JIDE Pivot Grid Developer Guide

Contents

Purpose of This Document

JIDE Pivot Grid is a Java/Swing implementation of pivot table. This developer guide is designed for developers who want to learn how to use *JIDE Pivot Grid* in their applications.

JIDE Pivot Grid heavily depends on features and components provided by *JIDE Grids*. So if you never used *JIDE Grids* before, we strongly recommend you read *JIDE Grids Developer Guide* first or at least refer to it while reading this developer guide.

Pivot Table Basic

A pivot table enables you to take what seems to be an indecipherable mass of facts and extract any trends and patterns buried in the data. You can organize and summarize your data, perform comparisons, and extract meaningful information that can be invaluable to you and your organization.

Microsoft® Office Excel and many other analytics applications provided pivot table feature. Below is a screenshot of pivot table in Excel. You can also play with Excel yourself and see how powerful a pivot table is. Once you've done that, you will start to think if you can have it in your Swing application. That's where *JIDE Pivot Grid* comes to help.
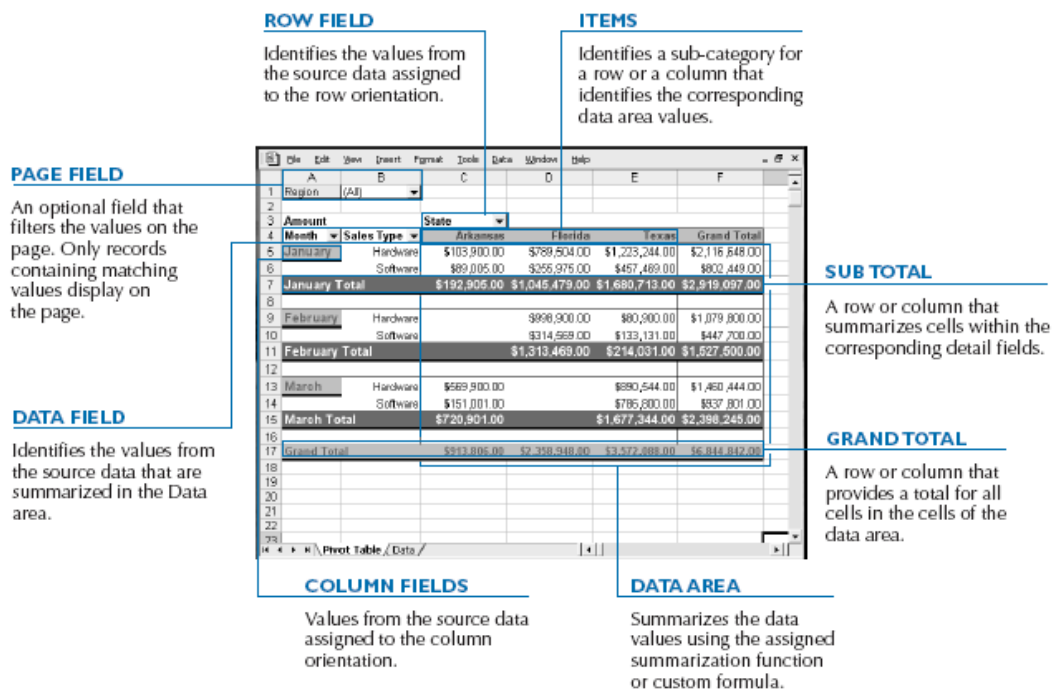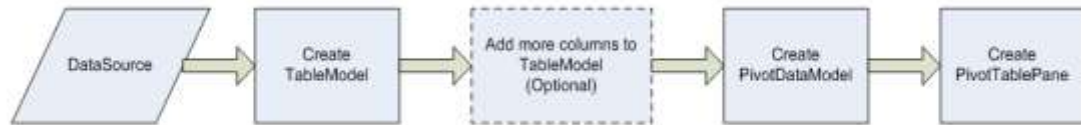

Figure 1 PivotTable in Microsoft® Office Excel

2

Overview

In order to add a pivot table to your applications, there are several steps you need to follow. Here is the flow. We will cover each step in details.

Data Source

You may read your data from any data sources such as a database table, a file, a piece of data in memory. The data should be in tabular format that can be converted to *TableModel* as defined in Java Swing. From *JIDE Pivot Grid* point of view, the only data it will accept is the *TableModel*. As long as you convert your raw data to *TableModel*, you can use it in *JIDE Pivot Grid*.

Prepare the Table Model

Pivot table provides a way to analysis the data. However, what you can actually get out of the pivot table is depending on your knowledge of the data. So after you convert your row data to a *TableModel*, most likely you need to customize the table model to add more columns and convert some columns that you will need in pivot table.

Calculated Columns

Frequently, you find the data in your table model doesn't contain all the columns you need. For example, your table model contains a shipped date column. However you are interested in the quarter value of the shipped date so that you can generate a quarterly report. Or, your table model contains quality and unit price columns but you are interested in the total amount which can be calculated as quality * unit price. With the help of calculated columns, you can easily add new columns to do what mentioned above.

CalculatedColumn is the interface for a calculated column. The main method in this interface is

```
Object getValueAt(int rowIndex);
```

This method will calculate on fly the value for the calculated column. Another method is

```
int[] getDependingColumns();
```

This method will tell us what columns the calculated column depends on. It is important to know because when the depending columns' value changed, we know we should recalculate the calculated column.

3

In order to use *CalculatedColumn*, you need *CalculatedTableModel*. *CalculatedTableModel* wraps the underlying table model to create a new table model so that you can add and remove columns.

```
CalculatedTableModel calculatedTableModel = new CalculatedTableModel(tableModel);
```

The code above will create a *CalculatedTableModel*. However it's empty at this point. You need add columns to it. You can add all the columns in the underlying table model to the *CalculatedTableModel*. Here is how.

```
calculatedTableModel.addAllColumns();
```

Just so you know, you don't have to add all the columns but just pick those columns that you will use in the pivot table. The less columns, the more efficient will be the calculation of pivot table data. There are many things we can do to this *CalculatedTableModel*. Before jumping in, let's talk more about *CalculatedColumn*.

