

JTables - Un autre regard

par Nicolas Zozol (<http://www.edupassion.com>)

Date de publication : 10/11/2007

Dernière mise à jour :

Ce tutoriel a pour but de comprendre le fonctionnement des JTables en utilisant un système d'explication distancié du pattern Modèle/Vue/Controlleur sans toutefois contester la validité de ce pattern. Il s'adresse aux débutants en Swing ou aux programmeurs plus expérimentés n'ayant utilisé principalement que les fonctions par défaut des JTables.

Introduction

I - Ce que l'utilisateur voit

I-A - L'utilisateur voit des cellules : les TableCellRenderer

I-B - L'utilisateur lit des données dans ces cellules

I-C - L'utilisateur comprend que les cellules sont rangées par colonne

I-D - La colonne comprend un entête

II - Ce que l'utilisateur fait

II-A - L'utilisateur va cliquer sur une case pour en modifier le contenu

II-B - L'utilisateur fait une modification

II-C - Le Model est modifié

III - Ce que JTable fait... et ne fait pas

III-A - Une couche de verre

III-B - Un exemple classique : Cliquer sur un bouton du Header, ce qui nous permettra de rajouter un Controle à notre Bulletin

III-C - Une conséquence : Utiliser autant que possible les objets et méthodes par défaut, afin de ne pas devoir reprogrammer les événements.

IV - Conclusion

IV-A - Récapitulatif

IV-B - Et quand j'aurai tout compris à ce tutoriel ?

Remerciements

Introduction

Les JTables sont des composants Java écrits par Sun Microsystems pour gérer les tableaux. Sun fournit des Objets par défaut permettant de les utiliser très simplement comme tableur. Mais il est également possible de mettre ce que l'on veut dans chacune des cases (même un autre JTable!) et d'aller plus loin qu'un tableur classique.

Vous trouverez plus bas du code tiré directement d'**Edupassion.com**, site web proposant entre autres des bulletins de note. Parfois le code est coloré en rouge, permettant de retrouver un même objet entre différents objets.

I - Ce que l'utilisateur voit

I-A - L'utilisateur voit des cellules : les TableCellRenderer

Si ce que l'on voit est un texte ou un nombre, on aura sans doute affaire à un **new DefaultTableCellRenderer()** qui hérite de JLabel. Chaque case peut être différente : c'est la fonction **TableCellRenderer.getTableRendererComponent(JTable table, Object value, boolean isSelected, boolean hasFocus, int row, int column)** qui définira le JComponent selon la ligne et la colonne. Si **getTableRendererComponent** renvoie un JButton, l'utilisateur pourra cliquer dessus. Si le JComponent est un JLabel, on peut lire une donnée.

Guillaume			
ve	ds2	Nouveau Controle	Moyen
Renée	12.0		
mandine	Absent		

JButton
Component JLabel

Voici le code pour obtenir le JComponent de type JButton dans lequel apparaît le nom de l'élève.

```

public class RendererEleve extends javax.swing.JButton implements
    javax.swing.table.TableCellRenderer {

    public RendererEleve() { /*Constructeur vide*/ }

    /** l'object value représente en principe l'élève */

    public java.awt.Component getTableCellRendererComponent(javax.swing.JTable table, Object value,
        boolean isSelected, boolean hasFocus, int row, int col)
    {
        String nom=((Eleve) value).toString();// Prenom + Nom de l'élève
        this.setText(nom);
        return this;
    }
} // Fin de la classe
    
```

I-B - L'utilisateur lit des données dans ces cellules

Ces données sont stockées dans le TableModel. La fonction **TableModel.getValueAt(int row, int column)** renvoie l'objet métier selon la colonne et la ligne.

```

public class TableModel {

    public Object getValueAt(int row, int column) {
    
```

```

        if (column==this.INDEX_COLONNE_LISTE_DES_ELEVES){ // Colonne des élèves
            Eleve eleve = (Eleve)listeDesEleves.get(row);
            return eleve;
        }

        if (column==this.INDEX_COLONNE_MOYENNE){ //Colonne des moyennes
            Eleve eleve = (Eleve)listeDesEleves.get(row);
            return trouveMoyenne(eleve);
        }
    }
}

```

L'objet renvoyé sera traité par la fonction `TableCellRenderer.getTableRendererComponent(JTable table, Object value, boolean isSelected, boolean hasFocus, int row, int column)` vue auparavant (I.A).

