Interface Graphique en Java 1.6

Tables

Sébastien Paumier

Les acteurs

- `JTable`: la vue graphique
- `TableModel`: le modèle de données des cellules
- `TableColumnModel`: le modèle gérant les colonnes (titre, position, sélection, ...)
- `JTableHeader`: rendu des en-têtes de colonnes

Les acteurs

- `ListSelectionModel`: le modèle de sélection des lignes de cellules
- `TableCellRenderer`: rendu des cases de la table
- `TableCellEditor`: l'éditeur de cellules

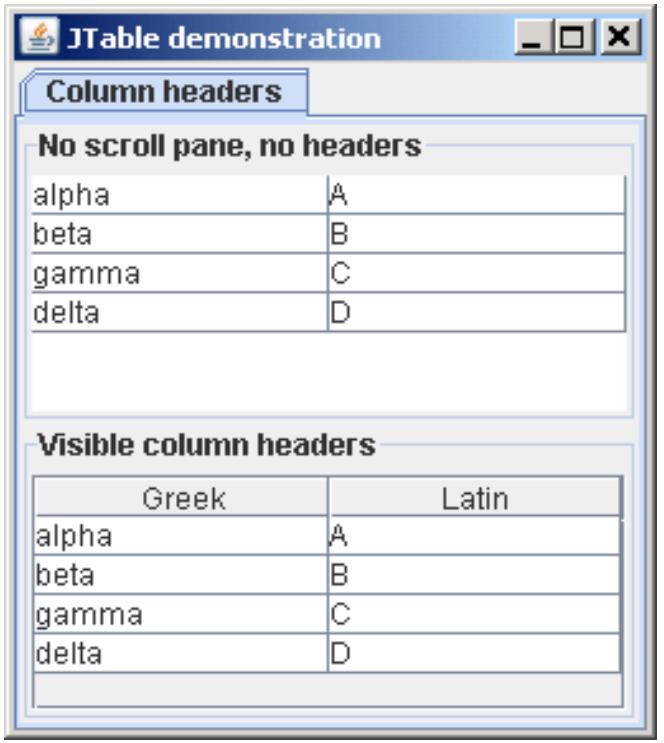
Créer une table

- plusieurs constructeurs:
  - `JTable(TableModel tm,TableColumnModel tcm,ListSelectionModel lsm);`
  - `JTable(TableModel tm);`
  - `JTable(TableModel tm,TableColumnModel tcm);`
  - `JTable(int rows,int columns);`
  - `JTable(Object[][] rowData,Object[] columnNames);`

Les titres de colonnes

- n'apparaissent que si l'on met la **JTable** dans un **JScrollPane**:

```
private static Component createFileTable1() {
    String[][] cells=new String[][]{
        {"alpha","A"},{"beta","B"},
        {"gamma","C"},{"delta","D"}};
    String[] names=new String[]{"Greek","Latin"};
    JPanel p=new JPanel(new GridLayout(2,1));
    JPanel up=new JPanel(new BorderLayout());
    up.setBorder(BorderFactory.createTitledBorder(
            "No scroll pane, no headers"));
    up.add(new JTable(cells,names));
    p.add(up);
    JPanel down=new JPanel(new BorderLayout());
    down.setBorder(BorderFactory.createTitledBorder(
            "Visible column headers"));
    down.add(new JScrollPane(new JTable(cells,names)));
    p.add(down);
    return p;
}
```



Le TableModel

- ses méthodes:
  - `int getRowCount()`
  - `int getColumnCount()`
  - `Object getValueAt(int row,int column)`
  - `String getColumnName(int column)`
  - `Class<?> getColumnClass(int column)`
  - `boolean isCellEditable(int row,int column)`
  - `void setValueAt(Object value,int row,int column)`

Le TableModel

- la gestion des listeners se fait avec `addTableModelListener` et `removeTableModelListener`
- le `AbstractTableModel`:
  - gère les listeners
  - `getColumnName`: "A", "B", ...
  - `getColumnClass`: `Object.class`
  - les cellules sont non éditables
  - `setValueAt` est sans effet

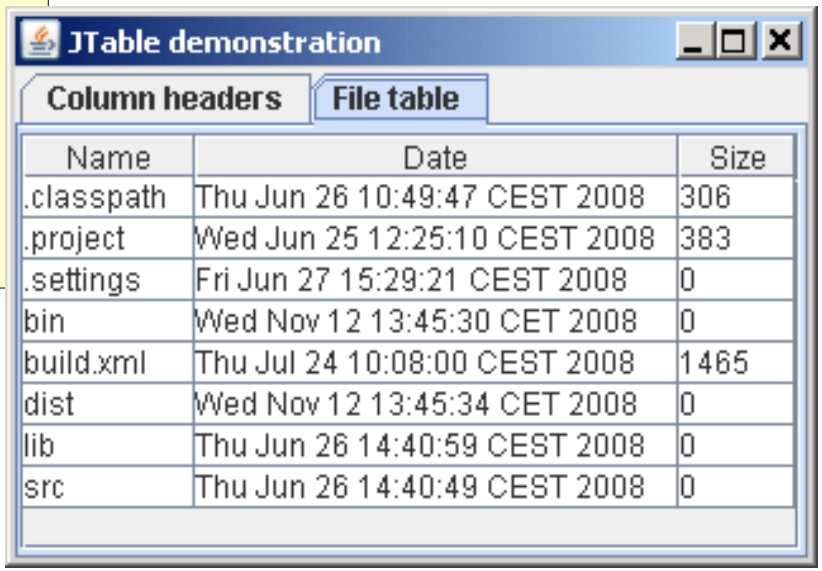
Le FileTableModel

- on crée une table à trois colonnes:

```
public class FileTableModel extends AbstractTableModel {

    private File[] files;
    private static String[] columnNames=new String[]{"Name","Date","Size"};

    public FileTableModel(File directory) {
        if (directory==null || !directory.isDirectory()) {
            throw new IllegalArgumentException("FileTableModel requires a directory");
        }
        files=directory.listFiles();
    }

    @Override
    public int getColumnCount() {
        return columnNames.length;
    }

    @Override
    public int getRowCount() {
        return files.length;
    }
...
}
```

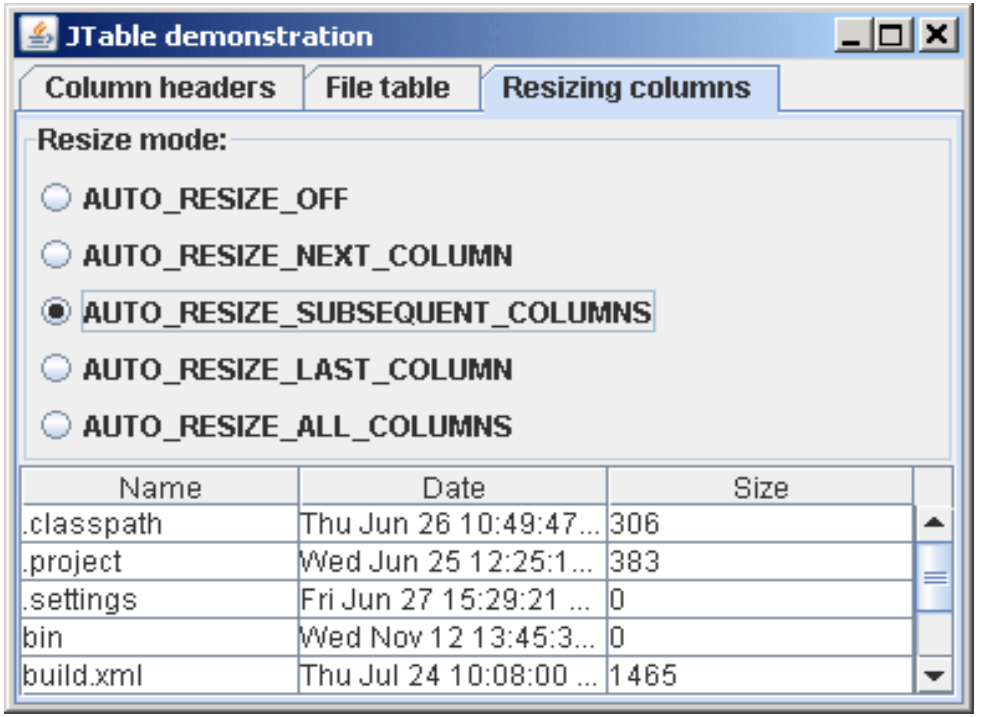
Le FileTableModel

```
public class FileTableModel extends AbstractTableModel {

...

    @Override
    public String getColumnName(int column) {
        return columnNames[column];
    }

    @Override
    public Object getValueAt(int rowIndex, int columnIndex) {
        File f=files[rowIndex];
        switch (columnIndex) {
        case 0: return f.getName();
        case 1: return new Date(f.lastModified());
        case 2: return f.length();
        default: throw new IllegalArgumentException(
                    "Invalid column index: "+columnIndex);
        }
    }
}
```



