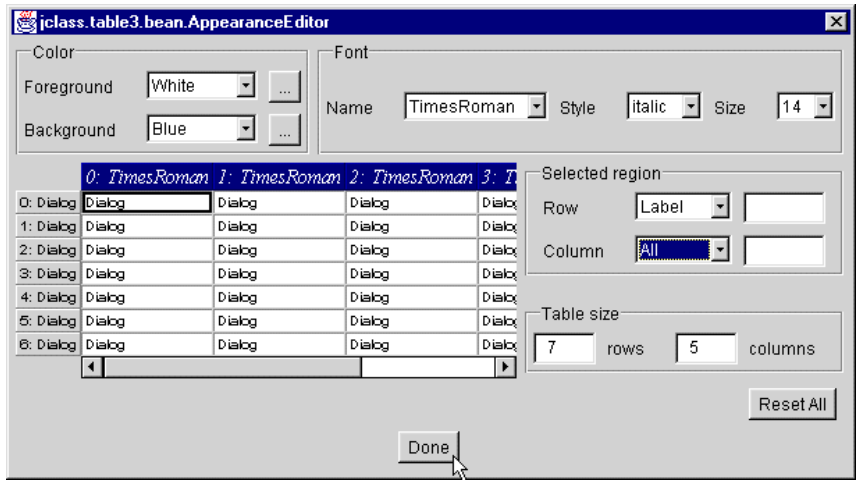


Figure 47 The Appearance property editor

To begin, you have to select the column labels:

- Choose **Label** from the **Row** pull-down menu in the **Selected region** box;
- Choose **All** from the **Column** pull-down menu

This will apply any settings you choose to the column labels. Next, you'll change the color of the label text:

- Choose `White` from the **Foreground Color** pull-down list;
- Choose `Blue` from the **Background Color** pull-down list;

Finally, choose the font, style, and size of the label text:

- Choose TimesRoman from the Font pull-down menu;
- Choose Italic from the Style pull-down menu;
- Choose 14 from the 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).

If you click on a cell that isn't currently selected, you'll be able to see your changes.

- Click the Done button 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.

Figure 48 Labels displayed with new colors and font

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.

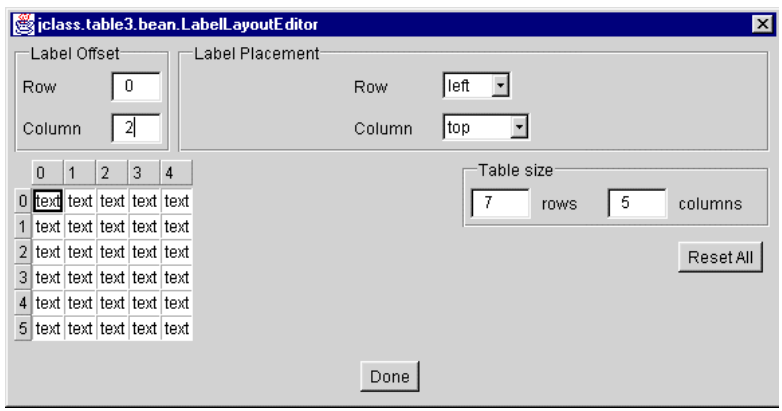


Figure 49 The LabelLayoutEditor

For this exercise, you are going to add some space between the column labels and the top of the table.

- Change the value of the **Column** text field from 0 to 2 in the **Label Offset** box.

The change is immediately reflected in the editor and the BeanBox.

- Click the Done button 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:

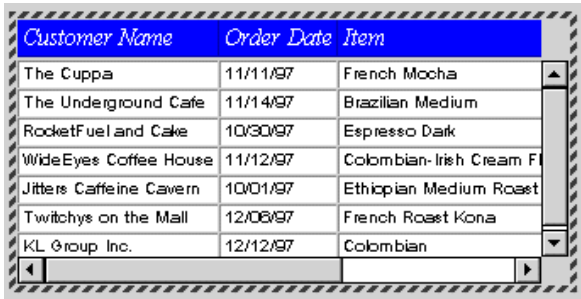


Figure 50 The table with the changes

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 above in the description of the `CellBorderType` property. For the example program, you're going to thicken the cell borders and change the border style. You will have to edit two properties: `CellBorderWidth` and `CellBorderType`.

To change the `CellBorderWidth` 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 `CellBorderType`:

- Choose `BORDER_IN` from the `CellBorderType` pull-down menu.

To set the BorderType for the column labels use the `ColumnLabelBorderType` property. The properties are separate so that you can set different borders for cells and labels:

- Set this property to `BORDER_OUT`.

The table should now resemble the following in the BeanBox:

Figure 51 Example table with new borders

9.4.3 Adding Interactivity

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

We'll add some basic user-interactivity to our example table to give you a sense of some of the things JClass LiveTable can do. You can explore user-interactivity further in Chapter 6, Programming User Interactivity.

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, KL Group has ordered 22,000 pounds of coffee. This is obviously a typographical error, but we're going to make sure KL Group gets all 22,000 pounds of coffee by not allowing that cell to be edited.

Invoke the `EditState` editor by clicking next to the `EditState` property:

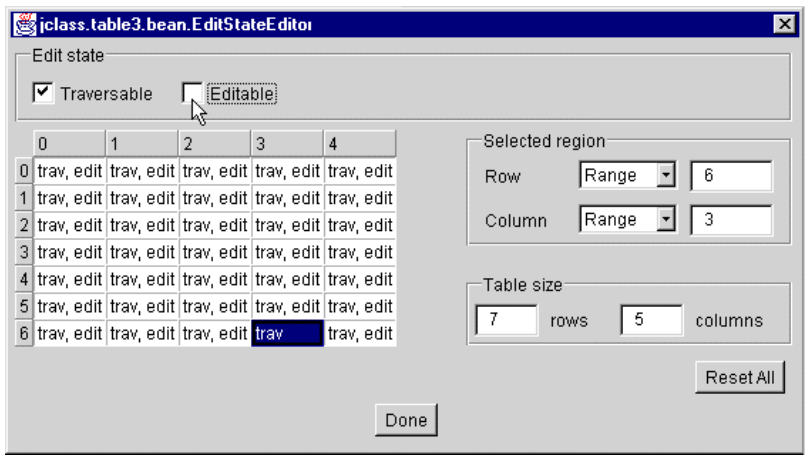


Figure 52 The EditState editor with cell 6,3 selected

In our original data, the cell containing the value 22,000 was located at row 6, column 3. You can either select this cell with the mouse, or type these values into 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:

- Uncheck the editable checkbox. This makes the cell traversable but not editable, as is displayed in the editor's table.
- Click the Done button to commit the changes and return to the BeanBox.

Now we can be sure that nobody will change KL Group'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.

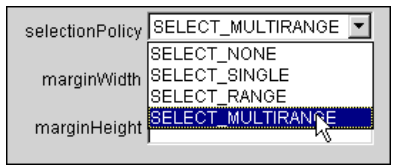


Figure 53 The SelectionPolicy menu

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 `ResizeByLabelsOnly` property to enable this feature. In the property sheet, change the `ResizeByLabelsOnly` property to `true`.

Enabling Column Sorting

It might be easier for your users to find certain information if they can sort the table based on cell values in a column. That way they can find a customer name alphabetically, or determine large orders by sorting the order amounts column.

A simple way to allow your users to sort a row or column is to set the `ColumnLabelSort` property. This property takes a boolean value. Set this to `true`.

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 `FocusRectColor` property:

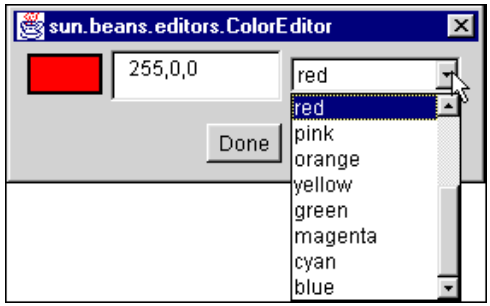


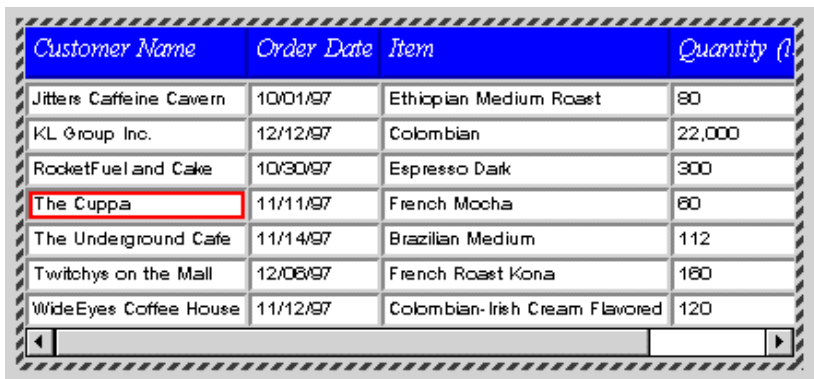
Figure 54 Choosing color for the focus rectangle

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

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

9.4.4 The Final Program

Your simple table program has evolved into an interactive, easy-to-understand utility. Although it's far from being a real order-tracking system, using a few more JClass LiveTable features, it soon could be. The following illustration shows all of the visual changes that you've accomplished. From here you can try out other properties and see how they affect the table's appearance and behavior.



9.5 Data Binding with IDEs

If you are using an IDE to develop Java applets and applications, the LiveTable data binding Beans allow you to bind a table with a JDBC–compliant data source, an ODBC data source (by using the JDBC–ODBC bridge), or an IDE–specific data source. Version 3.0 of LiveTable used the Model–View–Controller (MVC) data mechanism, with which table data was stored in a separate object. LiveTable 3.5's data binding Beans extend this principle, where the direct link between the table component and the IDE's data source offers an easier and more efficient way of representing and modifying data in your tables.

As outlined earlier in this chapter in JClass LiveTable and JavaBeans, LiveTable includes three data binding Beans: `JBdbTable` is used with JBuilder's DataSet, `VCdbTable` is used with Visual Café's QueryNavigator and `DSdbTable` is used with any JDBC data source (and JClass DataSource) in any IDE.

