JClass Field™

Programmer's Guide

Version 6.3
for Java 2 (JDK 1.3.1 and higher)

Complete Input and Validation for Popular Data Types

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

April 2004

RefNo: PRGDE-JCFLD/630-04/2004

Warranty

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

Trademarks

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

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

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

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

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

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

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

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

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

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

Table of Contents

Preface

Introducing JClass Field

JClass Field is a set of Java components that permits the collection, validation, and display of textual, calendar, and numeric data. You can use the components of JClass Field for data entry applications. You can present a list of pre-programmed choices in a combo field or in a spin field from which users make a selection, or you can permit them to type into various fields. In the latter case, you can provide both a validation format and a "prompt" format. The validation format accepts a certain class of characters at each input position, for example, three letters followed by four numbers. The prompt format gives the user an idea of what data the field is expecting by filling the field with a generic example. The user types over the prompt text, replacing it with valid data. Using JClass Field, your applications can collect calendar, numeric, and textual information. Built-in validation methods permit you to apply various consistency checks on the information and to give the end-user visual and audible feedback when the validator detects an incorrect entry.

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

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

Feature Overview

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

- the type of text that end-users are allowed to type in by using an input validation mask
- through the use of place holder characters, representative contents for the field that the end-user can overtype
- the look of the data display and edit formats for calendar, date and time fields
- data binding to display and edit field values from a database
- the association of words with integer values in integer combo boxes and integer spin fields–a useful feature for database applications where numeric indices are used internally to denote possibly lengthy field descriptors
- field appearance attributes including border, text alignment, font, and color

- user feedback, such as an audible beep and a change of color upon entry of invalid data
- cell editability and traversability: a field may be read-only, or it may accept changes only from a list of valid values in a spin or combo field
- the display or modification of time values using date and calendar popups
- the range of acceptable values in numeric fields

JClass Field also provides several methods which:

- contain capabilities for internationalization
- allow you to return data about fields inside a container
- set an area which dynamically displays prompt text for the fields in a container

Typographical Conventions in this Manual

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

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.

Overview of the Manual

Part I –Using JClass Field – describes how to program with the JClass Field components.

Chapter 1, JClass Field Basics, should be read by all programmers learning JClass Field. It introduces the JClass Field components, and provides basic terminology and conventions used throughout the documentation.

Chapter 2, JClass Field's Properties, describes the Java Bean properties that are exposed in the Beans Development Kit (BDK) and other integrated development environment (IDE) tools.

Chapter 3, Building a Field, provides hands-on examples of creating different kinds of fields and detailed information on data binding with JClass Field.

Chapter 4, Example Code for Common Fields, contains extensive description of the examples included in the distribution.

Part II –Reference Appendices – contains additional detailed technical reference on all JClass Field properties and other reference information related to programming with JClass Field.

Appendix A, JClass Field Property Listings, lists all of the available properties in JClass Field and their default values.

Appendix B, Distributing Applets and Applications on a Web Server, provides a method of releasing your applet or application to your users.

Appendix C, Porting JClass 3.6.x Applications, shows on how to convert your code created with earlier versions of JClass Field.

Appendix D, Using JCField's Autocomplete Feature, outlines the autocomplete mechanism in `JCComboField` which may be used to simplify selecting items in a combo box.

API Reference

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

Licensing

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

Related Documents

The following is a selection of useful references to Java and Java Beans programming:

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

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

About Quest

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

Contacting Quest Software

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

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

Customer Support

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

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

You can use SupportLink to do the following:

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

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

To Contact JClass Support

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

- Your name, email address, telephone number, company name, and country

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

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

Product Feedback and Announcements

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

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

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

Part I

Using JClass Field

1

JClass Field Basics

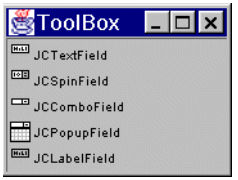
The following topics cover basic information that anyone who intends to create JClass Field objects should be familiar with. After you can recognize the basic JClass Field processes and vocabulary, you can begin using JClass Field's objects and data validators to simplify the development of your data entry applications.

1.1 Terminology

There are five basic graphical user interface (GUI) styles of visual components in JClass Field: Text, Spin, Combo, Popup and Label. Each of these styles is represented by one of Field's standard Beans: `JCTextField`, `JCSpinField`, `JCComboField`, `JCPopupField`, and `JCLabelField`, respectively.

One or more of the data types supported by JClass Field are handled by each of these Beans. Regardless of what data type the Bean handles, the name of the field is the same.

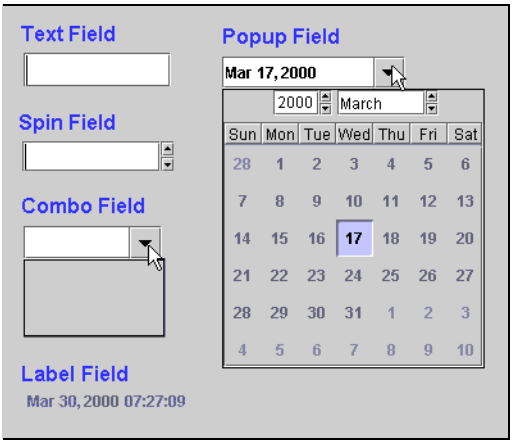
In all, there are five standard field components and five data-bound components. The following BeanBox illustration shows the standard JClass Field Bean components:



1.2 Overview of GUI Components and Field Data Types

1.2.1 JClass Field's GUI components

Your choice of the type of visual object will be based in part on the type of validation you wish to perform on the data. The following image shows the five types of visual objects in JClass Field:



- A *text field* is a visual component which displays a single piece of alphanumeric data entered by the user. The contents of the field and the position of the cursor within the field are under program control. You can validate all types of data in this field.
- A *spin field* allows the user to spin through a range of values. The range of values shown in the spin field is determined by one of two mechanisms.

  The upper and lower limits on the range can be determined by predefining *min* and *max* values. The intermediate values are controlled by the *increment* property.

  A *pick list* offers all the possible values that the spin field can contain. This has precedence over the increment method.

Only one value at a time is shown in the spin field. Up and down-arrow buttons are used to change the value by "spinning." The spin field has access to the same validator functions as the text field.

- A *combo field* contains a text field and a drop-down list in combination. If the number of items available for selection is small enough, the drop-down list can show all possible pre-assigned values at once. Otherwise, a vertical scrollbar is used to position the items in the drop-down list window. You can also allow users to enter data not contained in the list.
- A *popup field* is used to display a monthly calendar in rectangular format where the user can select the month and year using spin fields, and the day of the month from the resulting calendar. If you are using a time or date/time validator, the user can set the hour, minute, second, and meridiem using spin fields.
- A *label field* is similar to a text field in that it is used to display a single piece of alphanumeric data; however, it functions as a heading or label because its value cannot be changed by the user and it is displayed as a label. It can be bound to a database and present dynamic data.

1.2.2 Data Types Handled by JClass Field

The following table lists the data types handled by JClass Field:

JCField DataType	Stored As
Byte	`java.lang.Byte`
Integer	`java.lang.Integer`
Short	`java.lang.Short`
Long	`java.lang.Long`
Float	`java.lang.Float`
Double	`java.lang.Double`
BigDecimal	`java.math.BigDecimal`
String	`java.lang.String`
Calendar	`java.util.Calendar`

Date
SqlDate
	`java.util.Date`
	or
	`java.sql.Date`

SqlTime	`java.sql.Time`
SqlTimeStamp	`java.sql.TimeStamp`
JCIPAddress	`com.klg.jclass.util.JCIPAddress`

Note that the Date data type is stored as two different representations. You can select either type at design time, or you can set the type programmatically.

The SQL data types are usually used when binding to a database.

1.2.3 GUI Component Support for Data Types

The following table shows the commonly used data types and GUI component combinations:

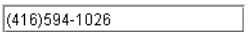
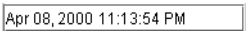
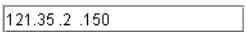
Data Type	Text Field	Spin Field	Combo Field	Popup Field	Label Field
java.lang.Byte	✔	✔	✔		✔
java.lang.Short	✔	✔	✔		✔
java.lang.Integer	✔	✔	✔		✔
java.lang.Long	✔	✔	✔		✔
java.lang.Float	✔	✔	✔		✔
java.lang.Double	✔	✔	✔		✔
java.math.BigDecimal	✔	✔	✔		✔
java.lang.String	✔	✔	✔		✔
java.util.Calendar	✔	✔		✔	✔
java.util.Date	✔	✔		✔	✔
java.sql.Date	✔	✔		✔	✔
java.sql.Time	✔	✔			✔
java.sql.Timestamp	✔	✔			✔
com.klg.jclass.util.JCIPAddress	✔	✔	✔		✔

In actuality, you can use any data type with any GUI component; however, some combinations may not be as useful as others. For example, a popup field that uses the byte data type will not display a popup.

1.3 JClass Field Components and Data Types

This section provides a brief description of the standard JClass Field components combined with each appropriate data type, and the databound components.

1.3.1 JCTextField

Data Type(s)	Description
java.lang.Byte java.lang.Integer java.lang.Short java.lang.Long java.lang.Double java.lang.Float java.math.BigDecimal	Numeric values in a JCTextField are displayed while in edit mode using an *edit pattern*, and displayed in a possibly different format after editing is complete. The field may be *editable*, in which case values may be typed in, or it may be set to simply display a value supplied by your program. Example of an integer type in a text field: 
java.lang.String	The String type in a text component permits entry of data that can be validated using a mask. For instance, the field may be for phone numbers, whose formats follow a fixed rule. Optionally, place-holder characters may be supplied to indicate the type of data that the field is programmed to accept. The user types over these characters using them as a guide when typing in valid data. Example of a String type in a text field: 
java.util.Calendar java.util.Date java.sql.Date java.sql.Time java.sql.Timestamp	The date and time data types in a JCTextField presents the date in a locale-specific format. The time text field may be programmed to accept any of a set of standard time formats, such as “h:mm:ss a”, which stands for colon-delimited hours, minutes, and seconds entries followed by an “a” or a “p,” standing for “AM” or “PM.” A property (maskInput) can be set so that the component insists on a pre-defined format for input, and dates cannot be entered in any other format. Example of a calendar type in a text field: 
com.klg.jclass.util. JCIPAddress	The IP address data type is created to allow validation of IP addresses in period-delimited subfields. Example of a IP address data type in a text field: 

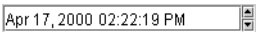
1.3.2 JCSpinField

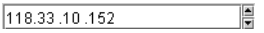
Data Type(s)	Description

`java.lang.Byte`
`java.lang.Integer`
`java.lang.Short`
	The spin field increments by integral values, between preset minimum and maximum values.
	Example of an integer type in a spin field:
	 

`java.lang.Double`
`java.lang.Float`
`java.math.BigDecimal`
	Values may be selected by using the spin arrows or by typing in the field. After a value has been entered, it can be checked to ensure that it lies in the acceptable range. Typically, the arrow buttons are be disabled when the top or bottom of the list is reached, indicating that there are no more data items in that direction.
	Example of a double type in a spin field:
	 

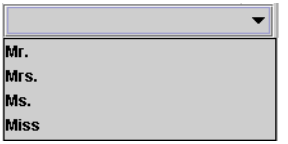
`java.lang.String`	The String data type in a `JCSpinField` is useful for providing a list of names or other Strings that can be accessed by spinning. By default the action of the spinners is set so that the data is accessible as if it were arranged in a continuous loop. Example of a String type in a spin field: 

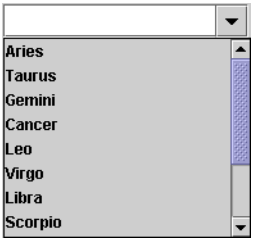
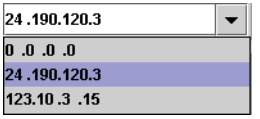
`java.util.Calendar`
`java.util.Date`
`java.sql.Date`
`java.sql.Time`
	This component permits the selection of other values by using the spin arrows in conjunction with the mouse. The subfield is selected using the mouse pointer, then the arrow buttons are used to change the value of this subfield.
	Example of a calendar type in a spin field:
	 

`com.klg.jclass.util.` `JCIPAddress`	The IP address data type is created to allow validation of IP addresses in period-delimited subfields. Example of a IP address data type in a spin field: 

1.3.3 JCComboField

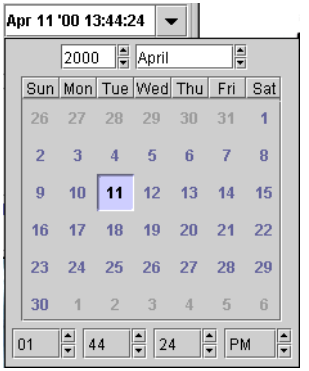
Data Type(s)	Description

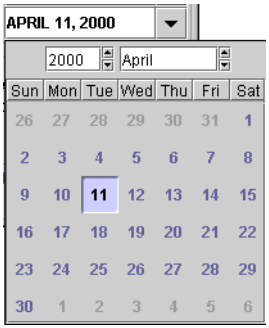
`java.lang.Byte`
`java.lang.Integer`
`java.lang.Short`
`java.lang.Long`
`java.lang.Float`
`java.lang.Double`
`java.math.BigDecimal`
	This combo field component can show choices expressed in textual form as well as numeric. No matter how the items in the combo field appear, they are associated with numeric values. In the example shown below, the item "Mr." has a value of 0, and "Mrs." has a value of 1.
	Example of an integer type in a combo field:
	 

`java.lang.String`	The String type in a combo field behaves similarly to its integer relative except that the values that appear in the field are the actual values. Example of a String type in a combo field: 
`com.klg.jclass.util.` `JCIPAddress`	The IP address data type is created to allow validation of IP addresses in period-delimited subfields. Example of a IP address data type in a combo field: 

Autocomplete Feature	Description
JClass Field's combo box autocomplete feature has three modes: ■ suggest – a drop-down list appears as soon as the end user begins typing ■ refine – the drop-down list updates itself to just those items that match what has already been typed ■ append – the first candidate that matches what has been typed appears in the text field. The completed part is highlighted See Appendix D, Using JCField's Autocomplete Feature, for a complete description of the combo box's autocomplete facility.	As an end user begins typing in a combo field with autocomplete on, those list items that match what has already been typed are presented. Pressing ENTER selects the choice currently in the text field. Another choice from the drop-down list may be selected by clicking it. 

1.3.4 JCPopupField

Data Type(s)	Description
`java.util.Calendar`	The Calendar data type displays the time and date, along with an arrow button in the popup component. Clicking on the arrow button produces a pop-down calendar with spin fields for the year, month, hour, minute, second, and meridian. When the year and month fields respond to mouse clicks by incrementing or decrementing their values each time an arrow button is pressed, the calendar display updates accordingly by showing the days of the month. The default value for all calendar components is the current date and time. Example of the calendar type in a popup field: 

Data Type(s)	Description
`java.util.Date` `java.sql.Date`	This combination is similar to the calendar type and popup field, except that there are no time spinners in this component, and it contains no time information. Month and year values are changed using the two spin boxes at the top of the calendar, and the day of the month is selected by clicking on it in the calendar. Example of the date type in a popup field: 

1.3.5 JCLabelField

Data Type(s)	Description

`java.lang.Byte`
`java.lang.Integer`
`java.lang.Short`
`java.lang.Long`
`java.lang.Double`
`java.lang.Float`
`java.math.BigDecimal`
	Numeric values in a label field can be displayed in different formats.
	Example of an integer type in a label field:
	 

`java.lang.String`	String values can be used in label fields to simulate headings or to display a constant value. Example of a String type in a label field: 

`java.util.Calendar`
`java.util.Date`
`java.sql.Date`
`java.sql.Time`
	Calendar data types in a label field can display dates and times in locale-specific and standard formats. You can also set the format to your own specifications.
	Example of a calendar type in a label field:
	 

`com.klg.jclass.util.` `JCIPAddress`	The IP address data type is created to allow validation of IP addresses in period-delimited subfields. Example of a IP address data type in a label field: 

1.3.6 Data Bound Components

JClass Field includes additional components that can be bound to a column in a JDBC or IDE-based database data source. These components are contained in separate JAR files

for each development environment. Please see the *JClass DesktopViews Installation Guide* for more information.

JCField Component	Description

`DSdbTextField (JClass DataSource)`
`JBdbTextField (Borland JBuilder)`
`DSdbSpinField (JClass DataSource)`
`JBdbSpinField (Borland JBuilder)`
`DSdbComboField (JClass DataSource)`
`JBdbComboField (Borland JBuilder)`
`DSdbPopupField (JClass DataSource)`
`JBdbPopupField (Borland JBuilder)`
`DSdbLabelField (JClass DataSource)`
`JBdbLabelField (Borland JBuilder)`
	JClass Field data bound Beans have virtually the same interactive behavior and properties as JClass Field's standard Beans. They are designed to be bound to a column in a JClass DataSource or Borland JBuilder data source component. Once bound, the data type is read from the database query results.

Note: "DSdb" components only bind with data sources included with JClass DataSource; "JBdb" components only bind with Borland JBuilder 3 data sources.

1.4 The Structure of a JClass Field Component

JClass Field components are comprised of four elements:

- a visual component: either `JCTextField`, `JCComboField`, `JCSpinField`, `JCPopupField`, or `JCLabelField`.
- the value model, which determines the data type of the component and contains the initial and current values of the field.
- the validator which supports the data type and contains properties specific to that data type.
- the InvalidInfo object, which contains additional properties that can be used by all data types and validators to direct the behavior of the field when it encounters an invalid value.

A field starts with the visual component. Each of the three objects that define the field's properties is contained within this visual component. The first object, the value model, specifies the data type of the values the field will hold. Once you declare the value model, the appropriate validator to support the data type is automatically determined. You can then declare that validator. The InvalidInfo object is automatically created when the field is declared.

You can create a field in several ways. If you want a standard field with the minimum customization, you can use the constructor that requires the data type class. For example, `JCComboField(java.util.Double.class c)`. This way, the selection of the data type automatically creates the correct value model and validator. To access the properties in

the validator, you would use the `getDataProperties()` method. If you want slightly more control over the field, you can declare the value model and validator separately from the field. See Example Programs, in Chapter 4, for an example of this code.

1.5 Validators

JClass Field validators are used to ensure, as much as possible, the integrity of the information in the field. Initially they parse the text in the field and create an object to be examined by the validator. Once the validity of the object is established, the validator formats the object into appropriate text for the field. You can use validators to enforce a standard format for data fields, and you can prevent the typing of unwanted variants, such as "three" instead of "3." For more information, refer to Section 1.5.2, The Validation Process.

JClass Field validators are available for all of the data types. Therefore, you should assign the component a validator that is appropriate for the type of information that you want the data field to contain.

Note that the `JCDateTimeValidator` supports both the `java.util.Calendar` data type and the `java.sql.Timestamp` data type. The `JCDateValidator` supports both `java.util.Date` and `java.sql.Date`. All other validators correspond exactly to the names of the data types they support.

1.5.1 Validator Functions

The following list describes some of the functions supported by the validators. For a more detailed description of the supported functions see The Validator Property, in Chapter 2.

- *display* pattern, *edit* pattern (numbers only)

  The way that numeric data is input and displayed can be modified to suit the circumstances. You can choose to display negative numbers by enclosing them in brackets, or you can add a fixed text String to the numeric field.

- *mask* (Strings only)

  This allows you to display dates using any of the standard display formats, and you can choose an input format that differs from the display format if you wish.

- *min* and *max* (numbers only)

  You can set limits on the range of numeric data the field will accept.

- a valid character list: *validChars,* an invalid character list: *invalidChars*

  There are times when you wish to restrict input to a short list of valid characters, and other times when there are only a few characters you wish to prohibit. Having both ways of setting character lists lets you choose the one that is shorter and more descriptive.

- *defaultValue*

  A field is given a default value when it is created. You might want to change this value, or you might not want a default to appear at all when the field is created. This property can also be used as a reference when a user types in invalid data.

1.5.2 The Validation Process

The following diagram shows the steps of parsing and validation that occur when a user enters text in a field. When the text is committed or the field loses focus, this process will be performed.

You can intercept the new value between the parse and validation processes. You might want to change the value a user has entered using the `JCValueModel.SetNewValue()` method, or simply monitor the user's data.

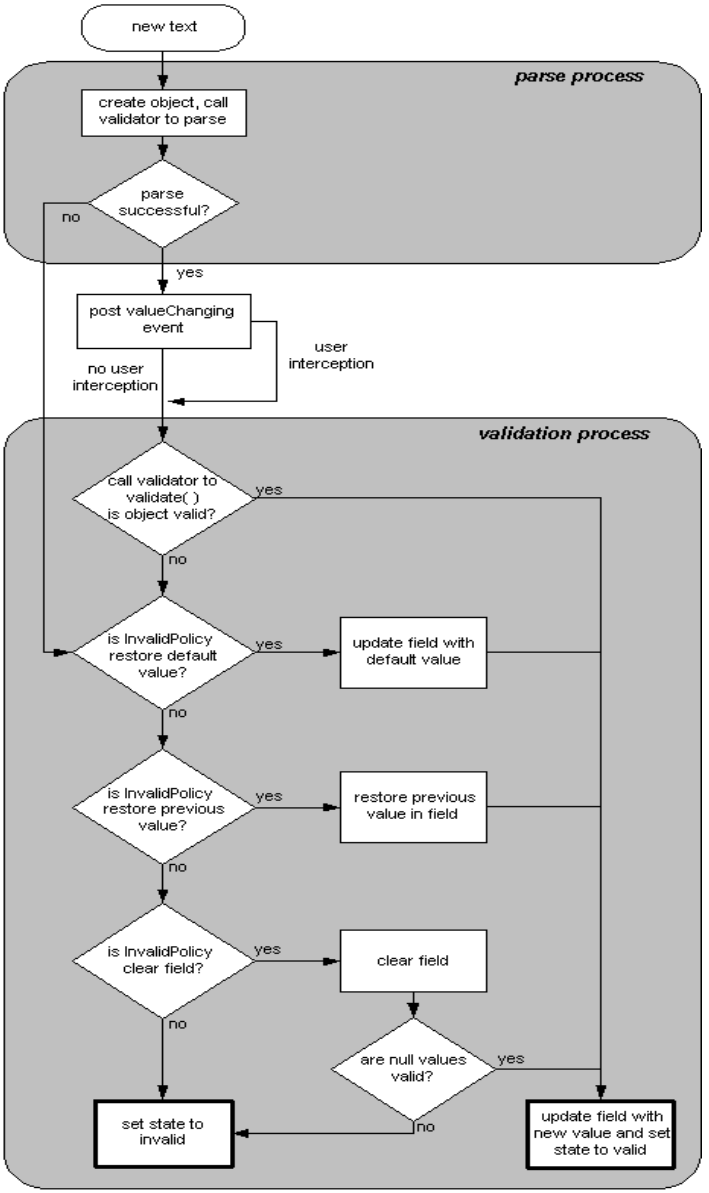


Figure 1 The validation and parsing process.