On top of *CalculatedColumn*, there is *AbstractCalculatedColumn* which implements all methods in *CalculatedColumn* except the two methods we mentioned above – *getValutAt* and *getDependingColumns*. We also have *SingleColumn* which you can use if the calculated column depends on only one column in the underlying table model. Then we have *ExpressionCalculatedColumn* which allows you to write a simple expression for the calculated column. For example *[Quality] * [unitPrice]* is the expression you can write for the example we used above.

To add a single column, you can call

```
calculatedTableModel.addColumn(new SingleColumn(tableModel, 0)
```

This line will add the first column in tableModel to *CalculatedTableModel*. Please note the second parameter is the column index in the underlying table model.

You can also use *AbstractCalculatedColumn* directly. See below. We create a new column which doubles the sales value which is the 3rd column in the underlying table model.

```
calculatedTableModel.addColumn(new AbstractCalculatedColumn(tableModel, "Double Sales", Float.class) {
  public Object getValueAt(int rowIndex) {
    Object valueAt = getActualModel().getValueAt(rowIndex, 2);
    if (valueAt instanceof Float) {
      return ((Float) valueAt).floatValue() * 2;
    }
    return "--";
  }

  public int[] getDependingColumns() {
    return new int[]{2};
  }
});
```

4

The value change in *CalculatedTableModel* is transparent. In the other words, whenever the underlying table model's cell value changes, *CalculatedTableModel* will update its value automatically. That's because we have the *getDependingColumns()* method in *CalculatedColumn* interface.

There is also a more complex way to define a *CalculatedColumn* using Expression. Please refer to "Expression for CalculatedColumn" section at the end of this document.

Grouping Values

You can group a range of values into one value so that the resulting column has less number of unique values. This is very important in pivot table because too many unique values will not only increase the memory needed to hold all the data but also causes noise because of the small variation in the data. For example if you have an Age column, you can group it into 0 to 18, 18 to 30, 30 to 50 etc groups. For 5-digit U.S. zip code, you can group them by the first two digits which will roughly tell you where the zip code is. For product name, you may want to group them aphetically into the product names beginning with A, beginning with B etc. See below for such an example.

```
SingleColumn column = new SingleColumn(tableModel, "ProductName");
column.setObjectGrouper(new ObjectGrouper() {
  public Class getType() {
    return String.class;
  }

  public Object getValue(Object value) {
    String name = value.toString();
    return name.substring(0, 1);
  }

  public String getName() {
    return "Alphabetical";
  }
});
```

ObjectGrouper is the interface that calculates group value from a value. You can define many *ObjectGrouper* and register them with *ObjectGrouperManager*. If you use JIDE *ObjectConverter* before, this is the same mechanism as *ObjectConverterManager*.

AbstractCalculatedTableColumn defined a method *setObjectGrouper(ObjectGrouper)*. Once you set a *ObjectGrouper*, the value of this column will be grouped.

CachedTableModel

Now you see some advanced features provided by *CalculatedTableModel* and the *ObjectGrouper*, you might start to worry about the performance as the values are calculated on fly. You are right. If the calculation is complex, you will face the performance issue. To address this problem, we introduce *CachedTableModel*. A the name indicates, it will cache the values that is returned from *getValueAt(int row, int column)*. So next time when *getValueAt* is called

again, it will return the cached value instead of calculating it again. The *CachedTableModel* is smarted enough to recalculate the value only when the underlying table model changes and only when *getValueAt* is called. And again, this is completely transparent to you.

Pivot Data Model

PivotDataModel is the main data structure for pivot table. It will convert a tabular data such as *TableModel* to pivot data.

The constructor of *PivotDataModel* can take **any** table model.

```
PivotDataModel pivotDataModel = new PivotDataModel(tableModel);
```

PivotDataModel is the data model that contains all we need in order to create a pivot table. As you can see from the code above, it can take any table model (including the *CalculatedTableModel* or *CachedTableModel* as we introduced above) as the input data and calculate pivot data so that Pivot Grid component can use it.

Fields in PivotDataModel

Each column in the *TableModel* is automatically become a *field* in *PivotDataModel*. There are four types of fields – row fields, column fields, filter fields[1] and data fields.

You can call *getField(String columnName)* to get a field. The *columnName* is the column name in *TableModel*. Hopefully you make them unique in your *TableModel*. In fact, you can also call *getField(int index)* to get a field if you know the index. However in most cases, you refer to a field by name, so *getField(String columnName)* is used more often.

```
pivotDataModel.getField("CategoryName").setAreaType(PivotField.AREA_ROW);
pivotDataModel.getField("CategoryName").setAreaIndex(1);
pivotDataModel.getField("CategoryName").setTitle("Category Name");
pivotDataModel.getField("CategoryName").setSubtotalType(PivotField.SUBTOTAL_AUTOMATIC);
```

The four line above configured "CategoryName" field. It makes this field a row field. The index is set to 1 which usually means it is the first field in row areas. It also changes the display title to more user friendly "Category Name" instead of "CaregoryName". The last line said we want to calculate subtotal for the field.

```
pivotDataModel.getField("ProductSales").setAreaType(PivotField.AREA_DATA);
pivotDataModel.getField("ProductSales").setSummaryType(PivotField.SUMMARY_SUM);
pivotDataModel.getField("ProductSales").setAreaIndex(1);
pivotDataModel.getField("ProductSales").setTitle("Sales");
pivotDataModel.getField("ProductSales").setConverterContext(CurrencyConverter.CONTEXT);
```

---

[1] In Excel, filter field is called page field.