The principles of data binding and connecting to a database in any environment are similar. The following sections assume that you are:

- familiar enough with your IDE or other development environment to create and work with basic application projects
- familiar with setting up database connectivity in your development environment's projects

Please note that the examples used in the following sections use a sample JClassDemo database (*demo.mdb*) that is only included with JClass DataSource. As such, these examples are primarily meant to illustrate data binding with IDEs, as you will not be able to duplicate them if you do not have the sample database.

9.5.1 Data Binding LiveTable with a JBuilder Data Source

To data bind to a JBuilder's DataSet using `JBdbTable`, you require:

- Inprise JBuilder 2.0 or greater
- JDK 1.1 or greater
- JClass LiveTable's `JBdbTable` Bean
- a data source properly set up in Windows' ODBC Data Source Administrator

Creating a Java application that contains a data bound table in JBuilder requires an understanding of database connectivity in a JBuilder project. Binding your table with a database in this IDE involves:

- creating the project and laying out the UI
- adding and configuring the Database component
- adding and configuring the QueryDataSet component
- adding and configuring the LiveTable data binding Bean (`JBdbTable`)

There are different methods and components with which a JBuilder project with database connectivity can be created. The following provides a general overview of data binding, as it is assumed that you are familiar with working with your IDE. For

specific information, please refer to your JBuilder documentation, where comprehensive tutorial and reference information can be found about setting up an application project, and adding the Database and QueryDataSet components to manage the JDBC connection, and communicating with the database.

Let's begin with a basic project in JBuilder, where the UI components are set up, readying the project for the addition of the database and data binding components.

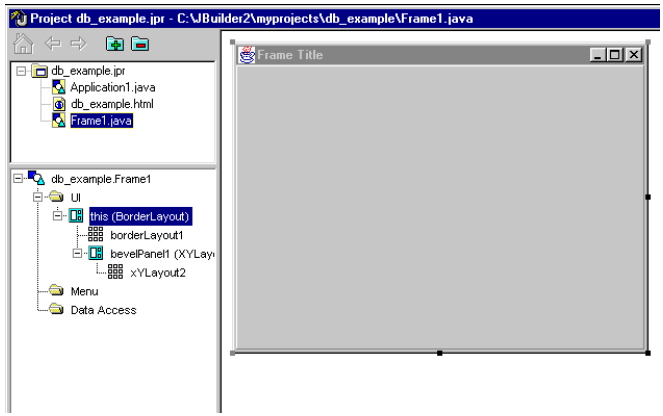


Figure 55 Example project work space with defined UI components, ready for database components

Adding the Database Component to the Project

In the Data Express tab of the Component Palette, click the `borland.sql.dataset.Database` object, designating it as the component to be added. Next, click an empty area of the Component Tree. The database object is added to your project.

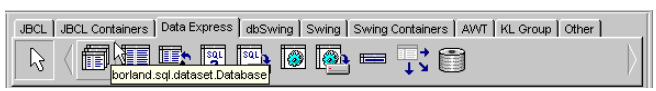


Figure 56 Selecting the Database component on the Component Palette

Setting the Connection Property for the Database Component

Now that the database object has been inserted into your project, you need to define the JDBC Connection information for this object. This is done by setting the connection property in the Inspector, when the database object is selected.

It is here that you select the data source that you want bound to your table component. All available data sources and DataGateway sources recognized by JBuilder are listed and available to be set as the main data source. These data sources are defined with Windows' ODBC Administrator. In this example, the *demo.mdb* database (JClassDemo) is selected.

Adding the Database component and setting the connection properties adds the following lines of code to your project:

```
import borland.sql.dataset.*;
...
Database database1 = new Database();
...
database1.setConnection(new borland.sql.dataset.ConnectionDescriptor
   ("jdbc:odbc:JClassDemo", "", "", false,
   "sun.jdbc.odbc.JdbcOdbcDriver"));
```

This code introduces the database component (in this example, it is named as *database1*) and points it to the data source that you define (in this example, the sample *demo.mdb* database, JClassDemo is used).

Adding the QueryDataSet Component to the Project

Now that you have set up the link between your project and the desired database, you need to interact with that database. In the Data Express tab of the Component Palette, click the `borland.sql.dataset.QueryDataSet` object, designating it as the component to be added. Next, click an empty area in the Component Tree. The database query component is added to your project.

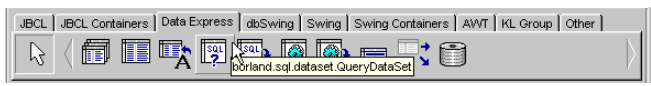


Figure 57 Selecting the QueryDataSet component on the Component Palette

Setting the Query Property for the QueryDataSet Component

Now that the QueryDataSet component has been added, you need to define which parts of which database will be used. To do this, you need to query the database with an SQL statement. This is done by setting the query property in the Inspector, when the QueryDataSet object is selected.

In the query property area, select the database you just added, and browse the tables if you have to get the proper information for your table. Enter the SQL statement that represents your needs. Test it to be sure that such a query will be successful.

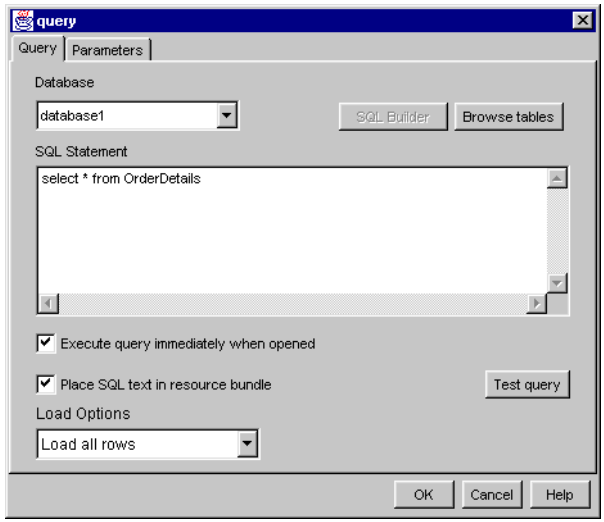


Figure 58 Entering the query statement to the selected database

Adding the QueryDataSet component and setting the query property adds the following lines of code to your project:

```
import java.util.*;
ReourceBundle sqlRes = ResourceBundle.getBundle("db_example.SqlRes");
...
QueryDataSet queryDataSet1 = new QueryDataSet();
...
queryDataSet1.setQuery(new borland.sql.dataset.QueryDescriptor
   (database1, sqlRes.getString("OrderDetails"),null,true,Load.ALL));
```

This code defines a new QueryDataSet component (in this example, named *QueryDataSet1*), which lets you read and write to and from the database by way of SQL statements. It also defines which part of the database is extracted, and bound to your table component. In this example, the OrderDetails table is selected with the query statement.

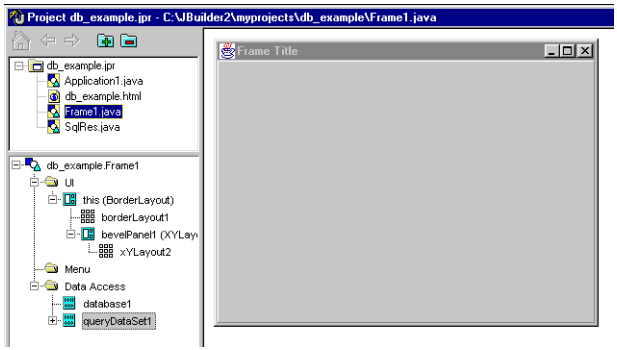


Figure 59 The project is ready for the table data binding component

Once the project's database and database connectivity components are in place and properly defined, the table component can be added, and the data binding can occur.

Adding the LiveTable Data Binding Bean to the Table

When installing LiveTable 3.5, its JBuilder components, including the LiveTable data binding Bean, were installed on JBuilder's Component Palette. If they are not there, please refer to Adding JClass LiveTable to Your IDE, in Chapter 1 for information on manually adding LiveTable components to JBuilder's Palette.

Click the table data binding Bean on the Component Palette, designating it to be added. Next, insert the data binding table component by clicking anywhere in the component tree (the table will be its default size), or by dragging and defining its size in the UI Designer.

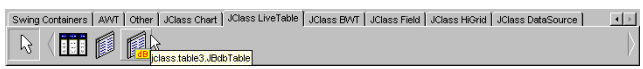


Figure 60 Selecting the data binding LiveTable component on the Component Palette

Setting the Dataset Property for the LiveTable Data Binding Component

Now that the data binding table component is part of the project, you need to define its DataSet. This is done by setting the dataSet property in the Inspector, when the table component is selected in the Component Tree.

It is here that you set the property to the QueryDataSet component name that is part of your project. In this example, the name of the component is *queryDataSet1*. This action adds these lines of code to your .java file:

```
import jclass.table3.db.jbuilder.*;
...
jclass.table3.db.jbuilder.JBdbTable jBdbTable1 = new
   jclass.table3.db.jbuilder.JBdbTable();
...
jBdbTable1.setDataSet(queryDataSet1);
...
this.add(jBdbTable1, BorderLayout.SOUTH);
```

This code introduces the LiveTable data binding Bean, and connects it to JBuilder's DataSet.

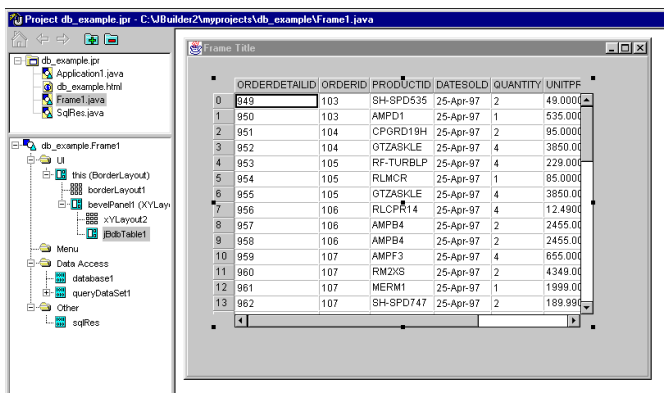


Figure 61 The project now contains a data bound table

Now that the table component has been linked to the QueryDataSet component, the data bound table is part of the project. The project can now be compiled and run, or continued to be developed.