1.5.3 The useFormatting flag

A `useFormatting` flag is available for `JCBigDecimalValidator`, `JCFloatValidator`, and `JCDoubleValidator`. It is used to dictate whether or not formatting is enforced. By default, the value for this flag is `true` for `JCFloatValidator` and `JCDoubleValidator`.

This flag is set to `false` by default for the `JCBigDecimalValidator`, as formatting may result in a loss of precision for `BigDecimal` data types. This risk is not applicable to the other two properties.

1.5.4 Validators and Value Models

Each validator can be used with specific value models; the following table indicates the value model to use with the different validators.

Validator	Supported Types	Value Model Class
JCBigDecimalValidator	BigDecimal	BigDecimalValueModel
JCByteValidator	Byte	ByteValueModel
JCDateTimeValidator	java.util: Calendar, Date, GregorianCalendar java.sql: Date, Time, TimeStamp	CalendarValueModel, DateValueModel, SqlValueModel, SqlTimeStampValueModel, SqlTimeValueModel
JCDateValidator	java.util: Calendar, Date, GregorianCalendar java.sql: Date	CalendarValueModel, DateValueModel, SqlValueModel
JCDoubleValidator	Double	DoubleValueModel
JCFloatValidator	Float	FloatValueModel
JCIntegerValidator	Integer	IntegerValueModel
JCIPAddressValidator	JCIPAddress	IPAddressValueModel
JCLongValidator	Long, Byte, Integer, Short	LongValueModel
JCShortValidator	Short	ShortValueModel
JCStringValidator	String, StringBuffer	StringValueModel
JCTimeValidator	java.util: Calendar, GregorianCalendar java.sql: Time, TimeStamp	CalendarValueModel, SqlTimeValueModel, SqlTimeStampValueModel

1.6 JClass Field Inheritance Hierarchy

JClass Field's visual components are subclassed entirely from Swing. In previous versions of Field, some components were subclassed from JClass BWT. Using Swing and JClass Field, you can do everything without BWT.

The following two diagrams show the inheritance hierarchy for the JClass Field Bean components. The first diagram shows the inheritance of the basic classes in JClass Field.

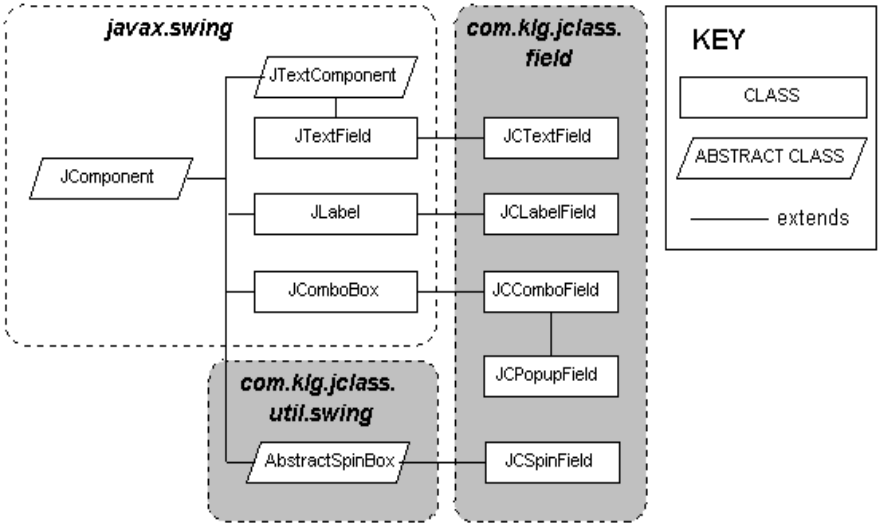


Figure 2 JClass Field's Component Classes - the basic classes.

Here is the hierarchy of classes within JClass Field:

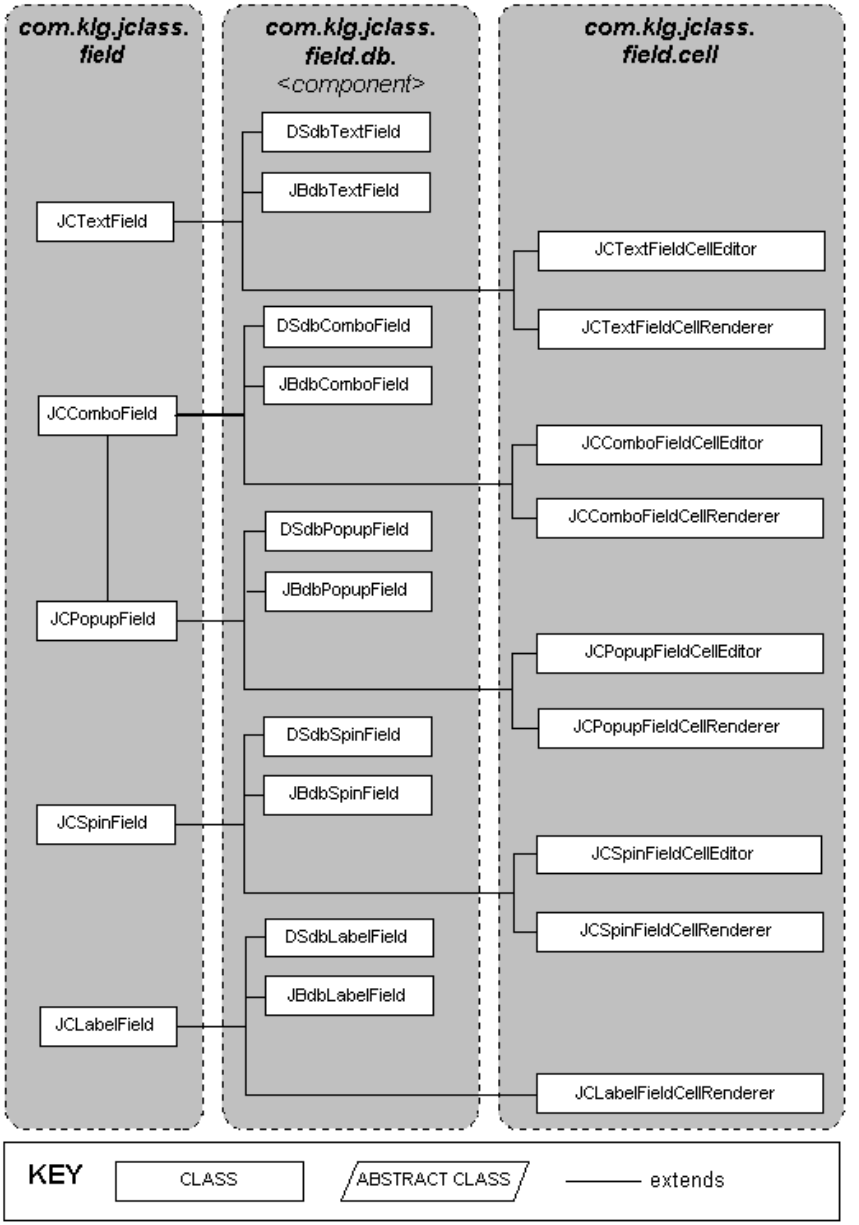


Figure 3 JClass Field's Component Classes - classes inside JClass Field.

The following diagram shows the inheritance properties for JClass Field's validators. The object at the top signifies `java.lang.Object`, from which all JCField's validators are subclassed.

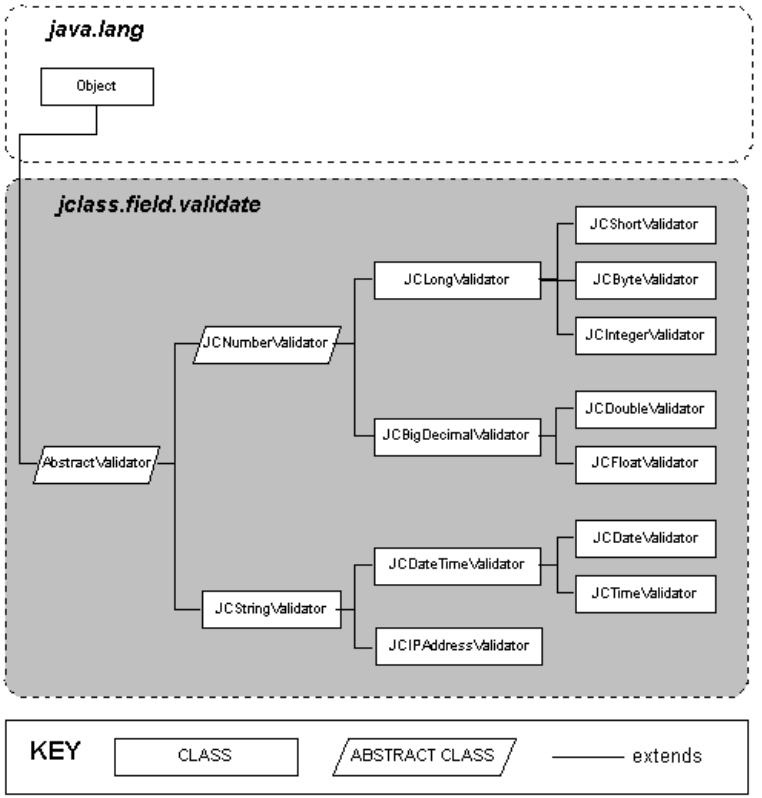


Figure 4 JClass Field's validator Classes.

1.7 Events

Events are a mechanism used to propagate state change notifications between a source object and one or more target listener objects. Events are typically used within windowing toolkits for delivering notifications of such things as mouse or keyboard actions, or other programmatically-defined actions.

Bean-compliant JClass products (like JClass Field) provide the means for an application to be notified when an event occurs through event listeners. It works as follows: if a component is acted upon by the user or from within the program, a `JCFooEvent` is fired (where "`Foo`" is the place holder for the actual event name). The `JCFooListener` (which has been registered by calling `addFooListener()` on the component) receives the

instance and enacts the action to be taken. The developer uses the `JCFooListener` to define what action or actions should take place when it receives the `JCFooEvent`.

JClass Field Events

The event listener that receives the events generated by the four editable Fields is called `JCValueListener`. Its methods are `valueChanging()` and `valueChanged()`. Changes to any one of the Fields are handled by invoking `addValueListener()`. You supply the code to implement the `JCValueListener` interface. To register the method, please see addValueListener, removeValueListener, in Chapter 2.

The methods of the JClass Field event listener are described below:

Event Methods	Description
`JCValueListener.valueChanging()`	The field has been edited, as signalled by the end-user pressing the **Return** key or leaving the field, or the `setValue()` method has been called programmatically, and the new value is about to replace the old value. `valueChanging()` is invoked whenever the `value` is about to change. Catching this event allows a change of the new value at this point if desired.
`JCValueListener.valueChanged()`	The value has been changed.

1.8 Keystroke Actions

Most key actions are intuitive, however there are some circumstances that may need explanation.

- The **Escape** key cancels any changes made in a field as long as the data has not been committed by pressing the **Return** key. However, if no valid data has previously been committed to the field and the default value is zero or null, then the **Escape** key will not cancel the changes.
- Arrow keys perform dual functions in calendar popups. If the calendar popup is visible, the arrow keys may be used to move to adjacent days. If the popup is hidden, the right and left arrow keys move the cursor through the date field, and the down arrow opens the popup.
- In combo fields, the down arrow key may be used to spin through the entries in the combo field's list. To actually pop down the list, use **CTRL**+*Down Arrow*.

1.9 An Example Program

Now that you have seen an overview of JClass Field's objects and validators, here is a sample program that illustrates several features of some JClass Field components.

You can create fields for your applications programmatically, or in an IDE. Either way, you are using the same Bean. The following sections describe building a field using an IDE. For more detailed information about the process of creating fields, both with an IDE or programmatically, see Building a Field, in Chapter 3.

The class `Examples` contains a `main` method and an `init` method and, therefore, it is both an applet and a program. The file is located in the *examples* directory. When run using the command `java examples.field.Examples`, the program produces the output shown in the figure below.

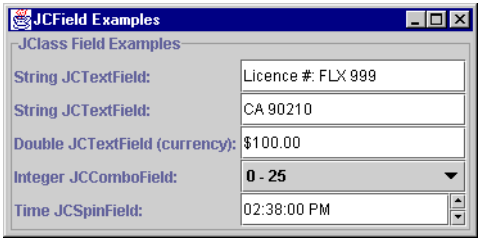


Figure 5 JClass Field examples.

Here is the program code:

```
package examples.field:

import com.klg.jclass.field.JCTextField;
import com.klg.jclass.field.JCComboField;
import com.klg.jclass.field.JCSpinField;
import com.klg.jclass.field.validate.*;
import com.klg.jclass.util.swing.JCExitFrame;
import com.klg.jclass.util.value.*;
import com.klg.jclass.field.JCInvalidInfo;
import com.klg.jclass.util.swing.JCListModel;

import javax.swing.*;
import javax.swing.border.TitledBorder;

import java.awt.GridLayout;
import java.awt.BorderLayout;
import java.awt.Color;
import java.util.Calendar;

public class Examples extends JApplet {

protected JCTextField text1, text2, text3, text4;
```

```
protected JCComboField combo1;
protected JCSpinField spin1;

public void init() {

     // set the layout
          getContentPane().setLayout(new BorderLayout());

     // place all the text fields in a panel
     JPanel p = new JPanel();
     getContentPane().add(p, BorderLayout.CENTER);
     p.setLayout(new GridLayout(5,2));
     p.setBorder(new TitledBorder("JClass Field Examples"));

     //
     // Example of a JCTextField using JCStringValidator
     //     with a mask set
     //
     p.add(new JLabel("String JCTextField: "));
     p.add(text1 = new JCTextField());

     // create the validator
     JCStringValidator sv1 = new JCStringValidator();

     // set the validator properties
     sv1.setMask("\\Licence \\#: UUU @@@");
     sv1.setPlaceHolderChars("Licence #: FLX 999");
     sv1.setAllowNull(false);

     // set the value model and validator
     text1.setValueModel(new StringValueModel());
     text1.setValidator(sv1);

     //
     // Example of a JCTextField using JCStringValidator
     //
     p.add(new JLabel("String JCTextField: "));
     p.add(text2 = new JCTextField());

     // create validator
     JCStringValidator sv2 = new JCStringValidator();

     // set validator properties
     sv2.setMask("AA @@@@@");
     sv2.setPlaceHolderChars("CA 90210");
     sv2.setAllowNull(true);

     // set the value model and validator
     text2.setValueModel(new StringValueModel());
     text2.setValidator(sv2);

     //
     // Example of a JCTextField using JCDoubleValidator
     //     with currency property
     //
```

```
p.add(new JLabel("Double JCTextField (currency): "));
p.add(text3 = new JCTextField());

// create validator
JCDoubleValidator cv = new JCDoubleValidator();

// set validator properties
cv.setAllowNull(true);
cv.setCurrency(true);
cv.setDefaultValue("0");

// set the invalidinfo properties
JCInvalidInfo ii1 = text3.getInvalidInfo();
ii1.setInvalidPolicy(JCInvalidInfo.CLEAR_FIELD);

// set the value model, invalidinfo and validator
text3.setValueModel(new DoubleValueModel(new Double(100.00)));
text3.setInvalidInfo(ii1);
text3.setValidator(cv);

//
// Example of a JCComboField using JCIntegerValidator
//
p.add(new JLabel("Integer JCComboField:"));
p.add(combo1 = new JCComboField());
combo1.setEditable(false);

// create validator
JCIntegerValidator iv = new JCIntegerValidator();

// set validator properties
Integer[] int_values = {new Integer(1), new Integer(2),
     new Integer(3), new Integer(4)};
String[] display = {"0 - 25", "26 - 50", "51 - 75", "75 - 100"};
iv.setAllowNull(true);
iv.setPickList(new JCListModel(int_values));
iv.setDisplayList(display);

// set the value model and validator
combo1.setValueModel(new IntegerValueModel());
combo1.setValidator(iv);
combo1.setSelectedIndex(0);

//
// Example of a JCSpinField using JCTimeValidator
//
p.add(new JLabel("Time JCSpinField: "));
p.add(spin1 = new JCSpinField());

// create validator
JCTimeValidator tv = new JCTimeValidator();

// set validator properties
```

```
        tv.setMaskInput(true);
        tv.setAllowNull(true);

        // set value model and validator
        spin1.setValueModel(new CalendarValueModel(
             Calendar.getInstance()));
        spin1.setValidator(tv);

    }
    public static void main(String[] args) {
        JCExitFrame frame = new JCExitFrame("JCField Examples");
        Examples t = new Examples();
        t.init();
        frame.getContentPane().add(t);
        frame.pack();
        frame.show();
    }
    }
```

1.9.1 Programming the Example

The five objects reside in a container and are introduced on the left by explanatory text labels. Neither the container itself nor the explanatory text is part of JClass Field, but they illustrate how JClass Field components and other JClass components are used with standard Java objects to achieve a desired result.

The `setMask()` method specifies the general format for any changes to the field. In the case of the `text1` object, the mask specifies that entered data will be validated against a pattern matching a North American license plate number. The @ symbol specifies that only digits are allowed in each of these positions and the U symbol specifies that only alphabetic characters are allowed in these positions. Any lowercase letters entered will be converted to uppercase.

The mask for `text2` specifies two letters followed by five digits. In this example, the `setPlaceHolderChars()` method displays a zip code in the field. This is not a default, but a prompt for the appropriate type entry.

The third object shows a default value of `$100.00`. This shows the use of a text field with type double and the `setCurrency()` method set to `true`. The field also uses the `setInvalidPolicy()` method, which is contained in the `InvalidInfo` object, to clear the field if the user enters an amount that is out of range. The range of valid values is determined by the `setRange()` method.

The field, `combo1`, uses the methods `setPickList()` and `setDisplayList()` to provide the user with meaningful choices while still using the type integer for the field value's data type.

The last object is a spin field that allows the user to change the time value. As for all objects with date or time types, the default, if none is set explicitly, is the current date and time.

1.9.2 The Property Sheet

It is possible to examine and experiment with the component properties of the JClass Field objects using the Bean Development Kit (BDK) or any IDE. You can use the Property sheet to change the properties of the JClass Field component under consideration.

Instructions on loading and running the BeanBox can be found online at *http://java.sun.com/beans/bdk_download.html*.

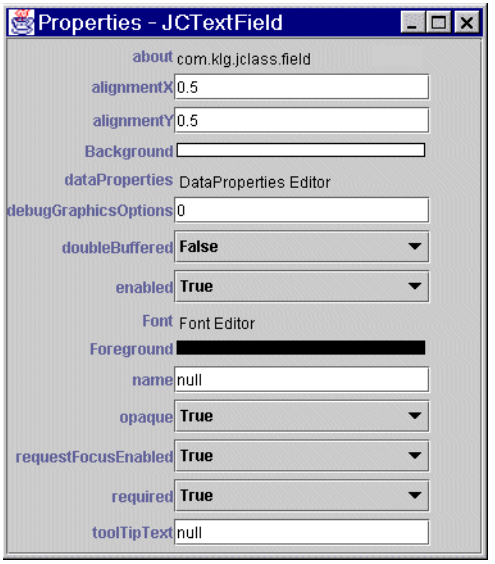


Figure 6 A JClass TextField component's properties as seen in the Property sheet.

The association between the property sheet and the design is dynamic, so any changes you make are automatically applied to the component in the BeanBox. Because the changes are immediately displayed as you edit properties, you can see how a change affects the JClass Field component without leaving the property sheet. You can continue to make changes and observe the results.

1.9.3 Using the Property Sheet

The following sections illustrate some steps for using the property sheet to customize the properties of a JClass Field Bean. You may wish to launch an appropriate program and execute the steps as they are described.

In the design window, click the JClass Field component you wish to customize. The property sheet's contents change to correspond to the component selected, as shown in the previous illustration.

1.9.4 Editing JClass Field Properties

The following lists the steps required to edit the properties for a JClass Field Bean:

1. In the design window, select the JClass Field component that you want to edit.

   All the editable Bean properties (and some read-only properties as well) appear in the Properties window.

2. Double click on the `DataProperties` property to invoke a property customizer that contains the `value`, `validator`, and `invalidinfo` object properties.

3. Before you can set the desired properties, you must select a data type from the list on the left of the **DataProperties** editor. Then you can customize the field using the properties in the **Value**, **Validator**, and **Invalid** tabs of the **DataProperties** editor.

4. When you make changes in the **DataProperties** editor, you must click **Apply** and then **Done** before the changes will be displayed.