I-C - L'utilisateur comprend que les cellules sont rangées par colonne

La plupart du temps, les éléments d'une colonne ont la même signification : une date, un nom, un prénom, et ici les notes. Les fonctionnalités d'une colonne sont gérées par une **TableColumn**.

```

public class TableColumnEleve extends TableColumn{
    /**
     * Crée un style pour les TableColumn des Eleves, et spécifie le modelIndex à 0.
     */
    public TableColumnEleve() {
        super(0); // le modelIndex est à 0, ce qui indique que les élèves seront représentés à la
        première colone
        setHeaderRenderer(new HeaderEleve()); //Le component sera un HeaderEleve, qui hérite d'un
        JLabel
        setCellRenderer(new RendererEleve()); //Le Component est un RendererEleve qui hérite d'un
        JButton
    }
} // Fin de TableColumnEleve

```

Ces TableColumn sont ensuite réparties lors de la création du JTable

```

public class Bulletin extends JTable {

    BulletinModel model;

    public Bulletin(Cours cours) {
        /* Création du modèle */
        this.setAutoCreateColumnsFromModel(false);
        Cours
        BulletinModel model= new BulletinModel(cours, this); //Le modèle dépendra des notes dans le
        this.setModel(model);

        /* Création des TableColumn */
        createColumns();
    }

    private void createColumns() {
        /* Première TableColumn : celle contenant la liste des Eleves */
        TableColumnEleve tableColumnEleve = new TableColumnEleve();
        this.addColumn(tableColumnEleve);

        /* Ajout des controles : On a une nouvelle colonne par controle donné aux élèves */

        for (int modelIndex=model.INDEX_PREMIER_CONTROLE;modelIndex <=
        model.INDEX_DERNIER_CONTROLE;modelIndex++){

```

```
        Controle controle= (Controle)
listeDesControles.get(modelIndex-model.INDEX_PREMIER_CONTROLE);
        TableColumnControle columnControle=new TableColumnControle(modelIndex,
controle);//Contrairement à TableColumnEleve , l'index est dans le constructeur
        this.addColumn(columnControle);
    }
        //On continue avec les autres colonnes
    }
} //Fin de la classe Bulletin
```

I-D - La colonne comprend un entête

L'entête est de la classe **HeaderRenderer** : il s'agit encore d'une **TableCellRenderer** qui par défaut est grisâtre (selon le JRE utilisé). Si votre table est grande, vous la ferez scroller vers le bas, mais l'entête reste. C'est pourquoi l'entête n'est pas géré graphiquement par le container du JTable, mais par le scrollPane parent.

```
        this.scrollPane.getViewport().add(this.bulletin);//Permet au scrollpane de gérer le
header - bulletin derive de JTable
```

Chaque colonne contient un Header, qui est donc choisi dans le TableColumn

```
public class TableColumnEleve extends TableColumn{
    /**
     * Créé un style pour les TableColumn des Eleves, et spécifie le modelIndex à 0.
     */
    public TableColumnEleve() {
        super(0); // le modelIndex est à 0, ce qui indique la première colone
        setHeaderRenderer(new HeaderEleve());//Le component sera un HeaderEleve, qui gèrite d'un
JLabel disabled
        setCellRenderer(new Renderereleve());//Le Component est un Renderereleve qui hérite d'un
JButton
    }
} // Fin de TableColumnEleve
```