9.5.2 Data Binding LiveTable with a Visual Café Data Source

To data bind to a Visual Café QueryNavigator using `VCdbTable`, you require:

- Symantec Visual Café 2.5 or greater (Database Development Edition)
- JDK 1.1 or greater
- JClass LiveTable's `VCdbTable` Bean
- a data source properly set up in Windows' ODBC Data Source Administrator

Creating a Java application that contains a data bound table in Visual Café requires an understanding of database connectivity in an IDE project. Binding your table with a database in Visual Café involves:

- creating a project and laying out the UI

- adding the QueryNavigator database component with dbNAVIGATOR
- adding the LiveTable data binding Bean (`VCdbTable`)
- setting the table and QueryNavigator connection to data bind

There are different methods and components with which a Visual Café project with database connectivity can be created. The following provides a general overview of data binding, as it is assumed that you are familiar with working with your IDE. For specific information, please refer to your Visual Café documentation, where comprehensive tutorial and reference information can be found about using the dbAWARE wizard to set up an application project, choosing the data source and establishing a database connection with dbNAVIGATOR and dbAWARE.

Let's begin with a basic Visual Café application project, where the UI components are set up, readying the project for the addition of the database and data binding components. Be sure that dbANYWHERE is running.

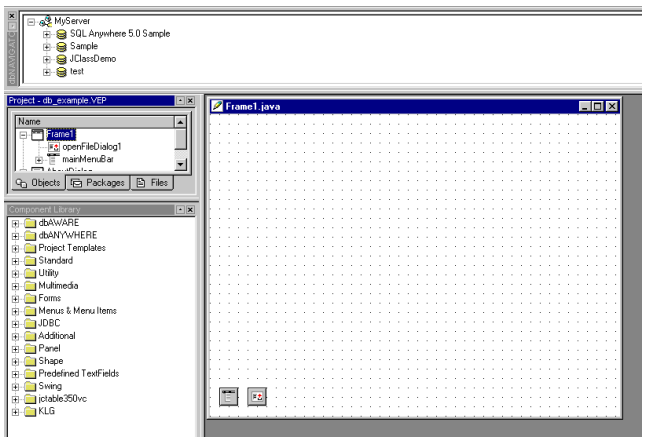


Figure 62 Example project with defined UI, and dbNAVIGATOR opened

Adding the QueryNavigator Database Component with dbNAVIGATOR

In order to perform this step, you must be running dbNAVIGATOR. Open it if you have not done so yet. dbNAVIGATOR lists all of the data sources that you have defined with Windows' ODBC Administrator. For this example, the *demo.mdb* database (JClassDemo) is selected.

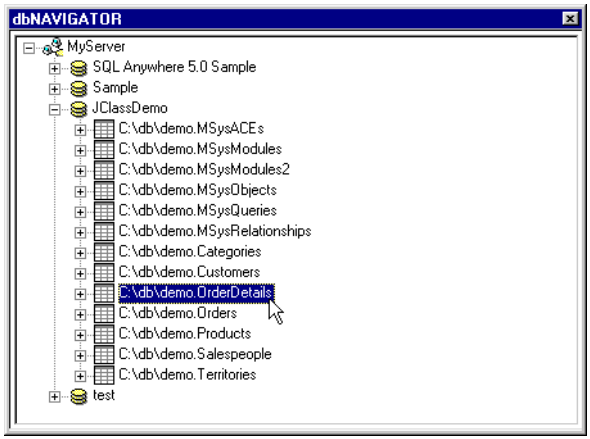


Figure 63 Making a selection from dbNAVIGATOR's listing of the example JClassDemo database

Click the database table that you want to use, and drag it to the Form Designer. The QueryNavigator appears where you placed it. This action adds the following lines of code to your project:

```
C__db_demo__OrderDetails_Navigator =
    new symantec.itools.db.beans.jdbcQueryNavigator();
C__db_demo__OrderDetails_Navigator.setAutoStart(true);
C__db_demo__OrderDetails_Navigator.setClassName
    ("C__db_demo__OrderDetails_Record");
C__db_demo__OrderDetails_Navigator.setAliasName
    ("C__db_demo__OrderDetails__Frame1_QNAlias");
...
symantec.itools.db.beans.jdbc.QueryNavigator
    C__jclass_devt_jclass_table3_examples_db_demo__OrderDetails_Navigator;
```

This code brings the QueryNavigator component into the program, and directs its queries to the chosen database and table (in this case OrderDetails in the JClassDemo sample database).

Now that the QueryNavigator component is part of the application, you are ready to add the LiveTable data binding Bean.

Adding the LiveTable Data Binding Bean

When installing LiveTable 3.5, its Visual Café components, including the LiveTable data binding Bean, were installed on Visual Café's Component Library. If they are not there, please refer to Adding JClass LiveTable to Your IDE, in Chapter 1 for information on manually adding LiveTable components to Visual Café's Palette.

Click the table data binding Bean on the Component Palette, designating it to be added. Next, insert the data binding table component by clicking anywhere in the Form Designer (the table will be its default size), or by dragging and defining its size.

Figure 64 Selecting the LiveTable data binding component

Inserting the table adds these lines of code to your project:

```
import jclass.table3.db.vcafe.VCdbTable;
...
vCdbTable1 = new jclass.table3.db.vcafe.VCdbTable();
add(vCdbTable1);
...
jclass.table3.db.vcafe.VCdbTable vCdbTable1;
```

This code simply introduces the LiveTable data binding bean (`VCdbTable`) into the program. However, the presence of the table component is not enough: you need to bind this table with the designated data source.

Setting the Table and QueryNavigator Connection to Data Bind

Click the QueryNavigator component in the Project window or Form Designer. In the Property List, highlight the component's Alias Name and copy it to the Windows clipboard.

Next, click the LiveTable data binding component in the Project Window or Form Designer. In the Property List, click the component's **DataBinding** property to access the **DataBinding Custom Editor**. In the **Query Navigator Alias** field, paste the text that you copied from the QueryNavigator's Alias Name property. Press **Tab** to have the Full Name field filled in automatically. Then, type `ALL` in the **Field Name** field to select all fields in the table.

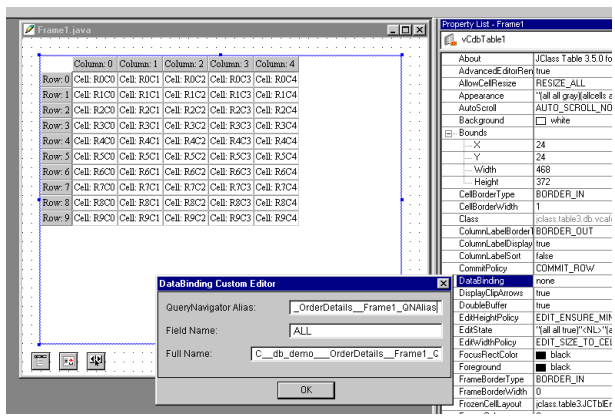


Figure 65 Setting the data binding property so that the table (with no data) knows where to bind

Once this has been done, you will not see any changes on the Form Designer, as Visual Café does not show these changes at design time. Defining the table's data binding property, and connecting it to a data source adds this line of code to your project:

```
vCdbTable1.setDataBinding("db_demo__OrderDetails__Frame1_QNAlias@ALL");
```

This line connects the table component to the desired table in the database, as defined by the QueryNavigator (in this case, the OrderDetails table from the JClassDemo sample database). You can now execute the project, or continue development.

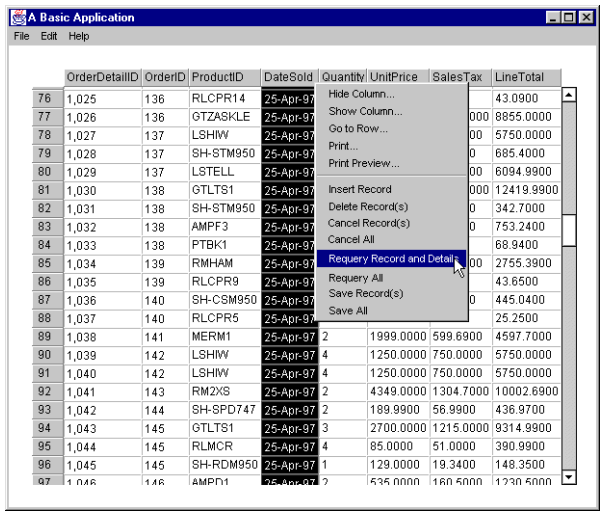


Figure 66 Executing the project produces a data bound table

9.5.3 Data Binding Using JClass DataSource

Using the DataSource data binding Bean (`DSdbTable`), you can bind a table component with any JDBC–compliant data source. The `DSdbTable` Bean works in any IDE[1] but can also be used if you are developing an applet or application without one. Data binding with the `DSdbTable` Bean requires:

- Sun Microsystems' Beans Development Kit or an IDE
- JDK 1.1 or greater
- JClass DataSource
- JClass LiveTable's `DSdbTable` Bean
- a data source properly set up in Windows' ODBC Data Source Administrator

1. If you are developing an application with JBuilder or Visual Café, you can use the specific data binding Beans that were designed for use with them (`JBdbTable` and `VCdbTable`, respectively).

When data binding your table component with a database, JClass DataSource manages the connection and querying to the database in your development project. Using the `DSdbTable` Bean creates a table component that connects with DataSource, thus completing the data binding link.

JClass DataSource uses two data binding Beans: the `DataBean` and the `TreeDataBean`. The `DataBean` allows data binding to flat data models, while the `TreeDataBean` allows data binding to hierarchical data models. The following example provides a general overview of data binding a LiveTable component to a database in Sun's Bean Development Kit.

It is assumed that you already are familiar with setting up a database connection with JClass DataSource. For specific information, please refer to your JClass DataSource documentation. Binding your table with a database involves:

- creating a project in an IDE or the Beans Development Kit
- establishing a database connection with `DataBean` or `TreeDataBean`
- inputting database query statements with the `DataBean` or `TreeDataBean`'s `DataComponentEditor`, or `TreeDataBeanComponentEditor`
- adding the `DSdbTable` data binding Bean to the work area

Establish a database connection with the DataBean

Insert the `DataBean` into the design area. Doing this will allow you to begin working with the `DataComponentEditor` in the BDK Properties window.