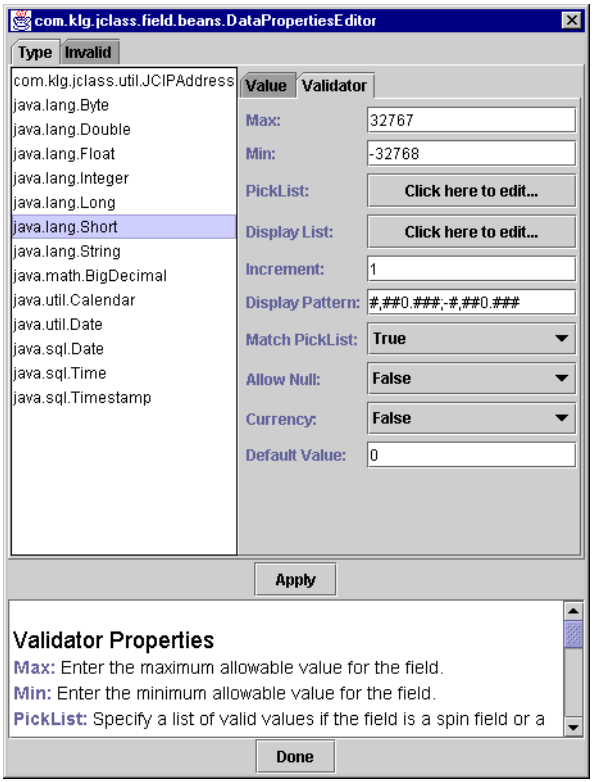


Figure 7 The property customizer of a JClass Field component.

5. Some properties, such as `about`, are read-only. They are included in the property sheet as information only. For example, double clicking on the `about` property will display the JClass help information. There is another read-only property, called `state`, that is used to describe whether the current contents of the field are valid, invalid, or in the process of being changed. Properties listed in Appendix A with a "(G)" after the property name have only a get method.

By now you should have a good idea of what JClass Field has to offer. The next chapter will explain in more detail the properties of the JClass Field components.

1.10 Internationalization

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

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

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

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

2

JClass Field's Properties

2.1 Introduction

This chapter discusses how to use JClass Field's key properties, and provides a quick summary of the properties arranged by data type. See Appendix A, JClass Field Property Listings, for a complete summary of JClass Field's properties, and the API for complete reference information.

Date, number, and format tables are included for reference purposes, since JClass Field components may be used for date and time information, for displaying and editing numbers and currencies, and for validating text entries. These formats follow standard Java conventions.

2.2 Field's Key Properties

The key properties of JClass Field components are contained in three objects: `JCValueModel`, `JCValidator`, and `JCInvalidInfo`. The properties, which you set to dictate the format and behavior of the JClass Field components, are not directly exposed by the Bean, but are accessed through the property customizer, DataProperties, or can be set programmatically.

JClass Field is designed to be used within an IDE, so almost all of its properties, including the property customizer, can be manipulated by using the property sheet in the IDE. Of course, you can set the properties programmatically as well.

2.2.1 The Value Model

value

`value` is the fundamental property of a field. A field's value changes dynamically as the end-user supplies data and has it validated. Because `value` is initially null, and `allowNull` is `false` by default, the field starts out in an invalid state.

The data member `value` is used to record both the initial value given to a field and its current value. It is initially set to null. If the `allowNull` property is `true`, the field may be left in this condition. However, the field is usually edited by the user and its null value changed to the value set by the user's input. After it has been inspected and approved by the field's validator, this change becomes the new value. Thus, in a typical application, `value` is updated through user interaction with the field. You see the current value displayed in the field and you can inspect `value` programmatically using the method `getValue()` to determine this central property of a field.

2.2.2 The Validator Property

By setting a data type with a JClass Field component, you associate a corresponding validator with that component. Here are some of the important properties contained in one or more of the JClass Field validators:

displayPattern, editPattern

You use these properties to control the appearance of values in number fields. When the field has focus, `editPattern` is in effect. When the field loses focus, `displayPattern` comes into effect. Both properties take their formats from one of the possible conventions for number formatting in the current locale, but you can choose a different format while in edit mode from that in display mode. Thus, you can allow the user to type a minus sign when entering a negative number, yet display it in the accounting fashion, using brackets instead of the minus sign.

It is possible to supply a type of feedback to users by appending a text String to the number that was entered, perhaps to remind users of the units that are expected in the field. For example, if the application is using a `JCTextField` with a property set for integer to collect a length measurement, and the required unit of measurement is feet, you could cause "ft." to display after the number by incorporating that String in the number format that you specify for `displayPattern`. The internal value of the field is unchanged; it remains a pure number, and no units are associated with it.

For JDK 1.3.1 and higher, numeric validators support scientific notation.

editFormats

This is a list of String formats that are used to match a user's input in fields that contain date and time values. The input is assumed to represent a partially completed field that the validator attempts to expand into a full date. There are a number of standard locale specific formats that can be used. The default edit formats are listed here:

- h:mm:ss 'o"clock' a z EEEE, MMMM d, yyyy
- h:mm:ss a z EEEE, MMMM d, yyyy
- h:mm:ss a EEEE, MMMM d, yyyy

- h:mm a EEEE, MMMM d, yyyy
- h:mm:ss 'o"clock' a z dd-MMM-yy
- h:mm:ss a dd-MMM-yy
- h:mm a dd-mm-yy
- h:mm:ss 'o"clock' a z M/d/yy
- h:mm:ss a M/d/yy
- MM d yy h:mm:ss or d MM yy or EE MM d

The symbols dd, mm, M, and so on, must follow the convention specified in the javadoc entry in the `setFormat` method of `JCDateTimeValidator`. You can construct a date format in any way you choose, so long as you assemble it from date format elements. This manual contains a copy of the date format conventions in Section 2.3.1, Date Formats, and in Appendix A, JClass Field Property Listings.

Note that the `maskInput` property potentially has an effect on `editFormats`. If you set `maskInput` to `true`, its format specification may override the one in `editFormats`.

mask, numMaskMatch

`mask` is a property of `JCStringValidator` and `JCDateTimeValidator`. You use it to specify the type of character that is admissible at each position. For example, in an instance of a `JCTextField` with a String validator, the line:

```
TextFieldInstance.setMask("\\Licence \\#: UUU @@@");
```

results in the following being displayed:

The cursor is placed at the first editable position, where an alphabetic character is expected. The effect of a U in the mask is to convert lower case input to upper case. After three letters are typed, which match the three Us in the mask, the cursor skips over the space, and then is ready to accept three numeric characters, represented by the three "at" signs (@) in the mask. Note that the prompt String, *Licence* #, contains both a letter L and a number sign. These are mask characters which have special meaning. To use them as character literals, you need to preface them with double backslashes. For a list of the special mask characters and their meaning, see Section 2.3.2, Mask Characters.

`numMaskMatch` is a property of `JCStringValidator`. With it, you set the number of characters to match against the mask, going from left to right. This number does not include any literals that you may have embedded between mask characters. If the value is negative, the entire mask will be matched. An example of the use of this property is in a

field that is designed to collect an office telephone number and extension. The mask might be:

```
TextFieldInstance.setMask("Phone \\#: (@@@) @@@-@@@@ Ext. @@@@");
```

If the value of `numMaskMatch` is set as follows:

```
TextFieldInstance.setNumMaskMatch(10);
```

then only the first ten digits are needed for the input to be considered valid, although fourteen digits will be accepted. Note that in the absence of a `placeHolderChars` String, the field will be blank; the pattern used in `setMask()` will not appear in the field.

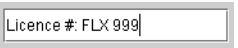
placeHolderChars

You can use the property `placeHolderChars` to override the appearance of a field when a mask has been used. Notice that in our mask, numMaskMatch example, the part of the field where the licence plate number is to be typed is blank. You can give the end-user a better idea of what the contents of the field should be by extending the prompt String to appear as follows.

Using a `JCTextField` with type String as an example, the command

```
TextFieldInstance.setPlaceHolderChars("Licence #: FLX 999");
```

results in the following being displayed:



The field now contains characters in the editable part of the field that give the user a good idea of what is expected.

If you use `placeHolderChars`, it is a good idea to match the mask String character by character. This means that you supply the same number of characters in the `placeHolderChars` String as there are in the `mask` String. If the `mask` String and `placeHolderChars` String have differing lengths or character placement, the field will use the character spacing of the mask and the display of the `placeHolderChars` String, which will likely confuse the user.

maskInput

The `maskInput` property exists for the date and time validators. If this Boolean property is `true`, the user's input must exactly match the date format that has been specified for the field. If it is `false`, the validator will attempt to complete a partial date as long as the input matches one of the currently set date formats.

When you use `placeHolderChars` with Calendar objects, `maskInput` must be `true`. The format is then transformed into a non-ambiguous form, and it is possible to use a `placeHolderChars` String as long as you are aware of the possible pitfall presented by ambiguous date formats. For example, if you use the ambiguous date format `h:mm:ss`, it is

internally converted to hh:mm:ss. Thus, an acceptable placeHolderChars String is shown in the following code fragment:

```
CalendarComponent.setMaskInput(true);
CalendarComponent.setPlaceHolderChars("HH:MM:SS");
```

The best approach is to use only non-ambiguous date format Strings.

Note that if you are using the JCDateTimeValidator or JCTimeValidator, the incrementField of the JCSpinFields depends on the maskInput property. In this case, if maskInput is set to true, the subfield will spin where the cursor is located. Otherwise, it will spin the subfield indicated by the incrementField property if it is explicitly set, or the hours subfield if it is not.

pickList, matchPickList, displayList

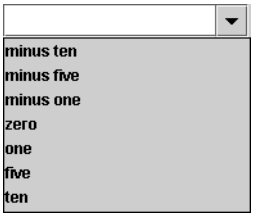
pickList is a property which provides a list of values for combo and spin fields with numeric and String data types. You can restrict the choices to just those given in the pick list by setting the Boolean property matchPickList to true. The property displayList provides further control over the way that the field displays its values. By defining a display list, values in the pick list are associated with the corresponding items in the display list. This additional capability is useful if you want to display Strings, yet couple them with integer values, in effect forming an associative array. The user sees the array element, and the field reports the index corresponding to that element as the value of the field. The following code snippet shows how:

```
// create validator
JCIntegerValidator IntVal= new JCIntegerValidator();

// set validator properties

Integer picklist[] = {new Integer(-10), new Integer(-5),
   new Integer(-1),new Integer(0), new Integer(1),
   new Integer(5), new Integer(10)};
String[] displaylist = {"minus ten", "minus five", "minus one", "zero",
   "one", "five", "ten"};
IntVal.setPickList(new JCListModel(picklist));
IntVal.setMatchPickList(true);
IntVal.setDisplayList(displaylist);
```

The code creates a JCIntegerValidator and declares a pick list of seven integer values. Since matchPickList is true, these values would be the only ones capable of being displayed in the field, except that a display list is also declared and set. Thus, instead of seeing the sequence of values from the pick list, -10, -5, -1, 0, 1, 10, in the combo box, the user sees the display list values, “minus ten”, “minus five”, “minus one”, “zero”, “one”, “five”, “ten”. The value of the field is the associated integer, not the String that is displayed.



defaultValue

All fields are given a default value when they are created. You may decide to change this value as part of your own initialization of the field, but one way or the other a field has an initial default value as well as a value. Typically, the default value is zero for numeric fields, null for String fields, and the current date and time for calendar fields. The data member `defaultValue` is used to hold a representative value for the field that does not get changed by user input or by the validation process. Note that the default value only displays in the field when the `JCInvalidInfo.RESTORE_DEFAULT` policy is in effect and the user has entered an invalid value. If you set the default value in a spin field without setting a value for the field, the field will be blank until the users clicks one of the arrows. The field will then spin from the default value to the next value.

You may wish to replace an invalid input with the default value as a way of providing the user with a reasonable starting place for further data entry. See invalidPolicy below for a description of how the `invalidPolicy` property is useful in this regard.

isCurrency

The `isCurrency` property allows you to specify whether a numeric type should be treated as currency. If the property is set to `true`, the value is displayed in the currency format appropriate to the set locale. The default value for this property is `false`. Note that if you set the property to `true` in a integer, short, byte, or long validator, the value will display with zero sub-units and the field will not allow the user to enter any fraction of the currency unit.

2.2.3 InvalidInfo Properties

invalidBackground, invalidForeground, beepOnInvalid

You use these three properties to provide visual and auditory warnings that an attempt has been made to enter invalid data in a field. By default, colors are inherited and `beepOnInvalid` is `true`. Examples are:

```
invalidInfoInstance.setInvalidBackground(Color.red);
invalidInfoInstance.setInvalidForeground(Color.white);
invalidInfoInstance.setBeepOnInvalid(false);
```

invalidPolicy

This property gives you four choices for the behavior of the field when invalid data is entered.

- `JCInvalidInfo.SHOW_INVALID` is the default value. It leaves the data in the field even after the field loses focus. To highlight the fact that the data is invalid, you can show it in different colors using the `invalidBackground` and `invalidForeground` properties.
- `JCInvalidInfo.RESTORE_DEFAULT` displays the default value. If the field loses focus, still containing an invalid value, setting `invalidPolicy` to this parameter causes the field to replace the invalid entry with the default.
- `JCInvalidInfo.RESTORE_PREVIOUS` replaces an invalid entry by the last valid value that was committed in the field.
- `JCInvalidInfo.CLEAR_FIELD` clears a field containing invalid data upon loss of focus. In this case, the field is blank and the value of the field is undefined.

2.2.4 Other Properties

state

`state` is one of the few read-only properties in JClass Field. Using `getState()`, you can determine if the value of the field is valid, invalid, or indeterminate. An indeterminate value arises when the field is currently being edited, so the validator must defer its decision until editing is complete. A field becomes "under edit" when the user types a key, and remains so until the field loses focus, the user presses the **Enter** key, or the field is resolved programmatically by the `commit()` method.

editable

This boolean property lets you decide whether or not a field can be edited via the keyboard. If you are concerned that it will be all too easy for the user to make a mistake if keyboard entry is allowed, you can set this property to `false` and restrict the user to employing the mouse. The items that you have placed in spin, combo, and popup fields contain (presumably) valid choices, so your users are constrained to one of these valid choices. You would set editable to `true` when the user must supply more generalized and unpredictable information, such as names and addresses.

You can make spin and combo fields extensible by allowing users to type new values into the field, but you are responsible for adding the programming code that adds these new values to the pick list. See Event Programming, in Chapter 4 for an example of this code. See pickList, matchPickList, displayList for a description of pick lists.

max and min

These properties set the minimum and maximum values of numeric fields. A convenience method, `setRange()`, allows you to set both properties in a single command. There are examples of its use in the code snippets in Chapter 4. In an IDE only the `min` and `max` properties are available.

2.2.5 addValueListener, removeValueListener

Changes to JClass Field are handled by invoking `addValueListener()`. Just as with any other listener, you supply the code to implement the `JCValueListener` interface, and add the event handler to the Field. For example:

```
myField.addValueListener(new MyJCValueListener);
```

The `removeValueListener()` method removes the named listener object.

2.3 Format Tables

The format Strings for date and time validators, the mask characters for the String validator, and number format characters for fields using numeric validators are listed in the next three sections.

2.3.1 Date Formats

Symbol(s)	Meaning
y	Year within the current century (1 or 2 digits).
yy	Year within the current century (2 digits).
yyyy	Year including century (4 digits).
M	Numeric month of year (1 or 2 digits).
MM	Numeric month of year (2 digits).
MMM	Abbreviated month name.
MMMM	Full month name.
EE	Day of the Week (abbreviated).
EEEE	Day of the Week (full name).
d	Numeric day of month (1 or 2 digits).
dd	Numeric day of month (2 digits).
h	Hour of day (1-12) (1 or 2 digits).
hh	Hour of day (1-12) (2 digits).
H	Hour of day (0-23) (1 or 2 digits).
HH	Hour of day (0-23) (2 digits).

Symbol(s)	Meaning
m	Minutes (1 or 2 digits).
mm	Minutes (2 digits).
s	Seconds (1 or 2 digits).
ss	Seconds (2 digits).
a	AM/PM representation.
p	AM/PM representation.
z	Time zone abbreviation.
zz	Time zone abbreviation.
zzzz	Time zone (full name).
D	Day in year (1, 2, or 3 digits).
DDD	Day in year (3 digits).

2.3.2 Mask Characters

Symbol	Meaning
#	Any digit, minus sign, comma, decimal point, or plus sign.
@	Any digit.
H	Any hexadecimal digit.
U	Any alphabetic character. Lower case characters will be converted to upper case.
L	Any alphabetic character. Upper case characters will be converted to lower case.
A	Any alphabetic character. No case conversion.
*	Any character.
^	An alphanumeric character – one of the set {0-9a-zA-Z}.
\\	The next character that follows is to be treated as a literal, even if it is one of the above characters.

2.3.3 Number Format Characters

Symbol	Meaning
0	Any digit, zeros show as zero.
#	A digit, zero shows as absent.
.	Placeholder for decimal separator.
,	Placeholder for grouping separator.
E	Separates mantissa and exponent for exponential formats.
;	Separates formats.
-	Locale-specific negative prefix.
X	Any other characters can be used in the prefix or suffix.
'	Used to quote special characters in a prefix or suffix.
other	Appears literally in the output.

Notes:
If there is no explicit negative sub-pattern, - is prefixed to the positive form. That is, "0.00" alone is equivalent to "0.00;-0.00".

Illegal formats, such as "#.#.#" or mixing '-' and '*' in the same format, will cause a `ParseException` to be thrown. From that `ParseException`, you can find the place in the String where the error occurred.

The grouping separator is commonly used for thousands, but in some countries for ten-thousands. The interval is a constant number of digits between the grouping characters, such as 100,000,000 or 1,0000,0000.

If you supply a pattern with multiple grouping characters, the interval between the last one and the end of the integer is the one that is used. So, the grouping interval for each of "#,##,###,####", "######,####", and "##,####,####" is four.

2.4 Property Summaries

The first property list shown below details the properties common to all fields. The following lists are organized according to properties contained in the three main validators of the JClass Field components and the `InvalidInfo` and `ValueModel` objects. You can use these lists for quick reference to the properties that a particular object possesses; however, the best reference is the API for a particular component.

These lists differ from the single list given in Appendix A, JClass Field Property Listings, where the property is listed and the JClass Field types which can be customized by it are listed in the second column.

A small number of properties are read-only variables, and therefore only have a *get* method. These properties are marked with a "(G)" following their property name. There is also one property that has only a *set* method, and is marked with an "(S)" f following the property name.

2.4.1 Properties for JClass Field Components

Property	Type	Default
about	String	com.klg.jclass.field 4.5.0
background	Color	inherited
doubleBuffered	boolean	false
editable	boolean	true
enabled	boolean	true
font	Font	inherited
foreground	Color	inherited
maximumSize	Dimension	dynamic
minimumSize	Dimension	dynamic
name	String	null
preferredSize	Dimension	dynamic
required	boolean	true
selectOnEnter	boolean	false
state (G)	int	N/A
toolTipText	String	null

2.4.2 Properties for Numeric and IPAddress Validators

Property	Type	Default
allowNull	boolean	false
casePolicy	int	JCValidator.AS_IS
continuousScroll	boolean	false

Property	Type	Default
currency	boolean	false
currencyLocale	Locale	locale dependent
currencySymbol (G)	String	locale dependent
defaultValue	Object	0
displayList[a]	String	null
displayPattern	String	locale dependent
editPattern	String	Byte, Short, Integer, Long: 0 Float, Double, BigDecimal: 0.###
firstValidCursor Position (G)	int	usually 0, but dependent on mask set
increment	Number	1
invalidChars	String	null
locale	Locale	Locale.getDefault
matchPickList	boolean	true
max	int	type dependent
min	int	type dependent
pickList	ListModel	null
pickListIndex (G)	Object	N/A
range (S)	int	type dependent
spinPolicy	int	JCValidator.SPIN_FIELD
useIntlCurrencySymbol	boolean	false
validChars	String	null

a. Only byte, short, integer and long types possess these properties.