Redimensionner les colonnes

- 5 modes possibles à définir avec **`setAutoResizeMode`**:



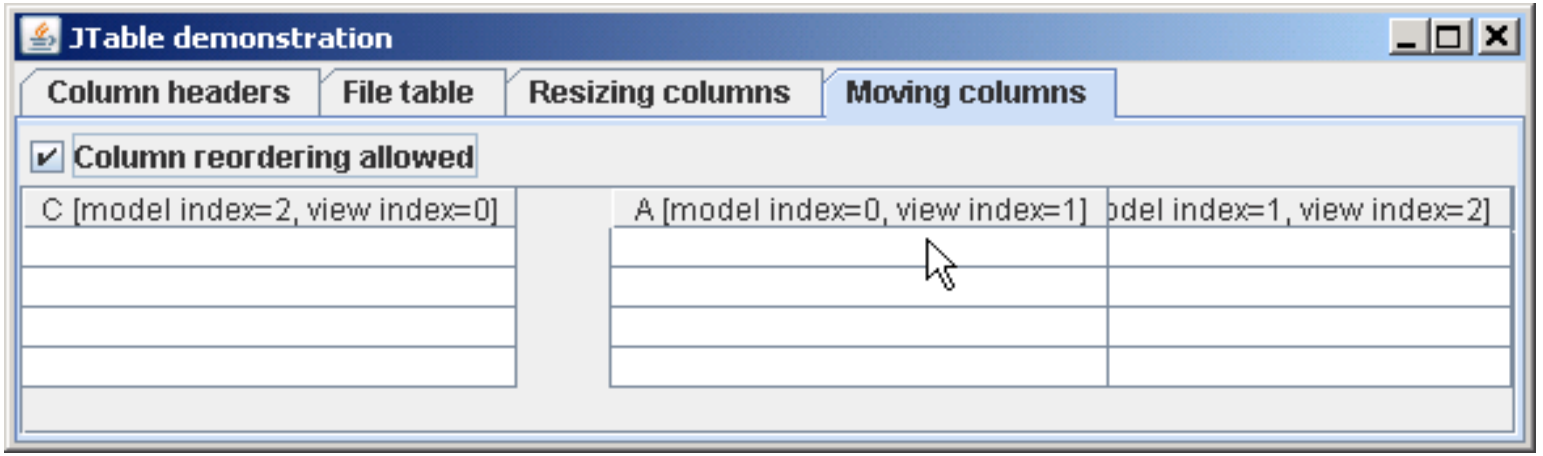
Déplacer les colonnes

- le **TableColumnModel** gère une indirection entre la position des colonnes dans le modèle et celle dans la vue:

```
private static Component createFileTable3() {
    final JTable t=new JTable(4,3);
    t.getColumnModel().addColumnModelListener(
        new TableColumnModelListener() {
        @Override public void columnAdded(TableColumnModelEvent e) {
            updateHeaders();
        }
...
        private void updateHeaders() {
             for (int i=0;i<t.getColumnCount();i++) {
                int viewIndex=t.convertColumnIndexToView(i);
                if (viewIndex==-1)  continue;
                TableColumn column=t.getColumnModel().getColumn(viewIndex);
                column.setHeaderValue(t.getColumnName(viewIndex)
                  +" [model index="+i+", view index="+viewIndex+"]");
             }
        }
    });
...

}
```

Déplacer les colonnes

- on peut ainsi savoir où sont les colonnes dans le modèle:




```
...
    JPanel p=new JPanel(new BorderLayout());
    JCheckBox check=new JCheckBox("Column reordering allowed",true);
    check.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            t.getTableHeader().setReorderingAllowed(((JCheckBox)e.getSource()).isSelected());
        }});
    p.add(check,BorderLayout.NORTH);
    p.add(new JScrollPane(t));
    return p;
}
```

TableColumn

- plus généralement, le `TableColumnModel` gère des `TableColumn` contenant:
  - l'index correspondant dans le modèle
  - une largeur
  - un éditeur
  - un renderer

```
TableColumn(int modelIndex,
  int width,
  TableCellRenderer cellRenderer,
  TableCellEditor cellEditor)
```

Les renderers

- on peut définir le renderer soit par colonne:
  - `tableColumn.setCellRenderer(TableCellRenderer renderer)`
- soit par type d'objet:
  - `table.setDefaultRenderer(Class<?> columnClass, TableCellRenderer renderer)`
- dans ce dernier cas, Swing utilise `getColumnClass` pour savoir quel renderer appeler

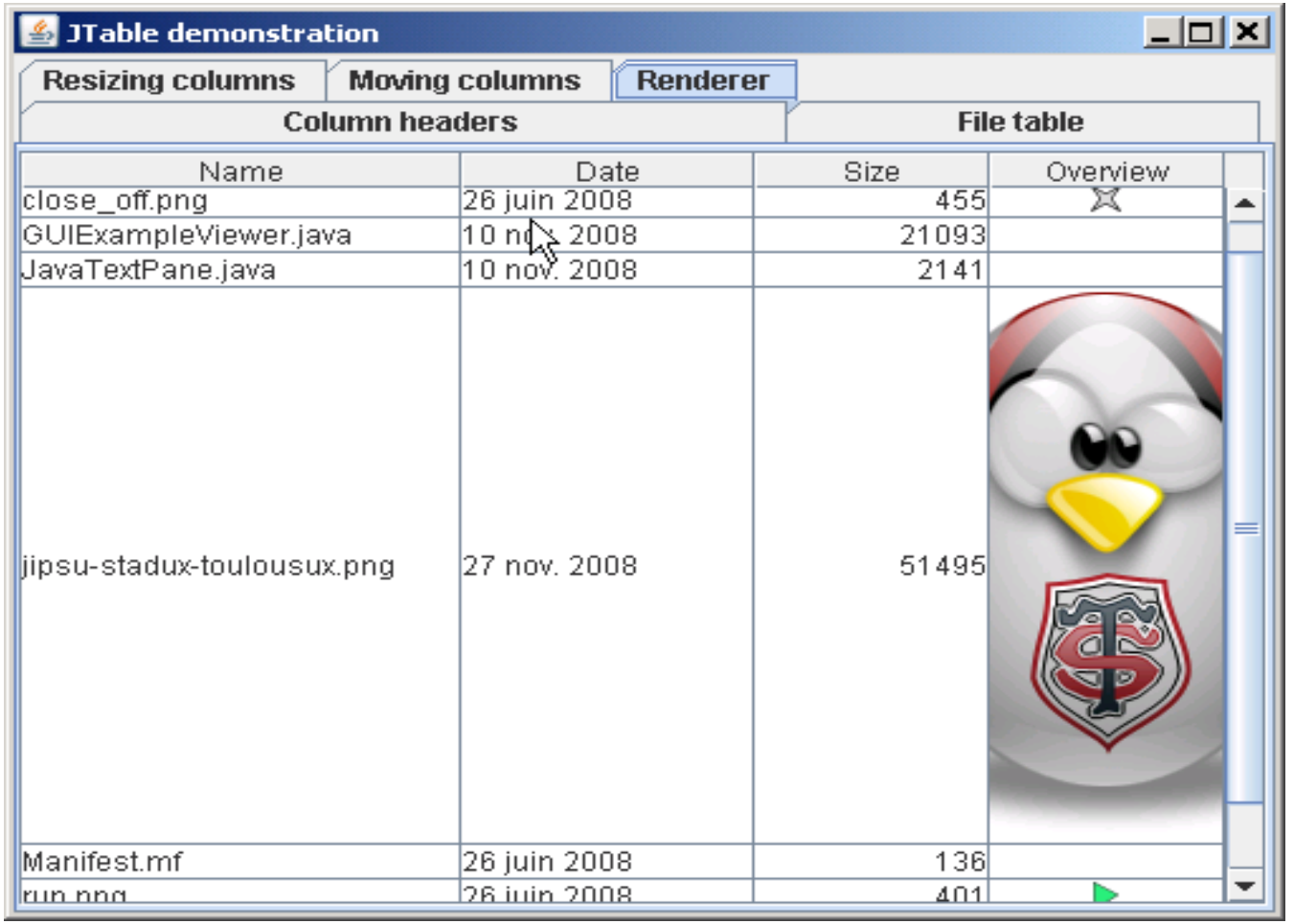
Les renderers

- exemple: renderer qui affiche une image

```
@SuppressWarnings("serial")
private static Component createFileTable4() {
    final JTable t = new JTable(new AdvancedFileTableModel(
                        new File("./src/fr/umlv/ig/example_viewer")));
    t.setDefaultRenderer(ImageIcon.class, new DefaultTableCellRenderer() {
        @Override public Component getTableCellRendererComponent(JTable table,
                    Object value, boolean isSelected, boolean hasFocus,
                    int row, int column) {
            ImageIcon icon = (ImageIcon) value;
            super.getTableCellRendererComponent(table, null, isSelected, hasFocus, row, column);
            setHorizontalAlignment(SwingConstants.CENTER);
            if (icon != null) {
                setIcon(icon);
                final int r = row;
                final int h = icon.getIconHeight();
                if (t.getRowHeight(r) < h) {
                    EventQueue.invokeLater(new Runnable() {
                        @Override
                        public void run() {
                            t.setRowHeight(r, h);
                        }
                    });
                }
            }
            return this;
        }
    });
    return new JScrollPane(t);
}
```