If desired, enter names in the **Description** and **Model Name** fields. In this example, we will leave the BDK's default names (**Node1** and **DataBean1**). On the **Serialization** tab, click **Save As** to save your serialization file. Next, click the **Data Model** tab to specify which database you want to connect to.

You need to specify the **Server Name** and **Driver** on the **Data Model** \ **JDBC** \ **Connection** tab. For the purposes of this example, we are using the *demo.mdb* (JClass Demo) database. In the **Server Name** field, enter or select `jdbc:odbc:JClassDemo`, and in the **Driver** field, enter or select `sun.jdbc.odbc.JdbcOdbcDriver`. Ensure that the **Prompt User For Login** checkbox is empty, and test the connection. When you receive confirmation that the database connection is successful, you can begin to set up the query statements.

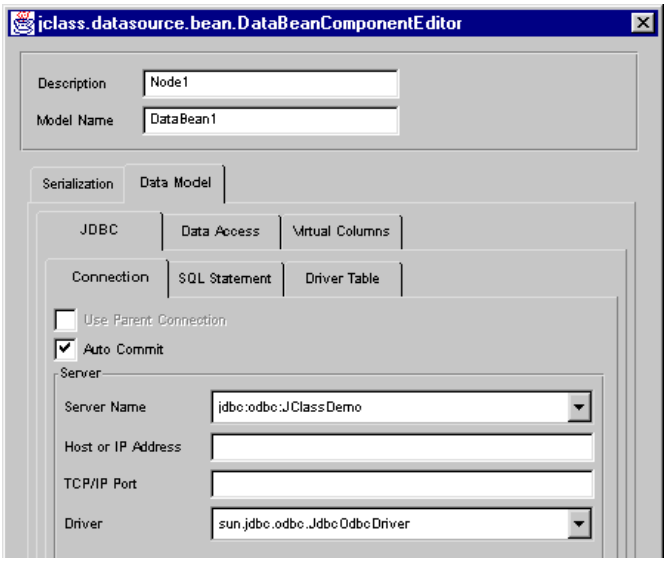
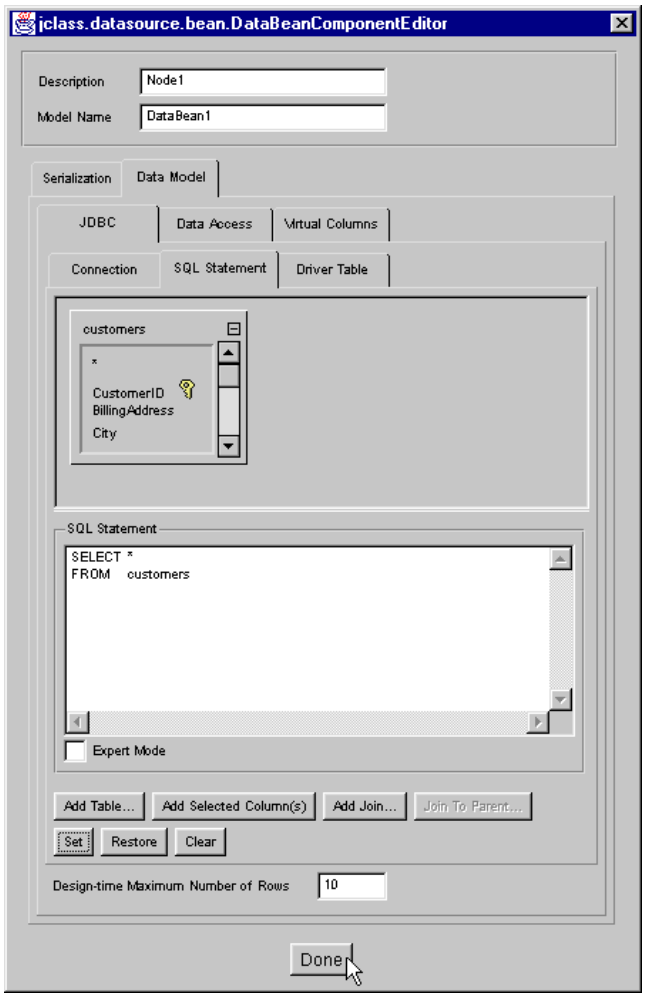


Figure 67 Connecting to the demo.mdb *database in Sun's Beans Development Kit*

Inputting database query statements with the DataBean

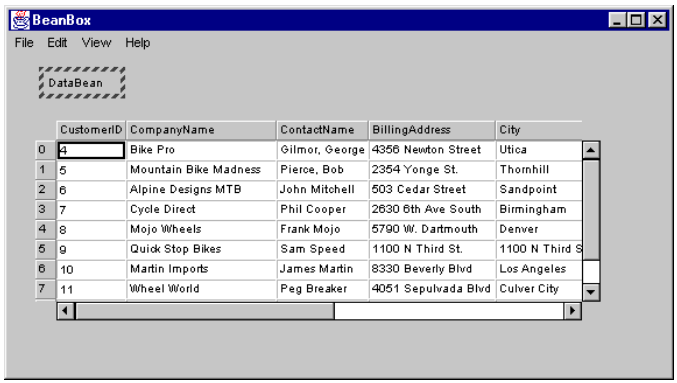
In order to properly query the database you have connected to, you need to input your query statement in the fields found on the **Data Model \ JDBC \ SQL Statement** tab. For this example, the *demo.mdb* database contains various tables, one of which is *Customers*. Enter `select * from Customers` to take all of the fields from the *demo.mdb*'s customers table, then click **Set**.

Now that you've successfully connected to, and queried the database, click **Done**.



Adding DSdbTable

The last step in creating a data bound table in your development project is adding the actual table component. In the BDK's Toolbox, click the DataSource data binding Bean (`DSdbTable`) and drop it into the BeanBox design area. Doing this will allow you to begin working with the `DSdbTable` properties in the Properties window. In the list of properties, click the **dataBinding** property, and set the connection to the appropriate data source. The data source is determined by what you entered in the **Description** and **Model Name** fields in the `DataBeanComponentEditor` (if you used the defaults in this example, they will be **Node1** and **DataBean1**). The table object will update to reflect the successful binding to the data source.



At this point, you have a table component in your design area, that is bound to the designated data source. You can now continue developing the rest of your application.

9.6 Interacting with Data Bound Tables

When a data bound table component has been successfully placed into your applet or application, you can interact with the table that takes advantage of the binding between the component and the data source.

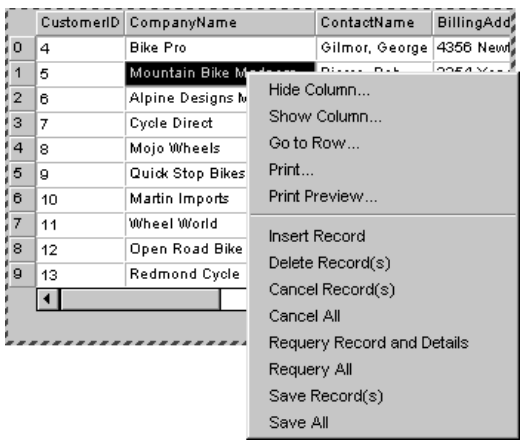


Figure 68 Interacting with the data bound table component

These actions are accessible through the table component's pop–up menu. By right–clicking a record, or multiple selected records, a list of possible actions is presented to the user.

Pop–up Menu Item	Function
Insert Record	Adds a new record to the current table and bound data source.
Delete Record(s)	Removes the selected row(s) from the current table and bound data source.
Cancel Record(s)	Cancels changes made to selected records.
Cancel All	Cancels all changes made to all records.
Requery Record and Details	Requeries the selected record(s) and any of its children from the database.
Requery All	Requeries all records in the table from the database.
Save Record(s)	Commits changes to selected records in the table, and updates the bound data source.
Save All	Commits all changes made in the table to the database

9.7 Property Differences Between the LiveTable and Data Binding Beans

Most of the common properties of the three data binding Beans (`JBdbTable`, `VCdbTable`, `DSdbTable`), are the same as the LiveTable Bean. By retaining most of the LiveTable Bean properties (outlined in LiveTable Properties), the new Beans provide feature–rich data binding table components.

The following data binding Bean properties are either unavailable, or have a new editors.

Data binding Bean	Property difference from LiveTable Bean
Appearance	same property; new editor
CellSize	unavailable in the data binding Beans
CommitPolicy	unavailable in LiveTable
Data	unavailable: replaced by specific data binding properties
FrozenRow	unavailable in the data binding Beans
LeftColumn	unavailable: data bound table always starts at column 0
RowLabelDisplay	unavailable in the data binding Beans
SpanningCells	unavailable in the data binding Beans
TopRow	unavailable: data bound table always starts at row 0
TraverseCycle	unavailable: always on in the data binding Beans

Part II

Reference Appendices

A

Event Summary

This table is a quick reference to JClass LiveTable's events and their corresponding event listeners. For details on how to use events and listeners in your programs, see Chapter 7, Events and Listeners.

Event	Listener Method	Description
JCAdjustmentEvent	addAdjustmentListener	Under JDK 1.1, it is a trivial extension of `java.awt.event.AdjustmentEvent`. Under JDK 1.0.2, it is a replacement for JDK 1.1's `java.awt.event.AdjustmentEvent`.
JCCellDisplayEvent	addCellDisplayListener	Posted when a cell's contents are to be displayed in the table.
JCCreateComponentEvent	addCreateComponentListener	Posted when a single component has been specified for multiple cells, or a component is specified in the componentList value in an HTML file.
JCDisplayComponentEvent	addDisplayComponentListener	Posted before drawing a component in a cell or label.
JCEnterCellEvent	addEnterCellListener	Posted before and after completing a traversal from one cell to another.
JCPaintEvent	addPaintListener	Posted when a portion of the table is painted.
JCPrintEvent	addPrintListener	The event posted for each page during printing.
JCResizeCellEvent	addResizeCellListener	Posted when the table is resized.
JCScrollEvent	addScrollListener	Posted when a user resizes a row and/or column.
JCSelectEvent	addSelectListener	Posted when a user selects cells.
JCSortEvent	addSortListener	Posted after a call to `sortByColumn`.