2.4.3 Properties for JCStringValidator

Property	Type	Default
allowNull	boolean	false
casePolicy	int	JCValidator.AS_IS
continuousScroll	boolean	false

Property	Type	Default
defaultValue	Object	null
firstValidCursor Position (G)	int	0
invalidChars	String	null
locale	Locale	Locale.getDefault
mask	String	null
maskChars	String	#@HULA*^\
matchPickList	boolean	true
numMaskMatch	int	-1
pickList	ListModel	null
pickListIndex (G)	Object	N/A
placeHolderChars	String	null
spinPolicy	int	JCValidator.SPIN_WRAP
validChars	String	null

2.4.4 Properties for Date/Time Validators

Property	Type	Default
allowNull	boolean	false
casePolicy	int	JCValidator.AS_IS
continuousScroll	boolean	false
defaultDetail	int	medium
defaultEditFormats (G)	String	N/A
defaultFormat (G)	String	N/A
defaultValue	Object	null
editFormats	String	locale dependent
firstValidCursor Position (G)	int	0
format	String	locale dependent
increment	int	1

Property	Type	Default
invalidChars	String	null
locale	Locale	Locale.getDefault
mask	String	null
maskChars	String	#@HULA*^\\
maskInput	boolean	true for JCSpinField with any date type; false otherwise
matchPickList	boolean	true
milleniumThreshold	int	69
numMaskMatch	int	-1
parsedMask (G)	String	N/A
pickList	ListModel	null
pickListIndex (G)	Object	N/A
placeHolderChars	String	null
spinPolicy	int	JCValidator.SPIN_SUBFIELD
timeZone	java.util.TimeZone	locale dependent
validChars	String	null

2.4.5 InvalidInfo Properties

Property	Type	Default
beepOnInvalid	boolean	true
invalidBackground	Color	null
invalidForeground	Color	null
invalidPolicy	int	JCInvalidInfo.SHOW_INVALID

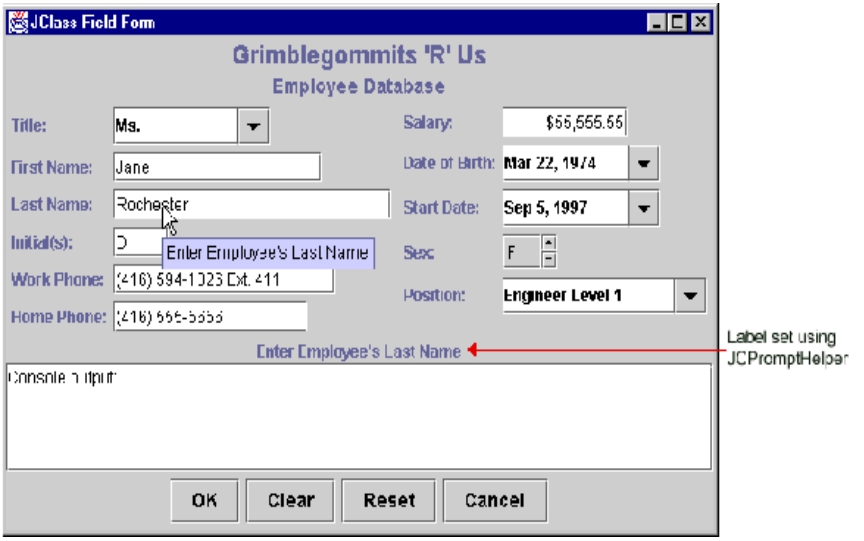
2.4.6 ValueModel Properties

Property	Type	Default
value	Object	null

`valueClass (G)`	`java.lang.Class`	N/A

2.5 Exploring the Form Demo

JClass Field includes more extended sample programs. For example, the Form demo implements a complete data-entry form containing all of the elements needed by such an application. The code is located in the *demos/field/form* directory.



This program uses two additional classes of JClass Field. The following sections describe them.

2.5.1 JCPromptHelper

`JCPromptHelper` extends the function of the `toolTipText` property. This class allows you to set a label in the specified container that takes the value of the tool tip text associated with the field in focus.

For example, in the illustration above, the text above the console output area is the tool tip text for the field containing the cursor **Last Name**.

2.5.2 JCFormUtil

The class `JCFormUtil` provides several useful methods for collecting different sets of information based on the JClass Field components in a container.

- The `clearFieldComponents()` method allows you to set the values in all of the JClass Field components in a specified container to null. The `resetFieldComponents()` method can be used to reset all fields to their default values.
- The `getFieldComponents()` method returns a list of the JClass Field components in a specified container.
- The `getInvalidRequiredFieldComponents()` method returns a list of the required JClass Field components in the container that are in an invalid state.
- The `getRequiredFieldComponents()` method returns a list of the required JClass Field components in a container.
- The `isFieldComponentContainerComplete()` method returns a value of `true` if all the required JClass Field components in a container have valid values.

The form program demonstrates all of these functions. You can view the return values in your command window. In addition, `demos.field.form.Form` has a scrollable text window where you can view the values directly. It also displays messages that inform the user of invalid input and incomplete entries.

3

Building a Field

3.1 Determining Which Technique to Use

JClass Field offers several options when it comes to modifying properties. The choice of technique is a personal preference; however, the following two sections illustrate some important points to consider when deciding which technique to use.

3.1.1 Using an Integrated Development Environment

JClass Field 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. To modify properties of the component in an IDE, you simply drag the component onto the form, then edit the properties exposed by the Bean and the properties in the **DataProperties** editor. You can use many of Field's default properties "as is" and set the few that are specific to your application.

3.1.2 Setting Properties Programmatically

Setting properties programmatically requires writing the actual Java code that will accomplish the task. This approach offers more control, because elements not exposed by the Bean model may be accessed.

As mentioned previously, most properties in JClass Field have `set` and `get` methods associated with them. For example, to retrieve the value of the `value` property in a `JCTextField` instance, do the following:

```
TextFieldValueModel.getValue();
```

3.2 Creating a New Field Component (Using an IDE)

The following steps provide an outline for building a new Field component in an IDE.

1. Add the field you want to build to your container.
2. Set the general bean properties available in the property editor.

3. Open the **DataProperties** editor and select a data type for your field. Now you can test your field by entering data into it.
4. Set the field's initial value and other properties under the **Invalid** and **Validator** tabs in the **DataProperties** editor. Refer to the bottom panel of the window for descriptive help on the properties.

   Since you can associate most validators to most JClass Field components, there are some validator properties that will have no effect on a particular field. For example, although you can set a pick list for a text field, the user will never see the pick list values.

Note that the changes to the JClass Field Component do not take effect until you click **Apply** and close the **DataProperties** editor.

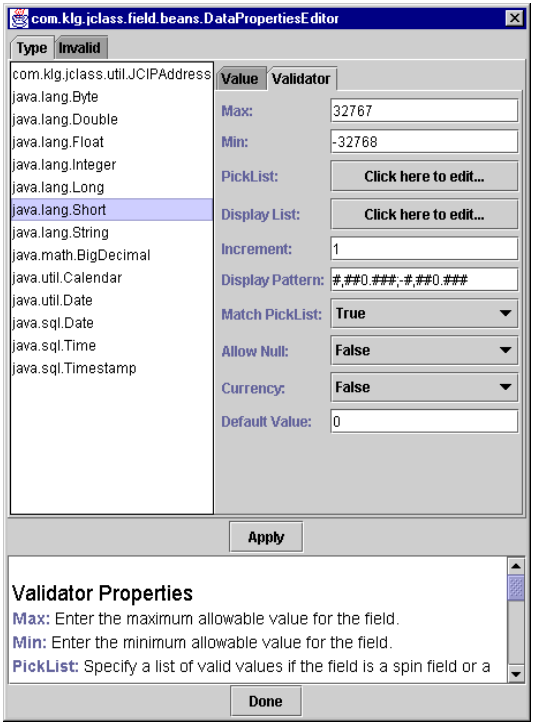


*Figure 8 The **DataProperties** editor.*

3.3 Creating a New Field Component (Programmatically)

You might want to use existing JClass Field example code as a starting point for the new object. The examples and demos provided with JClass Field distribution are a good

starting point. You can use the following steps as a general outline for creating a field component programmatically or start with the appropriate example field and modify to your specifications.

1. Create a container for your new field component.
2. Declare an instance of the type of field you want (JCTextField, JCSpinField, JCComboField, JCPopupField, or JCLabelField) and add it to the component.
3. Select the data type you want to use and declare the appropriate validator as follows:
   `JC<DataType>Validator val = new JC<DataType>Validator();`
4. Set the validator properties using the Property Summaries tables found in JClass Field's Properties, in Chapter 2.
5. If you want to define the behavior of the field when it receives an invalid entry, declare an InvalidInfo object:
   `JCInvalidInfo ii = NewField.getInvalidInfo();`
   then set the InvalidInfo Properties using the table found in Section 2.4.5.
6. Associate a value model, and the `validator` and the `InvalidInfo` objects with the field, as follows:
   ```
   NewField.setValueModel(new <DataType>ValueModel());
   NewField.setValidator(val);
   NewField.setInvalidInfo(ii);
   ```
   You can use the ValueModel declaration to set the initial value for the field.
7. You can also integrate the field with your application by associating it with events and by using utility classes such as `JCPromptHelper` and `JCFormUtil`.

3.3.1 Customizing a New Field Component

Now that you have created your field, you can modify it to suit your specific needs. The following lists present the most common ways to customize a field. Use them as a guide to customizing your field. Example code for some of the methods here appears in chapter 4.

JCString Validator Only

- Specify the valid characters at each position in the field using `setMask()`.
- Limit the number of characters to match with the mask using `setNumMaskMatch()`.
- Put in place-holder characters to provide a hint of the required format using `setPlaceHolderChars()`.
- Provide a list of valid or invalid characters, using `setValidChars()` or `setInvalidChars()`.

Numeric Validators Only

- Set the amount by which a spin field will increase or decrease when the user clicks an arrow using `setIncrement()`.

- Format the appearance of the field during and after editing using `editPattern()` and `displayPattern()`.
- Set a numeric field's value to be treated as currency using `isCurrency()`.
- Specify the valid values for a numeric field using `setRange()`.

Date/Time Validators Only

- Set the appearance of date and time fields using `setFormat()`.
- Allow several formats for entering date and time information using `editFormats()`.

Multiple Validators

- Specify values available in a spin or combo component and their displayed values using `setPickList()` and `setDisplayList()`.
- Determine whether the user can enter values that are not on the pick list using `matchPickList()`.
- Set a default value for the field using `setDefaultValue()`.
- Specify a case policy of upper case or lower case using `setCasePolicy()`.

JCInvalidInfo Customization

- Specify the behavior of a field when the user enters an invalid value using `setInvalidPolicy()`.
- Specify the colors of the field when it contains invalid data using `setInvalidForeground()` and `setInvalidBackground()`.
- Set an auditory warning for invalid entries using `setBeepOnInvalid()`.

Other JCField Customization

- Add prompt text for a mouse-over or for use with `JCPromptHelper` using `setToolTipText()`.
- Determine whether the component is editable using `isEditable()`.

3.4 Data Binding

JClass Field provides special components that connect and bind to IDE or JDBC-compliant data sources, including the database components that are part of Borland JBuilder 3.0 or later.

Fields are bound to a particular column of a query *result set* and display the value at the *current record*. You can enable users to change the value, and have the field validate the change before committing the change back to the database. You can also change the current record displayed in the field programmatically or by using a GUI query navigation component.

Preliminaries

There are five types of GUI components provided for data binding – a text field, a spin field, a combo field, a popup field, and a label field. JClass Field's data-bound Beans dynamically determine their data type at runtime, based on the data type of the column they are bound to.

The Beans are packaged in a separate JAR file for each IDE environment; be sure you are using the correct one for your environment (please see the *JClass DesktopViews Installation Guide* for details). The following table lists the data-bound Field Beans included with this release:

JClass Field Databound Bean	IDE Requirements and Data Source Compatibility

`DSdbTextField`
`DSdbSpinField`
`DSdbComboField`
`DSdbPopupField`
`DSdbLabelField`
	■ JClass DataSource 4.5 or higher
	■ Works with Data Bean and TreeData Bean data source components which connect to JDBC- or ODBC-compliant databases

`JBdbTextField`
`JBdbSpinField`
`JBdbComboField`
`JBdbPopupField`
`JBdbLabelField`
	■ Borland JBuilder 3.0
	■ Works with DataExpress data source components such as QueryDataSet.

Note: You must be using Borland JBuilder 3.0 or later to use the "JBdb" IDE-specific Beans. Earlier versions will not work.

Before proceeding, you should ensure that your IDE and database are configured correctly and that you can create simple database applications.

JClass DataSource

JClass DataSource is a platform-independent JDBC-compliant hierarchical data source product. With it, your applications can bind to databases without being locked into an IDE-specific data binding solution.

JClass DataSource is available as part of the JClass DesktopViews product bundle. Visit *http://www.quest.com* for more information and downloads.

3.4.1 Data Binding in Borland JBuilder

Binding a field to a database in Borland JBuilder involves adding a database connection and query functionality using JBuilder Data Express components, and then using a JClass Field "JBdb" component to connect to the dataset column and display the data. This section walks through these steps.

Note: Database setup, connection, and querying are handled by JBuilder components. Our coverage of these components is only intended as a guide. Consult your JBuilder documentation for detailed information on JBuilder database connectivity.

Step 1: Connect to a Database

Use JBuilder's `Database` Bean to create a database connection. This component is located on the **DataExpress** tab in the Component Palette.

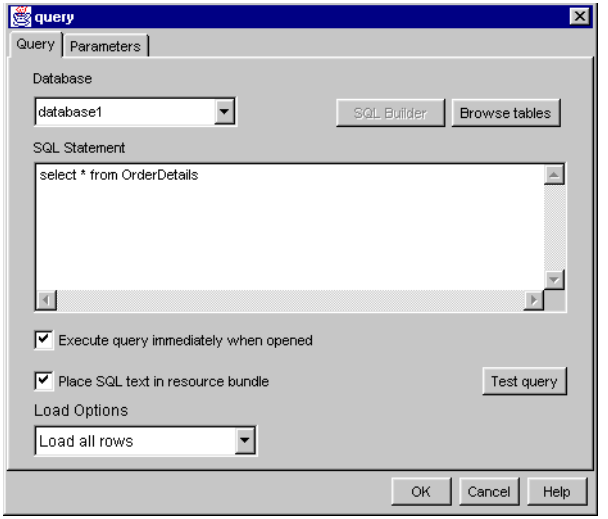
Add an instance to your frame. Then, use the `connection` property to specify the URL of the database you want to use.

Step 2: Query the Data

To query the database, add an instance of JBuilder's `QueryDataSet` (also on the **DataExpress** tab) to your frame.

Select the columns you want to retrieve with the `query` property editor. For example, to select all of the columns from a table named OrderDetails, you would use a statement similar to:

```
select * from OrderDetails
```



You can include all columns at this step, and then use a "JBdb" data-bound Bean to specify the column to display. Each column can be bound to a different "JBdb" field component.

Step 3: Bind a Field to the DataSet

With the database connection established and the query created, you can now add a data binding field Bean and connect it to the JBuilder DataSet to display the data. The data

binding properties of the `JBdbTextField`, `JBdbSpinField`, `JBdbComboField`, `JBdbPopupField`, and `JBdbLabelField` Beans are `dataSet` and `columnName`.

Add a "JBdb" Bean to your frame.

Select a query from the `dataSet` property's pull-down menu. If the database connection and query are set up correctly, there should be one or more queries in the list.

Then, select the column to display in the field using the `columnName` property. Enter the column name into the property editor. The case must match that of the column name in the table.

Step 4: Add Navigation Controls (optional)

The field displays the value at one particular record in the table; this is known as the *current record*. To display the value at another record, add a database navigation component such as the `borland.jbcl.control.NavigatorControl` component, and connect it to the `QueryDataSet`. You should then be able to traverse through the query, displaying each row in your data-bound fields.

With your connection established, you can then use the other Bean properties, such as `DataProperties`, to configure the field's validation behavior. Note that because the data type of a field is determined by the column to which it is bound, you cannot access its type dependent properties in the **DataProperties** editor until it is bound to a specific column.

3.4.2 Data Binding with JClass DataSource

The third way to add data binding to a JClass Field application is to use the data source components provided with JClass DataSource, a separately-available product from Quest. JClass DataSource is a platform-independent JDBC-compliant hierarchical data source product.

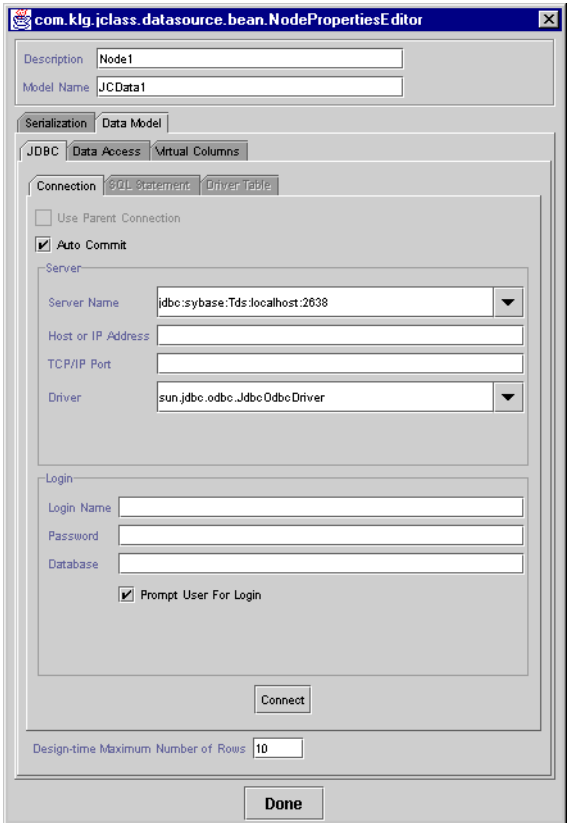
Binding a field to a database with JClass DataSource involves adding a database connection and query using JClass DataSource's `JCData` Bean and `JCTreeData` Bean components, and then using a JClass Field "DSdb" component to connect to the JClass DataSource and display the data. This section walks through these steps using the `JCData` Bean component.

Database setup, connection, and querying are handled by JClass DataSourcee. Our coverage of these components is only intended as a guide. Consult your JClass DataSource documentation for detailed information on configuring its components.

Step 1: Connect to a Database

Add a `JCData` Bean instance to your design area.

Then, use the `dataBeanComponent` property editor to specify the URL of the database you want to use and the database query.

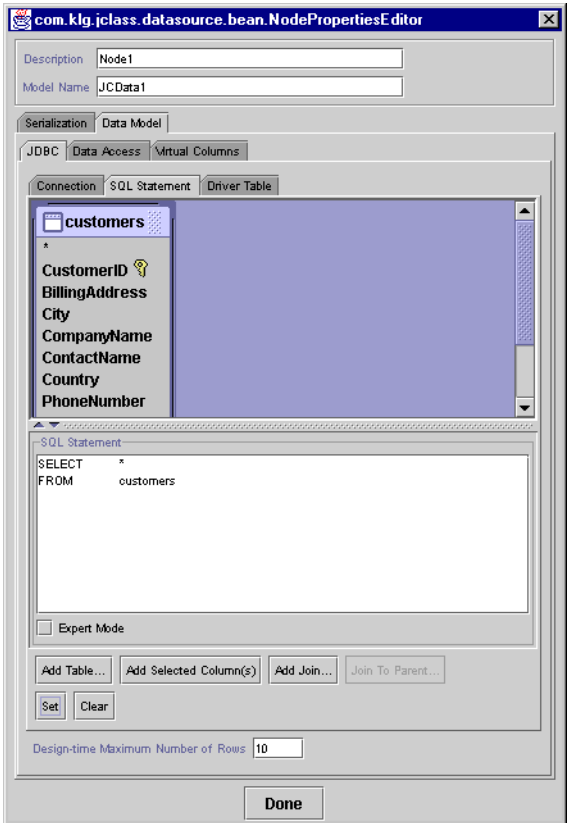


The first thing to do is to set up a serialization file under the **Serialization** tab. This file saves information and settings about the connection. You can then proceed to set up a connection.

To set up a database connection, display the **Data Model > JDBC > Connection** tab and specify the *Server Name* and *Driver* for the database you want to connect to. Test the connection. When the connection is successful you can proceed to set up a query. The JClass DataSource documentation contains complete details on using the `dataBeanComponent` property editor.

Step 2: Query the Data

Display the **Data Model > JDBC > SQL Statement** tab to show the query options:



You can create your entire query using mouse clicks (or you can enter it directly in the text window if you are proficient with SQL). First, add a table, and then create a query by selecting columns. When you have built the query, click **Set/Modify** and then **Done**.

You can include all columns at this step, and then use a "DSdb" data-bound Bean to specify the column to display. Each column can be bound to a different "DSdb" field component.

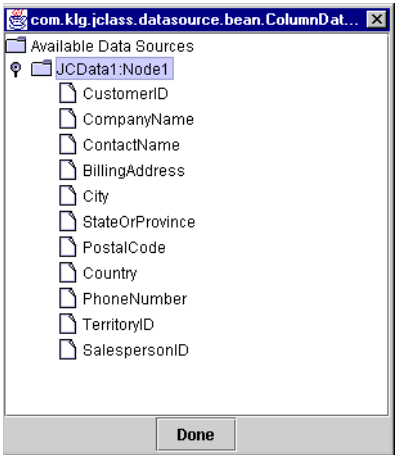
Step 3: Bind a Field to the Data Bean

With the database connection established and the query created, you can now add a data binding field Bean and connect it to the JClass DataSource `JCData` Bean to display the

data. The data binding property of the DSdbTextField, DSdbSpinField, DSdbComboField, DSdbPopupField, and DSdbLabelField Beans is dataBinding.

First, add a "DSdb" Bean to your design area.

Click the dataBinding property to display its property editor. If the JCData Bean's database connection and query are set up correctly, there should be one or more queries in the list.



Double-click a query to display the available columns. Select the column to display in the field and click **Done**.

Step 4: Add Navigation Controls (optional)

The field displays the value at one particular record in the table; this is known as the *current record*. You need to use a database navigation component to traverse to another record and display that value. JClass DataSource provides a Navigator Bean that you can use for this purpose.

Add a DSdbNavigator or DSdbJNavigator to the design area and connect it to the JCData Bean. You should then be able to traverse through the query, displaying each row in your data-bound fields.

With your connection established, you can then use the other Bean properties, such as validator, to configure the field's validation behavior. Note that because the data type of

a field is determined by the column to which it is bound, you cannot access its type dependent properties in the **DataProperties** editor until it is bound to a specific column.

Examples

JClass Field includes several sample programs that work with JClass DataSource, located in *examples/field/db*.

3.5 Handling Two-Digit Year Values

An application can use JClass Field to display and store abbreviated year values. For example, June 15, 1966 could be displayed as "15/06/66" in a date-type `JCTextField`.

Using two-digits for year values introduces an ambiguity about which century or millennium the date applies to, especially near the beginning or end of a century. That is, "15/06/66" could be interpreted as either June 15, 1966 or June 15, 2066.

The best approach for avoiding Year 2000 problems in your application is to use four digits to specify year values. If this is not possible, JClass Field provides a `milleniumThreshold` property that you can use to interpret two-digit years.