si nécessaire, on adapte la hauteur de la ligne à celle de l'image

Les renderers

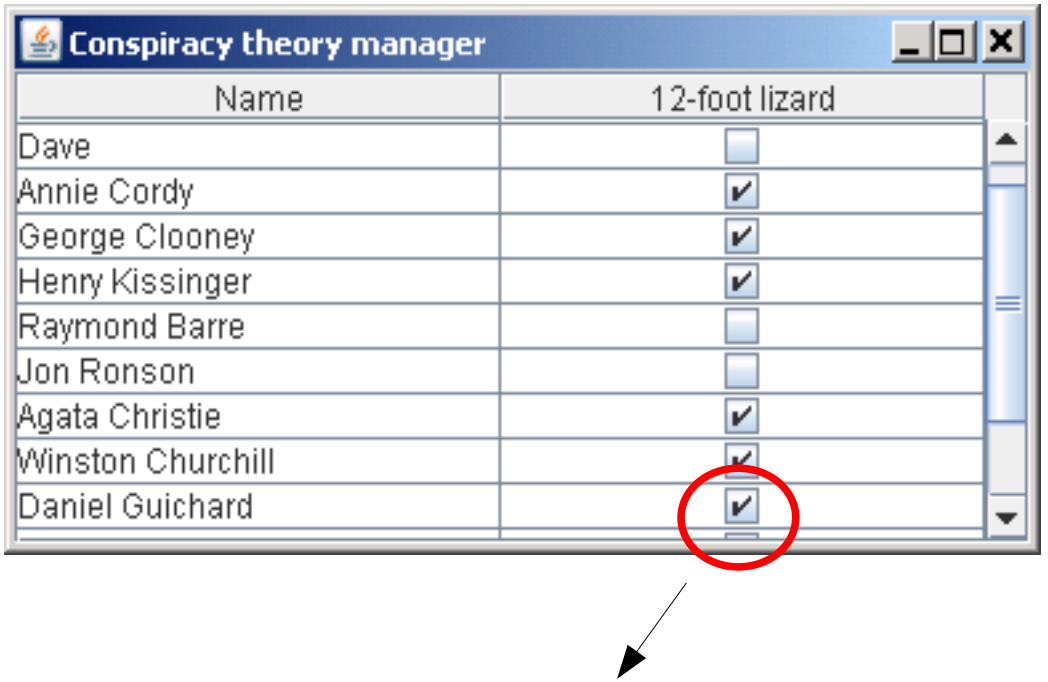


Les éditeurs

- pour éditer et modifier une cellule, il faut:
  - que la cellule soit déclarée éditable:
    ```
    boolean isCellEditable(int rowIndex,
    int columnIndex)
    ```
  - qu'on puisse modifier les données:
    ```
    void setValueAt(Object value,
    int rowIndex, int columnIndex)
    ```
  - qu'il y ait un éditeur installé (c'est le cas par défaut)

Les éditeurs

- exemple: édition de valeurs booléennes

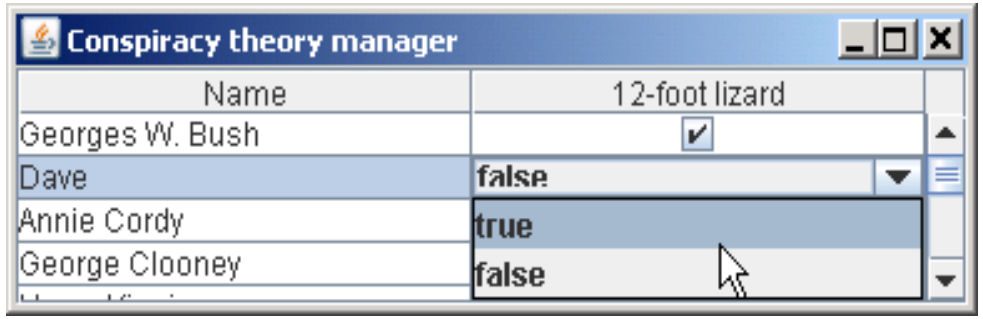


renderer/éditeur par défaut pour les **`Boolean`**

Les éditeurs

- on peut utiliser 3 sortes d'éditeurs prédéfinis:
  - **`DefaultCellEditor(JCheckBox checkBox)`**
  - **`DefaultCellEditor(JComboBox comboBox)`**
  - **`DefaultCellEditor(JTextField textField)`**

```
JTable t=new JTable(model);
t.setDefaultEditor(Boolean.class,
    new DefaultCellEditor(
        new JComboBox(
            new Boolean[]{true,false})));
```



Un composant dans une table

- on peut utiliser un pseudo-éditeur pour mettre un composant dans une **JTable**
- exemple avec un **JButton**:

```
public class MyTableCellEditor extends AbstractCellEditor implements TableCellEditor {

    private JButton button;
    private BilderbergTableModel model;

    public MyTableCellEditor(final Component parent,BilderbergTableModel model) {
        this.model=model;
        button=new JButton();
        button.addActionListener(new ActionListener() {
            @SuppressWarnings("synthetic-access")
            @Override public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(parent,"You don't really want to access to "
                                              +button.getText()+".","",JOptionPane.ERROR_MESSAGE);
                fireEditingStopped();
            }
        });
    }
...
}
```

on rend la main après avoir géré le clic

Un composant dans une table

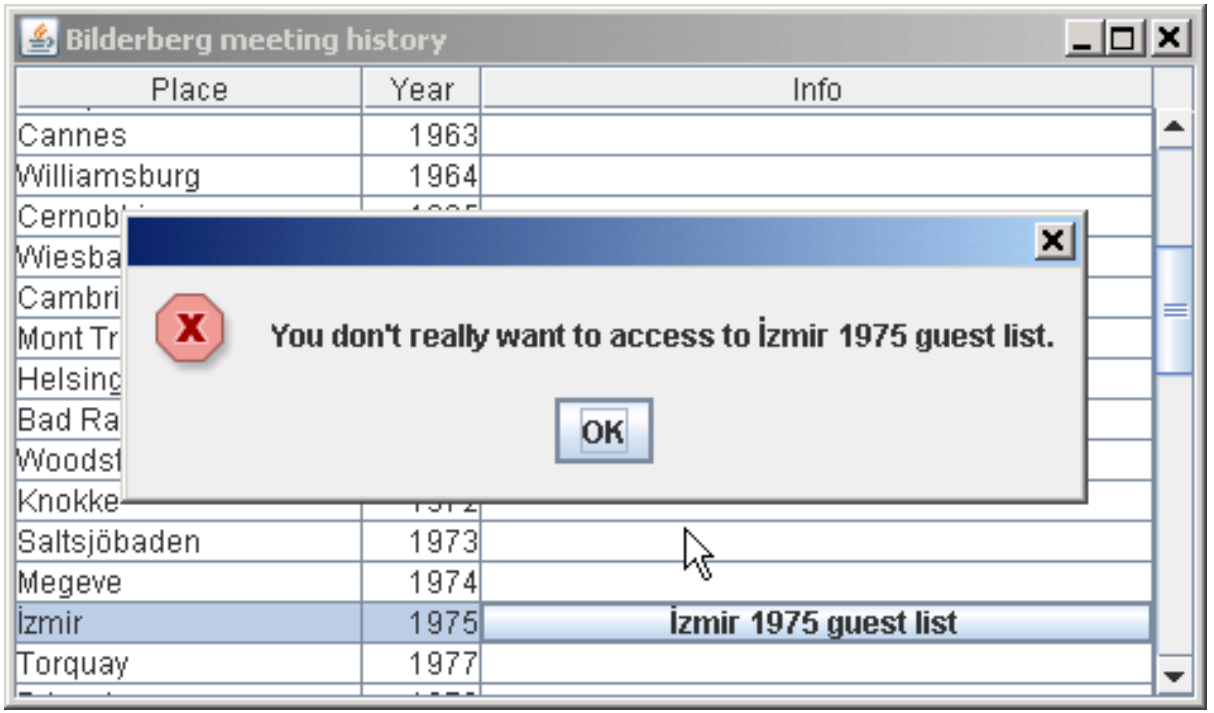
- `getTableCellEditorComponent` doit simplement mettre le texte du bouton à jour:

```
public class MyTableCellEditor extends AbstractCellEditor implements TableCellEditor {

...
    @Override
    public Component getTableCellEditorComponent(final JTable table, Object value,
                                                 boolean isSelected, int row, int column) {
        button.setText(model.getValueAt(row,0)+" "+model.getValueAt(row,1)+" guest list");
        return button;
    }

    @Override
    public Object getCellEditorValue() {
        return null;
    }
...
}
```

