

Java-Magazin 11/2001

Table Dance - JBuilder: Renderer und Editoren für JdbTable implementieren

von Henry Wolf

Oft reicht es nicht, aus einfachen Daten in einem *JdbTable* anzuzeigen bzw. mit den Standard-Möglichkeiten der JBuilder-Komponente *JdbTable* zu arbeiten. Will man mehr, muß man sich zwangsläufig mit eigenen Implementierungen von Renderern und Editoren beschäftigen. Wir möchten Ihnen an dieser Stelle einen Weg zeigen, wie man seine *JdbTable*'s ordentlich "aufmotzen" kann.

Die Darstellungsform einer Tabellenspalte in einem *JdbTable* als datensensitive Komponente wird bekanntlich in den Eigenschaften der Spalte der Datenquelle, z.B. eines *QueryDataSet*, definiert. In der JBuilder4 - Dokumentation wird zwar zu diesem Zwecke u.a. auf eine "Unter-Klasse" der *JdbTable*, die Klasse *JdbTable.DBCellRenderer*, verwiesen. Leider konnte deren Existenz bisher nicht nachgewiesen werden.

Kommen wir auf den Weg der Definition der Darstellung über die Eigenschaften der Tabellenspalten einer Datenquelle zurück. Für den "Editor", mit dem in einer Tabellen-Zelle ein Datenfeld bearbeitet werden soll ist die Eigenschaft *itemEditor* des Spaltenobjektes verantwortlich. Will man z.B. für eine Tabellenspalte eine Auswahl der Werte über eine *ComboBox* realisieren, instanziiert man ein Objekt vom Typ *DefaultCellEditor* und weist diese der Eigenschaft *itemEditor* der gewünschten Tabellenspalte zu.

```
private void jblnit() throws Exception {
    ...
    DefaultCellEditor comboBox = new DefaultCellEditor
        (new JComboBox(new String[] {"Halle/Saale", "Nordhausen", "Dresden", "Hamburg",
            "Bitterfeld", "Berlin"}));

    column2.setColumnName("ORT");
    column2.setDataType(com.borland.dx.dataset.Variant.STRING);
    column2.setItemEditor(comboBox); /* Cell-Editor -> ComboBox */
    column2.setPrecision(100);
    column2.setTableName("SIMPLE_LISTE");
    column2.setServerColumnName("ORT");
    column2.setSqlType(12);
}
```

In unserem Beispiel soll der Wert der Spalte Ort aus einer Auswahlliste zugewiesen werden.

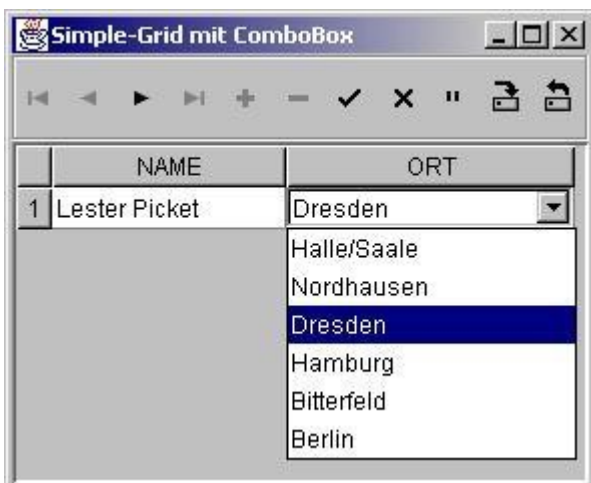


Abb. 1: Ein Ort wird zugewiesen

Betrachten wir nun etwas genauer, was wir mit der Klasse *DefaultCellEditor* anfangen können. Aus der Übersicht der möglichen Konstruktoren der Klasse ergibt sich, dass sich auf diesem Wege Spalten einfach als *ComboBox*, *CheckBox* für boolesche Datenbankfelder und als Textfeld darstellen lassen.

| Constuctor | Spaltendarstellung in einem jdbGrid |
|--|--|
| DefaultCellEditor (JCheckBox checkBox) | Darstellung als CheckBox für boolsche Felder |
| DefaultCellEditor (JComboBox comboBox) | Darstellung der Spalte als ComboBox / Auswahlliste |
| DefaultCellEditor (JTextField textField) | Darstellung als einfaches Textfeld (Standard) |

Tabelle 1: Konstruktoren der Klasse DefaultCellEditor

Der Klasse *JdbTable* ist es hoch anzurechnen, daß es im Normalfalle boolsche Felder automatisch als *CheckBox* darstellt. Will man einer Zelle jedoch eine andere Form als die der aufgeführten geben, können wir von dieser Klasse lernen, sie aber nicht so ohne weiteres verwenden.