Located in the `com.klg.jclass.field.validate.JCDateTimeValidator` class, `milleniumThreshold` controls the interpretation of two-digit years. Any two-digit year less than the value of this property is considered to be after the year 2000, while any value greater than or equal to the threshold is considered to be after the year 1900.

The default threshold is 69. This means that a year value of "95" is treated as 1995 and a "01" value is treated as 2001. The following image shows the effect of using milleniumThreshold (the bottom field is invalid because 1900 is not a leap year).

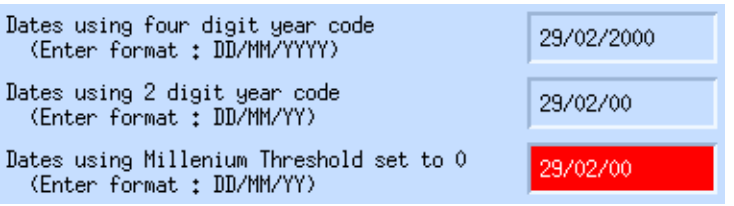


Figure 9 Four-digit year (top), two-digit year 2000 date (middle), and two-digit 1900 year (bottom).

4

Example Code for Common Fields

4.1 Example Programs

This chapter contains example code fragments that demonstrate the common uses of JClass Field components. In most cases, the properties used are exposed in IDEs, making the job of producing a GUI considerably easier. However, even if you are using an IDE, this code will extend a field's capabilities beyond the properties provided in the IDE.

The code listings below are snippets from the examples in the distribution, which contain a main method so that they can be run as an application as well as in a browser.

The table below provides a quick reference to the examples in this chapter.

For Code On…	See…
Using place holder characters to indicate the parts to be filled in;	Section 4.2.1, JCTextField with String Validator Section 4.3.1, JCSpinField with String Validator Section 4.6.1, JCLabelField with String Validator
Controlling the field's appearance before and during user edit;	Section 4.2.2, JCTextField with Integer Validator Section 4.3.7, JCSpinField with BigDecimal Validator Section 4.4.5, JCComboField with Byte Validator Section 4.6.2, JCLabelField with Integer Validator
Selecting the contents of the field whenever it receives focus;	Section 4.2.3, JCTextField with Long Validator Section 4.6.3, JCLabelField with Long Validator

For Code On…	See…
Defining a range of valid input, and providing a visual and audio warning to the user when the field is invalid;	Section 4.2.4, JCTextField with Short Validator Section 4.3.5, JCSpinField with Byte Validator Section 4.6.4, JCLabelField with Short Validator
Defining a range of valid input, and a default value when the user's input is invalid;	Section 4.2.5, JCTextField with Byte Validator Section 4.4.7, JCComboField with BigDecimal Validator Section 4.6.5, JCLabelField with Byte Validator
Displaying the content of a field as a currency of a given locale;	Section 4.2.6, JCTextField with Double Validator Section 4.3.6, JCSpinField with Double Validator Section 4.4.6, JCComboField with Double Validator Section 4.6.6, JCLabelField with Double Validator
Setting an invalid policy to restore the previous valid value;	Section 4.2.7, JCTextField with BigDecimal Validator Section 4.6.7, JCLabelField with BigDecimal Validator
Setting an invalid policy to clear the field when the user's input is invalid;	Section 4.2.8, JCTextField with Float Validator Section 4.3.2, JCSpinField with Integer Validator Section 4.3.8, JCSpinField with Float Validator Section 4.4.8, JCComboField with Float Validator Section 4.6.8, JCLabelField with Float Validator
Allowing date input in several formats, and to attempt to complete a partially entered date;	Section 4.2.9, JCTextField with DateTime Validator Section 4.6.9, JCLabelField with DateTime Validator

For Code On…	See…
Allowing date input in one format, specified by place holder characters, and converting all characters to uppercase;	Section 4.2.10, JCTextField with Date Validator Section 4.6.10, JCLabelField with Date Validator
Displaying and updating time information;	Section 4.2.11, JCTextField with Time Validator Section 4.3.11, JCSpinField with Time Validator Section 4.6.11, JCLabelField with Time Validator
Displaying IP addresses;	Section 4.2.12, JCTextField with IP Address Validator Section 4.3.12, JCSpinField with IP Address Validator Section 4.4.9, JCComboField with IP Address Validator Section 4.6.12, JCLabelField with IP Address Validator
Associating numeric field values in the pick list with text displayed in the spin field;	Section 4.3.3, JCSpinField with Long Validator
Allowing the user to enter a value not contained in the pick list;	Section 4.3.4, JCSpinField with Short Validator Section 4.4.1, JCComboField with String Validator Section 4.4.4, JCComboField with Short Validator
Allowing the user to set date and/or time;	Section 4.3.9, JCSpinField with DateTime Validator Section 4.5.1, JCPopupField with DateTime Validator Section 4.5.2, JCPopupField with Date Validator
Creating date and time formats for a spin field;	Section 4.3.10, JCSpinField with Date Validator
Allowing the user to pick only a value from the pick list; any other input is cleared;	Section 4.3.3, JCSpinField with Long Validator Section 4.4.3, JCComboField with Long Validator

For Code On…	See…
Presenting the user with a choice of items internally associated with ordinal numbers, for example for database applications;	Section 4.4.2, JCComboField with Integer Validator

4.2 Examples of Text Fields

The following code snippets are from *TextFields.java* found in the *examples/field* directory. Run the examples using the command:

```
java examples.field.TextFields
```

4.2.1 JCTextField with String Validator

This example demonstrates the effect of using place holder characters to supplement the more limited display capabilities of the mask property. You can fill the field with visible underscores to indicate the parts to be filled in.

```
p.add(new JLabel("String JCTextField: "));
p.add(text1 = new JCTextField());

// create the validator and set its properties
JCStringValidator sv = new JCStringValidator();
sv.setMask("(@@@)@@@-@@@@ Ext. @@@");
sv.setPlaceHolderChars("(___)___-____ Ext. ___");
sv.setAllowNull(true);

// set the value model and validator
text1.setValueModel(new StringValueModel());
text1.setValidator(sv);
```

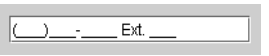


Figure 10 JCTextField with String validator.

4.2.2 JCTextField with Integer Validator

This example demonstrates how the `displayPattern` and `editPattern` properties determine the format of the field depending on whether it has focus.

```
p.add(new JLabel("Integer JCTextField: "));
p.add(text2 = new JCTextField());

// create validator and set its properties
JCIntegerValidator iv = new JCIntegerValidator();
iv.setAllowNull(true);
iv.setDisplayPattern("0 inches");
iv.setEditPattern("");
```

```
// set the value model and validator
text2.setValueModel(new IntegerValueModel(new Integer(100000)));
text2.setValidator(iv);
```

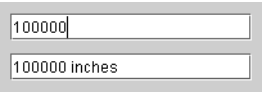


Figure 11 JCTextField with integer validator showing edit (top) and display formats.

4.2.3 JCTextField with Long Validator

This field uses a bean property to select the value in the field when it received focus. It also uses the default display and edit formats.

```
p.add(new JLabel("Long JCTextField: "));
p.add(text3 = new JCTextField());

// create validator and set its properties
JCLongValidator lv = new JCLongValidator();
lv.setAllowNull(true);

// set the value model and validator
text3.setValueModel(new LongValueModel(new Long(10000000000001)));
text3.setValidator(lv);
text3.setSelectOnEnter(true);
```

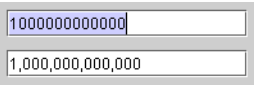


Figure 12 JCTextField with long validator showing edit (top) and display formats.

4.2.4 JCTextField with Short Validator

This example illustrates the use of a validator to confine user input to an acceptable range, and to provide a visual warning to the user when the field is invalid.

```
p.add(new JLabel("Short JCTextField: "));
p.add(text4 = new JCTextField());

// create the validator and set its properties
JCShortValidator shv = new JCShortValidator();
shv.setAllowNull(true);
shv.setRange(new Short((short)0), new Short((short)10));

// set the invalid info properties
JCInvalidInfo shii = text4.getInvalidInfo();
shii.setInvalidBackground(Color.red);

// set value model, validator, and invalidinfo
text4.setValueModel(new ShortValueModel(new Short((short)10)));
text4.setValidator(shv);
text4.setInvalidInfo(shii);
```

Figure 13 JCTextField with short validator is given an invalid entry.

4.2.5 JCTextField with Byte Validator

This example shows how the `invalidPolicy`, `JCInvalidInfo.RESTORE_DEFAULT`, forces the field to display the default value after the user attempts to commit a number to the field that is out of range.

```
p.add(new JLabel("Byte JCTextField: "));
p.add(text5 = new JCTextField());

// create the validator and set its properties
JCByteValidator bytev = new JCByteValidator();
bytev.setDefaultValue(new Byte((byte)5));
bytev.setAllowNull(true);
bytev.setRange(new Byte((byte)1), new Byte((byte)10));

// set the invalidinfo properties
JCInvalidInfo byteii = text5.getInvalidInfo();
byteii.setInvalidPolicy(JCInvalidInfo.RESTORE_DEFAULT);

// set the value model, validator, and invalidinfo
text5.setValueModel(new ByteValueModel(new Byte((byte)1)));
text5.setValidator(bytev);
text5.setInvalidInfo(byteii);
```

Figure 14 JCTextField with byte validator showing default value.

4.2.6 JCTextField with Double Validator

The double validator associated with this text field is augmented by the `isCurrency` property so that the value is treated as currency. The display format uses the currency format associated with the current locale.

```
p.add(new JLabel("Double JCTextField (currency): "));
p.add(text6 = new JCTextField());

// create validator and set its properties
JCDoubleValidator dv = new JCDoubleValidator();
dv.setAllowNull(true);
dv.setCurrency(true);

// set value and validator
text6.setValueModel(new DoubleValueModel(new Double(100.00)));
text6.setValidator(dv);
```

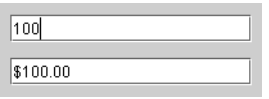


Figure 15 JCTextField with double validator and currency set showing edit (top) and display formats.

4.2.7 JCTextField with BigDecimal Validator

This field does not allow null values, so when the field is cleared, the `invalidPolicy` forces the field to display the previous valid value.

```
p.add(new JLabel("BigDecimal JCTextField: "));
p.add(text7 = new JCTextField());

// create validator and set its properties
JCBigDecimalValidator bdv = new JCBigDecimalValidator();
bdv.setAllowNull(false);

// set the invalidinfo properties
JCInvalidInfo bdii = text7.getInvalidInfo();
bdii.setInvalidPolicy(JCInvalidInfo.RESTORE_PREVIOUS);

// set the value model, validator, and invalidinfo
text7.setValueModel(new BigDecimalValueModel(new
   BigDecimal(100000.111)));
text7.setValidator(bdv);
text7.setInvalidInfo(bdii);
```

Figure 16 JCTextField with BigDecimal validator.

4.2.8 JCTextField with Float Validator

The `invalidPolicy` for this field forces it to clear when the user enters an invalid value.

```
p.add(new JLabel("Float JCTextField: "));
p.add(text8 = new JCTextField());

// create the validator and set its properties
JCFloatValidator fv = new JCFloatValidator();
fv.setRange(new Float((float)-10000), new Float((float)10000));
fv.setAllowNull(true);

// set the invalidinfo properties
JCInvalidInfo fii = text8.getInvalidInfo();
fii.setInvalidPolicy(JCInvalidInfo.CLEAR_FIELD);

// set the value model, validator, and invalidinfo
text8.setValidator(fv);
text8.setValueModel(new FloatValueModel(new Float(-3033.32)));
text8.setInvalidInfo(fii);
```

Figure 17 JCTextField with float validator.

4.2.9 JCTextField with DateTime Validator

This example allows the user to enter date values in several formats. Because the `maskInput` property is set to `false`, when the user enters a partial date that meets one of the allowed formats, the field attempts to complete the date.

```
p.add(new JLabel("DateTime(Calendar) JCTextField: "));
p.add(text9 = new JCTextField());

// create validator and set its properties
JCDateTimeValidator dtv = new JCDateTimeValidator();
dtv.setMaskInput(false);
dtv.setEditFormats(new String[] {"yyyy/MM/dd", "MMM d, yyyy"});
dtv.setAllowNull(true);

// set value model and validator
text9.setValueModel(new CalendarValueModel());
text9.setValidator(dtv);
```

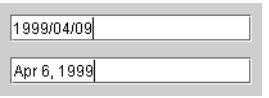


Figure 18 JCTextField with datetime validator showing two different edit formats.

4.2.10 JCTextField with Date Validator

The `format` property works as a mask for Date/Time validators. If you set `maskInput` to `true`, this field will only allow input that is in the format specified by the `format` property. It also prompts the user with place holder characters and uses the `casePolicy` property to convert all characters to uppercase.

```
p.add(new JLabel("Date JCTextField: "));
p.add(text10 = new JCTextField());

// create the validator and set its properties
JCDateValidator datev = new JCDateValidator();
datev.setFormat("MMM dd/yy");
datev.setMaskInput(true);
datev.setPlaceHolderChars("MMM DD/YY");
datev.setCasePolicy(JCDateValidator.UPPERCASE);
datev.setAllowNull(true);

// set value model and validator
text10.setValueModel(new DateValueModel());
text10.setValidator(datev);
```

Figure 19 JCTextField with date validator.

4.2.11 JCTextField with Time Validator

You use this field and validator combination to display and update time information. You can maintain a running clock if you wish. One way is to start a thread that sleeps for one second, then fires an event. You catch the event and update the time field using `setValue()`.

This example shows the `defaultDetail`'s `FULL` setting.

```
p.add(new JLabel("Time JCTextField: "));
p.add(text11 = new JCTextField());

// create the validator and set its properties
JCTimeValidator timev = new JCTimeValidator();
timev.setMaskInput(true);
timev.setDefaultDetail(JCTimeValidator.FULL);
timev.setAllowNull(false);

// set the invalidinfo properties
JCInvalidInfo timeii = text11.getInvalidInfo();
timeii.setInvalidPolicy(JCInvalidInfo.RESTORE_DEFAULT);

// set value model, validator, and invalidinfo
text11.setValueModel(new DateValueModel());
text11.setValidator(timev);
text11.setInvalidInfo(timeii);
```

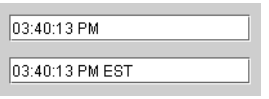


Figure 20 JCTextField with time validator showing default (top) and FULL display detail.

4.2.12 JCTextField with IP Address Validator

You use this field and validator combination to display IP addresses. The `setIPValidators()` method takes an array of `JCIntegerValidators` and uses their min and max values.

```
p.add(new JLabel("JCIPAddress JCTextField: "));
p.add(text12 = new JCTextField());

// create the validator and set its properties
JCIPAddressValidator ipv = new JCIPAddressValidator();
JCIntegerValidator[] validators = new JCIntegerValidator[4];
validators[0] = new JCIntegerValidator();
validators[0].setMin(new Integer(1));
validators[0].setMax(new Integer(128));
```

```
validators[1] = new JCIntegerValidator();
validators[1].setMin(new Integer(30));
validators[1].setMax(new Integer(50));
validators[2] = new JCIntegerValidator();
validators[2].setMin(new Integer(1));
validators[2].setMax(new Integer(10));
validators[3] = new JCIntegerValidator();
validators[3].setMin(new Integer(100));
validators[3].setMax(new Integer(200));
ipv.setIPValidators(validators);
// set value model and validator
text12.setValueModel(new IPAddressValueModel());
text12.setValidator(ipv);
text12.setValue(new JCIPAddress("121.35.2.150"));
```

Figure 21 JCTextField with IP address validator.

4.3 Examples of Spin Fields

The following code snippets are from *SpinFields.java* found in the *examples/field* directory. Run the examples with the command:

```
java examples.field.SpinFields
```

4.3.1 JCSpinField with String Validator

This example uses the `mask` property and place holder characters to provide clues about the kind of input the field is expecting.

String validators use `JCValidator.SPIN_WRAP` as the default spin policy.

```
p.add(new JLabel("String JCSpinField: "));
p.add(spin1 = new JCSpinField());

// create the validator and set its properties
JCStringValidator sv = new JCStringValidator();
String[] string_values = {"4165941026620", "8005551234567",
   "5195555941323"};
sv.setMask("(@@@)@@@-@@@@ Ext. @@@");
sv.setPlaceHolderChars("(___)___-____ Ext. ___");
sv.setAllowNull(true);
sv.setPickList(new JCListModel(string_values));

// set the value model and validator
spin1.setValueModel(new StringValueModel());
spin1.setValidator(sv);
```

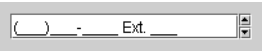


Figure 22 JCSpinField with String validator.

4.3.2 JCSpinField with Integer Validator

There is no display list associated with the pick list in this example. The pick list values themselves appear in the field. Since `matchPickList` is `true` by default, only four values are possible: 1, 2, 3, and 4. Any attempt by the user to type other values in the field will result in it being cleared.

```
p.add(new JLabel("Integer JCSpinField: "));
p.add(spin2 = new JCSpinField());

// create validator and set its properties
JCIntegerValidator iv = new JCIntegerValidator();
Integer[] int_values = {new Integer(1), new Integer(2),
   new Integer(3), new Integer(4)};
iv.setAllowNull(true);
iv.setPickList(new JCListModel(int_values));
iv.setSpinPolicy(JCIntegerValidator.SPIN_WRAP);

// create the invalidinfo and set its properties
JCInvalidInfo iii = spin2.getInvalidInfo();
iii.setInvalidPolicy(JCInvalidInfo.CLEAR_FIELD);

// set value model, validator, and invalidinfo
spin2.setValueModel(new IntegerValueModel());
spin2.setValidator(iv);
spin2.setInvalidInfo(iii);
```

Figure 23 JCSpinField with integer validator.

4.3.3 JCSpinField with Long Validator

This field uses the `displayList` property to associate the numeric field values in the pick list with text that will be displayed in the field. Notice that by default the top spin arrow is disabled when the last title in the array is reached.

```
p.add(new JLabel("Long JCSpinField: "));
p.add(spin3 = new JCSpinField());

// create validator and set its properties
JCLongValidator lv = new JCLongValidator();
Long[] long_values = {new Long(1), new Long(2), new Long(3),
   new Long(4)};
String[] long_display = {"Mr.", "Mrs.", "Ms.", "Miss", "Dr."};
lv.setMatchPickList(true);
lv.setAllowNull(true);
lv.setPickList(new JCListModel(long_values));
lv.setDisplayList(long_display);

// set the value model and validator
spin3.setValueModel(new LongValueModel());
spin3.setValidator(lv);
```

Figure 24 JCSpinField with long validator.

4.3.4 JCSpinField with Short Validator

In this example, the `matchPickList` property is set to `false`, so that the user is able to enter a value not contained in the pick list.

```
p.add(new JLabel("Short JCSpinField: "));
p.add(spin4 = new JCSpinField());

// create the validator and set its properties
JCShortValidator shv = new JCShortValidator();
Short[] short_values = {new Short((short)1), new Short((short)2),
   new Short((short)3), new Short((short)4)};
shv.setMatchPickList(false);
shv.setAllowNull(true);
shv.setPickList(new JCListModel(short_values));

// create the invalidinfo and set its properties
JCInvalidInfo shii = spin4.getInvalidInfo();
shii.setInvalidPolicy(JCInvalidInfo.RESTORE_DEFAULT);

// set value model, validator, and invalidinfo
spin4.setValueModel(new ShortValueModel());
spin4.setValidator(shv);
spin4.setInvalidInfo(shii);
```

Figure 25 JCSpinField with short validator.

4.3.5 JCSpinField with Byte Validator

Here, we set limits on the field using the `setRange()` method. Alternatively you can set limits in your IDE. The equivalent statements are:

```
bytev.setMin(0);
bytev.setMax(10);
```

The `setRange()` method makes the program slightly easier to maintain because the numerical limits are kept together in one statement.

The field also sets an invalid policy which turns the background red when the user enters a value that is out of range.

```
p.add(new JLabel("Byte JCSpinField: "));
p.add(spin5 = new JCSpinField());

// create the validator and set its properties
JCByteValidator bytev = new JCByteValidator();
bytev.setAllowNull(true);
```

```
bytev.setRange(new Byte((byte)0), new Byte((byte)10));

// create the invalidinfo and set its properties
JCInvalidInfo byteii = spin5.getInvalidInfo();
byteii.setInvalidBackground(Color.red);

// set the value model, validator, and invalidinfo
spin5.setValueModel(new ByteValueModel());
spin5.setValidator(bytev);
spin5.setInvalidInfo(byteii);
```

Figure 26 JCSpinField with byte validator.

4.3.6 JCSpinField with Double Validator

The `isCurrency` property in this field is set to `true` so the value will be treated as currency. The field also uses an increment value of five.

```
p.add(new JLabel("Double JCSpinField (currency): "));
p.add(spin6 = new JCSpinField());

// create validator and set its properties
JCDoubleValidator dv = new JCDoubleValidator();
dv.setAllowNull(true);
dv.setCurrency(true);
dv.setIncrement(new Double(5.0));

// set value and validator
spin6.setValueModel(new DoubleValueModel());
spin6.setValidator(dv);
```

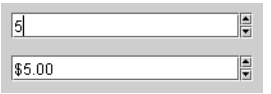


Figure 27 JCSpinField with double validator and currency set showing edit (top) and display formats.