The five lines above configured "ProductSales" field. As you can see, it's a data field. As sales is dollar amount, we set ConverterContext to use CurrentyConverter so that it displays a currency symbol in front of the numeric value. If you don't know what a Converter is, you can refer to JIDE Grids Developer Guide for more information.

```
pivotDataModel.getField("Year").setAreaType(PivotField.COLUMN);
pivotDataModel.getField("Year").setConverterContext(YearNameConverter.CONTEXT);
pivotDataModel.getField("Year").setAreaIndex(1);
pivotDataModel.getField("Year").setSubtotalType(PivotField.SUBTOTAL_CUSTOM);
pivotDataModel.getField("Year").setCustomSubtotals(new int[]{PivotField.SUMMARY_STDDEV,
PivotField.SUMMARY_SUM});
pivotDataModel.getField("Quarter").setAreaType(PivotField. COLUMN);
pivotDataModel.getField("Quarter").setConverterContext(QuarterNameConverter.CONTEXT);
pivotDataModel.getField("Quarter").setSubtotalType(PivotField.SUBTOTAL_CUSTOM);
pivotDataModel.getField("Quarter").setCustomSubtotals(new int[]{PivotField.SUMMARY_STDDEV,
PivotField.SUMMARY_SUM, PivotField.SUMMARY_MAX});
pivotDataModel.getField("Quarter").setAreaIndex(2);
pivotDataModel.getField("Month").setAreaType(PivotField.COLUMN);
pivotDataModel.getField("Month").setConverterContext(MonthNameConverter.CONTEXT);
pivotDataModel.getField("Month").setAreaIndex(3);
```

The lines above configured "Year", "Quarter" and "Month" fields as column fields. From the area index values, we can tell the "Year" is the first one, then "Quarter" and the last is "Month". So it looks like we create the pivot table in order to analysis sales data by the year/quarter/month.

```
pivotDataModel.setShowGrandTotalForColumn(true);
pivotDataModel.calculate();
```

Pivot Table

Once you have *PivotDataModel* configured and calculated, you are ready to display it in *PivotTablePane*, the main component for *JIDE Pivot Grid*.

```
PivotTablePane pivotTablePane = new PivotTablePane(pivotDataModel);
```

PivotTablePane is a component that you can add it anywhere in your application. As you can see below, each pivot table divides into eight areas – row field area, column field area, filter field area, data field area, row header area, column header area, data area and field chooser area.

Filter field area is the first area on top. It contains a number of fields. Each field has a drop down button to bring up a check box list. User can check or uncheck them to include or exclude the value. You will see the change automatically in the pivot when user makes the changes.

Data field area is the one right below Filter field area. It lists what fields are used as data fields.

Row field area is the one above the row header table. **Column field area** is the one above the column header table. They list what fields are used as row and column fields respectively.

There are actually three sub-tables in the body part of pivot table. The row header area contains the **row header table**. The column header area contains the **column header table**. The middle is the **data table**. The two scroll bars will synchronize those tables as if they are one table.

As last, *PivotTablePane* have a **field chooser area**. All fields that are not in filter field area, row header area, column header area or data field area are in this area. You can use this area to add the fields to the four other areas.


There are many things user can do to this table. Let's take a look.

Sort and Filter Fields

Each field is represented as a field box in filter field area, row field area, column field area and data field area of a pivot table. As you can see, each field box has three parts – the name, the sort arrow and a filter drop down button. You can click on the field box to toggle sorting. You can also click on filter drop down button to change the filter. Please note, not only you can change filter on filter fields as assumed, but also you can change filter in row fields and column fields.
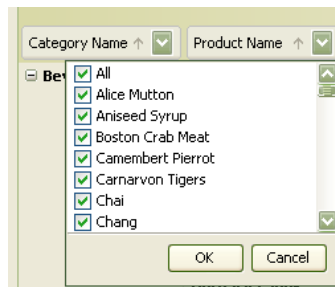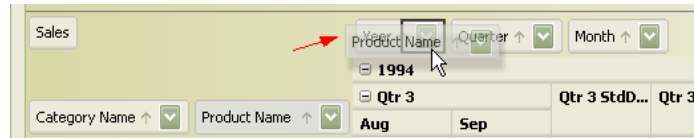

Rearrange Fields

You can also drag and drop the field box. Just grab any field, you will see a gesture indicating it is dragged.

8

If you move mouse slowly to different areas while holding the mouse left button, you will see indication of different drop area. See screenshot below, if you drag to the column field area, you see a black rectangle. It means if you drop now, the Product Name will be set to column field and put between Year and Quarter fields.

Here is what you get.

By drag-n-dropping, user can easily rearrange fields among the four field areas. In order to add a new field, user will need the field chooser area. As mentioned earlier, all columns in the table model become a field in pivot data model. Most fields are assigned as filter field, data field, row fields columns fields. For those unassigned fields, they will appear in field chooser area. Fields in field chooser area can be dragged as well. It can also take dragged fields from the four field areas.

The pivot table is updated immediately whenever the layout is changed. So user can what data table looks like and decides if the current layout is the one they want.

Totals and Summaries

JIDE Pivot Grid supports grand total on row and column fields. It can display a grand total row or column at the end of the data area. You can call *setShowGrandTotalForColumn(boolean)* and *setShowGrandTotalForRow(boolean)* methods on *PivotDataModel*.

Figure 2 Grand Total

JIDE Pivot Grid also supports subtotals on row or column fields. It will appear as one extra row or column after each field in row header table or column header table.

9

Figure 3 Subtotal Row

You can also customize it to show a different subtotal type either through UI or code. User can right click on a field and select "Field Settings…" then choose which subtotals they want to see.
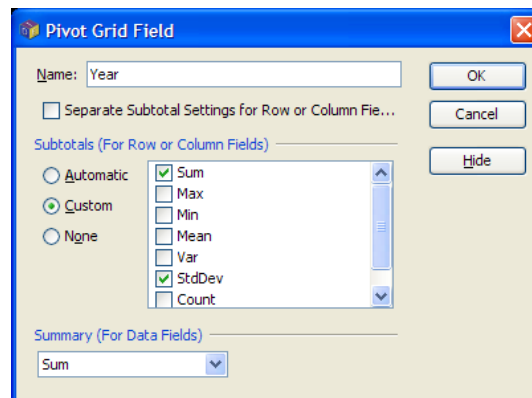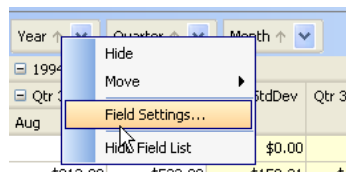

Figure 4 Subtotal Row

There are several settings user can set from this dialog.

● Subtotal: Subtotal is used for row and column fields. User can select automatic subtotal which will be Sum. Or they can pick Custom subtotal and select one or several statistics they want to calculate. See below for the area in red.