Stellen wir uns also folgende Aufgabe: Eine bestimmte Spalte in einer *JdbTable* soll als Edit / Button – Feld dargestellt werden. Wenn also der Anwender den Focus auf die entsprechende Zelle positioniert, soll er die Möglichkeit erhalten, über einen Button in der Zelle einen Unterdialog aufzurufen, aus dem er Werte auswählt / definiert und im Ergebnis der Datenfeld – Wert unserer Spalte gebildet wird. Was also ist zu tun?

Zuerst erstellen wir das allgemeine Layout unserer Zelle. Dafür erzeugen wir eine von *JPanel* abgeleitete Klasse als Container für unser Editfeld und unseren Button und fügen in diese das Panel ein.

```
public JavaClassEditButtonPanel()
{
    super();
    this.setLayout(borderLayout);
    TextField.setText("");
    Button.setMaximumSize(new Dimension(35, 5));
    Button.setMinimumSize(new Dimension(35, 5));
    Button.setMargin(new Insets(2, 5, 2, 1));
    this.add(Button, BorderLayout.EAST);
    this.add(TextField, BorderLayout.CENTER);
}
```

Zusätzlich erweitern wir die Klasse um einige *public*-Methoden, um einfach Eigenschaften der Elemente des Panel, also unseres Edit-Feldes und unseres Button, zu manipulieren. Als erstes erweitern wir die Panel-Klasse um die Funktionalität, die Werte den Edit-Feldes von "ausen" lesen und schreiben zu können.

```
/* holen des Wertes des Editors */
public Object getEditorValue ()
{
    return new String ( TextField.getText() );
}
/* setzen des Wertes des Editors */
public void setEditorValue ( Object value )
{
    if ( value != null ) TextField.setText( value.toString() );
}
```

Im zweiten Schritt implementieren wir Methoden, um außerhalb der Panelklasse das Button-Objekt und das Edit-Feld als Objekte voll verfügbar zu haben.

```
/* Rückgabe des Textfeldes / Komponente */
public JTextField getEditorComponent()
{
    return TextField;
}
/* Rückgabe des Buttons / Komponente */
public JButton getButtonComponent ()
{

```

```

return Button;
}

```

Um einfach bestimmte Eigenschaften der Elemente der Panelklasse verändern zu können, implementieren wird noch ein paar weitere *public*-Methoden, um z.B. den Button sichtbar oder unsichtbar zu machen. Wir haben zwar schon im vorherigen Schritt uns Objektreferenzen nach "außen" erzeugt, so macht es sich aber später etwas einfacher (siehe Listing 1).

Listing 1

```

/* setzen der Sichtbarkeit des Buttons */
public void setButtonVisible ( boolean status )
{
    Button.setVisible(status);
}
/*setzen des Bearbeitungsstatus des Textfeldes*/
public void isTextFieldReadOnly ( boolean status )
{
    TextField.setEditable(status);
}

/*setzen der Farben des Textfeldes*/
// Hintergrund
public void setEditorBackground ( Color color )
{
    TextField.setBackground ( color );
}

// Fordergrund
public void setEditorForeground ( Color color )
{
    TextField.setForeground(color);
}

//Schrift-Font des Textfeldes
public void setEditorFont ( Font value )
{
    TextField.setFont( value );
}
/

```

Nachdem wir nun ein Objekt mit dem Layout für unsere Zelle definiert haben, wollen wir nun auf dieser Grundlage einen Zelleditor erstellen. Zu diesem Zwecke leiten wir von unserer Klasse *JavaClassEditButtonPanel* ein entsprechendes Objekt ab und erweitern es um die Eigenschaften des Interfaces-Klasse *TableCellEditor*.

```

public class JavaClassEditButtonPanelEditor
    extends JavaClassEditButtonPanel
    implements TableCellEditor, ActionListener, FocusListener,
               KeyListener {