4.3.7 JCSpinField with BigDecimal Validator

Setting the display pattern, as in this field, gives the user the context for the value entered.

```
p.add(new JLabel("BigDecimal JCSpinField: "));
p.add(spin7 = new JCSpinField());

// create validator and set its properties
JCBigDecimalValidator bdv = new JCBigDecimalValidator();
bdv.setAllowNull(false);
bdv.setDisplayPattern("0.00 inches");
bdv.setEditPattern("");

// create the invalidinfo and set its properties
```

```
JCInvalidInfo bdii = spin7.getInvalidInfo();
bdii.setInvalidPolicy(JCInvalidInfo.RESTORE_DEFAULT);

// set the value model, validator, and invalidinfo
spin7.setValueModel(new BigDecimalValueModel());
spin7.setValidator(bdv);
spin7.setInvalidInfo(bdii);
```

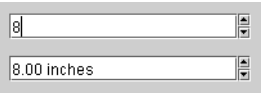


Figure 28 JCSpinField with BigDecimal validator showing edit (top) and display formats.

4.3.8 JCSpinField with Float Validator

This example sets the increment value to 0.1. The invalid policy will clear the field if the user enters an invalid value.

```
p.add(new JLabel("Float JCSpinField: "));
p.add(spin8 = new JCSpinField());

// create the validator and set its properties
JCFloatValidator fv = new JCFloatValidator();
fv.setIncrement(new Float(0.1));
fv.setAllowNull(true);

// create the invalidinfo and set its properties
JCInvalidInfo fii = spin8.getInvalidInfo();
fii.setInvalidPolicy(JCInvalidInfo.CLEAR_FIELD);

// set the value model, validator, and invalidinfo
spin8.setValidator(fv);
spin8.setValueModel(new FloatValueModel());
spin8.setInvalidInfo(fii);
```

Figure 29 JCSpinField with float validator.

4.3.9 JCSpinField with DateTime Validator

The default spin policy for date and time validators is `JCValidator.SPIN_SUBFIELD`, which allows the user to click a single set of arrow buttons to manipulate the subfields that comprise a complete date and time specification.

```
p.add(new JLabel("DateTime(Calendar) JCSpinField: "));
p.add(spin9 = new JCSpinField());

// create validator and set its properties
JCDateTimeValidator dtv = new JCDateTimeValidator();
dtv.setMaskInput(true);
dtv.setAllowNull(true);
```

```
// set value model and validator
spin9.setValueModel(new CalendarValueModel());
spin9.setValidator(dtv);
```

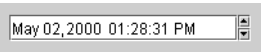


Figure 30 JCSpinField with datetime validator.

Note: The spin increment determines how many elements will be scrolled through for each spin.The spin increment can be set for a `JCSpinField` with Date validator to more than one, which is its default setting.

4.3.10 JCSpinField with Date Validator

The `format` property for date and time validators is useful for presenting the value of the field in a way that is familiar to a specific group of users.

```
p.add(new JLabel("Date JCSpinField: "));
p.add(spin10 = new JCSpinField());

// create the validator and set its properties
JCDateValidator datev = new JCDateValidator();
datev.setMaskInput(true);
datev.setFormat("MMMM d 'yy");

// set value model and validator
spin10.setValueModel(new DateValueModel());
spin10.setValidator(datev);
```

Figure 31 JCSpinField with date validator.

Note: The spin increment determines how many elements will be scrolled through for each spin.The spin increment can be set for a `JCSpinField` with Date validator to more than one, which is its default setting.

4.3.11 JCSpinField with Time Validator

A basic spin field with the time validator takes the current time as its default. This field presents the default time in full format.

```
p.add(new JLabel("Time JCSpinField: "));
p.add(spin11 = new JCSpinField());

// create the validator and set its properties
```

```
JCTimeValidator timev = new JCTimeValidator();
timev.setDefaultDetail(JCTimeValidator.FULL);

// create the invalidinfo and set its properties
JCInvalidInfo timeii = spin11.getInvalidInfo();
timeii.setInvalidPolicy(JCInvalidInfo.RESTORE_DEFAULT);

// set value model, validator, and invalidinfo
spin11.setValueModel(new DateValueModel());
spin11.setValidator(timev);
spin11.setInvalidInfo(timeii);
```

Figure 32 JCSpinField with time validator.

Note: The spin increment determines how many elements will be scrolled through for each spin.The spin increment can be set for a `JCSpinField` with Time validator to more than one, which is its default setting.

4.3.12 JCSpinField with IP Address Validator

You use this field and validator combination to display IP addresses. The `setIPValidators()` method takes an array of `JCIntegerValidators` and uses their min and max values.

```
p.add(new JLabel("JCIPAddress JCSpinField: "));
p.add(spin12 = new JCSpinField());

// create the validator and set its properties
JCIPAddressValidator ipv = new JCIPAddressValidator();
JCIntegerValidator[] validators = new JCIntegerValidator[4];
validators[0] = new JCIntegerValidator();
validators[0].setMin(new Integer(1));
validators[0].setMax(new Integer(128));
validators[1] = new JCIntegerValidator();
validators[1].setMin(new Integer(30));
validators[1].setMax(new Integer(50));
validators[2] = new JCIntegerValidator();
validators[2].setMin(new Integer(1));
validators[2].setMax(new Integer(10));
validators[3] = new JCIntegerValidator();
validators[3].setMin(new Integer(100));
validators[3].setMax(new Integer(200));
ipv.setIPValidators(validators);

// set value model and validator
spin12.setValueModel(new IPAddressValueModel());
spin12.setValidator(ipv);
spin12.setValue(new JCIPAddress("121.35.2.150"));
```

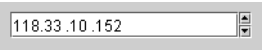


Figure 33 JCSpinField with IP address validator.

4.4 Examples of Combo Fields

The following code snippets are from *ComboFields.java* found in the *examples/field* directory. Run the examples using the command:

```
java examples.field.ComboFields
```

4.4.1 JCComboField with String Validator

This field has `matchPickList` set to `true`. Because users might have their own unique honorific (such as Lord or Count), you may want to add new entries to the pick list. To do this you would set `matchPickList` to `false` and write code to add the user's typed entry to the pick list. An example is shown in Section 4.7, Event Programming.

```
p.add(new JLabel("String JCComboField: "));
p.add(combo1 = new JCComboField());

// create the validator and set its properties
JCStringValidator sv = new JCStringValidator();
String[] string_values = {"Mr.", "Mrs.", "Ms.", "Miss", "Dr."};
sv.setMatchPickList(true);
sv.setAllowNull(true);
sv.setPickList(new JCListModel(string_values));

// set the value model and validator
combo1.setValueModel(new StringValueModel());
combo1.setValidator(sv);
```

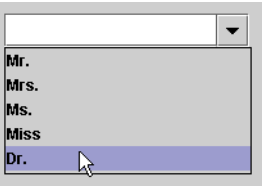


Figure 34 JCComboField with String validator.

4.4.2 JCComboField with Integer Validator

The `displayList` property is useful whenever you wish to present the user with a selection of items that are internally associated with ordinal numbers, perhaps for database applications. Note that the associated String value is displayed in the field, not its numerical value, even when the field loses focus.

```
p.add(new JLabel("Integer JCComboField: "));
p.add(combo2 = new JCComboField());

// create validator and set its properties
```

```
JCIntegerValidator iv = new JCIntegerValidator();
Integer[] integer_values = {new Integer(1), new Integer(2),
   new Integer(3), new Integer(4)};
String[] integer_display = {"apple", "banana", "orange", "pear"};
iv.setMatchPickList(true);
iv.setAllowNull(true);
iv.setPickList(new JCListModel(integer_values));
iv.setDisplayList(integer_display);

// create the invalidinfo and set its properties
JCInvalidInfo iii = combo2.getInvalidInfo();
iii.setInvalidPolicy(JCInvalidInfo.CLEAR_FIELD);

// set the value model, validator, and invalidinfo
combo2.setValueModel(new IntegerValueModel());
combo2.setValidator(iv);
combo2.setInvalidInfo(iii);
```

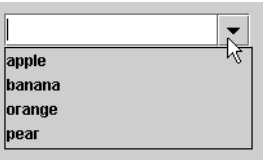


Figure 35 JCComboField with integer validator.

4.4.3 JCComboField with Long Validator

This example allows the user to choose an astrological sign. Since there are only 12 astrological signs, it makes sense that `matchPickList` is set to `true`.

```
p.add(new JLabel("Long JCComboField: "));
p.add(combo3 = new JCComboField());

// create validator and set its properties
JCLongValidator lv = new JCLongValidator();
Long[] long_values = {new Long(1), new Long(2), new Long(3),
   new Long(4), new Long(5), new Long(6), new Long(7),
   new Long(8), new Long(9), new Long(10), new Long(11),
   new Long(12)};
String[] long_display = {"Aries", "Taurus", "Gemini", "Cancer",
   "Leo", "Virgo", "Libra", "Scorpio", "Sagittarius",
   "Capricorn", "Aquarius", "Pisces"};
lv.setPickList(new JCListModel(long_values));
lv.setDisplayList(long_display);
lv.setMatchPickList(true);
lv.setAllowNull(true);

// set the value model and validator
combo3.setValueModel(new LongValueModel());
combo3.setValidator(lv);
```

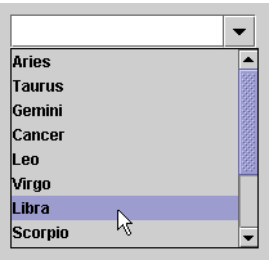


Figure 36 JCComboField with long validator.

4.4.4 JCComboField with Short Validator

In this example, the `matchPickList` property is set to `false`, so that the user is able enter a value not contained in the pick list.

```
p.add(new JLabel("Short JCComboField: "));
p.add(combo4 = new JCComboField());

// create the validator and set its properties
JCShortValidator shv = new JCShortValidator();
Short[] short_values = {new Short((short)1), new Short((short)2),
   new Short((short)3), new Short((short)4)};
shv.setMatchPickList(false);
shv.setAllowNull(true);
shv.setPickList(new JCListModel(short_values));

// set the value model and validator
combo4.setValueModel(new ShortValueModel());
combo4.setValidator(shv);
```

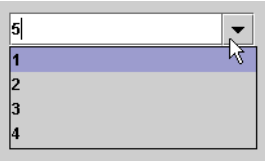


Figure 37 JCComboField with short validator.

4.4.5 JCComboField with Byte Validator

Setting the display pattern in a combo field allows the user to see the context of the value in the drop-down list.

```
p.add(new JLabel("Byte JCComboField: "));
p.add(combo5 = new JCComboField());

// create the validator and set its properties
JCByteValidator bytev = new JCByteValidator();
Byte[] byte_values = {new Byte((byte)10), new Byte((byte)20),
   new Byte((byte)30), new Byte((byte)40)};
```

```
bytev.setDisplayPattern("0 feet");
bytev.setEditPattern("");
bytev.setAllowNull(true);
bytev.setPickList(new JCListModel(byte_values));

// set the value model and validator
combo5.setValueModel(new ByteValueModel());
combo5.setValidator(bytev);
```

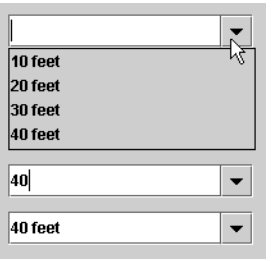


Figure 38 JCComboField with byte validator: dropdown list (top), edit format (middle), display format (bottom).

4.4.6 JCComboField with Double Validator

This example uses the `isCurrency` property to indicate the value is a currency amount. The pick list values are displayed in the default currency format for the present locale.

```
p.add(new JLabel("Double JCComboField (currency): "));
p.add(combo6 = new JCComboField());

// create validator and set validator properties
JCDoubleValidator dv = new JCDoubleValidator();
Double[] double_values = {new Double(100), new Double(200),
   new Double(300), new Double(400)};
dv.setAllowNull(true);
dv.setCurrency(true);
dv.setPickList(new JCListModel(double_values));

// set value model and validator
combo6.setValueModel(new DoubleValueModel());
combo6.setValidator(dv);
```

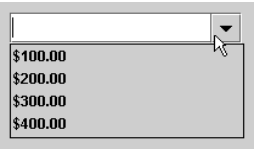


Figure 39 JCComboField with double validator.

4.4.7 JCComboField with BigDecimal Validator

This example shows how the `invalidPolicy`, `JCInvalidInfo.RESTORE_DEFAULT` forces the field to display the default value after the user attempts to enter an invalid number.

```
p.add(new JLabel("BigDecimal JCComboField: "));
p.add(combo7 = new JCComboField());

// create validator and set its properties
JCBigDecimalValidator bdv = new JCBigDecimalValidator();
BigDecimal[] bigdecimal_values = {new BigDecimal(10.0),
new BigDecimal(20.0), new BigDecimal(30.0), new BigDecimal(40.0)};
bdv.setDefaultValue(new BigDecimal(-1));
bdv.setAllowNull(false);
bdv.setPickList(new JCListModel(bigdecimal_values));

// create the invalidinfo and set its properties
JCInvalidInfo bdii = combo7.getInvalidInfo();
bdii.setInvalidPolicy(JCInvalidInfo.RESTORE_DEFAULT);
```

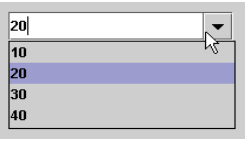


Figure 40 JCComboField with BigDecimal validator.

4.4.8 JCComboField with Float Validator

The `invalidPolicy` for this field forces it to clear when the user enters an invalid value.

```
p.add(new JLabel("Float JCComboField: "));
p.add(combo8 = new JCComboField());

// create the validator and set its properties
JCFloatValidator fv = new JCFloatValidator();
Float[] float_values = {new Float((float)100.101),
   new Float((float)200.202), new Float((float)300.303),
   new Float((float)400.404)};
fv.setAllowNull(true);
fv.setPickList(new JCListModel(float_values));
fv.setMatchPickList(true);

// create the invalidinfo and set its properties
JCInvalidInfo fii = combo8.getInvalidInfo();
fii.setInvalidPolicy(JCInvalidInfo.CLEAR_FIELD);

// set the value model, validator, and invalidinfo
combo8.setValidator(fv);
combo8.setValueModel(new FloatValueModel());
combo8.setInvalidInfo(fii);
```

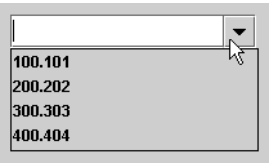


Figure 41 JCComboField with float validator.

4.4.9 JCComboField with IP Address Validator

You use this field and validator combination to display IP addresses.

```
p.add(new JLabel("JCIPAddress JCComboField: "));
p.add(combo9 = new JCComboField());

// create the validator and set its properties
JCIPAddressValidator ipv = new JCIPAddressValidator();
JCIPAddress[] ip_values = new JCIPAddress[3];
ip_values[0] = new JCIPAddress("0.0.0.0");
ip_values[1] = new JCIPAddress("24.190.120.3");
ip_values[2] = new JCIPAddress("123.10.3.15");
ipv.setPickList(new JCListModel(ip_values));

// set value model and validator
combo9.setValueModel(new IPAddressValueModel());
combo9.setValidator(ipv);
combo9.setValue(new JCIPAddress("121.35.2.150"));
```

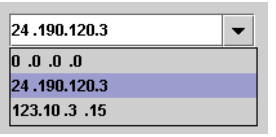


Figure 42 JCComboField with IP address validator.

4.5 Examples of Popup Fields

The following code snippets are from *PopupFields.java* found in the *examples/field* directory. Run the examples with the command:

```
java examples.field.PopupFields
```

4.5.1 JCPopupField with DateTime Validator

In this example you can spin the year, month, and time fields and select the date from the calendar display. This field also uses the `format` property to present the selected date and time in a suitable format.

```
p.add(new JLabel("Date Time JCPopupField: "));
p.add(popup1 = new JCPopupField());
```

```
// create the validator and set the validator properties
JCDateTimeValidator dtv = new JCDateTimeValidator();
dtv.setAllowNull(true);
dtv.setFormat("MMM d 'yy H:mm:ss");

// set the value model and validator
popup1.setValueModel(new CalendarValueModel(
   Calendar.getInstance()));
popup1.setValidator(dtv);
```

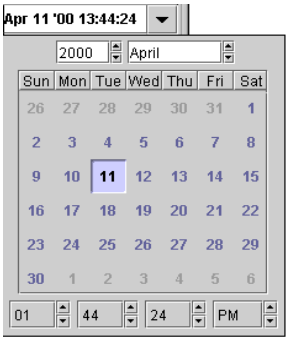


Figure 43 JCPopupField with datetime validator.

Note: If the format for the `JCDateTimeValidator` specifies the use of military hours (i.e. hours ranging from 0-23), the hour spinner in the popup field will also use military hours.

4.5.2 JCPopupField with Date Validator

Once the user selects the date, the value is displayed in the field with `defaultDetail` set to `JCValidator.LONG` and the `casePolicy` set to `JCValidator.UPPERCASE`.

```
p.add(new JLabel("Date JCPopupField: "));
p.add(popup2 = new JCPopupField());

// create the validator and set the validator properties
JCDateValidator dv = new JCDateValidator();
dv.setAllowNull(true);
dv.setDefaultDetail(JCDateValidator.LONG);
dv.setCasePolicy(JCDateValidator.UPPERCASE);

// set the value model and validator
popup2.setValueModel(new DateValueModel(new Date()));
popup2.setValidator(dv);
```

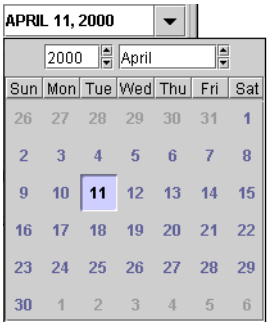


Figure 44 JCPopupField with date validator.

4.6 Examples of Label Fields

The following code snippets are from *LabelFields.java* found in the *examples/field* directory. Run the examples using the command:

```
java examples.field.LabelFields
```

4.6.1 JCLabelField with String Validator

This example demonstrates the effect of using the mask property.

```
p.add(new JLabel("String JCLabelField: "));
p.add(label1 = new JCLabelField());

// create the validator and set its properties
JCStringValidator sv = new JCStringValidator();
sv.setMask("(@@@)@@@-@@@@ Ext. @@@");
sv.setAllowNull(true);


// set the value model and validator
label1.setValueModel(new StringValueModel());
label1.setValidator(sv);
label1.setValue("4165941026");
```

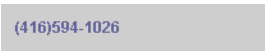


Figure 45 JCLabelField with String validator.

4.6.2 JCLabelField with Integer Validator

This example demonstrates how the `displayPattern` property determines the format of the field.

```
p.add(new JLabel("Integer JCLabelField: "));
p.add(label2 = new JCLabelField());
```

```
// create validator and set its properties
JCIntegerValidator iv = new JCIntegerValidator();
iv.setAllowNull(true);
iv.setDisplayPattern("0 inches");

// set the value model and validator
label2.setValueModel(new IntegerValueModel());
label2.setValidator(iv);
label2.setValue(new Integer(100));
```

Figure 46 JCLabelField with integer validator.

4.6.3 JCLabelField with Long Validator

This field displays a long value.

```
p.add(new JLabel("Long JCLabelField: "));
p.add(label3 = new JCLabelField());

// create validator and set its properties
JCLongValidator lv = new JCLongValidator();
lv.setAllowNull(true);

// set the value model and validator
label3.setValueModel(new LongValueModel(new Long(1000000000000)));
label3.setValidator(lv);
```

Figure 47 JCLabelField with long validator.

4.6.4 JCLabelField with Short Validator

This example illustrates the use of a validator to provide a visual warning to the user when the field is invalid, that is when the field contains a value that is out of the set acceptable range.

```
p.add(new JLabel("Short JCLabelField: "));
p.add(label4 = new JCLabelField());

// create the validator and set its properties
JCShortValidator shv = new JCShortValidator();
shv.setAllowNull(true);
shv.setRange(new Short((Short)0), new Short((Short)10));

// set the invalid info properties
JCInvalidInfo shii = text4.getInvalidInfo();
shii.setInvalidBackground(Color.red);

// set value model, validator, and invalidinfo
```

```
label4.setValueModel(new ShortValueModel(new Short((Short)10)));
label4.setValidator(shv);
label4.setInvalidInfo(shii);
```

Figure 48 JCLabelField with short validator is given an invalid entry.

4.6.5 JCLabelField with Byte Validator

This example shows how the `invalidPolicy`, `JCInvalidInfo.RESTORE_DEFAULT` forces the field to display the default value after the field receives a number that is out of range.

```
p.add(new JLabel("Byte JCLabelField: "));
p.add(label5 = new JCLabelField());

// create the validator and set its properties
JCByteValidator bytev = new JCByteValidator();
bytev.setDefaultValue(new Byte((Byte) 5));
bytev.setAllowNull(true);
bytev.setRange(new Byte((Byte) 1), new Byte((Byte) 10));

// set the invalidinfo properties
JCInvalidInfo byteii = text5.getInvalidInfo();
byteii.setInvalidPolicy(JCInvalidInfo.RESTORE_DEFAULT);

// set the value model, validator, and invalidinfo
label5.setValueModel(new ByteValueModel(new Byte(5));
label5.setValidator(bytev);
label5.setInvalidInfo(byteii);
label5.setValue(new Byte("11"));
```

Figure 49 JCLabelField with byte validator showing default value.

4.6.6 JCLabelField with Double Validator

The double validator associated with this text field is augmented by the `isCurrency` property so that the value is treated as currency. The display format uses the currency format associated with the current locale.

```
p.add(new JLabel("Double JCLabelField (currency): "));
p.add(label6 = new JCLabelField());

// create validator and set its properties
JCDoubleValidator dv = new JCDoubleValidator();
dv.setAllowNull(true);
dv.setCurrency(true);

// set value and validator
label6.setValueModel(new DoubleValueModel(new Double(100.00)));
label6.setValidator(dv);
```

Figure 50 JCLabelField with double validator and currency set.

4.6.7 JCLabelField with BigDecimal Validator

This field displays a BigDecimal type value.

```
p.add(new JLabel("BigDecimal JCLabelField: "));
p.add(label7 = new JCLabelField());

// create validator and set its properties
JCBigDecimalValidator bdv = new JCBigDecimalValidator();
bdv.setAllowNull(true);

// set the value model and validator
label7.setValueModel(new BigDecimalValueModel());
label7.setValidator(bdv);
label7.setValue(new BigDecimal("100000000.111"));
```

Figure 51 JCLabelField with BigDecimal validator.