comme on a un pseudo-éditeur, on se moque de sa valeur de retour

Un composant dans une table

- problème d'affichage: le bouton n'apparaît que le temps du clic



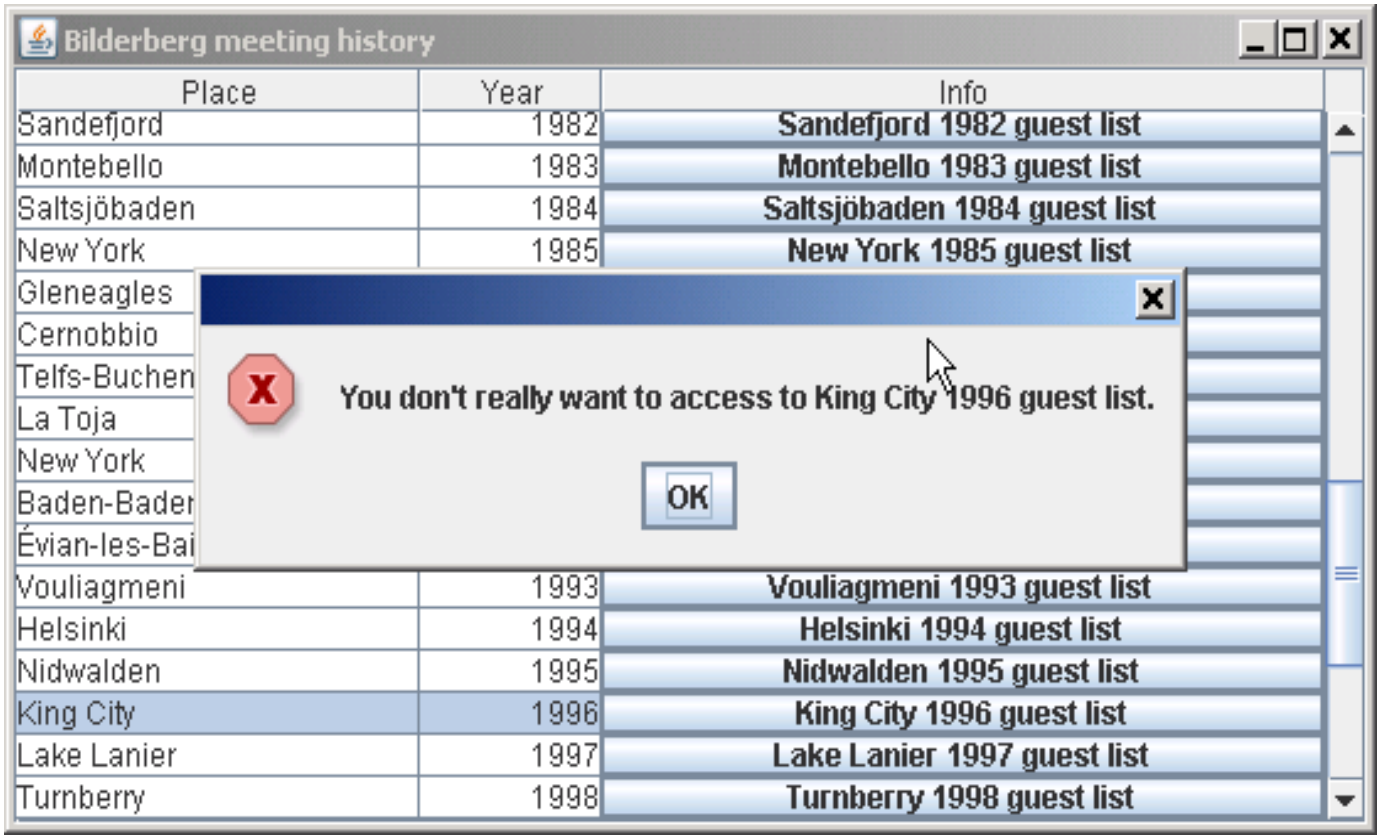
Un composant dans une table

- solution: mettre comme renderer un bouton semblable

```
JTable t=new JTable(model);
t.setDefaultRenderer(BilderbergTableModel.class,new DefaultTableCellRenderer() {

    JButton b=new JButton();

    @Override
    public Component getTableCellRendererComponent(JTable table, Object value,
                                                   boolean isSelected, boolean hasFocus,
                                                   int row, int column) {
        b.setText(model.getValueAt(row,0)+" "+model.getValueAt(row,1)+" guest list");
        return b;
    }
    }
);
```

Un composant dans une table

- résultat:



Le TableModelListener

- pour écouter les événements sur les données du **`TableModel`**:

```
void addTableModelListener(TableModelListener l)
```

- ce listener possède les méthodes suivantes:
  - **`int getFirstRow()`**
  - **`int getLastRow()`**
  - **`int getColumn()`**: *n* ou **`ALL_COLUMNS`**
  - **`int getType()`**: **`INSERT`**, **`UPDATE`**, **`DELETE`**

Le TableModelListener

- exemple: créer un adapter qui gère la mise en évidence des modifications des cellules

on relaie les événements du vrai modèle, mais en les manipulant

```
public class FlashyTableAdapter extends AbstractTableModel {

    public FlashyTableAdapter(TableModel model) {
        this.model=model;
        model.addTableModelListener(new TableModelListener() {
            @Override
            public void tableChanged(TableModelEvent e) {
                int firstRow=e.getFirstRow();
                int lastRow=e.getLastRow();
                int firstColumn=e.getColumn();
                int lastColumn=firstColumn;
                if (firstColumn==TableModelEvent.ALL_COLUMNS) {
                    firstColumn=0;
                    lastColumn=getColumnCount()-1;
                }
                for (int row=firstRow;row<=lastRow;row++) {
                    for (int c=firstColumn;c<=lastColumn;c++) {
                        markCellAsModified(row,c);
                    }
                }
            }
        });
    ...
    }
...
}
```

Le TableModelListener

- pour une modification effective dans le modèle, on va en simuler plusieurs pour provoquer des rafraîchissements avec une couleur évanescente

```
public void markCellAsModified(final int row,final int column) {
    if (row==-1 || column<=0) return;
    if (color[row][column]>=0) {
        /* The cell is already being updated, we have nothing to do */
        return;
    }
    color[row][column]=backgrounds.length-1;
    new Timer(40,new ActionListener() {

        @Override
        public void actionPerformed(ActionEvent e) {
            color[row][column]--;
            if (color[row][column]<0) {
                ((Timer)e.getSource()).stop();
                return;
            }
            fireTableCellUpdated(row,column);
        }
    }).start();
}
```

Le TableModelListener

- il suffit que l'adapter ait une méthode chargée de renvoyer la couleur:

```
public Color getBackground(int row,int column) {
    int n=color[row][column];
    if (n<0) {
        return backgrounds[0];
    }
    return backgrounds[n];
}
```

- et que le renderer de la table s'en serve:

```
t.setDefaultRenderer(Integer.class,new DefaultTableCellRenderer() {
    {   /* Initialization block, because we can't
         * put this into a constructor */
        setOpaque(true);
    }
    @Override public Component getTableCellRendererComponent(JTable table, Object value,
                            boolean isSelected, boolean hasFocus, int row, int column) {
        super.getTableCellRendererComponent(table,value,isSelected,hasFocus,row,column);
        setBackground(adapter.getBackground(row,column));
        return this;
    }
});
```

Le TableModelListener

- on ajoute un **`Timer`** pour modifier aléatoirement les données du modèle, et voilà le résultat:

Trading for dummies by J. K.