```

```

.....
}

```

TableCellEditor ist die Basis-Klasse zur Erstellung eigener Cell-Renderers. Natürlich müssen wir die Interface – Klasse entsprechend implementieren. Dies geschieht über die Implementierung der Methode *getTableCellEditorComponent* (siehe Listing 2):

Listing 2

```

/*CellEditor Implementation*/
public Component getTableCellEditorComponent

```

```

(JTable table, Object value, boolean isSelected,
                                     int row, int column)
{
if (table.getModel() instanceof DBTableModel)
{
    dbTableModel = (DBTableModel) table.getModel();
    intTable = (JdbTable)table;
    setDefaultFont(intTable.getFont());
}
this.setButtonVisible(isSelected);
columnIndex = column;
rowIndex = row;
touched = false;
ignoreModelChange = true;
acceptKeyReleasedEvent = true;
this.setEditorValue( value );
    /*aktuellen Wert zuordnen*/
ignoreModelChange = false;
return this;
}

```

Diese Methode sorgt dafür, dass der Zellen-Editor mit dem Model der Datenquelle verknüpft wird und die Belegung mit dem entsprechenden Wert der aktuellen Zeile erfolgt.

Neben der Implementierung der Schnittstellen – Methode *getTableCellEditorComponent* müssen noch weitere Methoden der Interface-Klasse *TableCellEditor* implementiert werden. Dabei handelt es sich um die folgenden:

- *public void addCellEditorListener(CellEditorListener l)*
Hinzufügen von Event-Listener, welche beim Beginn, während bzw. am Ende der Bearbeitung der Zelle registriert sein müssen.
- *public void removeCellEditorListener(CellEditorListener l)*
Entfernen der Event-Listener
- *protected void stopCellEditing()*
Beenden der Bearbeitung einer Zelle, Änderungen werden übernommen
- *protected void cancelCellEditing()*
Beenden der Bearbeitung einer Zelle, Änderungen werden nicht übernommen
- *public Object getCellEditorValue()*
Rückgabe der Wertes des Editors an das DataModel
- *public boolean isCellEditable(EventObject anEvent)*
Auskunft über Bearbeitungsmöglichkeit des Inhaltes der Zelle
- *public boolean shouldSelectCell(EventObject anEvent)*
Ist Zelle aktiv oder nicht.

Um dafür zu sorgen, daß der Wert des durch den vom Button aufgerufenen Dialogs ordentlich übernommen oder die Bearbeitung korrekt abgebrochen wird, implementieren wir die Methoden *fireEditingStopped* und *fireEditingCancelled*. Zum externen Zugriff auf diese Methoden verpacken wir diese noch in ein "public"-Schale (siehe Listing 3).

Listing 3

```

// normales Beenden der Bearbeitung einer Zelle
protected void fireEditingStopped()
{
    // sichert non-Null-Array
    Object[] listeners = listenerList.getListenerList();
    // Event-Listener werden informiert
    for (int i = listeners.length-2; i>=0; i-=2)
    {

```

```

if (listeners[i]==CellEditorListener.class)
{
    ((CellEditorListener)listeners[i+1]).editingStopped(changeEvent);
}
}
}

// Methode für public-Zugriff auf fireEditingStopped
public void PubfireEditingStopped ()
{
    fireEditingStopped();
}

// Abbruch der Bearbeitung einer Zelle
protected void fireEditingCancelled()
{
    // Guaranteed to return a non-null array
    Object[] listeners = listenerList.getListenerList();
    // Process the listeners last to first, notifying
    // those that are interested in this event
    for (int i = listeners.length-2; i>=0; i-=2)
    {
        if (listeners[i]==CellEditorListener.class)
        {
            ((CellEditorListener)listeners[i+1]).editingCanceled(changeEvent);
        }
    }
}
}

```

Um die vollständige Kontrolle über unseren Zell-Editor zu erhalten, müssen wir ebenfalls die Interface – Klassen *ActionListener*, *FocusListener* und *KeyListener* implementieren. Über diese erhalten wir die vollständige Kontrolle über Tastatur-Eingabe und wir können auf Ereignisse, wie das Zuweisen des Fokus auf unseren Zell-Editor reagieren, wenn wir z.B. dem Editor in Abhängigkeit vom Eingabefokus bestimmte Eigenschaften verpassen wollen.