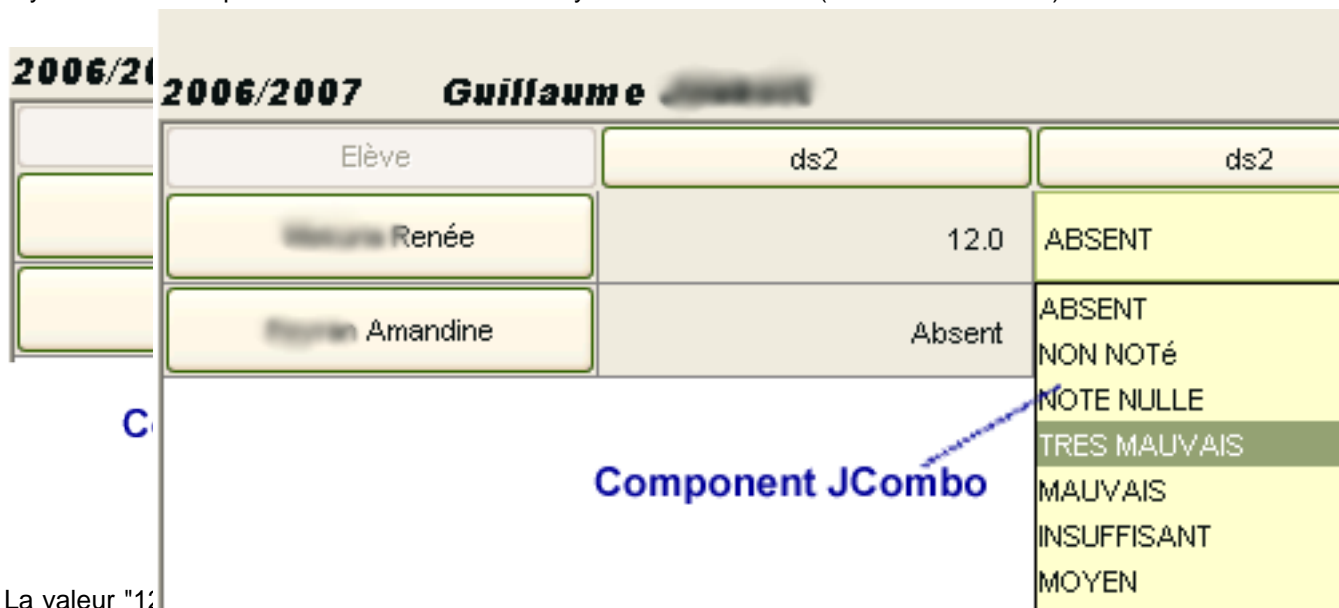
II - Ce que l'utilisateur fait

II-A - L'utilisateur va cliquer sur une case pour en modifier le contenu

Si la case est éditable, alors on rentre dans le mode d'édition de la cellule, géré par un objet de classe **TableCellEditor**. Il apparaît à l'écran un nouvel objet : le **CellEditorComponent**. Il s'agit d'un JComponent classique tel un JTextField ou un JComboBox.

Style de note Européenne :NoteEUEditor

Style de note Primaire (NotePrimaireEditor)



La valeur "12.0" reçoit l'objet **value** puis le programmeur écrit du code pour afficher cet objet à sa façon. On a choisit ici de présélectionner le texte, mais on pourrait par exemple afficher `value/2`. Le malheureux élève aurait sa note divisée par deux chaque fois qu'un professeur clique sur cette case.

On obtient le **CellEditorComponent** d'édition dans **TableCellEditor** : **public Component getTableCellEditorComponent(JTable table, Object value, boolean isSelected, int row, int column)**

```

public class NoteEUEditor extends AbstractCellEditor implements TableCellEditor{

    public NoteEU noteEU;
    JTextField fieldNoteEU=null; // C'est le JComponent que l'utilisateur va voir quand il entrera
    dans le mode Editor.
    JTable table;

    /** Constructeur pour NoteEUEditor
     * Cet éditeur permet de modifier une note sur /20 avec un seul clic.
     */
    public NoteEUEditor() {
        this.fieldNoteEU=new JTextField ();
    }

    /**

```

```

    * Cette fonction permet l'affichage sous forme de String de la note de l'élève
    * L'objet value est référencé dans le TableModel.getValueAt().
    */
    public Component getTableCellEditorComponent(JTable table, Object value, boolean isSelected,
    int row, int column)
    {
        NoteEU noteEU=(NoteEU) value; //L'objet Value est ce qui apparait AVANT que l'on
    modifie la valeur - ici c'est une note sur 20
        String stringNote= ((Float)noteEU.getValue()).toString();//StringNote est la chaine de
    caractère que l'on va afficher dans le JTextField

        /* Traitement conditionnel selon les spécificités métier */
        if (noteEU.getValue()<-2000) fieldNoteEU.setText("");//Cas pour un élève absent,
    représenté par la note -10002
        else labelNoteEU.setText(stringNote);

        /* Une fois que tous les calculs sont faits, on sélectionnera les chiffres de la note */

        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                fieldNoteEU.requestFocus();
                fieldNoteEU.selectAll();// utilisé pour que tout soit sélectionné, mais ça marche
    pas...
            }
        });

        return fieldNoteEU; //Le JTextField contient maintenant ce que l'on veut.
    }
}