Share	Wall Street	Shangai	Paris	Tokyo	Quicampoix
Arcelor	934	394	216	489	707
EDF	392	847	388	500	958
Enron	927	634	92	47	42
Microsoft	965	161	75	621	156
Ford	482	761	633	778	827
Reuters	794	401	948	207	331
MacDonald	726	912	423	641	602
Carambar	526	173	861	205	709
BNP Paribas	368	588	240	474	104
Lloyd's	40	533	866	180	518
EMI	172	858	717	227	834
Google	892	586	728	49	834
Danone	503	928	715	877	769

Le TableColumnModelListener

- pour écouter les événements sur les données du **TableColumnModel**:

```
void addTableColumnModelListener(TableColumnModelListener l)
```

- voici les méthodes de ce listener:

```
void columnAdded(TableColumnModelEvent e)

void columnMarginChanged(ChangeEvent e)

void columnMoved(TableColumnModelEvent e)

void columnRemoved(TableColumnModelEvent e)

void columnSelectionChanged(ListSelectionEvent e)
```

Le TableColumnModelListener

- exemple: modification dynamique des colonnes

```
private static Component createCheckBox(final TableColumnModel columnModel,int columnIndex) {
    final TableColumn column=columnModel.getColumn(columnIndex);
    final JCheckBox c=new JCheckBox(column.getHeaderValue().toString(),true);
    c.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            if (c.isSelected()) {
                columnModel.addColumn(column);
            } else {
                columnModel.removeColumn(column);
            }
        }
    });
    return c;
}
```

Le TableColumnModelListener

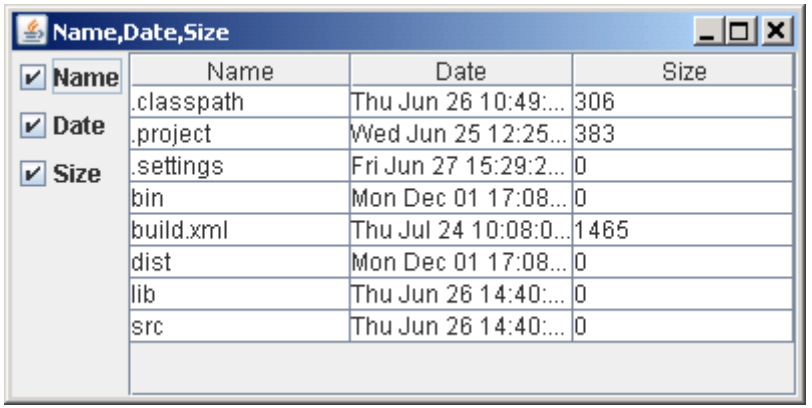
- on écoute les modifications pour mettre le titre de la fenêtre à jour

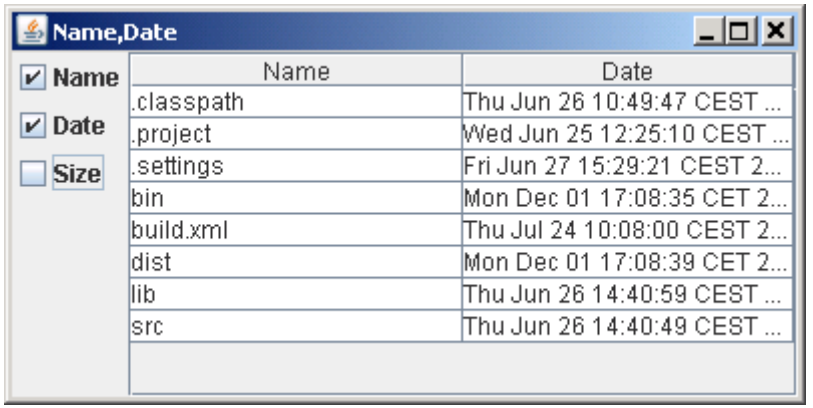
```
t.getColumnModel().addColumnModelListener(new TableColumnModelListener() {

    @Override public void columnAdded(TableColumnModelEvent e) {
        updateTitle(t.getColumnModel(),f);
    }
...

});
```

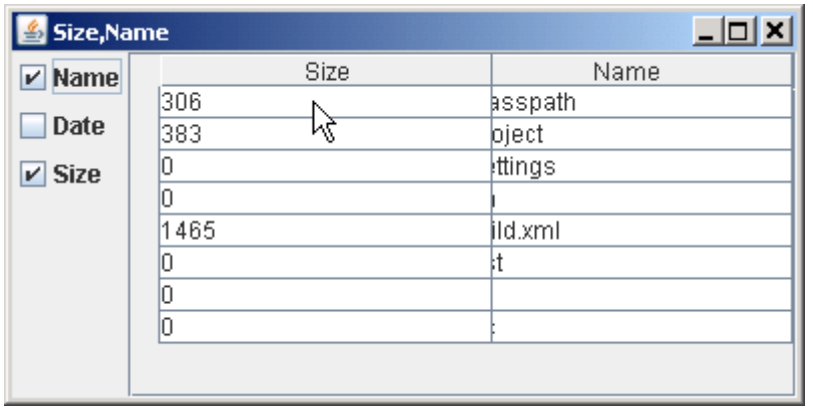
```
static void updateTitle(TableColumnModel columnModel, JFrame f) {
    String title="";
    int i,n=columnModel.getColumnCount();
    if (n!=0) {
        for (i=0;i<n-1;i++) {
            title=title+columnModel.getColumn(i).getHeaderValue().toString()+",";
        }
        title=title+columnModel.getColumn(i).getHeaderValue();
    }
    f.setTitle(title);
}
```

Le TableColumnModelListener

- résultat:





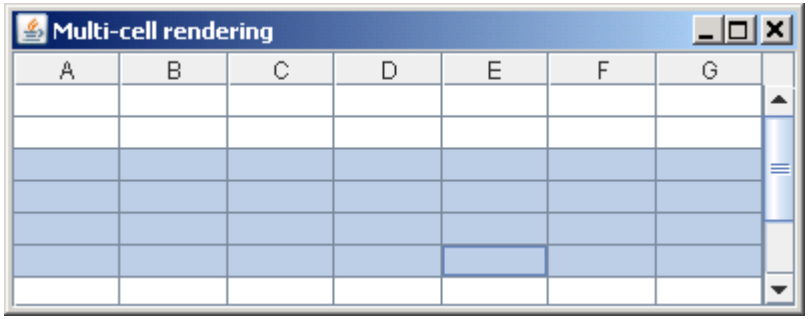


Multi-cell rendering

- on veut personnaliser le rendu d'une zone de sélection s'étendant sur plusieurs cellules
- principe: fabriquer un composant personnalisé à la taille totale de la zone
- pour chaque cellule, le renderer montrera la portion appropriée de ce composant

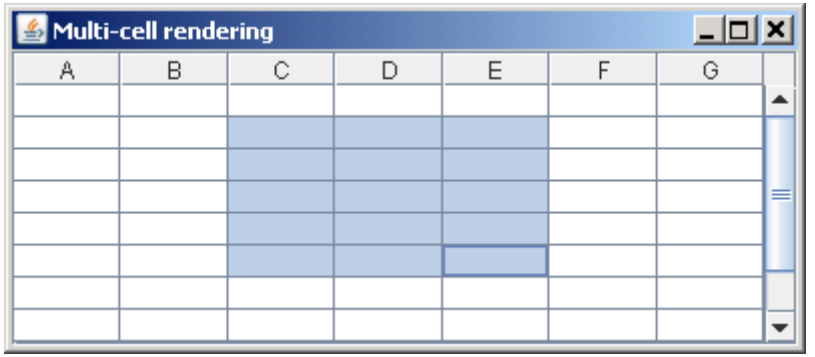
Multi-cell rendering

- par défaut, la sélection est en lignes:



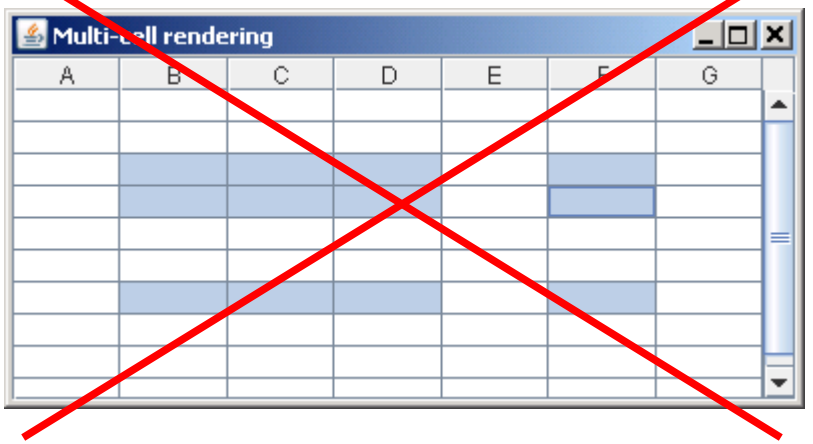
- on doit la passer en zone de cellules:

```
final JTable t=new JTable(10,7);
t.setCellSelectionEnabled(true);
```



Multi-cell rendering

- on veut n'autoriser qu'une seule zone de sélection:



- on impose l'intervalle simple pour les lignes et pour les colonnes:

```
t.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);
t.getColumnModel().getSelectionModel().setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);
```

Multi-cell rendering

- on va maintenant écouter les changements de sélection pour calculer la taille de notre composant de rendu:

```
ListSelectionListener listSelectionListener = new ListSelectionListener() {
    @Override
    public void valueChanged(ListSelectionEvent e) {
        updateSelectionComponent(t);
    }
};
t.getColumnModel().getSelectionModel().addListSelectionListener(listSelectionListener);
t.getSelectionModel().addListSelectionListener(listSelectionListener);
```

le même listener sert pour écouter les changements de sélection en ligne et en colonne

Multi-cell rendering

- on calcule les dimensions du composant ainsi que la position de son coin supérieur gauche:

```
static int width,height;
static int upperX,upperY;

static void updateSelectionComponent(JTable t) {
    int[] rows=t.getSelectedRows();
    if (rows.length==0) return;
    int[] columns=t.getSelectedColumns();
    if (columns.length==0) return;
    width=0;
    upperX=t.getCellRect(rows[0],columns[0],true).x;
    upperY=t.getCellRect(rows[0],columns[0],true).y;
    for (int i=0;i<columns.length;i++) {
        width=width+t.getCellRect(rows[0],columns[i],true).width;
    }
    height=0;
    for (int i=0;i<rows.length;i++) {
        height=height+t.getCellRect(rows[i],columns[0],true).height;
    }
    t.repaint();
}
```

Multi-cell rendering

- on définit le renderer en gardant le comportement par défaut pour les cases non sélectionnées
- pour les autres, on indique le décalage correspondant à la position de la cellule dans la vue

```
t.setDefaultRenderer(Object.class,new DefaultTableCellRenderer() {
    @Override public Component getTableCellRendererComponent(JTable table, Object value,
                    boolean isSelected, boolean hasFocus, int row, int column) {
        if (!isSelected) return super.getTableCellRendererComponent(table,value,
                                                    isSelected,hasFocus,row,column);
        Rectangle rect=table.getCellRect(row,column,true);
        shiftX=-rect.x;
        shiftY=-rect.y;
        return multiCellRenderer;
    }
});
```

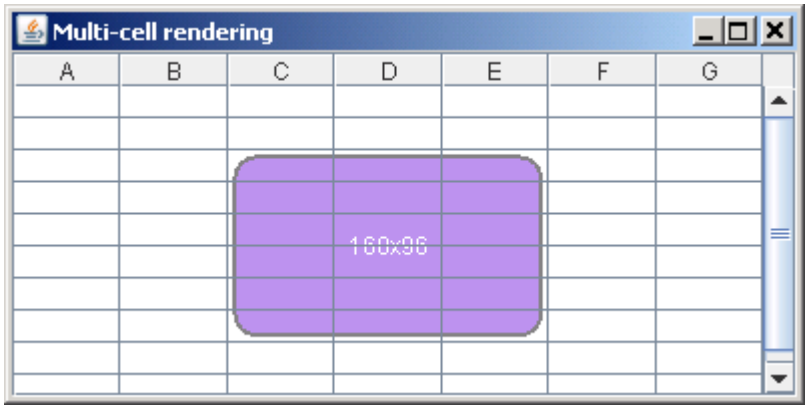
Multi-cell rendering

- il ne reste plus qu'à écrire le renderer:

```
final static JComponent multiCellRenderer=new JComponent() {
    {
        setVisible(true);
        setBorder(BorderFactory.createEmptyBorder(4,4,4,4));
    }
    BasicStroke stroke=new BasicStroke(4);
    Color purple=new Color(185,147,236);

    @Override protected void paintComponent(Graphics g) {
        Graphics2D g2=(Graphics2D)g.create();
        Insets insets=getInsets();
        try {
            g2.translate(insets.left+upperX+shiftX,insets.top+upperY+shiftY);
            int w=width-insets.left-insets.right;
            int h=height-insets.top-insets.bottom;
            int arc=(w<h)?w:h;
            arc=(arc>20)?20:5+arc/2;
            g2.setColor(Color.GRAY);
            g2.setStroke(stroke);
            g2.drawRoundRect(0,0,w-1,h-1,arc,arc);
            g2.setColor(purple);
            g2.fillRoundRect(0,0,w-1,h-1,arc,arc);
            g2.setColor(Color.WHITE);
            drawCenteredString(width+"x"+height,g2,w,h);
        } finally {
            g2.dispose();
        }
    }
...
}
```

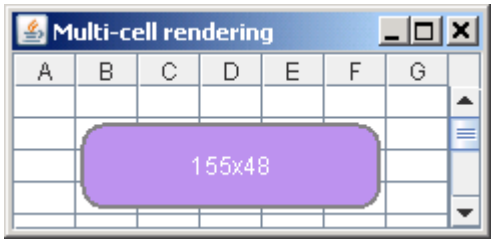
Multi-cell rendering

- problème de rendu:



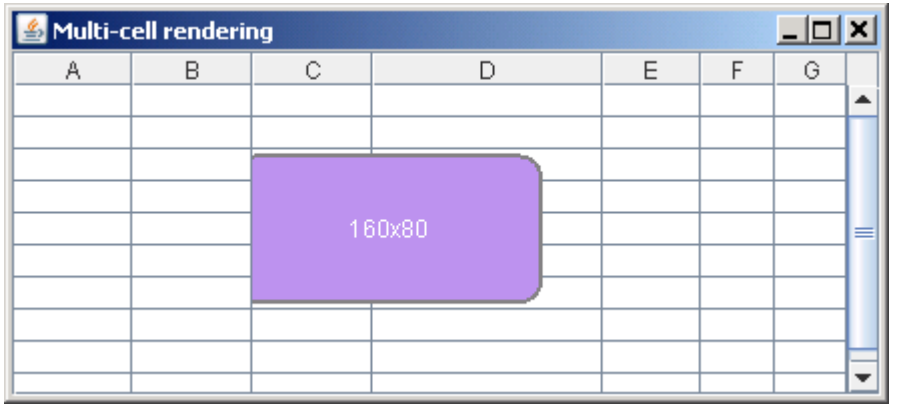
- il faut enlever tout espacement entre les cellules:

```
t.setIntercellSpacing(new Dimension(0,0));
```



Multi-cell rendering

- nouveau problème de rendu: pas de rafraîchissement quand les colonnes sont redimensionnées



- il faut écouter ces modifications pour réagir en conséquence

Multi-cell rendering

- on utilise un **PropertyChangeListener** qu'on place sur toutes les **TableColumn**:

```
PropertyChangeListener widthListener=new PropertyChangeListener() {
    @Override public void propertyChange(PropertyChangeEvent evt) {
        if ("width".equals(evt.getPropertyName())) {
            updateSelectionComponent(t);
        }
    }};
for (int i=0;i<t.getColumnCount();i++) {
    t.getColumnModel().getColumn(i).addPropertyChangeListener(widthListener);
}
```

- c'est gagné !
- si l'on est perfectionniste, on écoutera aussi les changements de hauteur des lignes (plus compliqué)

Trier les lignes

- on définit un gestionnaire de tri sur une **JTable** grâce à la méthode:

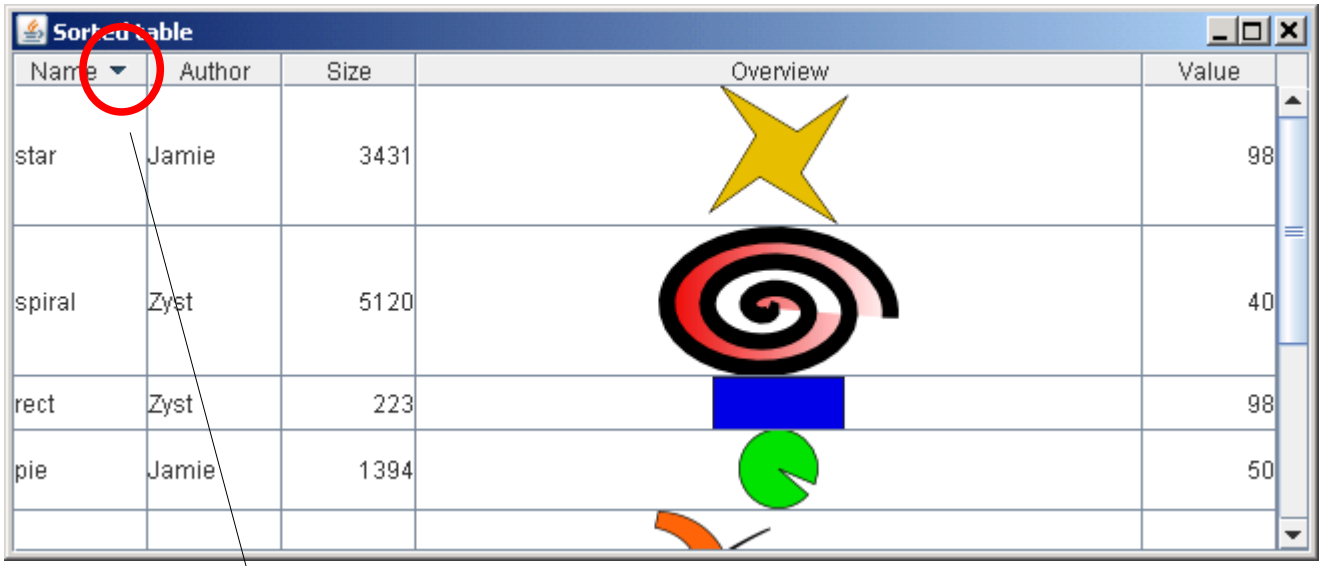
```
void setRowSorter(
  RowSorter<? extends TableModel> sorter)
```