| Interface-Klasse | Implementierung |
|------------------|--|
| ActionListener | public void actionPerformed(ActionEvent e) |
| FocusListener | public void focusGained(FocusEvent e) public void focusLost(FocusEvent e) |
| KeyListener | public void keyPressed (KeyEvent e) public void keyTyped (KeyEvent e) |

Wollen wir nun unseren Editor benutzen, müssen wir eine Instanz des Objektes anlegen und der gewünschten Spalte die Instanz des Objektes als *ItemEditor* zuweisen.

```

public class DataModule1 implements DataModule {
..
    JavaClassEditButtonPanelEditor  EditorButtonEdit = new
    JavaClassEditButtonPanelEditor();
}
private void jblnit() throws Exception {
..
    column3.setItemEditor(EditorButtonEdit);
..
}

```

Wenn wir den Button in dem Zellen-Editor drücken würden, würde bis jetzt nichts passieren. Das heißt, wir müssen für die Betätigung des Buttons noch eine auszuführende Aktion definieren. Hierbei kommen wir auf unsere Methode *getButtonComponent()* unserer Basisklasse zurück (siehe Listing 4).

Listing 4

```
private void jblnit() throws Exception {
EditorButtonEdit.getButtonComponent().addActionListener(new java.awt.event.ActionListener(){

    public void actionPerformed(ActionEvent e)
    {
        jButton1_actionPerformed(e);
    }
});

void jButton1_actionPerformed(ActionEvent e)
{
    SUB_Dialog Dialog = new SUB_Dialog();
    Dimension dlgSize = Dialog.getPreferredSize();
    Dialog.setModal(true);
    Dialog.show();
    EditorButtonEdit.getEditorComponent().setText ( Dialog.RetValue ( ) );
    EditorButtonEdit.PubfireEditungStopped (); /* für ordentliche Übernahme
des Wertes */
}
```

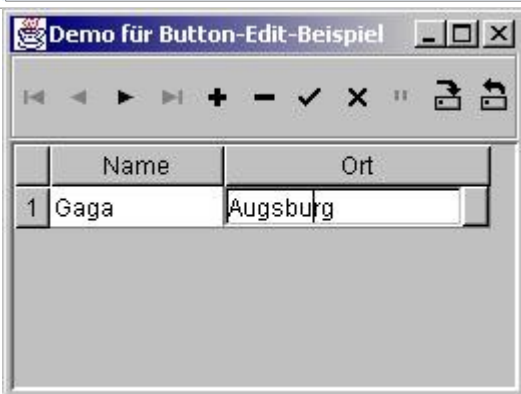


Abb. 2: Der Cell-Renderer im Einsatz

Ist dies realisiert, sollte alles klappen. Vergleichen wir nun unseren eigenen Editor mit der Deklaration der Klasse *DefaultCellEditor*, werden wir viele Gemeinsamkeiten feststellen. Kann auch gar nicht anders sein.

Nach dem wir unsere Editor-Klasse erfolgreich erstellt haben, wollen wir unserer Tabellenspalte noch ein paar individuelle Anzeigeeigenschaften zuordnen. Hierfür verwendet man sogenannte Renderer-Klassen. Die Zuordnung erfolgt über eine Eigenschaft der Spalte unseres DataSet's, dem Property *itemPainter*. Die erstellte Rander-Klasse für die Spaltendarstellung in einer *JdbTable* muß die Interface – Klasse *TableCellRender* implementieren. Vor dem Aufbau der eigenen Randerer-Klasse lohnt es sich auch hier mit der Dokumentation der Standard-Renderer-Klasse *DefaultCellRender* zu beschäftigen.

Die Deklaration einer eigenen Randerer-Klasse in unserem Beispiel würde wie folgt aussehen:

```
public class JavaClassEditButtonPanelRender
    extends JavaClassEditButtonPanel
    implements TableCellRender, CustomPaintSite
{
...
}
```

Die Randerer-Klasse leiten wir von unserer Basisklasse *JavaClassEditButtonPanel* ab und implementieren die Interface – Klassen *TableCellRender* und *CustomPaintSite*. *TableCellRender* um unserer Klasse als Randerer-Klasse für eine *JdbTable* verwenden zu können und *CustomPaint* um den Prozess des Zeichnens der Spalten-Klasse individuell gestalten zu können.