Event	Listener Method	Description
JCTraverseCellEvent	addTraverseCellListener	Posted when a user traverses from one cell to another.
JCValidateCellEvent (JClass LiveTable 2.x compatibility)	JCTable.addValidateCellListener	Posted when a cell has been edited, both before and after an edit has been committed to cell value structures.
JCValueEvent (JClass LiveTable 2.x compatibility)	JCTable.addCellValueListener JCTable.addLabelValueListener	Posted whenever the table displays an empty cell or label (one which has no value as described by the internal vector of the table). This event will be posted during the initial table display, scrolling and table repaint.
TableDataEvent	TableData.addTableDataListener	Describes changes to a `TableData` object.
TableInitialEvent	addInitialEventListener	`InitialEvent` objects are used to tell `CellEditor` classes what event started the edit.

B

JClass LiveTable Property Listing

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 `JCTblEnum` or an example of a value for setting the property. Default values are marked with an asterisk (*).

B.1 Properties of jclass.table3.Table

Properties controlling table design elements

Name	Description	Values/Examples

Alignment	The `Alignment` property specifies the alignment of cell or label text or images. This property can also be set for a `JCCellRange` of cells.	`JCTblEnum.TOPLEFT*`
		`JCTblEnum.TOPCENTER`
		`JCTblEnum.TOPRIGHT`
		`JCTblEnum.MIDDLELEFT`
		`JCTblEnum.MIDDLECENTER`
		`JCTblEnum.MIDDLERIGHT`
		`JCTblEnum.BOTTOMLEFT`
		`JCTblEnum.BOTTOMCENTER`
		`JCTblEnum.BOTTOMRIGHT`

Background	Background color for the entire table.	array of colors
CellBorderColor	Specifies border colors	single color value

Properties controlling table design elements

Name	Description	Values/Examples

CellBorderSides	Visible border sides (defined by `CellBorderType`) for individual cells.	`JCTblEnum.BORDERSIDE_NONE`
		`JCTblEnum.BORDERSIDE_ALL*`
		`JCTblEnum.BORDERSIDE_LEFT`
		`JCTblEnum.BORDERSIDE_RIGHT`
		`JCTblEnum.BORDERSIDE_TOP`
		`JCTblEnum.BORDERSIDE_BOTTOM`

CellBorderType	Border types for individual cells.	`JCTblEnum.BORDER_ETCHED_IN`
		`JCTblEnum.BORDER_ETCHED_OUT`
		`JCTblEnum.BORDER_FRAME_IN`
		`JCTblEnum.BORDER_FRAME_OUT`
		`JCTblEnum.BORDER_IN`
		`JCTblEnum.BORDER_OUT`
		`JCTblEnum.BORDER_PLAIN`
		`JCTblEnum.BORDER_NONE`

CellBorderWidth	Sets the shadow thickness around each cell.	integer: number of pixels
CharHeight	Height in characters of individual cells.	specific row number, `JCTblEnum.LABEL`, or `JCTblEnum.ALL`, plus the number of characters
CharWidth	Width of column in characters.	specific column number, `JCTblEnum.LABEL`, or `JCTblEnum.ALL`, plus the number of characters

ClipArrows	Determines whether clip arrows are shown, and where, when the contents of the cell do not fit in the cell frame.	`JCTblEnum.CLIP_ARROWS_BOTH`
		`JCTblEnum.CLIP_ARROWS_RIGHT`
		`JCTblEnum.CLIP_ARROWS_LEFT`
		`JCTblEnum.CLIP_ARROWS_NONE`

ColumnLabelDisplay	Determines whether the column labels display in the table.	boolean (default: true)
ColumnLabelOffset	Distance between column labels and table cells.	pixels (default: 0) e.g., `setColumnLabelOffset(4)`
ColumnLabelPlacement	Location of the column labels (top or bottom of the table).	`JCTblEnum.PLACE_TOP*` `JCTblEnum.PLACE_BOTTOM`

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`

Properties controlling table design elements

Name	Description	Values/Examples
FocusRectColor	The color of the focus rectangle.	context and color name, e.g., `setFocusRectColor(Color.blue);`
Font	Sets the font for the entire table.	array of colors
Foreground	Sets the foreground color for the entire table.	array of colors

FrameBorderType	Border type for the frame around the table. `FrameBorderWidth` must be set to greater than zero for the border to be visible.	`JCTblEnum.BORDER_IN*`
		`JCTblEnum.BORDER_NONE`
		`JCTblEnum.BORDER_ETCHED_IN`
		`JCTblEnum.BORDER_ETCHED_OUT`
		`JCTblEnum.BORDER_OUT`
		`JCTblEnum.BORDER_PLAIN`

FrameBorderWidth	Thickness of the frame around the entire table.	pixels (default:0), e.g., `setFrameBorderWidth(5);`
FrozenColumnPlacement	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.	`JCTblEnum.PLACE_LEFT*` `JCTblEnum.PLACE_RIGHT`
FrozenColumns	Specifies the number of columns from the start of the table that are not horizontally scrollable.	number of columns to freeze e.g., `setFrozenColumns(3);`
FrozenRowPlacment	Specifies the location of all frozen rows.	`JCTblEnum.PLACE_TOP*` `JCTblEnum.PLACE_BOTTOM`
FrozenRows	Specifies the number of rows from the start of the table that are not vertically scrollable.	number of rows to freeze, e.g., `setFrozenRows(2);`
IgnoreContainerSize	Specifies if the container determines the size of the table. Should be set to true when the table is set on a scroll pane, as the table will then take the maximum amount of space to display all of the cells.	boolean (default: false)
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 e.g., `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 e.g., `setMarginWidth(3);`
MaxLength	Maximum content length for individual cells.	integer, (default: MAXINT)
MinCellVisibility	Minimum visible percentage of a cell.	integer: 1 to 100

Properties controlling table design elements

Name	Description	Values/Examples
Mode	Determines table operation - table or list.	`JCTblEnum.MODE_LIST` `JCTblEnum.MODE_TABLE`

PixelHeight	Row height in pixels. This property controls the height unless set to `JCTblEnum.NOVALUE`.	integer value (pixels)
		Special values:
		`JCTblEnum.VARIABLE`
		`JCTblEnum.AS_IS`
		`JCTblEnum.VARIABLE_ESTIMATE`

PixelWidth	Column width in pixels. This property controls the width unless set to `JCTblEnum.NOVALUE`.	integer value (pixels)
		Special values:
		`JCTblEnum.VARIABLE`
		`JCTblEnum.AS_IS`
		`JCTblEnum.VARIABLE_ESTIMATE`

PopupMenuEnabled	Determines whether or not to display the table pop menu.	boolean value default: false for LiveTable and JCTableComponent default: true for JBdbTable, VCdbTable and DSdbTable
RepeatBackgroundColors	Repeat pattern for background colors.	array of colors
RepeatForegroundColors	Repeat pattern for foreground colors.	array of colors
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.	`JCTblEnum.PLACE_LEFT` `JCTblEnum.PLACE_RIGHT`
TopRow	Indicates the non-frozen row at least partially visible at the top of the table.	integer: row number
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 controlling user interaction

Name	Description	Values/Examples
ActionKey	Sets the key and modifier to initiate an action on the table.	`JCTblEnum.ACTION_COPY` `JCTblEnum.ACTION_PASTE`
AdvancedEditorRenderers	Determines if the advanced field editors and renderers are used.	boolean (default: true)

AllowCellResize	The `AllowCellResize` property specifies whether an end user is able to resize rows and columns.	`JCTblEnum.RESIZE_ALL*`
		`JCTblEnum.RESIZE_NONE`
		`JCTblEnum.RESIZE_COLUMNS`
		`JCTblEnum.RESIZE_ROWS`

AutoScroll	Specifies the table's scrolling behavior when user selection or dragging moves past the bounds of the table.	`JCTblEnum.AUTO_SCROLL_NONE*`
		`JCTblEnum.AUTO_SCROLL_ROW`
		`JCTblEnum.AUTO_SCROLL_COLUMN`
		`JCTblEnum.AUTO_SCROLL_BOTH`

CellEditor	Retrieves the component associated with the current editor. Returns null value if no cell selected.	Based on type Non–advanced: Boolean – BooleanCellEditor Date – DateCellEditor Double – DoubleCellEditor Integer – IntegerCellEditor JCString – JCStringCellEditor String – TextCellEditor
CellRenderer		Based on type Non–advanced: Image – ImageCellRenderer JCString – JCStringCellRenderer String – String CellRenderer Object – StringCellRenderer
CellTrigger	Maps user events into cell label actions.	none
ColumnTrigger	Sets the value of the `ColumnTrigger` property. The `ColumnTrigger` property maps user events into column label actions. The `ColumnTrigger` property is made up of a number of `LabelTrigger` objects.	Requires Event and Action parameters: **Events include:** Event.SHIFT_MASK Event.CTRL_MASK Event.ALT_MASK Event.META_MASK **Actions are:** LabelTrigger.SORT LabelTrigger.DRAG Under JDK 1.1, you can specify a mouse button for the Event: InputEvent.Button1_MASK InputEvent.Button2_MASK InputEvent.Button3_MASK e.g. `setColumnTrigger(Event.SHIFT_MASK, LabelTrigger.SORT);`
Component	AWT components in individual cells.	row and column index, component

Properties controlling user interaction

Name	Description	Values/Examples
DataSource	Specifies the table's data source (any object that implements the `TableData` interface).	data source object
DoubleBuffer	Determines whether double buffering is used.	boolean value (default:false)
Editable	Editable attribute for individual cells, `JCTblEnum` values, or `JCCellRange` ranges. Contingent on whether the data source is editable. Useful for disabling editing for a single cell.	row and column + boolean value (default: true) e.g., `setEditable(3, 2, false);`

EditHeightPolicy	Vertical sizing policy for cell editors.	`JCTblEnum.EDIT_SIZE_TO_CELL`
		`JCTblEnum.EDIT_ENSURE_MINIMUM_SIZE`
		`JCTblEnum.EDIT_ENSURE_PREFERRED_SIZE`