```

II-B - L'utilisateur fait une modification

Si on valide le changement, on renvoie un objet grace à la fonction **public Object getCellEditorValue()** dans le même TableCellEditor.

```


public class NoteEUEditor extends AbstractCellEditor implements TableCellEditor{

    (... Voir plus haut ...)

    /**
    * Renvoie un objet une fois APRES MODIFICATION par l'utilisateur
    * L'objet renvoyé est très basique et sera ensuite traité par TableModel.setValueAt().
    */
    public Object getCellEditorValue()
    {
        String str=labelNoteEU.getText();
        if
    (str.equalsIgnoreCase("ABS")||str.equalsIgnoreCase("a")||str.equalsIgnoreCase("absent")) return
    Note.FLOAT_ABSENT; //On renvoie -10002
        if (str.equalsIgnoreCase("")) return Note.FLOAT_NON_NOTE;
        try{
            Float f=new Float(str);
            if (f.floatValue()<=Note.FLOAT_NA.floatValue()) return Note.FLOAT_NA;
            else return f;
        }
        catch (Exception e) {e.printStackTrace();return Note.FLOAT_NA;}//Si l'utilisateur a rentré
    n'importe quoi (ce qui arrive souvent ; )
    }
}

```

Arrive alors un processus invisible pour l'utilisateur grâce aux différents Events préprogrammés par nos amis de Sun Microsystems. L'objet renvoyé par **AbstractCellEditor** : **public Object getCellEditorValue()** arrive à **TableModel.setValueAt(Object obj, int row, int col)**, ce que nous détaillerons plus bas.

 *Il est vivement conseillé de passer par AbstractCellEditor afin de ne pas reprogrammer des méthodes comme fireEditingStopped(). En effet l'AbstractCellEditor fera souvent la démarche logique souhaitée.*

II-C - Le Model est modifié

Les observateurs auront notés que la fonction **getCellEditorValue()** ne fait pas référence à la position de la cellule dans le tableau. Cela a pour avantage de pouvoir réutiliser le TableCellEditor dans d'autre JTable (voire JTree ou JList avec une utilisation mineure de l'héritage). L'influence de la position dans le tableau dépend logiquement du traitement métier, et est traitée dans le **TableModel.setValueAt()**.

```
public void setValueAt(Object obj, int row, int col) {
    if (col==this.INDEX_COLONNE_APPRECIATION) this.setValueForAppreciation(obj, row);
    if (col==this.INDEX_COLONNE_MOYENNE) this.setValueForMoyenne(obj, row, col);
    else
        this.setValueForControle(obj, row, col);
}
```

L'Editor renvoie un Objet, et en fonction de celui-ci, le TableModel modifie ses données. Les systèmes d'événement programmés par Sun mettront à jour la partie visuelle en faisant appel à **TableModel.getValueAt()** avec les nouvelles valeurs.

III - Ce que JTable fait... et ne fait pas

III-A - Une couche de verre

Lorsque l'on clique sur un bouton de la JTable, on ne clique pas sur le bouton, mais sur la JTable, qui transmet ensuite l'événement au bouton... si on l'a programmé. On dit parfois qu'il y a une glace de verre par dessus la Table : il est possible de voir ce qu'il y a en dessous, mais en appuyant à un endroit, on appuie sur toute la table.