Damit unsere eigene Klasse als *TableCellRender* funktioniert, müssen wir die Methode

getTableCellRendererComponent implementieren. Nach "ausssen" liefert diese Interface-Methode vor dem Zeichnen die Eigenschaften der aktuellen Zelle. Dabei können wir durch die Implementierung der Schnittstellen-Klasse *CustomPaintSite* die Anzeigeeigenschaften wie Farben, Schrift, Ausrichtung u.a. individuell an unsere Anforderungen anpassen (siehe Listing 5).

Listing 5

```
public Component getTableCellRendererComponent(JTable table, Object value,
        boolean isSelected, boolean hasFocus,
        int row, int column)
{
    selectStatus = isSelected; // für interne Nutzung !!
    focusStatus = hasFocus; // für interne Nutzung !!

    this.setButtonVisible ( selectStatus );

    if ( table instanceof JdbTable ) { jdbTable = (JdbTable) table; }
    if ( defaultForeground == null )
    {
        if ( selectStatus ) this.setEditorForeground(jdbTable.getSelectionForeground());
        else this.setEditorForeground (jdbTable.getForeground());
    }
    if ( defaultBackground == null )
    {
        if ( selectStatus ) this.setEditorBackground(table.getSelectionBackground());
        else this.setEditorBackground(jdbTable.getBackground());
    }

    setDefaultFont(jdbTable.getFont());

    if ( hasFocus )
    {
        {
            setBorder(UIManager.getBorder("Table.focusCellHighlightBorder"));
            if ( table.isCellEditable(row, column) && table instanceof JdbTable )
            {
                if ( jdbTable.getEditableFocusedCellForeground() != null )
                {
                    setDefaultForeground(jdbTable.getEditableFocusedCellForeground());
                }
                if ( jdbTable.getEditableFocusedCellBackground() != null )
                {
                    setDefaultBackground(jdbTable.getEditableFocusedCellBackground());
                }
            }
        }
    }
    else
    {
        {
            setBorder(noFocusBorder);
        }
    }
    this.setEditorValue(value);
    return(this);
}
```

Wollen wir unsere Renderer-Klasse für unsere Tabellenspalte verwenden, weisen wir einfach eine Instanz dieser Klasse der Eigenschaft *itemPainter* unserer Tabellenspalte zu.

Listing 6

```
public class DataModule1 implements DataModule {
    private static DataModule1 myDM;
    Database database1 = new Database();
}
```

```

QueryDataSet simple_liste = new QueryDataSet();
Column column1 = new Column();
Column column2 = new Column();
Column column3 = new Column();
JavaClassEditButtonPanelEditor EditorButtonEdit = new JavaClassEditButtonPanelEditor();
JavaClassEditButtonPanelRenderer EditorButtonRend = new JavaClassEditButtonPanelRenderer();
private void jblnit() throws Exception {
EditorButtonEdit.getButtonComponent().addActionListener(new java.awt.event.ActionListener(){
    public void actionPerformed(ActionEvent e)
    {
        jButton1_actionPerformed(e);
    }
});
column3.setColumnName("ORT");
column3.setDataType(com.borland.dx.dataset.Variant.STRING);
column3.setItemEditor(EditorButtonEdit);
column3.setItemPainter(EditorButtonRend);
/* alternativ column3.setItemPainter(new JavaClassEditButtonPanelRenderer()); */
column3.setPrecision(100);
column3.setTableName("SIMPLE_LISTE");
column3.setServerColumnName("ORT");
column3.setSqlType(12);
...
}
..
}

```

Auf der Grundlage unseres vorgestellten Beispiels kann man sich mit etwas Mühe eine ordentliche Klassenbibliothek für alle möglichen Zwecke aufbauen und ist dabei nicht nur auf Zellen einer *JdbTable* beschränkt. Das gleiche Konzept funktioniert für alle DBSwing-Klassen. Man muß nur seine eigene Klasse von den jeweiligen Interface – Klassen ableiten. Java setzt uns wie immer fast keine Grenzen ;-).