EditWidthPolicy	Horizontal sizing policy for cell editors.	`JCTblEnum.EDIT_SIZE_TO_CELL`
		`JCTblEnum.EDIT_ENSURE_MINIMUM_SIZE`
		`JCTblEnum.EDIT_ENSURE_PREFERRED_SIZE`

HorizSB	Specifies a horizontal scrollbar component other than the default provided with JClass LiveTable. Note: the specified component must implement `JCAdjustable`	e.g., `setHorizSB(java.awt.Scrollbar);`
HorizSBAttachment	Attach point for horizontal scrollbar. When set to `ATTACH_CELLS`, the scrollbar ends at the edge of the visible cells. When set to `ATTACH_SIDE`, the scrollbar is attached to the whole side of the table.	`JCTblEnum.ATTACH_CELLS*` `JCTblEnum.ATTACH_SIDE`

HorizSBDisplay	Determines when to display horizontal scrollbar.	`JCTblEnum.SBDISPLAY_ALWAYS`
		`JCTblEnum.SBDISPLAY_NEVER`
		`JCTblEnum.SBDISPLAY_AS_NEEDED*`

HorizSBOffset	Distance between the table and horizontal scrollbar in pixels.	integer: number of pixels e.g., `setHorizSBOffset(3);`
HorizSBPosition	Position of horizontal scrollbar. When set to `SBPOSITION_CELLS`, the scrollbar is attached to the visible cells. When set to `SBPOSITION_SIDE`, the scrollbar is attached to the whole side of the table.	`JCTblEnum.SBPOSITION_CELLS*` `JCTblEnum.SBPOSITION_SIDE`

JumpScroll	Determines whether the table will visually scroll smoothly or whether the display will 'jump' to display the cells scrolled to.	`JCTblEnum.JUMP_NONE*`
		`JCTblEnum.JUMP_HORIZONTAL`
		`JCTblEnum.JUMP_VERTICAL`
		`JCTblEnum.JUMP_ALL`

Mode	Sets table resources to list mode or table mode.	`JCTblEnum.MODE_LIST` `JCTblEnum.MODE_TABLE`
Multiline	Sets whether data is displayed with multiple lines.	boolean (default: false)

Properties controlling user interaction

Name	Description	Values/Examples
Repaint	Sets whether the table should be redrawn and recomputed whenever one of its properties is set.	boolean, (default: true)
ResizeByLabelsOnly	This property controls whether resizing can only be done with labels. If set to true, users can resize rows and columns only by resizing their respective labels.	boolean value (default: false)
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)
RowTrigger	Sets the value of the `RowTrigger` property at a specified index. The `RowTrigger` property is an indexed property, and *int* contains all the information necessary to map user events into column label actions. The `RowTrigger` property is made up of a number of `RowTrigger` objects.	Requires Event and Action parameters: **Events include:** Event.SHIFT_MASK Event.CTRL_MASK Event.ALT_MASK Event.META_MASK **Actions are:** LabelTrigger.SORT LabelTrigger.DRAG Under JDK 1.1, you can specify a mouse button for the Event: InputEvent.Button1_MASK InputEvent.Button2_MASK InputEvent.Button3_MASK e.g. `setRowTrigger(Event.CTRL_MASK, LabelTrigger.DRAG);`
SelectedBackground	Background color for cells that have been selected. The default is the cells' foreground color.	`Color` value: e.g., `setSelectedBackground(Color.yellow);`
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: e.g., `setSelectedForeground(Color.blue);`
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)

SelectionMode	Specifies the mode for selecting, based on cells, rows or columns.	`JCTblEnum.SELECT_CELL*`
		`JCTblEnum.SELECT_ROW`
		`JCTblEnum.SELECT_COLUMN`

Properties controlling user interaction

Name	Description	Values/Examples

SelectionPolicy	Sets the type of allowable selection.	`JCTblEnum.SELECT_NONE`
		`JCTblEnum.SELECT_SINGLE`
		`JCTblEnum.SELECT_RANGE`
		`JCTblEnum.SELECT_MULTIRANGE`

SortSeries	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)
Spans	Array of currently-spanned ranges of cells or labels.	array

StringCase	String case (upper/lower/as-is) for cells, `JCTblEnum` values, or `JCCellRange` ranges.	`JCTblEnum.CASE_LOWER`
		`JCTblEnum.CASE_AS_IS`
		`JCTblEnum.CASE_UPPER`

TrackCursor	Determines whether the cursor changes.	boolean (default: true)
TrackJCStringURL	Determines whether URLs are detected when JCStrings are in the table.	boolean (default: true)
Traversable	Allows traversal of individual cells.	boolean (default: true)
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)
Userdata	Sets a user data object that the application can attach (to what?).	null for all cells
VariableEstimateCount	Sets the number of cells to use in estimating variable pixel calculations.	default: JCTblEnum.ALL
VertSB	Specifies a vertical scrollbar component other than the default provided with JClass LiveTable. Note: the specified component must implement `JCAdjustable`	`e.g., setVertSB(component);`
VertSBAttachment	Attach point for vertical scrollbar. When set to `ATTACH_CELLS`, the scrollbar ends at the edge of the visible cells. When set to `ATTACH_SIDE`, the scrollbar is attached to the whole side of the table.	`JCTblEnum.ATTACH_CELLS` `JCTblEnum.ATTACH_SIDE`

VertSBDisplay	Determines when to display vertical scrollbar.	`JCTblEnum.SBDISPLAY_ALWAYS`
		`JCTblEnum.SBDISPLAY_NEVER`
		`JCTblEnum.SBDISPLAY_AS_NEEDED`

VertSBOffset	Distance between the table and vertical scrollbar.	integer: number of pixels e.g., `setVertSBOffset(4);`

Properties controlling user interaction

Name	Description	Values/Examples
VertSBPosition	Position of vertical scrollbar. When set to SBPOSITION_CELLS, the scrollbar is attached to the visible cells. When set to SBPOSITION_SIDE, the scrollbar is attached to the whole side of the table.	JCTblEnum.SBPOSITION_CELLS JCTblEnum.SBPOSITION_SIDE

B.2 Properties of jclass.table3.LiveTable

Name	Description
about	Displays component version and contact information
advancedEditorRenderers	Determines if the advanced field editors and renderers are used
allowCellResize	Determines whether end-user can resize cells at run-time
appearance	Specifies cell/label fonts and foreground/background colors
autoScroll	Determines whether table scrolls during selection/traversal
background	Specifies component background color
cellBorderType	Specifies cell/label border types
cellBorderWidth	Specifies width of cell/label borders
cellSize	Specifies row heights and column widths
columnLabelBorderType	Specifies column label border types
columnLabelDisplay	Determines whether to display column labels
columnLabelSort	Determines whether end-user can sort table by clicking column labels
data	Specifies table data, data source, and row/column labels
displayClipArrows	Determines whether arrows display when cell contents are clipped
doubleBuffer	Determines whether to double-buffer table repaints
editHeightPolicy	Determines height control of cell editing components
editState	Determines whether cells are traversable and editable
editWidthPolicy	Determines width control of cell editing components
focusRectColor	Specifies color of line drawn around current cell
foreground	Specifies component foreground color
frameBorderType	Specifies frame border type
frameBorderWidth	Specifies frame border width
frozenCellLayout	Determines the position of frozen rows/columns
frozenColumns	Specifies the number of frozen columns
frozenRows	Specifies the number of frozen rows
horizSBDisplay	Determines when to display horizontal scrollbar
jumpScroll	Determines whether to scroll smoothly or jump by whole row/column

Name	Description
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 to display table popup menu
printSettings	Sets the pop menu printing options
resizeByLabelsOnly	Determines whether end-user can resize cells by labels only
resizeEven	Determines whether end-user resizing affects all rows/columns
rowLabelBorderType	Specifies row label border types
rowLabelDisplay	Determines whether to display row labels
selectedBackground	Specifies the background (highlight) color of selected cells
selectedForeground	Specifies the foreground (highlight) color of selected cells
selectIncludeLabels	Determines whether selection includes row/column labels
sBLayout	Determines the space between scrollbars and cells
selectionPolicy	Determines type of cell selection allowed
spanningCells	Specifies cell ranges to treat as spanned cells
swingDataModel	Sets the table's data source to use a specified Swing TableModel object, instead of using the `data` property
textPositioning	Specifies cell/label text alignments
topRow	Specifies first row displayed on screen
trackCursor	Determines whether mouse pointer is tracked over the table
traverseCycle	Determines whether traversal can cycle to opposite side of table
vertSBDisplay	Determines when to display vertical scrollbar

B.3 Properties of jclass.table3.db.jbuilder.JBdbTable

Name	Description
about	Displays data aware component version and contact information
advancedEditorRenderers	Determines whether to use advanced field editor and renderers
appearance	Specifies column fonts and foreground/background colors
allowCellResize	Determines whether end-user can resize cells at run-time
autoScroll	Determines whether table scrolls during selection/traversal
background	Specifies component background color
cellBorderType	Specifies cell/label border types
cellBorderWidth	Specifies width of cell/label borders
columnLabelBorderType	Specifies column label border types

Name	Description
columnLabelDisplay	Determines whether to display column labels
columnLabelSort	Determines whether end-user can sort table by clicking column labels
dataSet	Specifies table data source
displayClipArrows	Determines whether arrows display when cell contents are clipped
doubleBuffer	Determines whether to double-buffer table repaints
editHeightPolicy	Determines height control of cell editing components
editState	Determines whether cells are traversable and editable
editWidthPolicy	Determines width control of cell editing components
focusRectColor	Specifies color of line drawn around current cell
foreground	Specifies component foreground color
frameBorderType	Specifies frame border type
frameBorderWidth	Specifies frame border width
frozenCellLayout	Determines the position of frozen columns
frozenColumns	Specifies the number of frozen columns
horizSBDisplay	Determines when to display horizontal scrollbar
jumpScroll	Determines whether to scroll smoothly or jump by whole row/column
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 to display table popup menu
printSettings	Configures the popup menu printing options
resizeByLabelsOnly	Determines whether end-user can resize cells by labels only
resizeEven	Determines whether end-user resizing affects all rows/columns
rowLabelBorderType	Specifies row label border types
rowLabelDisplay	Determines whether to display row labels
selectedBackground	Specifies the background (highlight) color of selected cells
selectedForeground	Specifies the foreground (highlight) color of selected cells
selectIncludeLabels	Determines whether selection includes row/column labels
sBLayout	Determines the space between scrollbars and cells
selectionPolicy	Determines type of cell selection allowed
textPositioning	Specifies cell/label text alignments
trackCursor	Determines whether mouse pointer is tracked over the table
useDataSourceEditable	Determines whether the editable column state is defined by the data source or the table
vertSBDisplay	Determines when to display vertical scrollbar