III-B - Un exemple classique : Cliquer sur un bouton du Header, ce qui nous permettra de rajouter un Controle à notre Bulletin

On va créer un MouseListener qui va écouter ce que notre souris clique.

```
public class EvtAjoutEtModifDeControle implements MouseListener{

    Bulletin bulletin;
    JTableHeader header;

    /**
     * Creates a new instance of EvtAjoutEtModifDeControle
     */
    public EvtAjoutEtModifDeControle(Bulletin bulletin) {
        this.bulletin=bulletin;
    }

    public void mouseClicked(MouseEvent e) {

        /* Ligne clé ! c'est ici qu'on sait où on clique */
        int indexDeColonneSelected =
bulletin.convertColumnIndexToModel(bulletin.columnAtPoint(e.getPoint()));

        /* code une fois que l'on a su où l'on cliquait */
        if (indexDeColonneSelected==bulletin.getBulletinModel().INDEX_COLONNE_AJOUT_DEVOIR)
            MaFactory.creerControle(bulletin);
    }

    /* Les autres méthodes non prises en charge */
    public void mousePressed(MouseEvent e) { }
    public void mouseReleased(MouseEvent e) { }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
}
```

Et dans le constructeur de la JTable, on rajoute la ligne :

```
this.getTableHeader().addMouseListener(new EvtAjoutEtModifDeControle(this));
```

Pour continuer avec notre allégorie, il faut ordonner à chaque JComponent de "regarder" ce qui se passe au dessus de la glace posée sur la Table.

III-C - Une conséquence : Utiliser autant que possible les objets et méthodes par défaut, afin de ne pas devoir reprogrammer les événements.

J'ai donné tantôt un exemple de JComboBox fourni lorsque l'on clique sur une cellule. On voudrait qu'il se passe ceci : User choisit dans le combo un **objet_choisit** -> public void monJComboActionPerformed() -> public **Object** monEditor.getCellEditorValue() -> monModel.setValueAt (**objet_choisit** , row, col)

Malheureusement, une fois que l'on choisit l'élément de notre JComboBox, la fonction programmée JComboBoxActionPerformed() ne s'exécute pas : l'événement n'est pas transmis, et il faudrait reprogrammer les enchaînement des événements. C'est faisable, mais pénible, c'est pourquoi Sun Microsystem nous donne un DefaultCellEditor (JComboBox combo) - il est possible d'avoir un DefaultCellEditor basé sur d'autres JComponents.