4.6.8 JCLabelField with Float Validator

This field displays a float data type value.

```
p.add(new JLabel("Float JCLabelField: "));
p.add(label8 = new JCLabelField());

// create the validator and set its properties
JCFloatValidator fv = new JCFloatValidator();
fv.setAllowNull(true);

// set the value model and validator
label8.setValidator(fv);
label8.setValueModel(new FloatValueModel());
label8.setValue(new Float("1000.0000"));
```


Figure 52 JCLabelField with float validator.

4.6.9 JCLabelField with DateTime Validator

This example shows date and time values. The `setValue()` method gives the field the current date and time as its initial value.

```
p.add(new JLabel("DateTime(Calendar) JCLabelField: "));
p.add(label9 = new JCLabelField());

// create validator and set its properties
JCDateTimeValidator dtv = new JCDateTimeValidator();
```

```
dtv.setMaskInput(true);
dtv.setAllowNull(true);

// set value model and validator
label9.setValueModel(new CalendarValueModel());
label9.setValidator(dtv);
label9.setValue(Calendar.getInstance());
```

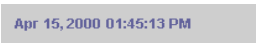


Figure 53 JCLabelField with datetime validator.

4.6.10 JCLabelField with Date Validator

The `format` property for date and time validators is useful for presenting the value of the field in a way that is familiar to a specific group of users.

The `format` property works as a mask for Date/Time validators. The field also uses the `casePolicy` property to convert all characters to uppercase.

```
p.add(new JLabel("Date JCLabelField: "));
p.add(label10 = new JCLabelField());

// create the validator and set its properties
JCDateValidator datev = new JCDateValidator();
datev.setMaskInput(true);
datev.setFormat("MMMM d 'yy");
datev.setCasePolicy(JCDateValidator.UPPERCASE);

// set value model and validator
label10.setValueModel(new DateValueModel());
label10.setValidator(datev);
label10.setValue(new Date());
```


Figure 54 JCLabelField with date validator.

4.6.11 JCLabelField with Time Validator

You use this field and validator combination to display and update time information. You can maintain a running clock if you wish. One way is to start a thread that sleeps for one second, then fires an event. You catch the event and update the time field using `setValue()`.

This example shows the `defaultDetail`'s `FULL` setting.

```
p.add(new JLabel("Time JCLabelField: "));
p.add(label11 = new JCLabelField());

// create the validator and set its properties
JCTimeValidator timev = new JCTimeValidator();
timev.setMaskInput(true);
timev.setDefaultDetail(JCTimeValidator.FULL);
```

```
timev.setAllowNull(false);

// set value model and validator
label11.setValueModel(new DateValueModel());
label11.setValidator(timev);
label11.setValue(new Date());
```

Figure 55 JCLabelField with time validator.

4.6.12 JCLabelField with IP Address Validator

You use this field and validator combination to display IP addresses.

```
p.add(new JLabel("JCIPAddress JCLabelField: "));
p.add(label12 = new JCLabelField());

// create the validator and set its properties
JCIPAddressValidator ipv = new JCIPAddressValidator();

// set value model and validator
label12.setValueModel(new IPAddressValueModel());
label12.setValidator(ipv);
label12.setValue(new JCIPAddress("121.35.2.150"));
```

Figure 56 JCLabelField with IP address validator.

4.7 Event Programming

A class can be notified both before and after a field's value has changed by implementing `com.klg.jclass.util.value.JCValueListener` and registering itself with the field via `addValueListener()`. In this code snippet, `combo` is an instance of an editable `JCComboField` with a `JCStringValidator`. If the user types a new value into the field instead of choosing one of the values in the combo field, the code shown below adds the new information to the pick list.

First, the line of code that registers the field with the listener:

```
combo.addValueListener(this);
```

Now the event handling code:

```
public void valueChanged(JCValueEvent e) {
// Gets the contents of the combo box's text field
   String newValue = ((String) combo.getValue()).trim();
   boolean found = false;
   int position = 0;
   // Make sure there is something in the text field
   if(newValue != null && newValue.length() > 0){
```

```
        for (int i = 0; i < dm.getSize(); i++){
        // See if the pick list contains an item matching the text field
        if (newValue.compareTo((String)dm.getElementAt(i)) == 0) {
            found = true;
        } else {
            // Set the insertion point for a new item
            if (newValue.compareToIgnoreCase(
                 (String)dm.getElementAt(i)) > 0) {
                position = i + 1;
            }
         }
     }
    }
    // Add a new item to the data model
    if (!found && newValue != null && newValue.length() > 0) {
        dm.add(position, (String) newValue);
        combo.setPickList(dm);
        combo.setSelectedIndex(position);
    }
}
```

Items may be appended to the list in the combo field with autocomplete off or on. It is recommended that the append mode in autocomplete be turned off because end users may find interaction with the text field awkward.

Removing items from a `JCComboField`'s list model does not require implementing the `JCValueListener` interface. All that is required is a reference to the combo box's list model and a core Java listener. For instance, if you provide a button that allows the end user to remove the item currently in the text field, the code might be something like:

```
JButton removeButton = new JButton("Remove Selection");
removeButton.addActionListener(this);
...
public void actionPerformed(ActionEvent e){
    String newValue = null;
     if (combo.getValue() != null){
        newValue = ((String) combo.getValue()).trim();
     }
     boolean found = false;
     int position = 0;
     if(newValue != null && newValue.length() > 0){
        for (int i = 0;i < dm.getSize();i++){
        if (newValue.compareTo((String)dm.getElementAt(i)) == 0) {
            found = true;
            position = i;
            dm.removeElementAt(i);
            combo.setPickList(dm);
            combo.setValue("");
            break;
        }
    }
    }
}
```

The following figure shows how an item may be removed with the use of a button whose action listener uses the code shown above.

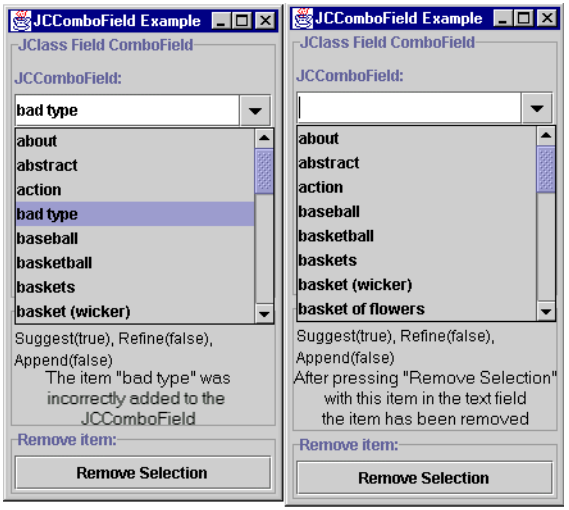


Figure 57 A combo field before and after the removal of an item.

The combo field shown on the left of Figure 57 illustrates the addition of an unwanted item. A user has typed the words “bad type” and has pressed the *Enter* key, thus adding the entry to the combo field’s list model. Realizing the error, the user has pressed the *Remove Selection* button. The item in the text box is removed from the list, no longer appearing in the right-hand combo field of Figure 57.

Part II

Reference Appendices

Appendix A

JClass Field Property Listings

The following is a listing of most of the available properties in JClass Field and their default values. The properties are arranged alphabetically by property name. The second entry on any given row details the group or groups for which the property is appropriate. The third entry names the data type of the method's argument. A small number of properties are read-only variables, and therefore only have a *get* method. These properties are marked with a "(G)" following their property name. There is also one property that has only a *set* method. It is marked with an "(S)" following the property name.

For a list of properties categorized by validator, component, and invalid, see Property Summaries, in Chapter 2.

Property	JCField Group	Type	Default
about (G)	**All**	**String**	**JClass Field X.Y.Z**
	A read-only variable that contains the version number for this release of JClass Field. This read-only property is supplied as a convenience function.		
allowNull	**All**	**boolean**	**false**
	Describes whether or not a null value is to be interpreted as a valid value. See state (G).		
background	**All**	**Color**	**inherited**
	The background color of the field. Typically, the color is `lightGray`.		
beepOnInvalid	**All**	**boolean**	**true**
	If `beepOnInvalid` is `true` then the field will beep whenever the state is switched to `INVALID`.		

Property	JCField Group	Type	Default
casePolicy	**Date/Time String**	**integer**	**JCValidator.AS_IS**

Determines the case of the characters displayed in the field. When the case policy is set to `JCValidator.AS_IS`, typed characters are left alone, while in the other two cases typed input is converted as required. The possible property values are listed below with their corresponding meanings.

- `JCValidator.AS_IS` – Leave characters as entered.
- `JCValidator.UPPERCASE` – Convert characters to upper case.
- `JCValidator.LOWERCASE` – Convert characters to lower case.

columnName	**Data bound**	**String**	**null**

Specifies the column in the data source to which the field is bound. **Borland JBuilder** only.

continuousScroll	**All**	**boolean**	**false**

Determines how selection is handled when the mouse button is held down on a spin arrow button. If `continuousScroll` is `true`, the component scrolls continuously through the items in the scroll box until the mouse button is released. If `continuousScroll` is `false`, a separate mouse click is required to select the next item in the scroll box.

currency	**Numeric**	**boolean**	**false**

Controls whether the value in a numeric field is treated as currency.

currencyLocale	**Numeric**	**Locale**	***locale dependent***

The currency locale controls the display of currency in a numeric field by using the currency formatting conventions of the given locale.

currencySymbol (G)	**Numeric**	**String**	***locale dependent***

A read-only variable that contains the currency symbol used in the given currency locale.

dataBinding	**Data bound**	**String**	**null**

Binds the field to a data source component. The value specifies the name of the data source component combined with the table column to bind to. The format of the value is specific to each type of data source. **JClass DataSource** only.

Property	JCField Group	Type	Default
dataSet	**Data bound**	**borland.jbcl.dataset.DataSet**	**null**

Binds the field to a data source component.
Borland JBuilder only.

defaultDetail	**Date/Time**	**integer**	**JCCalendarValidator.MEDIUM**

Specifies the detail level of the *default* format for date/time validators. Has no effect when the format property has been changed. The possible property values are listed below with examples of their display.

```
JCCalendarValidator.FULL — Fri, Apr 30, 1999 01:33:05 PM EST
JCCalendarValidator.LONG — Apr 30, 1999 01:34:21 PM EST
JCCalendarValidator.MEDIUM — Apr 30, 1999 01:35:08 PM
JCCalendarValidator.SHORT — 04/30/99 01:35 PM
```

defaultEditFormats (G)	**Date/Time**	**String**	***locale dependent***

This property contains the edit format determined by the locale, for displaying date and time values.

defaultFormats (G)	**Date/Time**	**String**	***locale dependent***

This property contains formats determined by the locale, for date and time values.

defaultValue	**All**	**Object**	**null**

The default value of the field which is used if `invalidPolicy`'s value is `JCInvalidInfo.RESTORE_DEFAULT`. (See invalidPolicy.)

displayList	**byte, short, integer, long**	**String**	**null**

Given an array of Strings, the `setDisplayList()` method associates the elements of the array with the corresponding integers in the pick list. (See pickList) The end-user sees String-type choices in the combo field or spin field, yet the value returned by `getValue()` method is that of the associated integer.

Property	JCField Group	Type	Default
displayPattern	**Numeric**	**String**	***locale dependent***
Method to set a pattern on the Decimal Format object used by the validator. This display format is in effect when the field does not have focus. See Number Format Characters. (See also editPattern.)			
doubleBuffered	**All**	**boolean**	**true**
Controls whether double-buffering is used when displaying and updating the component.			
editFormats	**Date/Time**	**String**	***locale dependent date/time***
A list of Strings that are used in an attempt to match the user's input to date and time formats. (See Date Formats.) Given an incomplete String, the calendar validator attempts to fill in the rest. If the validation fails, the field is marked as invalid.			
editPattern	**Numeric**	**String**	**Byte, Short, Integer, Long: 0 Float, Double, BigDecimal: 0.###**
The display formatting pattern used when the field has focus. A number's format may be different while it is being edited from the format that is used when the field loses focus. An example is allowing the end-user to type in a leading hyphen (minus sign) to denote a negative number in a financial application, yet showing a bracketed number when editing is complete. (See displayPattern.)			
enabled	**All**	**boolean**	**true**
Inherited from `awt.Component`, enables or disables this component, depending on the value of the boolean parameter.			
firstValidCursor Position (G)	**All**	**integer**	***mask dependent***
This read-only variable contains the number corresponding to the first space in a field in which a user can enter data. Its value will vary depending on the mask set.			
font	**All**	**Font**	***inherited***
Specifies the font of the component.			
foreground	**All**	**Color**	***inherited***
Controls the foreground color of the field.			

Property	JCField Group	Type	Default
format	**Date/Time**	**String**	***locale dependent date/time***

Controls the format currently being used to display the date/time values. The format String uses these conventions:

Symbol(s)	Meaning
y	Year within the current century (1 or 2 digits).
yy	Year within the current century (2 digits).
yyyy	Year including century (4 digits).
M	Numeric month of year (1 or 2 digits).
MM	Numeric month of year (2 digits).
MMM	Abbreviated month name.
MMMM	Full month name.
EE	Day of the Week (abbreviated).
EEEE	Day of the Week (full name).
d	Numeric day of month (1 or 2 digits).
dd	Numeric day of month (2 digits).
h	Hour of day (1-12) (1 or 2 digits).
hh	Hour of day (1-12) (2 digits).
H	Hour of day (0-23) (1 or 2 digits).
HH	Hour of day (0-23) (2 digits).
m	Minutes (1 or 2 digits).
mm	Minutes (2 digits).
s	Seconds (1 or 2 digits).
ss	Seconds (2 digits).
a	AM/PM representation.
z	Time zone abbreviation.
zz	Time zone abbreviation.
zz	Time zone (full name).
D	Day in year (1, 2, or 3 digits).

Property	JCField Group	Type	Default
	DDD	Day in year (3 digits).	
increment	**Numeric, Date/Time**	**Number**	**Byte, Short, Integer, Long: 1 Float, Double, BigDecimal: 1.0**
	Controls the amount by which to increment or decrement the field's value with each click on the spinner. The increment must be a non-zero positive number.		
invalidBackground	**All**	**Color**	***inherited***
	Controls the background color used in the visual component if the field is invalid. By default, this value is inherited from the background color of the component.		
invalidChars	**String**	**String**	**null**
	Describes a `String` of characters which are not allowed to be used as input in the current field. There is a associated property called `validChars`. Use this one if the list of invalid characters is shorter than the set of valid characters. (See validChars.)		
invalidForeground	**All**	**Color**	***inherited***
	Controls the foreground color used in the visual component if the field is invalid. By default this value is inherited from the foreground color of the component.		
invalidPolicy	**All**	**integer**	**JCField.SHOW_INVALID**
	The `invalidPolicy` governs what happens when a user enters invalid data into a field. The possible property values are listed below with their corresponding meanings. `JCInvalidInfo.SHOW_INVALID` – Shows invalid values, using `invalidBackground` and `invalidForeground` colors. `JCInvalidInfo.RESTORE_DEFAULT` – Restores the default value. (See defaultValue.) `JCInvalidInfo.RESTORE_PREVIOUS` – Restores the value to the field's previous valid value. `JCInvalidInfo.CLEAR_FIELD` – Clears the field if given invalid input.		
IPValidators	**IP Address**	**JCInteger Validator**	**null**
	Specifies the validators used for each subfield of the IP. For example: xxx.xxx.xxx.xxx You can associate one validator for each subfield.		

Property	JCField Group	Type	Default
locale	**All**	**Locale**	**locale.getDefault()**

Controls the display of time and date values according to the given locale.

Property	JCField Group	Type	Default
mask	**String, Date/Time**	**String**	**null**

The mask to be used to validate a String field.

Symbol	Meaning
#	Any digit, minus sign, comma, decimal point, or plus sign.
@	Any digit.
H	Any hexadecimal digit.
U	Any alphabetic character. Lower case characters will be converted to upper case.
L	Any alphabetic character. Upper case characters will be converted to lower case.
A	Any alphabetic character. No case conversion.
*	Any character.
^	Any alphanumeric character, that is, any character from the set {0-9a-zA-Z}.
\\	The next character that follows is to be treated as a literal, even if it is one of the above characters.

Property	JCField Group	Type	Default
maskChars	**String, Date/Time**	**String**	**"#@HULA*^\\"**

Use this property to reassign the mask characters. The meaning assigned to a character at a given position remains the same, but the character used to designate that meaning changes.

Example: `setMaskChars("!9HUXA*^\\")` remaps the mask characters so that an exclamation point(!) represents the extended digit, 9 represents a digit, and X represents a lower case character. All other mask characters remain the same. Note that you must use a mapping String that has the same number of characters as the default mask. For the meaning of the mask characters, see the table in the Mask Characters section.

Property	JCField Group	Type	Default
maskInput	**Date/Time**	**boolean**	**false**
If `maskInput` is `true`, the user is required to enter characters that conform exactly to the specified format. Some Java date formats are ambiguous. As part of its operation, JClass Field expands any ambiguous pattern it encounters into an internal pattern in which the ambiguity is removed.			
matchPickList	**All**	**boolean**	**true**
Controls whether values must match those in the pick list of the validator or not. The default is `true`, but this property is only applicable if there are elements stored in the pick list. (See pickList.)			
max	**Numeric**	**Numeric**	***type dependent***
Controls the maximum possible value of the numeric object currently being checked by the validator.			
maximumSize	**All**	**Dimension**	***dynamic***
The maximum size of the field.			
milleniumThreshold	**Date/Time**	**integer**	**69**
Controls the interpretation of two-digit years. Any two-digit date less than the threshold is considered to be after the year 2000 while any value greater than or equal to the threshold is considered to be after the year 1900. The default is 69 so that, for example, '96 is treated as 1996 and '10 is treated as 2010.			
min	**Numeric**	**Numeric**	***type dependent***
Controls the minimum possible value of the numeric object currently being checked by the validator.			
minimumSize	**All**	**Dimension**	***dynamic***
The minimum size of the field.			
name	**All**	**String**	***variable***
Gives a name to the component.			
numMaskMatch	**String**	**integer**	**-1**
Controls the number of characters to match with the mask from left to right. This number does not include any literals. If the value is -1, the entire mask will be matched.			

Property	JCField Group	Type	Default
preferredSize	**All**	**Dimension**	***dynamic***
The preferred size of the field.			
pickList	**All**	**ListModel**	**null**
A list of valid values for the field. If used in conjunction with `matchPickList` set to `true`, it represents the *only* valid values for the field. (See matchPickList)			
pickListIndex (G)	**All**	**Object**	***N/A***
The get method for this property returns the list of entries in the pick list of a given field.			
placeHolderChars	**Date/Time String**	**String**	**null**
Describes the place holder String, which specifies the prompt characters to be used in place of blanks (spaces) in a masked field. If the place holder character String is null, or if an empty character exists after the number of characters provided, then the field uses a space character. **Note:** Use the `placeHolderChars` property with date and time validators only if `maskInput` is `true` and you know the exact format being used. (The format for a date object can be ambiguous because a format like *h:mm:ss* is expanded internally to *hh:mm:ss*.)			
range (S)	**Numeric**	**integer**	***type dependent***
This property allows you to set both min and max at the same time. Use the get methods for min and max to return the values that determine the range.			
required	**All**	**boolean**	**true**
Controls whether a field on a given form must have a valid value to before the form can be submitted.			
selectOnEnter	**All**	**boolean**	**false**
Controls whether the value in a field is selected upon the field gaining focus.			
size	**All**	**Dimension**	***dynamic***
The width and height dimensions in pixels of the component.			

Property	JCField Group	Type	Default
spinPolicy	**All**	**integer**	***validator dependent***

Controls the action of the spin-arrow buttons. The possible property values are listed below with their corresponding meanings.

`JCValidator.SPIN_FIELD` – Allows spinning up and down between the maximum and minimum values. (Default for numeric validators.)

`JCValidator.SPIN_SUBFIELD` – Allows context sensitive spinning if it is allowed. (Default for date and time validators.)

`JCValidator.SPIN_WRAP` – Like spin field but allows continuous spinning. A value wraps from its maximum value to its minimum when spinning in the "up" direction, and from minimum to maximum in the other direction. (Default for String validators.)

state (G)	**All**	**integer**	***dynamic***

Describes the state of the field. The possible values of this property are listed below with their corresponding meanings.

`JCField.VALID` – The field is valid.

`JCField.INVALID` – The field is invalid.

`JCField.UNDEREDIT` – The field is currently under edit and hence the state is indeterminate

timeZone	**Date/Time**	**java.util. TimeZone**	***locale dependent***

The `timeZone` property controls the time value using conventions of the given time zone.

toolTipText	**All**	**String**	**null**

This property is used to store a short informative prompt describing the field to help end-users know what type of data to enter. `ToolTipText` can be used in conjunction with the utility `JCPromptHelper`, to associate the prompt text for given fields with an area in the display window.

useIntlCurrency Symbol	**Numeric**	**boolean**	**false**

Controls whether a numeric field with currency set displays its value using the international currency symbol for a given locale or using the symbol used by convention in that locale

Property	JCField Group	Type	Default
validChars	**All**	**String**	**null**

Describes a String of characters which are allowed as input in the current field. Use this property if the number of valid characters is less than the number of invalid ones. (See invalidChars.)

Property	JCField Group	Type	Default
value	**All**	**Object**	**null**

This is the fundamental property of a field. It fully describes the object's value. A field's value is set as a result of some valid action on the field that changes its data and has been approved by the associated validator.