User also has the option to display different subtotal for row fields and column fields. See below.

- Summary: Summary is also a kind of statistic just like those used in subtotal except this is only used when the field is a data field. For example, if you look at the following screenshot, the value in the red rectangle is actually a sum of several values in the original table model. It shows as sum because we selected Sum as the default statistics for Sales data field.

Not only user can configure those, you as a developer can also write code to do it using APIs. The method is *setSubtotalType* on *PivotField*. If the value is *PivotField.SUBTOTAL_CUSTOM*, *setCustomSubtotals* method will allow you to set the custom subtotals as user can set using the check box list in the dialog above. PivotField#setSummaryType is to change the last combobox in the dialog above.

Advanced Features

Customized Cell Renderers

In Pivot Table, each field usually one contains data of one type. You can provide your own converter[2] to convert from the data type to string so that it can be displayed in the cell.

The screenshot above actually shows three different types of converters. The first one is CurrencyConverter which converts from a double or float to a string with $ prefix. Inside the converter, it simply uses NumberFormat.getCurrencyInstance() to do the conversion. The second one is a MonthNameConverter which converts an int with value from 0 to 11 to the Month name. The third one is a QuarterNameConverter which converts from 0 to 3 to a string

---

[2] Converter is referred to *ObjectConverter*. Please read *JIDE Grids Developer Guide* for more information.

like "Qtr 1", "Qtr 2", etc. If you are familiar with JIDE Grids, you will know we have a manager called ObjectConverterManager where you can register the ObjectConverters. ObjectConverterManager already contain a lot of converters for common data types. If you need more, you can always register it yourself. See code below where we register several converters we mentioned above to ObjectConverterManager.

```
CurrencyConverter converter = new CurrencyConverter();
ObjectConverterManager.registerConverter(Double.class, converter, CurrencyConverter.CONTEXT);
ObjectConverterManager.registerConverter(double.class, converter, CurrencyConverter.CONTEXT);
MonthNameConverter monthNameConverter = new MonthNameConverter();
ObjectConverterManager.registerConverter(Integer.class, monthNameConverter,
MonthNameConverter.CONTEXT);
ObjectConverterManager.registerConverter(int.class, monthNameConverter, MonthNameConverter.CONTEXT);
QuarterNameConverter quarterNameConverter = new QuarterNameConverter();
ObjectConverterManager.registerConverter(Integer.class, quarterNameConverter,
QuarterNameConverter.CONTEXT);
ObjectConverterManager.registerConverter(int.class, quarterNameConverter,
QuarterNameConverter.CONTEXT);
```

You may notice we use *CurrencyConverter.CONTEXT*, *MonthNameConverter.CONTEXT* and *QuarterNameConverter.CONTEXT* when we registered the converters. The reason we need the context so that we can register several converters for the same data type. For example both *MonthNameConverter* and *QuarterNameConverter* convert int type.

In order to tell *PivotField* to use certain converter, you just call *setConverterContext* on it. See code below.

```
pivotDataModel.getField("Year").setConverterContext(YearNameConverter.CONTEXT);
pivotDataModel.getField("Quarter").setConverterContext(QuarterNameConverter.CONTEXT);
pivotDataModel.getField("Month").setConverterContext(MonthNameConverter.CONTEXT);
```

Setting *ConverterContext* should be enough for most of the data types that need to be displayed as string. However, there are still cases that you need to use a customized cell renderer. For example, you want to display a percentage using a color bar. Obviously a converter is not enough. In this case, you can use *setEditorContext*[3] method on *PivotField*. See code below where we register a *PercentageCellRenderer* and tell *PivotField* to use it.

```
PercentageCellRenderer percentageCellRenderer = new PercentageCellRenderer();
CellRendererManager.registerRenderer(int.class, percentageCellRenderer, PercentageCellRenderer.CONTEXT);
pivotDataModel.getField("Percentage").setEditorContext(PercentageCellRenderer.CONTEXT);
```

---

[3] You may wonder why we named the method setEditorContext, not setRendererContext. The reason is EditorContext is an old concept in JIDE Grids which we use for both cell editor and cell renderer. We called *EditorContext* to cover both cases. In Pivot Table, the cells are not editable so only renderers will be used. However we still want to keep the name *EditorContext* just to be compatible with JIDE Grids.

Customize Cell Styles

Customizing cell styles sounds very similar to customize cell renderers but in fact they are two different aspects of making the data cells easier for users to read. The cell style concept again comes from *JIDE Grids* where there is *CellStyleTable* which you can use a *StyleModel* to customize the cell background, foreground, font, border, alignments etc without touching the cell renderers.

In Pivot Table, PivotDataModel has a method called setCellStyleProvider. In this provider, you can provide CellStyle for data table, column header table and row header table respectively. Let's see an example.

First, let's define some CellStyles as constants to make it efficient.

```
static CellStyle HIGH_STYLE = new CellStyle();
static CellStyle LOW_STYLE = new CellStyle();
static CellStyle SUMMARY_STYLE = new CellStyle();
static CellStyle DEFAULT_STYLE = new CellStyle();
static CellStyle HEADER_STYLE = new CellStyle();

static {
    HIGH_STYLE.setForeground(Color.WHITE);
    HIGH_STYLE.setBackground(Color.RED);

    LOW_STYLE.setForeground(Color.BLACK);
    LOW_STYLE.setBackground(Color.YELLOW);

    SUMMARY_STYLE.setForeground(Color.BLACK);
    SUMMARY_STYLE.setBackground(new Color(255, 255, 215));

    DEFAULT_STYLE.setForeground(Color.BLACK);
    DEFAULT_STYLE.setBackground(Color.WHITE);

    HEADER_STYLE.setFontStyle(Font.BOLD);
    HEADER_STYLE.setForeground(Color.BLACK);
}
```

Basically, we want to display summary row or column using SUMMARY_STYLE which has a yellow background. Then for cells that have lower or higher than normal value, we use a different color background. Last for all headers such as row header or column, we want to use bold font.

In *setCellStyleProvider* method, we had

```
pivotDataModel.setCellStyleProvider(new PivotCellStyleProvider() {
    public CellStyle getDataTableCellStyleAt(DataTableModel model, FieldTableModel rowHeaderModel,
FieldTableModel columnHeaderModel, int rowIndex, int columnIndex) {
        if (rowHeaderModel.isSubtotalRow(rowIndex) || columnHeaderModel.isSubtotalRow(columnIndex)
            || rowHeaderModel.isGrandTotalRow(rowIndex) ||
columnHeaderModel.isGrandTotalRow(columnIndex)) {
```

14

```
                return SUMMARY_STYLE;
            }
        else {
            Object value = model.getValueAt(rowIndex, columnIndex);
            if (value instanceof Double && ((Double) value).doubleValue() > 10000) {
                return HIGH_STYLE;
            }
            else if (value instanceof Double && ((Double) value).doubleValue() < 50) {
                return LOW_STYLE;
            }
            else {
                return DEFAULT_STYLE;
            }
        }
    }

    public CellStyle getColumnHeaderCellStyleAt(FieldTableModel model, int rowIndex, int columnIndex) {
        return HEADER_STYLE;
    }

    public CellStyle getRowHeaderCellStyleAt(FieldTableModel model, int rowIndex, int columnIndex) {
        return HEADER_STYLE;
    }
});
```

Here is what you get.

As you can see, with a few lines of code, you customized the cell styles to make result much easier to read.

15

Data Editing

PivotTable is usually read-only because the data table shows the summary of several rows in the original table model. If user tries to edit the data table directly, it is not clear how to modify the original table model to reflect the data change. In order to solve this problem, we introduce *PivotDataEditingProvider*. See the interface below.

```
public interface PivotDataEditingProvider {
   /**
    * Returns true if the cell at <code>rowIndex</code> and
    * <code>columnIndex</code> of the DataTableModel in PivotDataModel
    * is editable.  Otherwise, <code>setValueAt</code> on the cell will not
    * change the value of that cell.
    *
    * @param dataModel   the pivot data model.
    * @param rowIndex    the row whose value is to be queried
    * @param columnIndex the column whose value is to be queried
    * @return true if the cell is editable
    */
   boolean isCellEditable(PivotDataModel dataModel, int rowIndex, int columnIndex);


   /**
    * This method is called when user changes the the value in the cell at <code>columnIndex</code> and
    * <code>rowIndex</code> to <code>aValue</code>. You should implement this method and modify the
    * corresponding cell or cells in data source so that the change can be persisted.
    *
    * @param dataModel   the pivot data model.
    * @param aValue      the new value
    * @param rowIndex    the row whose value is to be changed
    * @param columnIndex the column whose value is to be changed
    * @see #isCellEditable
    */
   void setValueAt(PivotDataModel dataModel, Object aValue, int rowIndex, int columnIndex);
}
```

The purpose of this interface is to let you to decide how to apply the data change from the data table to the underlying table model. You can use isCellEditable method to determine which cell is editable. If a cell is editable and user changes the value, the value will be passed to *setValueAt* method. You can decide how to accept the new value.

There are three on *PivotDataEditingProvider* on *PivotDataModel*. They are for the data table, row header table and column header table respectively.

We also introduced *PivotEditingProvider* which is a higher level abstraction. This is an interface that further simplifies the data editing to just three basic operations – addRow, removeRow and setValueAt. These three operations are on the original table model. If you are using *DefaultTableModel*, you can pretty much delegate to the three same-name methods on *DefaultTableModel*. We even made *DefaultPivotEditingProvider* for it. If not, you will have to implement the three methods yourself using *AbstractPivotEditingProvider*.

When user edits a cell in the data table (or row header table and column header table), *PivotEditingProvider* will find the rows behind the cell and change values for all the rows. You can also add a button to your user interface which will call *addRow* or *removeRow* method of *PivotEditingProvider* if user needs to add and remove rows from the original table model.

Date Calculation and Auto-update

The calculation from the data in the original table model to the data displayed in the pivot table is a complex process. It is done in *PivotTableModel#calculate()* method. If the original table model has a huge number of rows, *calculate()* could take several minutes or even longer.

Although in most cases, pivot table is used to do static data analysis. However, we notice many customers use it to do real-time data analysis. That is the data in the original table model changes constantly. If every change requires the *calculate()* to be called to update the display, it is certainly not acceptable. To solve this problem, we introduce a flag in *PivotDataModel*. If you call *PivotDataModel#setAutoUpdate(true)*, we will listen to the *TableModelEvent* fired in the original table model and update the pivot data incrementally. The result is instead taking a few minutes to do *calculate()*, it will only take a few seconds to when a row in the original table model is added, deleted or modified. If the event is a large scale event such as tableDataChanged or tableStructureChanged, we will still call *calculate()*. So if you should fire a few small scale events, please do that instead of firing one large scale event. Saying that, you many need to do some experiments in order to find out the best solution. For example, *calculate()* takes one minute and each row updated event takes 1 second. If 900 out of 1000 rows are updated at one time, you probably should fire tableDataChanged instead of firing 900 tableRowUpdated events because 1 minute is less than 900 seconds.

Filtering

We already mentioned user can click on the filter button on the fields (except data field) to uncheck the value from the *CheckBoxList* popup to do filtering. There is of course one way to filter data. However this filtering is done after *PivotDataModel* processed all the data. As you may know, if you have a huge data, *PivotDataModel*'s *calculate()* method will spend a lot of time to read and calculate the pivot data. In the other word, the user filtering way doesn't help to relieve the burden on *PivotDataModel* at all.

We would like to introduce a class called *FilterableTableModel*, a class in *JIDE Grids*, to combine use with *PivotDataModel*. Here is how it works.

```
FilterableTableModel filterModel = new FilterableTableModel(tableModel)
filterModel.addFilter(…);
filterModel.refresh();
PivotDataModel pivotModel = new PivotDataModel(filterModel);
```

As you can see, instead of passing the tableModel, which is huge, directly to *PivotDataModel*, we piped it to *FilterableTableModel* first. You can add many filters to *FilterableTableModel*. The refresh() method will filter away all unnecessary rows so that *PivotDataModel* sees a table model which has much less rows. A typical use case is to use fitler to filter away incomplete or obviously wrong data. People sometimes called it data cleansing.

17

PivotDataSource

As Pivot Table could potentially deal with large amount of data, it is important to optimize how the data is retrieved and filtered. Obviously the most used storage for large data source is still the database. All databases have the index feature which makes the filter a lot faster than our *FilterableTableModel*. This is the reason we introduced *PivotDataSource* interface.

PivotDataSource is very similar to *TableModel* interface. By introducing this interface, it is possible to use any compatible data as the data source for *PivotDataModel*. *PivotDataSource* also provides methods related to filtering. If you use this interface, you can totally skip the *FilterableTableModel* we used internally in *PivotDataModel*. If you can provide a more efficient filtering engine to implement this interface in the right way, it could make the pivot table faster.

Persistent the Pivot Layout as XML Document

PivotTablePersistenceUtils is the class that can save *PivotTablePane* as xml file and load them back. The reason we didn't build this feature into *PivotTablePane* itself but provide a separate class for it is because we think you might have your own to persistent it as xml format in your application.

All methods on *PivotTablePersistenceUtils* are static. There are several overload *load()* methods which can load shortcut definition from an xml file, or InputStream, or Document (org.w3c.dom.Document). You can also load shortcut definition from multiple files by setting "append" parameter to true in the *load()* method.

The save() method will save shortcut definition as Document or file. Please note *save()* method will use classes *XMLSerializer* and *OutputFormat*. Those two classes are part of xerces.jar before but now they are also part of JDK5. If you are using JDK5 and above, there is nothing you need to do. However if you are still using JDK1.4.2 or below, you need to include xerces.jar in your classpath. Otherwise you will get NoClassDefFoundError during runtime.

Performance of PivotDataModel

As *PivotDataModel* deals large amount of data, the performance is a very important factor for consideration. We did some benchmark (based on JIDE release 2.2.2). See table below.

You can do this performance test yourself. We used PivotTableDemo.java as the test case. You change one line in *createProductReportsTableModel* method to add more rows. For example, if the value of the loop is 1000, there will be 2082 * 1000 rows.

```
for (int i = 0; i < 1000; i++) {
    data.add(lineData);
}
```

This result is tested on a Dell XPS 600, Windows XP, Intel Pentium D 3.00GHz CPU, 2GB RAM. Actual result could vary on your computer.

# of rows	2082	20,820	208,200	2,082,000	10,410,000	20,820,000
Extra memory (in KB)	~*	23	2560	26,640	141,253	271,475
calculate() (in ms)	47	156	1,032	9,797	45,406	111,656

* We used *Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()* to find out how much extra memory is used. It is not very accurate and the best way which is why we couldn't measure the memory usage of the 2082-row case correctly.

As you can see from the table above, the time increases linearly with the number of rows. The performance gets bad once there are more than 2 million rows with around 10 seconds. As *calculate()* is called for many UI operations (such as add, remove or rearrange a field, apply a filter), it means that's time that user will have to wait for those operations.

The *calculate()* method spent about half of the total time reading the data from the original *TableModel*. The rest of time is spent on adding the row keys and column keys to a map. So the speed of *TableModel.getValueAt()* has a large impact on the performance. In fact, in *PivotTableDemo*, we used *CalculatedTableModel*. It is slower as we have to calculate the month, year and date from a *Date* object. If your table model uses *DefaultTableModel*, the performance could be better than this performance test.

The more row and column fields, the slower is *calculate()*.

The initial startup time is just one aspect of the performance. After the pivot table is initialized, when you scroll up and down to view the pivot table data, the performance is again determined by the *TableModel.getValueAt()*. You can use *CachedTableModel* class we mentioned earlier to boost. However this will add to the initial startup time because it has to prepare a cache for the cell values. It will also increase the memory consumption. Use it with caution.

```
PivotDataModel pivotDataModel = new PivotDataModel(new CachedTableModel(tableModel));
```

Subtotal and grand total cells also make the viewing the table slower. It is because the time takes to collect all the values of the data fields that match with the sub or grand total condition.

19

AggregateTable

Before introducing AggregateTable, let's look at a SortableTable. See below.

CategoryName	ProductName	ProductSales	ShippedDate
Dairy Products	Queso Cabrales	168	Aug 16, 1994
Grains/Cereals	Singaporean Hokkien Fried Mee	98	Aug 16, 1994
Dairy Products	Mozzarella di Giovanni	174	Aug 16, 1994
Produce	Tofu	167.4	Aug 10, 1994
Produce	Manjimup Dried Apples	1,696	Aug 10, 1994
Seafood	Jack's New England Clam Chowder	77	Aug 12, 1994
Produce	Manjimup Dried Apples	1,261.4	Aug 12, 1994
Condiments	Louisiana Fiery Hot Pepper Sauce	214.2	Aug 12, 1994
Grains/Cereals	Gustaf's Knäckebröd	95.76	Aug 15, 1994
Grains/Cereals	Ravioli Angelo	222.3	Aug 15, 1994
Condiments	Louisiana Fiery Hot Pepper Sauce	336	Aug 15, 1994
Confections	Sir Rodney's Marmalade	2,462.4	Aug 11, 1994
Dairy Products	Geitost	47.5	Aug 11, 1994
Dairy Products	Camembert Pierrot	1,088	Aug 11, 1994
Dairy Products	Gorgonzola Telino	200	Aug 16, 1994
Beverages	Chartreuse verte	604.8	Aug 16, 1994
Confections	Maxilaku	640	Aug 16, 1994
Beverages	Guaraná Fantástica	45.9	Aug 23, 1994
Meat/Poultry	Pâté chinois	342.72	Aug 23, 1994
Produce	Longlife Tofu	168	Aug 23, 1994
Beverages	Chang	304	Aug 15, 1994
Confections	Pavlova	486.5	Aug 15, 1994
Seafood	Inlagd Sill	380	Aug 15, 1994
Dairy Products	Raclette Courdavault	1,320	Aug 15, 1994
Meat/Poultry	Perth Pasties	393	Aug 17, 1994

Figure 5 A TableModel

As you can see, it is a pretty large table model. There are many cells with the same value in *CategoryName* and *ProductName* columns. You can find see that once we sort the first column and the second column. See below. As we used JIDE Grids *SortableTable* which supports multiple sorted columns, we can see those same value cells.

20

Figure 6 A Table Model with the first column sorted

After you look at this sorted table, you will start to think that maybe we can use *CellSpan* to group the same value cells to make it easier to read. Or we can even collapse the *CellSpan* to save space. That's exactly the reason for *AggregateTable*.

See below for an example which is the aggregate table for the table model above. In this case, we aggregated on *CategoryName* and *ProductName* columns.

Figure 7 AggregateTable

21

As the name indicated, *AggregateTable* aggregates cells of the same value into one cell span and add +/- button to those aggregated cell. You can decide which columns to aggregate. See below for an example code.

```
AggregateTable aggregateTable = new AggregateTable(tableModel);
aggregateTable.aggregate(new String[]{"CategoryName", "ProductName"});
```

Internally, we will create a *PivotDataModel* to do the calculation just like in PivotTablePane. However we represent the calculation result through a new *AggregateTableModel* class which is then used by *AggregateTable*.

Difference from PivotTablePane

Even though *AggregateTable* is built on top *PivotDataModel* and both of them can use the same *TableModel* as data source, *PivotTablePane* and *AggregateTable* are used for two totally difference purpose.

PivotTablePane is used to display summaries of the original data. User usually doesn't care too much about the original data but cares more about the statistical summary of the original data. In fact, there is no way see the original data through the *PivotTablePane* UI. Sometimes user doesn't even know what kind of information in the original data and wants to use *PivotTablePane* as an experimental tool to analysis the data.

AggregateTable is used to display the original data in an aggregated view. Each value in the original data will find its place in *AggregateTable*. User (or developer) also knows beforehand what columns are in the table model so that they can decide which columns are aggregated columns. For example, *CategoryName* and *ProductName* can be used as aggregate columns. If you try to aggregate on the *Sales* column, it probably doesn't make any sense.

Even though it looks like a regular Table, *AggregateTable* is actually a special pivot table. You can view it as a pivot table without column fields. Aggregated columns are row fields and non-aggregated columns are data fields. To prove this statement, here is a screenshot of the same *TableModel* in *PivotTablePane*. You can compare the two screenshots to see the difference.

Sales	Year	Quarter	Month	Week	Drop Column Fields Here				
Category Name ↑ ⌄		Product Name ↑ ⌄		Sales	Year	Quarter	Month	Week	
⊟ Beverages		Chai		$1,065.60				127	
		Chang		$2,052.00				150	
		Chartreuse verte		$2,001.60				204	
		Guaraná Fantástica		$288.54				231	
		Ipoh Coffee		$3,532.80				216	
		Lakkalikööri		$1,153.44				166	
		Laughing Lumberjack...		$42.00				40	
		Outback Lager		$1,089.00				205	
		Rhönbräu Klosterbier		$322.40				170	
		Sasquatch Ale		$380.80				85	
		Steeleye Stout		$2,786.40				207	
Beverages Total				$14,714.58				1,801	
⊟ Condiments		Aniseed Syrup		$240.00				40	

Note that the cells in the data table actually summaries. For example, the first Sales cell $1065.60 is actually Summary of three values. You can see them in the table below. Other three

22

cells don't display any information is because the data type and converter context don't match with the summary type.

CategoryName	ProductName	ProductSales	ShippedDate	Year	Quarter	Month	Week
⊟ Beverages	⊟ Chai	518.4	Sep 26, 1994	1994	Qtr 3	Sep	40
		259.2	Oct 6, 1994	1994	Qtr 4	Oct	41
		288	Nov 10, 1994	1994	Qtr 4	Nov	46
	⊟ Chang	304	Aug 15, 1994	1994	Qtr 3	Aug	34
		608	Aug 23, 1994	1994	Qtr 3	Aug	35
		532	Sep 23, 1994	1994	Qtr 3	Sep	39
		608	Oct 12, 1994	1994	Qtr 4	Oct	42
	⊟ Chartreuse verte	604.8	Aug 16, 1994	1994	Qtr 3	Aug	34
		86.4	Aug 22, 1994	1994	Qtr 3	Aug	35
		864	Oct 11, 1994	1994	Qtr 4	Oct	42
		388.8	Nov 9, 1994	1994	Qtr 4	Nov	46
		57.6	Nov 14, 1994	1994	Qtr 4	Nov	47
	⊟ Guaraná Fantástica	45.9	Aug 23, 1994	1994	Qtr 3	Aug	35
		100.8	Aug 31, 1994	1994	Qtr 3	Aug	36
		41.04	Sep 9, 1994	1994	Qtr 3	Sep	37
		43.2	Oct 13, 1994	1994	Qtr 4	Oct	42
		21.6	Sep 21, 1994	1994	Qtr 3	Sep	39

Appendix: Expression for ExpressionCalculatedColumn

In the current release, we leveraged an expression parser from an open source project called prefuse (http://prefuse.org). Prefuse is a visualization toolkit which has nothing to do with *JIDE Pivot Grid*. However since they have an expression parser which is exactly what we need, we just use it for now until we decide to make one on our own. Prefuse is released under the BSD (Berkeley Standard Distribution) license, making it free to use for both commercial and non-commercial purposes.

What we need is a way to calculate a value from the value of an existing column. Let's use the same example we used before. Your table model contains quality and unit price columns but you are interested in the total amount which can be calculated as quality * unit price. If you are writing java code to do it, you can certainly use *AbstractCalculatedColumn* class and do the calculation in your code. However if you want to expose this feature to your end user who doesn't know how to write javadoc, you will have to use this *ExpressionCalculatedColumn*.

```
calculatedTableModel.addColumn(new ExpressionCalculatedColumn(tableModel, "Amount", " [Quality] *
[UnitPrice]"));
```

Below is the detailed document adapted from prefuse's javadoc.

Literal Values and Data Field References

The fundamental building blocks of the expression language, representing data values.