B.4 Properties of jclass.table3.db.vcafe.VCdbTable

Property	Description
about	Displays data aware component version and contact information
advancedEditorRenderers	Determines whether to use advanced field editor and renderers
appearance	Specifies column fonts and foreground/background colors
allowCellResize	Determines whether end-user can resize cells at run-time
autoScroll	Determines whether table scrolls during selection/traversal
background	Specifies component background color
cellBorderType	Specifies cell/label border types
cellBorderWidth	Specifies width of cell/label borders
columnLabelBorderType	Specifies column label border types
columnLabelDisplay	Determines whether to display column labels
columnLabelSort	Determines whether end-user can sort table by clicking column labels
dataBinding	Specifies the table data source
displayClipArrows	Determines whether arrows display when cell contents are clipped
doubleBuffer	Determines whether to double-buffer table repaints
editHeightPolicy	Determines height control of cell editing components
editState	Determines whether cells are traversable and editable
editWidthPolicy	Determines width control of cell editing components
focusRectColor	Specifies color of line drawn around current cell
foreground	Specifies component foreground color
frameBorderType	Specifies frame border type
frameBorderWidth	Specifies frame border width
frozenCellLayout	Determines the position of frozen columns
frozenColumns	Specifies the number of frozen columns
horizSBDisplay	Determines when to display horizontal scrollbar
jumpScroll	Determines whether to scroll smoothly or jump by whole row/column
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 to display table popup menu
printSettings	Configures the popup menu printing options
resizeByLabelsOnly	Determines whether end-user can resize cells by labels only
resizeEven	Determines whether end-user resizing affects all rows/columns
rowLabelBorderType	Specifies row label border types

Property	Description
rowLabelDisplay	Determines whether to display row labels
selectedBackground	Specifies the background (highlight) color of selected cells
selectedForeground	Specifies the foreground (highlight) color of selected cells
selectIncludeLabels	Determines whether selection includes row/column labels
sBLayout	Determines the space between scrollbars and cells
selectionPolicy	Determines type of cell selection allowed
textPositioning	Specifies cell/label text alignments
trackCursor	Determines whether mouse pointer is tracked over the table
useDataSourceEditable	Deteremines whether the editable column state is defined by the data source or the table
vertSBDisplay	Determines when to display vertical scrollbar

B.5 Properties of jclass.table3.db.datasource.DSdbTable

Property	Description
about	Displays data aware component version and contact information
advancedEditorRenderers	Determines whether to use advanced field editor and renderers
appearance	Specifies column fonts and foreground/background colors
allowCellResize	Determines whether end-user can resize cells at run-time
autoScroll	Determines whether table scrolls during selection/traversal
background	Specifies component background color
cellBorderType	Specifies cell/label border types
cellBorderWidth	Specifies width of cell/label borders
columnLabelBorderType	Specifies column label border types
columnLabelDisplay	Determines whether to display column labels
columnLabelSort	Determines whether end-user can sort table by clicking column labels
dataBinding	Specifies table data source
displayClipArrows	Determines whether arrows display when cell contents are clipped
doubleBuffer	Determines whether to double-buffer table repaints
editHeightPolicy	Determines height control of cell editing components
editState	Determines whether cells are traversable and editable
editWidthPolicy	Determines width control of cell editing components
focusRectColor	Specifies color of line drawn around current cell
foreground	Specifies component foreground color
frameBorderType	Specifies frame border type
frameBorderWidth	Specifies frame border width
frozenCellLayout	Determines the position of frozen columns

Property	Description
frozenColumns	Specifies the number of frozen columns
horizSBDisplay	Determines when to display horizontal scrollbar
jumpScroll	Determines whether to scroll smoothly or jump by whole row/column
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 to display table popup menu
printSettings	Configures the popup menu printing options
resizeByLabelsOnly	Determines whether end-user can resize cells by labels only
resizeEven	Determines whether end-user resizing affects all rows/columns
rowLabelBorderType	Specifies row label border types
rowLabelDisplay	Determines whether to display row labels
selectedBackground	Specifies the background (highlight) color of selected cells
selectedForeground	Specifies the foreground (highlight) color of selected cells
selectIncludeLabels	Determines whether selection includes row/column labels
sBLayout	Determines the space between scrollbars and cells
selectionPolicy	Determines type of cell selection allowed
textPositioning	Specifies cell/label text alignments
trackCursor	Determines whether mouse pointer is tracked over the table
useDatasourceEditable	Deteremines whether the editable column state is defined by the data source or the table
vertSBDisplay	Determines when to display vertical scrollbar

C

Moving from JClass LiveTable 2.x to 3.x

C.1 Overview

We have made significant changes and improvements to JClass LiveTable 3.x from LiveTable 2.x. These changes range from fundamental operations to new method names. These changes include:

- A new data source mechanism where data is stored in an external object; see Chapter 4, Working with Table Data.
- A new way of rendering and editing cell data; see Chapter 5, Displaying and Editing Cells.
- Property name changes to clarify the property functions.
- Added features.
- New JavaBeans (`LiveTable`) and Bean property editors; see Chapter 9, JClass LiveTable Beans and IDEs.
- A compatibility class to allow you to use your JClass LiveTable 2.x code in the new LiveTable 3.x package (see Using the Transitional JCTable Class below).

The major differences between JClass LiveTable 2.x and 3.x are outlined in What's Different, below.

C.2 What's New

JClass LiveTable 3.x contains some new features that are not available in JClass LiveTable 2.x. The following is a brief explanation of the new features:

Control Label Display

By default, row and column labels are not displayed in JClass LiveTable 3.x. To display row and column labels, set the `RowLabelDisplay` and `ColumnLabelDisplay` properties to `true`.

Control Editor Component Display

LiveTable 3.x contains methods that allow you to control how cell editors behave with respect to the cells in which they are displayed. The `editHeightPolicy` and `editWidthPolicy` properties determine how the editor fits according to the height and width of the cell, respectively.

Control Focus Rectangle Color

JClass LiveTable 3.x gives you control over the color of the focus rectangle using the `setFocusRectColor()` method. This method can take any of the 13 AWT color constants as its value.

Customizable Scrollbars

You can now implement the `JCAdjustable` interface to attach your own scrollbars to tables.

New Resize Behavior

You can set LiveTable 3.x to make it possible for users to resize rows and columns only by dragging the borders between row and column labels. They will not be able to resize the rows and columns by dragging individual cell borders. This behavior is similar to some popular grid-based software currently available.

In addition, you can set the table so that when a user resizes a column or row, all of the rows or columns in the table resize to the same width or height. To do so, set the `ResizeEven` property to `true`.

New Selection Behavior

Two new properties have been added to allow more detailed control of cell selection in JClass LiveTable applications and applets (see Chapter 6, Programming User Interactivity for details):

`SelectIncludeLabels` specifies whether the label is included in the selection when a user clicks on a row or column label to select the row or column.

`SelectionMode` allows you to control whether selections occur based on rows, columns, or cells.

New Traversal Behavior

A new property—`TraverseCycle`—has been included with JClass LiveTable 3.x to set traversals to wrap around the table to the other side. For example, if the user

traverses to the last column in the table, then traverses right once more, the current cell focus will be back at the beginning of the table on the next row down. The property takes a `boolean` value.

Column and Row Event Triggers

In JClass LiveTable 3.x, you can set triggers on column and row labels to respond to key+mouse-click combinations (or under JDK 1.1, alternate mouse button clicks) to perform sorting and dragging actions. You can specify a key-click combination to enable dragging on rows or columns, or sorting on columns. These actions are specified by the `RowTrigger` and `ColumnTrigger` properties (see Chapter 6, Programming User Interactivity for more information).

Cell Editors/Renderers

The way cells are drawn and edited in JClass LiveTable has changed substantially. Please refer to Chapter 5, Displaying and Editing Cells for full details.

Table Data Sources

The core functionality of JClass LiveTable's data handling has changed. LiveTable now uses a data source stored as a separate object. For full details, please see Chapter 4, Working with Table Data.

C.3 What's Removed

The following features have been removed but are still available as part of the transitional `JCTable` class:

- `DataType`, previously in JClass LiveTable 2.x, is now determined by data source in LiveTable 3.x. `DataType` is still available as part of the `JCTable` compatibility class (see below).
- Previously, JClass LiveTable 2.x used `JCValueEvent`, `JCLabelValueListener`, and `JCCellValueListener` to minimize the data stored in LiveTable. This is no longer necessary with LiveTable 3.x and these have been removed.
- `JCValidateCellListener` has no equivalent in LiveTable 3.x. Data validation is provided by `CellData` objects.

C.4 What's Different

Some of the features of JClass LiveTable 3.x have changed from their implementation in LiveTable 2.x. Additionally, some method names and values have changed. The following lists these changes:

- The package name has changed from `jclass.table` to `jclass.table3`. You will need to change `import` statements to use the new package name.
- JClass LiveTable 3.x does not suppress row/column labels if there is no data. The methods `setRowLabelVisible()` and `setColumnLabelVisible()` take `boolean` values to display or hide labels. The default value is `true`.