- il suffit d'utiliser celui par défaut qui prend le modèle de la table en paramètre:

```
JTable t=new JTable(model);
TableRowSorter<TableModel> sorter=new TableRowSorter<TableModel>(t.getModel());
t.setRowSorter(sorter);
```

Trier les lignes

- cela suffit à rendre les lignes triables en cliquant sur les headers de colonnes:



on peut désactiver cette possibilité avec:

```
void setSortable(int column, boolean sortable)
```

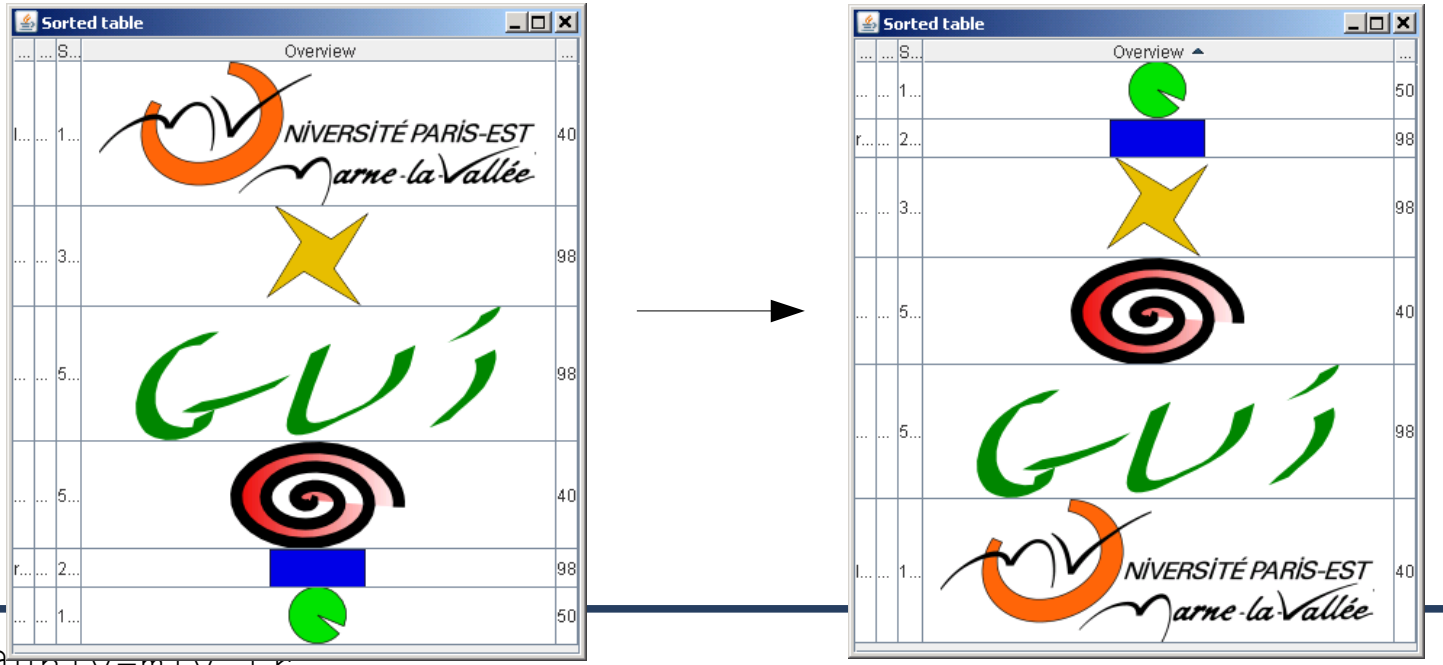
Trier les lignes

- pour trier les lignes d'une colonne, le sorter utilise les règles suivantes:
  - s'il y a un `Comparator` explicite, on l'utilise
  - sinon, si `getColumnClass()` renvoie `String`, on compare les chaînes
  - sinon, si `getColumnClass()` renvoie un objet `Comparable`, on utilise sa méthode `compareTo`
  - sinon, on compare les chaînes obtenues en faisant `toString()` sur les objets

Trier les lignes

- exemple: comparer des `ImageIcon` en fonction de leurs largeurs

```
/* 3 is the index of the column */
sorter.setComparator(3,new Comparator<ImageIcon>() {
    @Override public int compare(ImageIcon img1,ImageIcon img2) {
        return img1.getIconWidth()-img2.getIconWidth();
    }
});
```



Trier les lignes

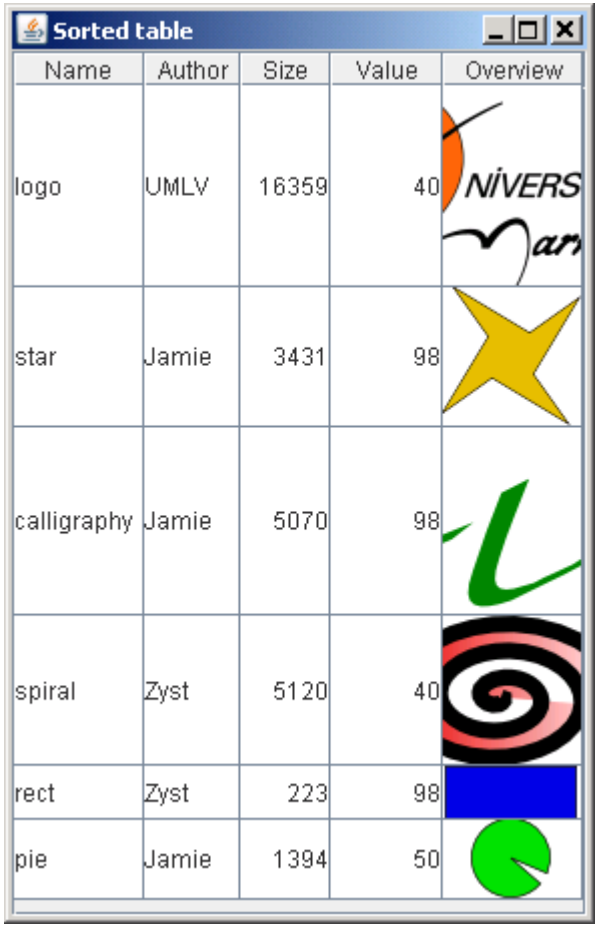
- par défaut, le tri se fait selon 3 colonnes:
  - en cas d'égalité pour une valeur dans une colonne, on regarde la colonne suivante pour trancher
- paramétrable avec:

```
void setMaxSortKeys(int max)
```

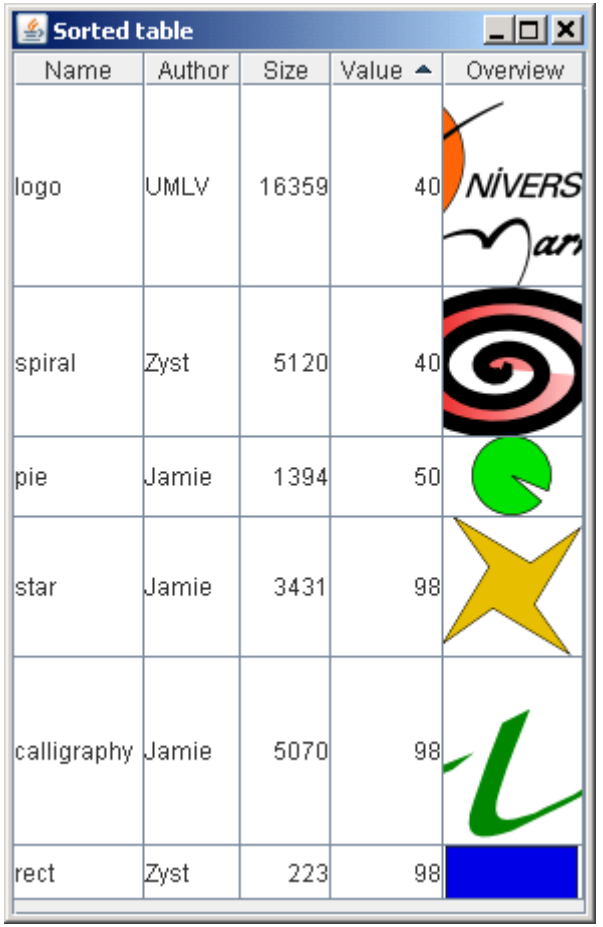
- ces clés de tri changent quand on clique sur les en-têtes de colonnes:
  - la nouvelle clé est la n°1; les anciennes clés sont décalées de 1; la dernière n'est plus prise en compte