Le DefaultCellEditor s'occupe des événements, et il vous reste à faire le code métier. Dans notre cas, la dernière ligne des bulletins est occupée par les moyennes du Controle et il n'y a donc pas de choix possible : voyons la nouvelle implémentation pour l'éditeur de notes type Ecole Primaire.

```
public class NotePrimaireEditor extends DefaultCellEditor{

    /** Creates a new instance of NoteUSEditor */
    public NotePrimaireEditor(JComboBox combo) {
        super (combo);
    }

    public Component getTableCellEditorComponent(JTable table, Object value, boolean isSelected, int
row, int column) {

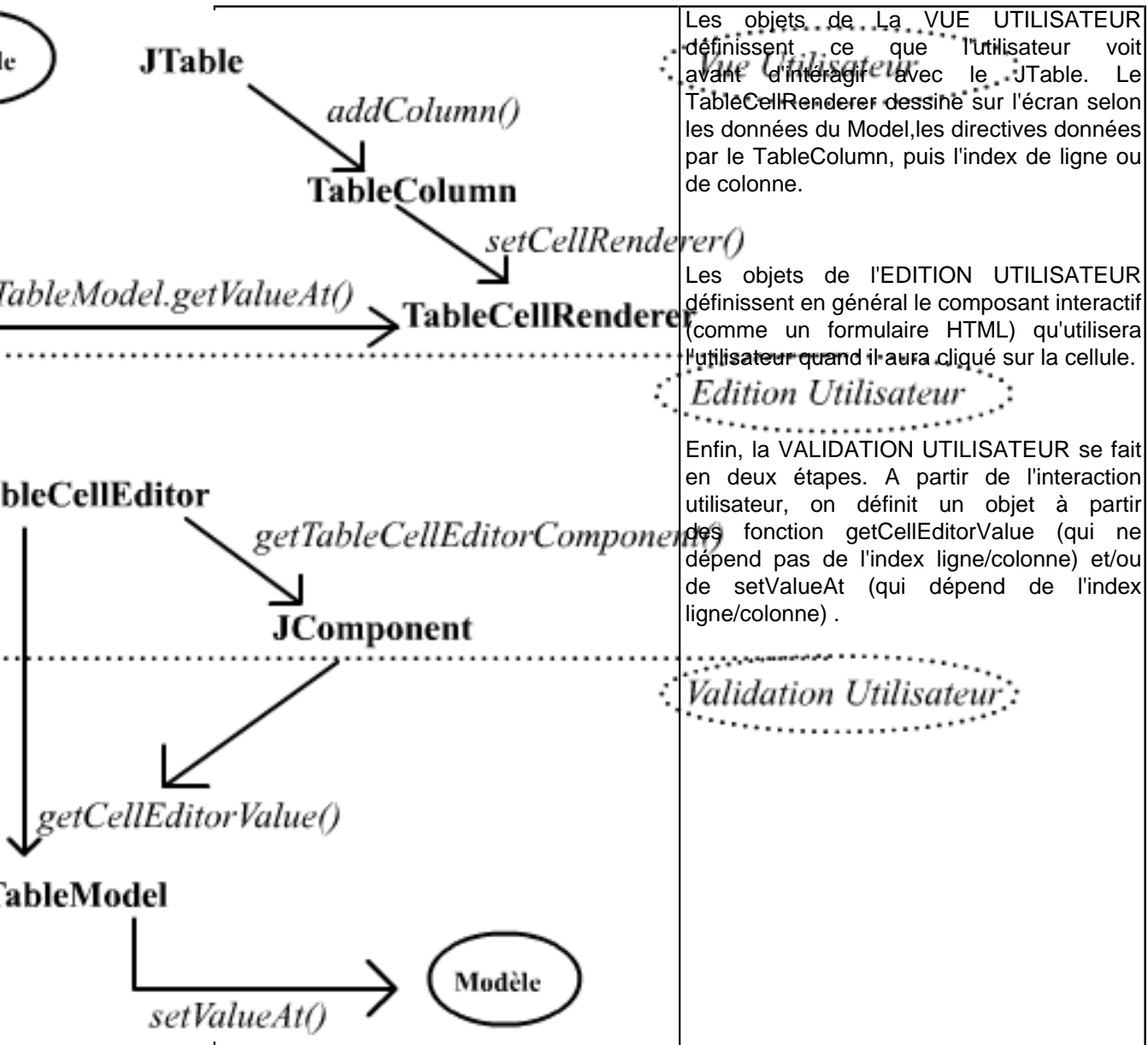
        /* Si c'est la dernière ligne, on affiche la moyenne */
        if (row == ((BulletinModel)table.getModel()).INDEX_DERNIERE_LIGNE){
            if (value!=null)
                return new JLabel(value.toString());
            else return new JLabel();
        }
        else //On peut afficher le JComboBox
            return super.getTableCellEditorComponent(table, value, isSelected, row, column);

    }

    /** La fonction est à implémenter si vous voulez renvoyer un autre objet que celui présent dans
le JComboBox (par exemple à partir d'une Factory).
    * Sinon, c'est le DefaultCellEditor qui renvoie l'objet contenu dans le JComboBox. C'est le
traitement par défaut car c'est le plus logique.
    *public Object getCellEditorValue() {}
    */
}
```

IV - Conclusion


IV-A - Récapitulatif



IV-B - Et quand j'aurai tout compris à ce tutoriel ?

Voici de quoi faire pas mal de chose avec les JTables. Cependant, les JTables étant des composants Swing, les bonnes pratiques de Swing s'appliquent aussi, notamment une bonne gestion des Threads (cf. SwingUtilities en section II-A).

- Si votre programme a des accès longs vers la base de données, vous devrez utiliser d'autres fonctions des JTables, comme `EditCellAt()`.
- Si votre programme est complexe (volontairement ou non), vous devrez peut-être redéfinir les Events, ou réécrire les fonctions `fireXXX()` des objets fournis par Sun.

 *Mon avis est que l'utilisation des JTables est déjà assez compliquée, et si vous souhaitez que votre code soit maintenable par une autre personne, évitez au maximum de réécrire ce que Sun Microsystem a déjà fait. Concrètement, cela consiste à écrire des classes qui héritent des DefaultXXX ou AbstractXXX plutôt que d'implémenter à nouveau les Interfaces.*

Remerciements

Merci à **Ricky81** pour l'aide technique et **LineLe** pour la correction