- The `ClipArrows` (`DisplayClipArrows` in LiveTable 2.x) property takes a different type of value. Previously, it required a `boolean`, and now it requires a `JCTblEnum`. value of one of `CLIP_ARROWS_BOTH`, `CLIP_ARROWS_RIGHT`, `CLIP_ARROWS_LEFT`, or `CLIP_ARROWS_NONE`.
- The new version uses a different sorting mechanism from the previous version: sorting must be implemented using the JClass LiveTable 3.x methods. In JClass LiveTable 3.x, sorting a column does not automatically select it. The `JCSortInterface` from JClass LiveTable 2.x is no longer needed. Use the new `JCSortable` interface instead.
- Different events (see Chapter 7, Events and Listeners for details).
- The following table outlines property name changes[1] from JClass LiveTable 2.x to LiveTable 3.x:

LiveTable 2.x Property Name	LiveTable 3.x Property Name
DisplayClipArrows	ClipArrows
BorderSides	CellBorderSides
BorderType	CellBorderType
MultiLine	Multiline
ShadowThickness	CellBorderWidth
FrameShadowThickness	FrameBorderWidth

- Subclassing the default editors or renderers may be incompatible. In particular, overriding any AWT event–handling method when subclassing with LiveTable 3.x is not guaranteed to work with LiveTable 2.x.
- In version 3.x, calling JCTable.setCell now validates. If you set validatePolicy to anything other than NEVER, calling setCell initiates a validateCell event.
- The behaviour of `SelectedBackground` and `SelectedForeground` now make cell selections easier for the user. By default, selected cells' foreground is Table's default background color (gray), and the selected cells' background is the default foreground color (black). When either or both of these properties are set to `null`, selected cells do not change in color.

C.5 Using the Transitional JCTable Class

JClass LiveTable 3.x uses a compatibility class called `JCTable` that attempts to make using JClass LiveTable 2.x code seamless within JClass LiveTable 3.x. Similarly, JClass LiveTable 3.x contains a compatibility Bean called `JCTableComponent`. To port LiveTable 2.x applications to use the `JCTable` compatibility class:

1. These name changes are handled by the JCTable compatibility class if you are running a LiveTable 2.x program in LiveTable 3.x. This list is a quick-reference to help you use the new Table class.

- Change import statements to use the new package name, `jclass.table3`, instead of `jclass.table`.
- Change code to use the transitional `JCTable` class instead of `Table`.

Some of the JClass LiveTable 2.x functionality cannot be duplicated.

- Various fixes to the listeners may result in extra/different events from JClass LiveTable 2.x. For example, all actions that generate an `enterCellBegin` call will generate an `enterCellEnd` call. See the Events chapter for more info.
- JClass LiveTable 3.x does not support the `JCValidateCellListener`, `JCLabelValueListener` and `JCCellValueListeners`. These listeners are available in the `JCTable` transitional class but you will need to make changes to your code when migrating to the new version. In JClass LiveTable 3.x, the `JCCellDisplayListener` class allows you to modify the display string at run-time. Unlike the `JCLabelValueListener` and `JCCellValueListener`, `JCCellDisplayListener` does not provide methods to save the data. There is no equivalent listener to `JCValidateCellListener` in JClass LiveTable 3.x. Custom `CellData` data types replace this functionality.
- Since LiveTable 3.x uses `JCSortable` instead of LiveTable 2.x's `JCSortInterface`, user-defined sorting will not be compatible. Methods like the following:

  ```
  boolean compare(Object o1, Object o2);
  ```

  will need to be changed to:

  ```
  long compare(Object o1, Object o2)
  ```

  where the `return` value should be:

  `JCSortable.LESS_THAN` if `o1 < o2`
  `JCSortable.GREATHER_THAN` if `o1 > o2`, and
  `JCSortable.EQUAL` if `o1 == o2`.
- JClass LiveTable 3.x does not support user data nor the `getDatatype()` or `setDatatype()` methods. These methods are available in the transitional class, `JCTable`.

We are interested in hearing about additional differences and problems our users may have with our JClass LiveTable 2.x transitional class. See Product Feedback and Annoucements, on page 15 for feedback information.

D

JCString Properties

Most JClass LiveTable components support a rich text format called "JCString", which allows a mixture of hypertext, images and text within JClass LiveTable components. Text can also appear in a variety of colors, fonts and styles, including underline and strikeout.

The following section describes the types of JCString properties available, and provides examples of their use.

Alignment

If a cell contains an image, the line height can, in some cases, be greater than the height of the text. Text can be aligned vertically relative to the image using the `ALIGN` property. Valid values include `TOP`, `BOTTOM` and `MIDDLE`. The following example uses all three possible `ALIGN` values:

```
([IMAGE=smiley.gif][ALIGN=TOP]top
[IMAGE=smiley.gif][ALIGN=MIDDLE]middle
[IMAGE=smiley.gif][ALIGN=bottom]bottom)
```

Color

Different text colors can be specified by using the `COLOR` property. The JCString shown below displays text using red, green and blue colors.

```
([COLOR=red]Red, [COLOR=green]Green, [COLOR=blue]Blue,
[DEFAULT_COLOR]Default)
```

In addition to these colors, any color referenced in Colors and Fonts can be used, including RGB color values.

Note: The property `DEFAULT_COLOR` resets the text color in the rest of the table to the browser's regular text color.

Fonts

Different fonts can be specified within a single cell or label by using the FONT property. The following JCString example displays text using a variety of fonts and font styles.

```
([FONT=timesroman-plain-20]TimesRoman-20,
 [FONT=timesroman-bold-12]TimesRoman-12 bold,
 [DEFAULT_FONT]Default)
```

Note: The property DEFAULT_FONT resets the fonts in the rest of the table to the browser's regular font.

Horizontal and Vertical Spacing

Vertical and horizontal spacing can be modified by using the VERT_SPACE and HORIZ_SPACE tags. VERT_SPACE offsets the current line by a number of pixels, and HORIZ_SPACE offsets the line from the margin by a set number of pixels.

The example below makes use of the HORIZ_SPACE and VERT_SPACE tags.

```
([VERT_SPACE=10]Vertical offset=10
 [HORIZ_SPACE=10] Horizontal offset=10)
```

Hypertext

Hypertext links can be specified within a cell. The link appears underlined, and the browser display will show the target URL when the mouse cursor passes over the linked text. Hypertext uses the HREF property, and trackCursorPosition must be set to true in order for the link to work. The example below links to KL Group's home page:

```
(Click [HREF=http://www.klg.com]here[/HREF] for tech support)
```

Images

Images can be specified in a cell by using the IMAGE property. A URL or file name must be provided. If a relative path is given, the document base for the page is assumed.

The example below mixes an image with text.

```
(Tech Support: [IMAGE=http://www.klg.com/images/technical.gif])
```

Reset

The RESET property resets the font and color to the default value used by the browser. The following example changes the COLOR and FONT values back to the default:

```
([COLOR=green][FONT=timesroman-plain-20]Big text[RESET]Regular Text)
```

Strikethrough Text

Text can be crossed-out using the `STRIKEOUT` property. The following incorporates text that has been struck through:

```
(This text is [ST]crossed-out[/ST].)
```

Underlined Text

Text can be underlined using the JCString `UNDERLINE` property. The following example incorporates underlined lined text:

```
(This text is [UL]underlined[/UL].)
```

E

Colors and Fonts

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

E.1 Colorname Values

The following lists all the colornames that can be used within Java programs. The majority of these colors will appear the same (or similar) across different computing platforms.

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

E.2 RGB Color Values

The following lists all the main RGB color values that can be used within JClass LiveTable. RGB color values are specified as three numeric values representing the red, green and blue color components; these values are separated by dashes ("-").

The following RGB color values describe the colors available to Unix systems. It is recommended that you test these color values in a JClass program on a Windows or Macintosh system before utilizing them.

The list begins with all of the variations of white, then blacks and greys, and then describes the full color spectrum ranging from reds to violets.

Example code from an HTML file:

```
<PARAM NAME=backgroundList VALUE="(4, 5 255-255-0)">
```

RGB Value	Description
255-250-250	Snow
248-248-255	Ghost White
245-245-245	White Smoke
220-220-220	Gainsboro
255-250-240	Floral White
253-245-230	Old Lace
250-240-230	Linen
250-235-215	Antique White
255-239-213	Papaya Whip
255-235-205	Blanched Almond
255-228-196	Bisque
255-218-185	Peach Puff
255-222-173	Navajo White
255-228-181	Moccasin
255 248-220	Cornsilk
255-255-240	Ivory
255-250-205	Lemon Chiffon
255-245-238	Seashell
240-255-240	Honeydew
245-255-250	Mint Cream
240-255-255	Azure
240-248-255	Alice Blue
230-230-250	Lavender
255-240-245	Lavender Blush
255-228-225	Misty Rose
255-255-255	White
0-0-0	Black

RGB Value	Description
47-79-79	Dark Slate Grey
105-105-105	Dim Gray
112- 128-144	Slate Grey
119- 136-153	Light Slate Grey
190- 190-190	Grey
211- 211-211	Light Gray
25-25-112	Midnight Blue
0-0-128	Navy Blue
100- 149 237	Cornflower Blue
72-61-139	Dark Slate Blue
106-90-205	Slate Blue
123- 104 238	Medium Slate Blue
132-112- 255	Light Slate Blue
0-0-205	Medium Blue
65-105-225	Royal Blue
0-0-255	Blue
30-144-255	Dodger Blue
0-19 -255	Deep Sky Blue
135-206-235	Sky Blue
135-206-250	Light Sky Blue
70-130-180	Steel Blue
176-196- 222	Light Steel Blue
173-216-230	Light Blue
176-224-230	Powder Blue
175-238-238	Pale Turquoise
0-206-209	Dark Turquoise
72-209-204	Medium Turquoise
64-224-208	Turquoise
0-255-255	Cyan
224-255-255	Light Cyan
95-158-160	Cadet Blue
102-205-170	Medium Aquamarine
127-255-212	Aquamarine
0-100-0	Dark Green

RGB Value	Description
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
205-133-63	Peru
222-184- 135	Burlywood
245-245-220	Beige
245-222-179	Wheat

RGB Value	Description
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
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

E.3 Fonts

There are five different font names that can be specified in any Java program. They are:

- `Courier`
- `Dialog`
- `DialogInput`
- `Helvetica`
- `TimesRoman`

Note: Font names are case-sensitive.

There are also four standard font style constants that can be used. The valid Java font style constants are:

- `bold`
- `bold+italic`
- `italic`
- `plain`

These values are strung together with dashes ("`-`") when used with the `VALUE` attribute. You must also specify a point size by adding it to other font elements. To display a text using a 12-point italic Helvetica font, use the following:

```
Helvetica-italic-12
```

All three elements (font name, font style and point size) must be used to specify a particular font display; otherwise, the default font is used instead.

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

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W