Boolean literals (TRUE, FALSE)

The boolean literals representing true and false conditions, parsed to type boolean

Integer literals (1, -5, 12340)

Undecorated, non-decimal numbers are parsed as numbers of type int

Long literals (1L, -5L, 12340L)

Integer values decorated with the suffix "L" are parsed as numbers of type long

Double literals (1.0, 3.1415, 1e-35, 2.3e6)

Numbers with decimals or exponents in scientific notation are parsed as numbers of type double

Float literals (1.0f, 3.1415f, 1e-35f, 2.3e6f)

Floating-point values decorated with the suffix "f" are parsed as numbers of type float

String literals ("some text", 'a label')

Text strings placed in double (") or single (') quotations are parsed as String literals

Null literal (null)

The string null is parsed as an ObjectLiteral of type null.

Data field references (_strokeColor, [a data field])

Free-standing strings or those placed within brackets are parsed as a reference to the data field of that name. Brackets are required for any fields that include spaces or other unusual characters in their name (e.g., characters like +, -, *, etc), or conflict with an existing keyword For example, true parses to a boolean literal while [true] parses to a reference to a data field named 'true'.

Operators and Control Flow

Basic operators and control flow structures for the expression language.

x + y (addition)

Add x and y

x - y (subtraction)

Subtract y from x

x * y (multiplication)

Multiply x and y

x / y (division)

Divide x by y

x ^ y (exponentiation, pow)

Raise x to the exponent y

x % y (modulo)

Return the remainder of x divded by y

x = y, x == y (equality)

Indicates if x and y are equal

x != y, x <> y (inequality)

Indicates if x and y are not equal

x > y (greater than)

Indicates if x is greater than y

x >= y (greater than or equal to)

Indicates if x is greater than or equal to y

x < y (less than)

Indicates if x is less than y

x <= y (less than or equal to)

Indicates if x is less than or equal to y

x AND y, x && y (and)

Indicates if both x and y are true

x OR y, x || y (or)

Indicates if either x or y is true

NOT x, !x (not)

Indicates if the negation of x is true

x XOR y (exclusive or)

Indicates if one, but not both, of x or y is true

IF test THEN x ELSE y (if-then-else)

Evaluates the predicate test, and if true evaluates and returns the expression x, and if false evaluates and returns the expression y

() (parentheses)

Groups expressions together to enfore a particular order of evaluation. For example, 1+2*3 evaluates to 7, while (1+2)*3 evaluates to 9.

Mathematical Functions

Functions for performing mathematical calculations.

ABS(x)

Returns the absolute value of x

ACOS(x)

Returns the inverse cosine (arc cosine) of a x

ASIN(x)

Returns the inverse sine (arc sine) of a x

ATAN(x)

Returns the inverse tangent (arc tangent) of a x

ATAN2(y, x)

For the Cartesian coordinates x, y return the polar coordinate angle theta

CEIL(x), CEILING(x)

Returns the nearest integer value greater than or equal to x.

COS(x)

Returns the cosine of x

COT(x)

Returns the cotangent of x

DEGREES(x)

Converts x from radians to degrees

EXP(x)

Returns the value of e (the base of natural logarithms) raised to the x power

FLOOR(x)

Returns the nearest integer value less than or equal to x.

LOG(x), LOG(b, x)

With one argument, returns the natural logarithm (logarithm base e) of x

With two arguments, returns the logarithm of x for the provided base b

LOG2(x)

Returns the logarithm base 2 of x

LOG10(x)

Returns the logarithm base 10 of x

MAX(a, b, c, ...)

Returns the maximum value among the provided arguments

MIN(a, b, c, ...)

Returns the minimum value among the provided arguments

MOD(x, y)

Returns x modulo y (the remainder of x divided by y)

PI()

Returns the constant π (= 3.1415926535...), the ratio between the circumference and diameter of a circle

POW(x, y), POWER(x, y)

Return the value of x raised to the exponent y

RADIANS(x)

Converts x from degrees to radians

RAND()

Returns a random floating-point value between 0 and 1

ROUND(x)

Returns the value of x rounded to the nearest integer

SIGN(x)

Returns the sign of x: 1 for positive, -1 for negative

SIN(x)

Returns the sine of x

SQRT(x)

Returns the square root of x

SUM(a, b, c, ...)

Returns the sum of the provided input value

TAN(x)

Returns the tangent of x

SAFELOG10(x)

Returns a "negative safe" logarithm base 10 of x, equivalent to SIGN(x) * LOG10(ABS(x))

POW(x)

Returns a "negative safe" square root of x, equivalent to SIGN(x) * SQRT(ABS(x))

String Functions

Functions for processing text strings.

CAP(str)

Capitalize words in the string str. Individual words/names will be given uppercase first letters, with all other letters in lowercase.

CONCAT(a, b, c, ...)

Concatenate the input strings into one resulting string.

CONCAT_WS(sep, a, b, c, ...)

Concatenate with separator. Concatenates the input strings into one resulting string, placing the string sep between each of the other arguments

FORMAT(x, d)

Format the number x as a string of the type "#,###.##", showing d decimal places

INSERT(str, pos, len, newstr)

Replaces the substring of length len starting at position pos in input string str with the string newstr

LEFT(str, len)

Returns the leftmost len characters of string str

LENGTH(str)

Returns the length, in characters, of the input string str

LOWER(str), LCASE(str)

Returns the string str mapped to lowercase letters

LPAD(str, len, pad)

Pad the left side of string str with copies of string pad, up to a total padding of len characters

MID(str, pos, len)

Return a substring of str of length len, starting at position pos

POSITION(substr, str)

Returns the starting position of the first occurrence of substring substr in the string str. Returns -1 if the substring is not found.

REVERSE(str)

Returns a reversed copy of the input string str

REPEAT(str, count)

Returns a string consisting of str repeated count times

REPLACE(str, orig, replace)

Returns a copy of str in which all occurrences of orig have been replaced by replace

RIGHT(str, len)

Returns the len rightmost characters of stringstr

RPAD(x)

Pad the right side of string str with copies of string pad, up to a total padding of len characters

SPACE(n)

Returns a string consisting of n whitespace characters

SUBSTRING(str,pos), SUBSTRING(str,pos,len)

For two arguments, returns the substring of str starting at position pos and continuing to the end of the string.

For three arguments, returns the substring of str of length len, beginning at position pos

UPPER(str), UCASE(str)

Returns the string str mapped to uppercase letters