Object Type	Value Type
String	String
Double	Double
Integer	Integer
Calendar	Calendar
Date	Calendar
Time	Calendar

Property	JCField Group	Type	Default
valueClass (G)	**All**	**java.lang. Class**	**N/A**

This read-only variable contains the class of the value in the field.

Appendix B

Distributing Applets and Applications on a Web Server

B.1 Using JarMaster to Customize the Deployment Archive

The size of the archive and its related download time are important factors to consider when deploying your applet or application.

When you create an applet or an application using third-party classes such as JClass components, your deployment archive will contain many unused class files unless you customize your JAR. Optimally, the deployment JAR should contain only your classes and the third-party classes you actually use. For example, the *jcfield.jar*, which you used to develop your applet or application, contains classes and packages that are only useful during the development process and that are not referenced by your application. These classes include the Property Editors and BeanInfo classes. JClass JarMaster helps you create a deployment JAR that contains only the class files required to run your application.

JClass JarMaster is a robust utility that allows you to customize and reduce the size of the deployment archive quickly and easily. Using JClass JarMaster you can select the classes you know must belong in your JAR, and JarMaster will automatically search for all of the direct and indirect dependencies (supporting classes).

When you optimize the size of the deployment JAR with JClass JarMaster, you save yourself the time and trouble of building a JAR manually and determining the necessity of each class or package. Your deployment JAR will take less time to load and will use less space on your server as a direct result of excluding all of the classes that are never used by your applet or application.

For more information about using JarMaster to create and edit JARs, please consult its online documentation.

JClass JarMaster is included with JClass DesktopViews. For more details please refer to *Quest Software's Web site*.

Appendix C

Porting JClass 3.6.x Applications

There have been significant structural changes to JClass Field beginning in its 4.x version. These modifications allow more flexibility and control over the composition of fields. Although the changes are noteworthy, you can easily convert any code created with 3.6.x versions to version 4.x and higher. The following sections will describe how to upgrade your code to JClass Field 4.5 and higher.

C.1 Key Concept Differences

In earlier versions of JClass Field, each field consisted of a visual component and a validator together. The validator portion determined what type of data the field expected. The names of the fields indicated their visual aspect and supported data type. For example, a text field that contained integers and a text field that held String values were named `JCIntTextField` and `JCStringTextField` respectively.

Now the five basic styles of visual components, which are represented by one of JClass Field's standard Beans: `JCTextField`, `JCSpinField`, `JCComboField`, `JCPopupField`, and `JCLabelField`, are separated from the validators and the supported data types. To use a field, you must associate it with a validator and declare an appropriate value model. The following table lists the a few examples of the combination of components, validators, and value models in JClass Field 4.x that are equivalent to fields in earlier versions:

Field in JClass Field 3.6.3 and earlier	Equivalent Field in JClass Field 4.x
`JCIntTextField`	`JCTextField + JCIntegerValidator + IntegerValueModel`
`JCTimeSpinField`	`JCSpinField + JCTimeValidator + TimeValueModel`
`JCStringComboField`	`JCComboField + JCStringValidator + StringValueModel`
`JCCalendarPopup`	`JCPopupField + JCDateTimeValidator + CalendarValueModel`
`JCCurrencySpinField`	`JCSpinField + JCDoubleValidator + DoubleValueModel + isCurrency property set to true`

You can duplicate all the fields contained in earlier versions by selecting the corresponding field, validator, and value model. In fact, you can create even more fields since JClass Field 4.x and higher expands the list of supported validators to include `java.lang.byte`, `java.lang.short`, `java.lang.long`, `java.lang.float`, `java.math.BigDecimal`, `java.sql.date`, and `java.sql.timestamp`, and introduces a new GUI component, `JCLabelField`. This new field can be used to simulate a heading or to display uneditable data.

C.2 Code Differences

The following table shows the differences in code between JClass Field 4.x and higher, and previous versions for a text field containing a String value.

JClass Field 3.6.3 and earlier	JClass Field 4.0 and higher
`JCStringTextField`	`JCTextField + JCStringValidator + StringValueModel`

`1 JCStringTextField text1 = new JCStringTextField();`
`2`
`3 text1.setMask("(@@@) @@@-@@@@");`
`4 text1.setPlaceHolder Chars("(___) ___-____");`
`5 text1.setValue("4165941026");`
	`1 JCTextField text1 = new JCTextField();`
	`2 JCStringValidator sv = new JCStringValidator();`
	`3 sv.setMask("(@@@)@@@-@@@@");`
	`4 sv.setPlaceHolderChars("(___)___-____");`
	`5 text1.setValueModel(new StringValueModel("4165941026"));`
	`6 text1.setValidator(sv);`

C.2.1 Converting Your Code

This section breaks down the above code listings and gives a line-by-line description of the differences.

Line 1	Similar for both versions; it simply creates the field, `text1`.
Line 2	Declares the validator, in version 4.x and higher.
Line 3	Sets the `mask` property for the field in earlier versions and for the validator for version 4.x and higher.
Line 4	Sets the `placeHolderChars` property for the field in earlier versions and for the validator for version 4.x and higher.

Line 5	Sets the initial value of the field, using the value property in earlier versions and using the value model declaration in version 4.x and higher. Although you do not have to set the value using the value model, the value model declaration and association with the field is necessary.
Line 6	Associates the validator with the field in version 4.x and higher.

C.3 Property Changes

Since the introduction of the `validator` and `invalidInfo` objects, the properties have been divided between these two objects and the field component, which in earlier

versions contained all the properties. The following table shows how the JClass Field 4.x and higher properties are allocated.

Validator Properties	Invalid Properties	Field Component Properties (same as earlier versions)
allowNull casePolicy continuousScroll currency currencyLocale currencySymbol (G) defaultDetail defaultEditFormats (G) defaultFormat (G) defaultValue displayList displayPattern editFormats editPattern firstValidCursorPosition (G) format increment invalidChars iPValidators locale mask maskChars maskInput matchPickList max milleniumThreshold min numMaskMatch parsedMask (G) pickList pickListIndex (G) placeHolderChars range (S) spinPolicy timeZone useIntlCurrencySymbol validChars	invalidPolicy invalidBackground invalidForeground beepOnInvalid	about background doubleBuffered editable enabled font foreground maximumSize minimumSize name preferredSize required selectOnEnter state (G) toolTipText

C.4 Porting Guidelines

The following list gives a general outline of the steps you should follow to port your code to JClass Field 4.x and higher from earlier versions.

- Determine the field, value model, and validator that correspond to your existing field. Create each of these objects and associate the value model and validator with the field.
- Separate any JClass Field properties you use into field component, validator and invalid properties.
- If you have any invalid properties, declare the field's `invalidInfo` object and set the properties using the new `invalidInfo`.
- Set any other properties using the field component or validator objects.

C.5 Event Handling Changes

JClass Field events have also undergone significant change in version 4.x and higher.

The event listener that receives the events generated by the four editable Fields is now called `JCValueListener` instead of `JCFieldListener`. Its methods are `valueChanging()` and `valueChanged()` instead of `valueChangedBegin()`, `valueChangedEnd()`, and `stateIsInvalid`.

Changes to any one of the Fields are handled by invoking `addValueListener()`. You supply the code to implement the `JCValueListener` interface. To register the method see addValueListener, removeValueListener, in Chapter 2.

The methods of the JClass Field event listeners are compared below:

JCFieldListener: Event Methods (earlier versions)	JCValueListener: Event Methods (JClass Field 4.x and higher)
`JCFieldListener.valueChangedBegin`	`JCValueListener.valueChanging()`
`JCFieldListener.valueChangedEnd`	`JCValueListener.valueChanged().`
`JCFieldListener.stateIsInvalid`	`no equivalent (see below)`

Although the `stateIsInvalid()` method is not available in `JCValueListener`, you can use a Field component's `addPropertyChangeListener()` method to determine changes to the state of a field.

Appendix D

Using JCField's Autocomplete Feature

D.1 Using Autocomplete in a JCComboField

The autocomplete mechanism in JCField's `JCComboField` may be used to simplify selecting items in a combo box. In addition to providing a facility for narrowing the range of possible matches as each character is typed, and thus anticipating what choice the end user really wants, the prefix mechanism can be used to simplify typing Web addresses, directory paths, or other choices that begin with a common String.

Here is what you need to do to use the autocomplete facility. A code snippet is included in each step:

1. Create or reference a combo field.
   ```
   JCComboField combo = new JCComboField());
   ```
2. Create a String validator, or a validator derived from a String validator. These are the only validators that may be used with the autocompletion mechanism: `JCStringValidator`, `JCDateValidator`, `JCDateTimeValidator`, `JCTimeValidator`, `JCIPAddressValidator`:
   ```
   JCStringValidator sv = new JCStringValidator();
   ```
3. Create or update the list Strings that will populate the combo field's drop down.
   ```
   String[] string_list = {string1, string2, ...};
   ```
4. Create a list model with the items.
   ```
   JCListModel autoCompleteListModel = new JCListModel(string_list);
   ```
5. Set this list model on the validator.
   ```
   sv.setPickList(autoCompleteListModel);
   ```
6. Set the validator on the combo box.
   ```
   combo.setValidator(sv);
   ```
7. Once a validator containing the list model has been set on a chosen combo field, the method call for invoking auto completion is:

   ```
   JCComboField combo;
   boolean autoComplete = true;
   combo.setAutoComplete(autoComplete, autoAppend, autoSuggest,
        autoRefinement, prefix_list);
   ```

The first four parameters are Booleans, while the last is an array of `String`s. The only way to set the modes and a prefix list is through a call to `setAutoComplete()`. The parameters for `setAutoComplete()` are:

- `autoComplete` – autocomplete is active. The combo box tries to autocomplete as the user types. If this parameter is `false`, auto completion is disabled.
- `autoAppend` – a candidate String appears in the text field itself. The drop down list does not appear unless `autoSuggest` is also `true`. What the end user has already typed appears as normal text and the rest of the autocompleted String appears in reverse video.
- `autoSuggest` – a drop down list appears as soon as the end user starts typing. All Strings in the list model appear in the list. The selected item is updated as the user types. The list updates itself if `autoRefinement` in on. If `autoAppend` is `false` there is no candidate completed String in the text field, only the characters the end user has typed so far.
- `autoRefinement` – to use refine, suggest mode must be on. It operates on the drop down list part of the combo box. If `autoRefinement` is `true`, the list updates itself as the end user continues to type, eliminating choices that are no longer relevant. Also, it adds what is currently in the text box to the list. If `autoRefinement` is `false`, the drop down list retains all items and the first possible match is highlighted. Autorefinement always reacts to adding or deleting a character anywhere in the String as the autocompletion mechanism matches possibilities with what has been typed.
- `prefix_list` – eliminates the need for the end user to type a common first part of the input, such as *http://www.* in a URL, or *C:/JClass/com/klg/class* in a directory path. Set a prefix list by calling `setAutoComplete()` with the array of prefix list Strings as the last parameter. If the typed-in String matches the letters immediately following one of the Strings in the prefix list, the item or items matched will appear in the combo box, along with items that begin with the typed-in String. If you are using a prefix list, set the Boolean properties `autoSuggest` and `autoRefinement` to `true` to avoid behavior that might be non-intuitive to the end user. With these two properties set, matching begins as soon as the end user begins typing. The drop down list shows the first matched list item highlighted along with any other possible match. The drop down list updates itself as the end user continues to type, showing only those items that are possible completions to what has already been typed.

  Example: The prefix list contains the Strings {“water”, “snow”}, and the combo box list contains items “police,” “polish,” “waterpic,” “waterpolo.” The combo box has `autoSuggest` and `autoRefinement` set to `true`. The end user begins by typing a “p” in the text box. All the items mentioned above appear in the drop down list. The last item is the letter “p” that has just been typed in. The end user types an “o,” so the text box contains “po.” The drop down list updates itself to contain just the matched items “police,” “polish,” “waterpolo,” and “po.” By the time the end user has typed “polo,” only two list items remain, “waterpolo,” and “polo.”

D.1.1 Cursor Behavior

Placing the cursor within the String sets the insertion point for the next typed character. The appearance of the cursor and its behavior depend on the mode autocomplete is in. The following table lists the state of `autoSuggest` (S), `autoRefine` (R), and `autoAppend` (A). The behaviors of the cursor, the text box, and the drop down list are described for each state. Assume in each case that `autoComplete` is `true`.

S	R	A	Behavior
Off	Off	Off	Autocomplete is on, but there are no visual clues.
Off	Off	On	There is no automatic invocation of the drop down list. As the end user begins typing, the first matched item appears in full in the text box. Since append mode is on, the autocompleted portion of the item is highlighted. A blinking cursor is shown at the end of the highlighted text. If the end user types another character, it is placed between what has already been typed and the highlighted portion, not at the blinking cursor. The drop down list is updated if a new match is found. If instead the end user types the backspace key, the entire highlighted portion of the String is removed. Further backspaces remove individual characters. If the end user clicks on the String to change the insertion point, the autocompleted portion of the text remains and highlighting is turned off.
Off	On	Off	Autocomplete is on, but there are no visual clues. This combination should not be used.
Off	On	On	Same as (Off, Off, On). This combination should not be used.
On	Off	Off	Just the end user's typing (without autocompletion) appears in the text box. The drop down list appears as soon as the end user begins typing. The first matched item is highlighted. All other items are available in the drop down list's scroll pane. The cursor appears in the text box at the end of the typed-in String. If the end user changes the insertion point and begins typing there, the drop down list adjusts by highlighting the new match, if there is one. If there is no match, the list stays the same.
On	Off	On	Cursor behavior is similar to (Off, Off, On). The full item list appears in the drop down list with the first matched item highlighted.

S	R	A	Behavior
On	On	Off	Since `autoAppend` is inactive, cursor behavior is the same as (On, Off, Off). Because `autoRefine` is on, the drop down list is restricted to potential matches. If the end user uses the left cursor key to move the insertion point and inserts a character that results in a new word that also is a partial match for some list items, the drop down list updates itself.
On	On	On	Cursor behavior is the same as (Off, Off, On). If a match occurs, an autocompleted item appears in the text box. The drop down list contains only potential matches.

Backspacing

To summarize, if `autoAppend` is `false`, backspacing deletes a character as usual. There is no autocompletion, so the cursor is at the end of the String. If `autoAppend` is `true`, there are effectively two cursors when a match is first found, one for inserting text and one for deleting text. Characters are inserted just before the highlighted text. Backspaces cause the highlighted text to be erased, after which previous typing is erased.

D.2 Autocomplete Methods

The methods listed here are of use with the autocomplete function:

Autocomplete Method	Description
`setAutoComplete()`	Controls whether the autocomplete function is on or off, and sets its modes of operation, which are described in the next section.
`isAutoComplete()`	Returns `true` if the field has `autoComplete` turned on.
`isAutoSuggest()`	Returns `true` if the field has `autoSuggest` turned on.
`isAutoAppend()`	Returns `true` if the field has `autoAppend` turned on.
`isAutoRefinement()`	Returns `true` if the field has `autoRefinement` turned on.
`getPrefixList()`	Returns a `String` array of prefixes that need not be typed to be matched, so long as the characters following the prefix do match an item in the combo box's list model.

D.3 Autocomplete Modes

The parameters in `setAutoComplete()` control the autocomplete modes:

Autocomplete Mode	Function
`autoAppend`	The text field shows a candidate String from its autocomplete list. Append the completion to the partial completed text shown in reverse video
`autoComplete`	Determines whether the auto complete mode is enabled. If this value is `false`, the values of the attributes are ignored.
`autoRefinement`	If `autoSuggest` is true, refine the popup list to include all possible matches as well as the currently typed text. Note that if `AutoSuggest` is false, this attribute will also be set to `false`.
`autoSuggest`	Pop up the combo box's popup as a suggestion list upon typing a character.
`prefixList`	Sets the `prefixList`. If non-null, the combo box will ignore the given prefixes when matching items. The longest matching prefix is always used. Note that although it is permitted to use a prefix list without `autoSuggest` being `true`, the resulting behavior maybe be quite non-intuitive to the end user.

The autocomplete mechanism for a `JCComboField` is turned on and off by a call to `setAutoComplete()`. The method's first parameter is the flag that controls whether autocomplete is enabled or not, and the other parameters set the autocomplete modes and the prefix list. If you do not want a prefix list, set the prefix list parameter to null. Thus, a typical call to `setAutoComplete()` looks like this:

```
combo.setAutoComplete(true, append, suggest, refine, prefix_list);
```

D.4 Code Examples

The following example shows a method that returns a `JCComboField`. Its parameters are `string_list`, the list of items for the combo box, `prefix_list`, a list of ignorable prefixes, and three Booleans for the autocomplete modes, `suggest`, `refine`, and `append`.

```
1.  public JCComboField createComboField(String []   string_list,
           String []     prefix_list,
           boolean       suggest,
           boolean       refine,
           boolean      append)
    {
       // Example of a JCComboField using a JCStringValidator
```

```
2.      JCComboField combo = new JCComboField();

        // create the validator and set its properties
3.      JCStringValidator sv = new JCStringValidator();
        sv.setMatchPickList(false);
        sv.setAllowNull(true);
4.       JCListModel dm = new JCListModel(string_list);
5.       sv.setPickList(dm);
        // set the value model and validator
        combo.setValueModel(new StringValueModel());
6.      combo.setValidator(sv);
        // No need to call this if all autocomplete mode flags are false
        if (suggest || refine || append) {
7.      combo.setAutoComplete(true,
            append,
            suggest,
            refine,
            prefix_list);
        }
        return combo;
    }
```

D.4.1 Explaining the Code

This section further explains the code in the previous section. The line number keys specify which line is being described.

Line 1	The method call.
Line 2	Instantiates a new `JCComboField`. There is no constructor for enabling the autocomplete mechanism, so it must be configured by calling `setAutoComplete()`.
Line 3	Creates a `JCStringValidator`, through which the list items are set on the combo field.
Line 4	Creates the data model that holds the list items.
Line 5	In a `JCComboField`'s design, a pick list may be derived from a data model. The pick list is set on one of JClass Field's validators, and the validator is associated with the combo box.
Line 6	Associates the validator with the combo box.
Line 7	Call `setAutoComplete()`, specifying the modes and the prefix list.

D.5 Setting and Updating the List of Autocomplete Strings

The autocomplete candidates are the Strings that populate the combo box's drop down list. Among the ways of setting the list of Strings on the swing data model, these three deserve notice.

Setting the list items:

- from predefined Strings set in the application itself
- as the result of a database query
- read from a file containing the appropriate items

Once initialized, you may wish to update the list:

- as the end user types new information
- as information changes as the result of a database query
- because the context of the application has changed

Updating from user input

One way of updating the data model is by adding a `JCValueListener` to the combo field. When the end user commits a choice by typing the *Enter* key, the listener's `valueChanging()` and `valueChanged()` methods are invoked. In the `valueChanged()` method, get the result of the user's input with

```
JCComboField combo;
...
String newValue = (String) combo.getValue();
```

You will have to check that the entry is different from ones already in the list. If it is, the new entry may be added to the data model at the position you deem appropriate, thereby updating the list. Please see Event Programming, in Chapter 4 for an example of modifying the combo box's pick list from user input as well as the example that follows in this section.

Updating from a database

You can connect a data-aware `JCComboField` component, such as `DSdbComboField`, to a data source and populate the data model form an SQL query. Please see the Data Binding, in Chapter 3, for information on data binding.

When the database is updated, the data source should fire an event to inform the combo box so that the change may be reflected in its item list.

Populating the list from a file

The tokens representing the list elements you want may be delimited in various ways in your source file. When reading the file, you will form the array of `String`s that you pass to a `JCListModel`, and you may want to use `JCStringTokenizer` in JClass Elements to simplify the task.

JClass Elements is available as part of the JClass DesktopViews product bundle. Visit *http://www.quest.com* for more information and downloads.

Adding an Item from End User Input

If you wish to allow end users to add to the list items, you can have the combo field respond to a value change event. Here is a suggestion.

In the class that is registered as a listener for `JCValue` events, implement the two required methods, `valueChanging()` and `valueChanged()`.

```
public void valueChanging(JCValueEvent e) {// May be empty }

public void valueChanged(JCValueEvent e) {
    String newValue = ((String) combo.getValue()).trim();
    boolean found = false;
    int position = 0;
    if(newValue != null && newValue.length() > 0){
        for (int i = 0; i < dm.getSize(); i++){
            if (newValue.compareTo((String)dm.getElementAt(i)) == 0) {
                found = true;
            } else {
                    if (newValue.compareToIgnoreCase(
                            (String)dm.getElementAt(i)) > 0) {
                        //place the new item in its sorted place
                        position = i + 1;            }
            }
        }
    }
    if (!found && newValue != null && newValue.length() > 0) {
        dm.add(position, (String) newValue);
        combo.setPickList(dm);
        combo.setSelectedIndex(position);
    }
}
```

The event handler looks for an existing list item that is essentially the same as the one the end user typed. If one is found, no addition is made. If the user's input is different from each item in the existing list, the new item is added to the data model and to the combo field's pick list.

Note: If `autoRefinement` is `true`, the `ValueChanged` event is not passed on to your class simply by pressing the *Enter* key. If you wish to allow end users to update the list of combo box items using `actionPerformed()` while `autoRefinement` is in effect, they will have to click on the newly-typed item. Since it does not match any previous item, it will be the only one remaining in the drop down list. This generates a `ValueChanged` event that is passed to your `ValueListener`.

D.6 Porting Guidelines

There should be no porting issues for your applications that employ a JClass `JCComboField`, version 4.5.1 or earlier, when you update to a current release. Your code should continue to function, and you may add autocompletion if you wish.

Index

A

B

C

D

E

F

G

I

J

K

L

M

N

O

P

Q