Trier les lignes

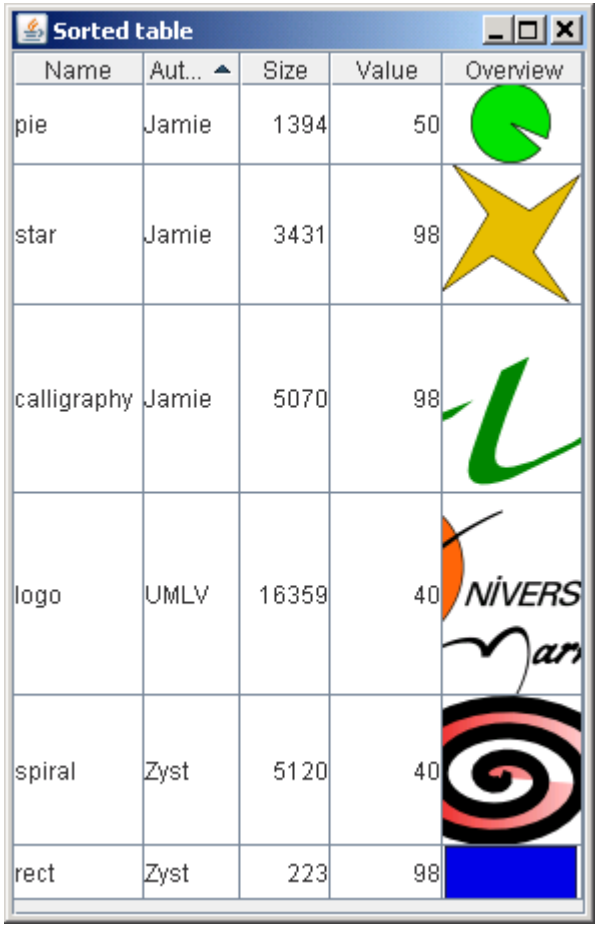
- exemple avec deux clés de tri:



pas de clé



valeur



auteur, valeur

Trier les lignes

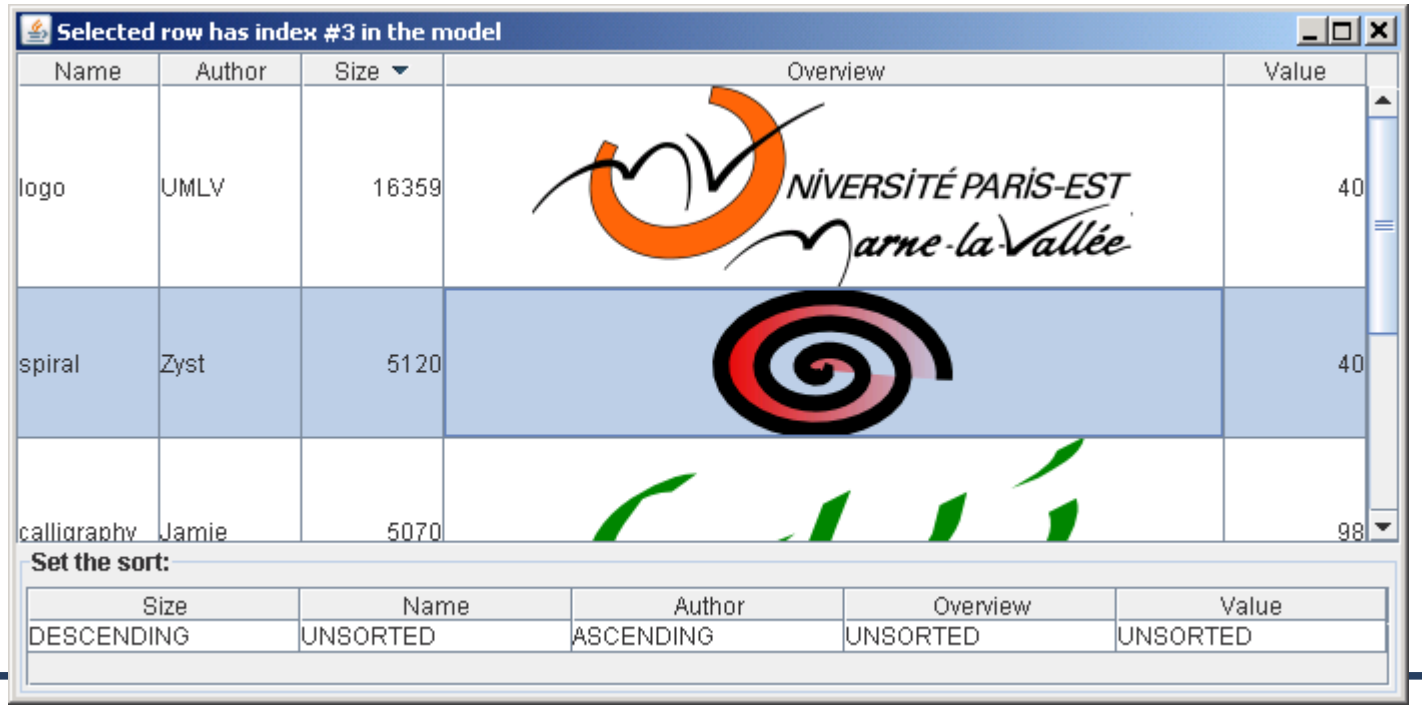
- on peut définir soi-même la liste des clés de tri avec:
  - `void setSortKeys(`
    `List<? extends RowSorter.SortKey> sortKeys`
- une **`SortKey`** est paramétrée par un indice de colonne (dans le modèle) et par un **`SortOrder`**:
  - **`SortOrder.UNSORTED`** (ordre dans le modèle)
  - **`SortOrder.ASCENDING`**
  - **`SortOrder.DESCENDING`**

Trier les lignes

- exemple d'application complexe: paramétrage du tri d'une table

Trier les lignes

- les indices des lignes sélectionnées sont relatifs à la vue
- `convertRowIndexToModel` permet d'obtenir les indices dans le modèle



Filtrer les lignes

- on peut indiquer un filtre avec une méthode de **`DefaultRowSorter<M,I>`**, dont hérite **`TableRowSorter<M>`**:

```
public void setRowFilter(
  RowFilter<? super M,? super I> filter)
```

  - **`M`** représente le **`TableModel`**
  - **`I`** représente le type des clés qui permettent d'identifier un élement:
    - pour une **`JTable`**, c'est **`Integer`**, car on désigne les lignes par des indices entiers
    - pour un **`JTree`**, c'est **`TreeNode`**

Filtrer les lignes

- `RowFilter.Entry<M,I>` permet d'accéder à une ligne de la table
  - `I getIdentifier()` : indice de la ligne
  - `Object getValue(int n)` : valeur de la $n^{\text{ème}}$ cellule de la ligne
  - `int getValueCount()` : nombre de cellules de la ligne
  - `M getModel()` : le modèle de la table

Filtrer les lignes

- il existe des filtres prédéfinis, que l'on peut obtenir avec des factory
- `regexFilter(String regex,int... indices)`
  - accepte les lignes dont au moins une valeur est matchée par l'expression régulière
- `numberFilter(RowFilter.ComparisonType type, Number number, int... indices)`
  - filtre par rapport à un nombre donné
- etc.

Filtrer les lignes

- le filtre est appliqué avant le tri éventuel
- `null`=pas de filtre
- on peut créer son propre filtre
- exemples:

```
final Object[] filters=new Object[] {
    null, /* no filter*/
    RowFilter.regexFilter("i",0), /* name contains "i" */
    RowFilter.numberFilter(RowFilter.ComparisonType.AFTER,2000,2), /* size>2000 */
    new RowFilter<TableModel, Integer>() { /* image is opaque */
        @Override
        public boolean include(RowFilter.Entry<? extends TableModel, ? extends Integer> entry) {
            int t=model.getTransparency(entry.getIdentifier());
            return t==Transparency.OPAQUE;
        }
    }
};
```

Filtrer les lignes

- exemples: