Création interface graphique avec Swing : les tableaux (JTable)

Par Baptiste Wicht

Date de publication : 20 décembre 2009

Cet article va vous présenter la création de tableaux avec Swing en Java via la classe JTable. On y verra les concepts élémentaires des JTable ainsi que comment faire certaines opérations telles que le tri/filtrage ou encore l'ajout dynamique de données dans un tableau.

Version anglophone de cet article - English version of this article: ***Creation of Swing User Interface : Tables (JTable)***

1 - Introduction

Cet article est la suite de **Création interface graphique avec Swing : les bases** qui vous a présenté les bases du développement d'interfaces graphiques avec Swing. Je vous conseille de lire le précédent article indiqué ci-dessus avant de lire celui-ci qui en est le prolongement.

Cet article va vous présenter la création de tableaux en Swing. Ce composant pose souvent des problèmes lorsqu'on débute avec Swing. Je vais donc tenter d'expliquer les différents concepts liés à l'utilisation de tableaux en Swing. Dans cet article, nous verrons donc les concepts de base du composant JTable, la définition de modèle de tableaux, la modification dynamique du contenu du tableau, la façon de modifier l'affichage des différentes cellules du tableau, la modification directe du contenu du tableau et finalement le tri et le filtrage du tableau.

2 - Concepts de base

Un JTable est donc un composant Swing permettant d'afficher un tableau formé d'un certain nombre de lignes et d'un certain nombre de colonnes. En plus des lignes de contenu, le JTable a également une ligne d'en-tête présentant un titre pour chaque colonne.

Un JTable a donc d'un côté des données et de l'autre des données d'en-tête. On peut voir les données comme un tableau à deux dimensions dans lequel chaque valeur correspond à la valeur d'une cellule du tableau. Quant aux en-têtes, on peut les voir comme un tableau de chaînes de caractères.

Le JTable utilise différents concepts de Swing :

- Un modèle pour stocker les données. Un JTable utilise une classe implémentant **TableModel**. Nous verrons plus loin comment spécifier un modèle pour un JTable.
- Un *renderer* pour le rendu des cellules. On peut spécifier un **TableCellRenderer** pour chaque classe de données. Nous verrons plus loin ce que ça signifie exactement.
- Un éditeur pour l'édition du contenu d'une cellule. On peut spécifier un **TableCellEditor** pour chaque classe de données. Encore une fois, nous approfondirons ce concept plus loin.

Tout au long de cet article, on va développer un petit programme très simple permettant de gérer une liste d'amis. Voici les caractéristiques d'un ami :

- Un nom et un prénom (classe String).
- Une couleur préférée (classe Color).
- Un sexe (booléen homme/femme).
- Un sport qu'on pratique avec lui (énumération Sport).

Voici notre énumération Sport :

```
public enum Sport {
    TENNIS,
    FOOTBALL,
    NATATION,
    RIEN;
}
```

On va donc commencer par une première version, des plus basiques, de notre application.

La façon la plus simple, mais pas la meilleure, est de passer directement à JTable un tableau à deux dimensions pour les données et un tableau à une dimension pour l'en-tête de chaque colonne.

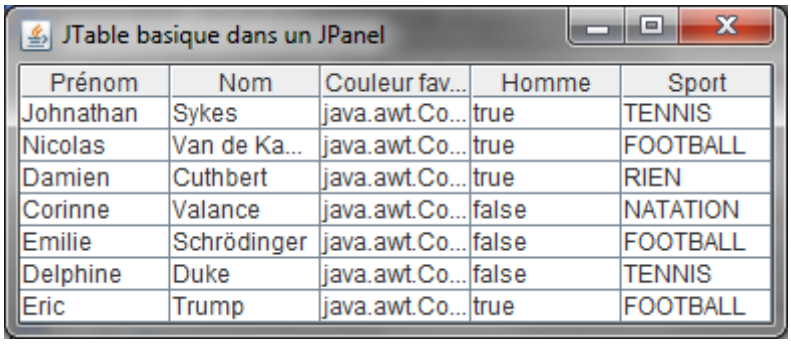
Voici donc l'implémentation la plus basique qui soit de notre programme :

```
public class JTableBasiqueAvecPanel extends JFrame {
    public JTableBasiqueAvecPanel() {
        super();

        setTitle("JTable basique dans un JPanel");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        Object[][] donnees = {
                {"Johnathan", "Sykes", Color.red, true, Sport.TENNIS},
                {"Nicolas", "Van de Kampf", Color.black, true, Sport.FOOTBALL},
                {"Damien", "Cuthbert", Color.cyan, true, Sport.RIEN},
                {"Corinne", "Valance", Color.blue, false, Sport.NATATION},
                {"Emilie", "Schrödinger", Color.magenta, false, Sport.FOOTBALL},
                {"Delphine", "Duke", Color.yellow, false, Sport.TENNIS},
                {"Eric", "Trump", Color.pink, true, Sport.FOOTBALL},
        };
```

```
        String[] entetes = {"Prénom", "Nom", "Couleur favorite", "Homme", "Sport"};

        JTable tableau = new JTable(donnees, entetes);

        getContentPane().add(tableau.getTableHeader(), BorderLayout.NORTH);
        getContentPane().add(tableau, BorderLayout.CENTER);

        pack();
    }

    public static void main(String[] args) {
        new JTableBasiqueAvecPanel().setVisible(true);
    }
```

On utilise donc le constructeur **JTable(Object[][] data, Object[] entetes)** pour gérer nos données et nos en-têtes. Pour ajouter le JTable dans un JPanel, il faut ajouter séparément le header du tableau et le tableau en lui-même.

Cela nous donnera le résultat suivant :



Prénom	Nom	Couleur fav...	Homme	Sport
Johnathan	Sykes	java.awt.Co...	true	TENNIS
Nicolas	Van de Ka...	java.awt.Co...	true	FOOTBALL
Damien	Cuthbert	java.awt.Co...	true	RIEN
Corinne	Valance	java.awt.Co...	false	NATATION
Emilie	Schrödinger	java.awt.Co...	false	FOOTBALL
Delphine	Duke	java.awt.Co...	false	TENNIS
Eric	Trump	java.awt.Co...	true	FOOTBALL

Version basique avec JTable dans un JPanel et données statiques

Avec très peu de code, nous avons donc un tableau fonctionnel. Néanmoins, cette première implémentation souffre de plusieurs défauts :

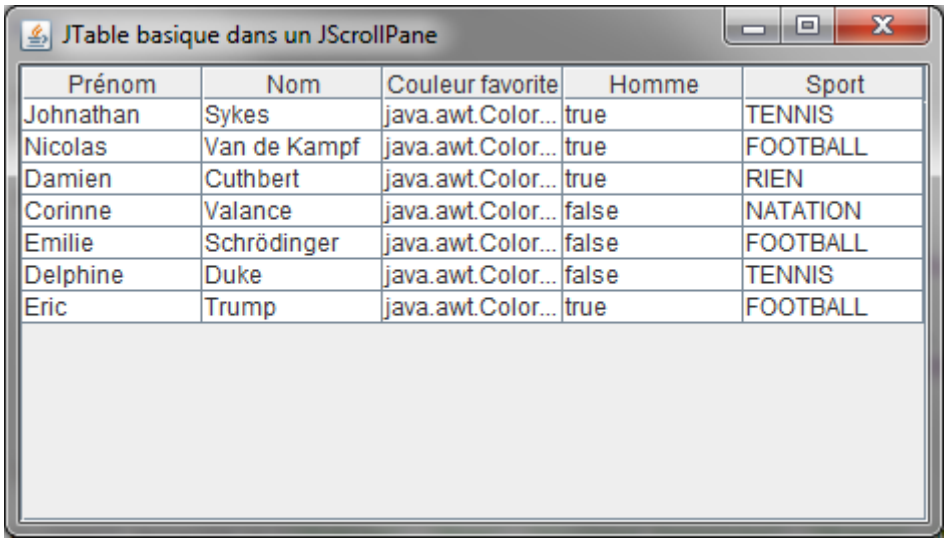
1. On ne peut pas afficher plus de lignes qu'il n'y a d'espace disponible sur la fenêtre.
2. Les données sont statiques et immuables.
3. On ne peut pas gérer la façon dont seront affichées les données.
4. Aucune distinction entre les données et la vue.
5. Les colonnes couleurs et Homme ne sont pas très esthétiques.

Le point 1 est très vite corrigible. En fait, la bonne façon d'ajouter un JTable dans un JPanel est de passer par un JScrollPane qui permettra d'afficher plus de lignes que la fenêtre ne le permet. Pour la suite de cet article, on va donc toujours utiliser un JScrollPane. Voici donc une deuxième version avec JScrollPane :

```
public class JTableBasiqueAvecScrollPane extends JFrame {
    public JTableBasiqueAvecScrollPane() {
        super();

        setTitle("JTable basique dans un JScrollPane");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        Object[][] donnees = {
                {"Johnathan", "Sykes", Color.red, true, Sport.TENNIS},
                {"Nicolas", "Van de Kampf", Color.black, true, Sport.FOOTBALL},
                {"Damien", "Cuthbert", Color.cyan, true, Sport.RIEN},
                {"Corinne", "Valance", Color.blue, false, Sport.NATATION},
                {"Emilie", "Schrödinger", Color.magenta, false, Sport.FOOTBALL},
                {"Delphine", "Duke", Color.yellow, false, Sport.TENNIS},
                {"Eric", "Trump", Color.pink, true, Sport.FOOTBALL},
        };

        String[] entetes = {"Prénom", "Nom", "Couleur favorite", "Homme", "Sport"};
```

```
        JTable tableau = new JTable(donnees, entetes);

        getContentPane().add(new JScrollPane(tableau), BorderLayout.CENTER);

        pack();
    }

    public static void main(String[] args) {
        new JTableBasiqueAvecScrollPane().setVisible(true);
    }
}
```

Cette fois, on ajoute directement le JTable dans le JScrollPane. Voici le résultat :



JTable basique avec JScrollPane et données statiques

Cette version est déjà meilleure que la précédente, mais n'est de loin pas encore parfaite, nous allons encore l'améliorer dans les chapitres suivants.

3 - Le modèle du tableau

Une chose indispensable à faire lorsqu'on utilise des JTable est d'utiliser un modèle de tableau pour stocker les données. Il faut donc créer une classe étendant *TableModel*. En pratique, on implémente rarement directement *TableModel*, mais on hérite plutôt d'*AbstractTableModel* et l'on redéfinit les méthodes nécessaires. Pour commencer, voici les méthodes qu'il faudra redéfinir pour notre modèle de données statique :

- int getRowCount() : doit retourner le nombre de lignes du tableau
- int getColumnCount() : doit retourner le nombre de colonnes du tableau
- Object getValueAt(int rowIndex, int columnIndex) : doit retourner la valeur du tableau à la colonne et la ligne spécifiées
- String getColumnName(int columnIndex) : doit retourner l'en-tête de la colonne spécifiée

On va donc créer notre premier modèle en reprenant les données dans un tableau à deux dimensions pour commencer :

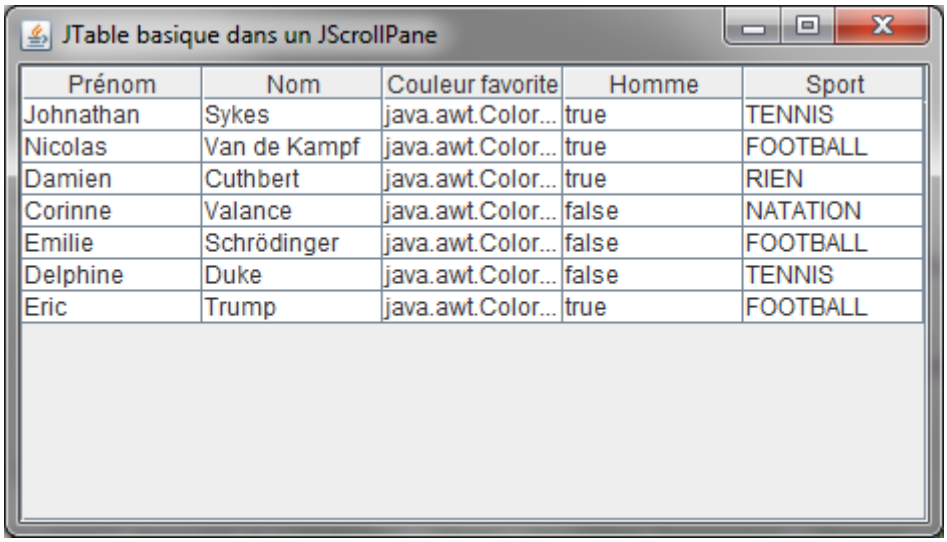
```
public class ModeleStatique extends AbstractTableModel {
    private final Object[][] donnees;

    private final String[] entetes = {"Prénom", "Nom", "Couleur favorite", "Homme", "Sport"};

    public ModeleStatique() {
        super();

        donnees = new Object[][]{
                {"Johnathan", "Sykes", Color.red, true, Sport.TENNIS},
                {"Nicolas", "Van de Kampf", Color.black, true, Sport.FOOTBALL},
                {"Damien", "Cuthbert", Color.cyan, true, Sport.RIEN},
                {"Corinne", "Valance", Color.blue, false, Sport.NATATION},
                {"Emilie", "Schrödinger", Color.magenta, false, Sport.FOOTBALL},
                {"Delphine", "Duke", Color.yellow, false, Sport.TENNIS},
                {"Eric", "Trump", Color.pink, true, Sport.FOOTBALL},
        };
    }

    public int getRowCount() {
        return donnees.length;
    }

    public int getColumnCount() {
        return entetes.length;
    }

    public String getColumnName(int columnIndex) {
        return entetes[columnIndex];
    }

    public Object getValueAt(int rowIndex, int columnIndex) {
        return donnees[rowIndex][columnIndex];
    }
}
```

Et on modifie cette fois notre JTable pour utiliser ce modèle :

```
public class JTableBasiqueAvecModeleStatique extends JFrame {
    public JTableBasiqueAvecModeleStatique() {
        super();

        setTitle("JTable avec modèle statique");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JTable tableau = new JTable(new ModeleStatique());

        getContentPane().add(new JScrollPane(tableau), BorderLayout.CENTER);
```

```
        pack();
    }

    public static void main(String[] args) {
        new JTableBasiqueAvecModeleStatique().setVisible(true);
    }
}
```

On a donc créé une classe héritant de **AbstractTableModel** et redéfinissant les méthodes indispensables. Les données sont toujours stockées de la même manière, mais cette solution est plus souple et beaucoup plus propre. Si l'on regarde la JFrame, on peut voir qu'il n'y a plus aucune donnée dans cette classe, ce qui est donc beaucoup plus propre en termes de découplage. En plus de cela, on est maintenant maîtres de nos données et de la façon dont elles sont stockées. Mais au niveau de l'affichage, rien n'a changé :



JTable basique avec modèle statique

On a maintenant une bonne base, mais on va encore l'améliorer. En règle générale, il est extrêmement rare de voir des données sous la forme de tableaux à deux dimensions. En général, Java étant un langage orienté objet, on manipule des objets. On va donc créer un objet Ami qui va représenter un de nos amis :

```
public class Ami {
    private String nom;
    private String prenom;
    private Color couleur;
    private boolean homme;
    private Sport sport;

    public Ami(String nom, String prenom, Color couleur, boolean homme, Sport sport) {
        super();

        this.nom = nom;
        this.prenom = prenom;
        this.couleur = couleur;
        this.homme = homme;
        this.sport = sport;
    }

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public String getPrenom() {
        return prenom;
```

```
    }

    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }

    public Color getCouleur() {
        return couleur;
    }

    public void setCouleur(Color couleur) {
        this.couleur = couleur;
    }

    public boolean isHomme() {
        return homme;
    }

    public void setHomme(boolean homme) {
        this.homme = homme;
    }

    public Sport getSport() {
        return sport;
    }

    public void setSport(Sport sport) {
        this.sport = sport;
    }
}
```

Une simple classe de données toute bête. Et on va prendre en compte ceci dans notre modèle :

```
public class ModeleStatiqueObjet extends AbstractTableModel {
    private final Ami[] amis;

    private final String[] entetes = {"Prénom", "Nom", "Couleur favorite", "Homme", "Sport"};

    public ModeleStatiqueObjet() {
        super();

        amis = new Ami[]{
                new Ami("Johnathan", "Sykes", Color.red, true, Sport.TENNIS),
                new Ami("Nicolas", "Van de Kampf", Color.black, true, Sport.FOOTBALL),
                new Ami("Damien", "Cuthbert", Color.cyan, true, Sport.RIEN),
                new Ami("Corinne", "Valance", Color.blue, false, Sport.NATATION),
                new Ami("Emilie", "Schrödinger", Color.magenta, false, Sport.FOOTBALL),
                new Ami("Delphine", "Duke", Color.yellow, false, Sport.TENNIS),
                new Ami("Eric", "Trump", Color.pink, true, Sport.FOOTBALL)
        };
    }

    public int getRowCount() {
        return amis.length;
    }

    public int getColumnCount() {
        return entetes.length;
    }

    public String getColumnName(int columnIndex) {
        return entetes[columnIndex];
    }

    public Object getValueAt(int rowIndex, int columnIndex) {
        switch(columnIndex){
            case 0:
                return amis[rowIndex].getPrenom();
            case 1:
                return amis[rowIndex].getNom();
```

```
            case 2:
                return amis[rowIndex].getCouleur();
            case 3:
                return amis[rowIndex].isHomme();
            case 4:
                return amis[rowIndex].getSport();
            default:
                return null; //Ne devrait jamais arriver
        }
    }
}
```

Cette fois, le code commence à devenir intéressant. C'est là qu'on commence à comprendre l'utilité d'utiliser un modèle et non pas directement le constructeur de JTable. Si maintenant par exemple, on veut inverser deux colonnes et mettre "prénom" après "nom", il suffit d'inverser les deux colonnes dans la liste des colonnes et d'inverser les deux return de la méthode **getValueAt()** alors que ceci aurait été beaucoup plus difficile et contraignant avec un tableau d'objets.

Pour ce qui est de l'affichage, il suffit d'utiliser le nouveau modèle au lieu de l'ancien :

```
public class JTableBasiqueAvecModeleStatiqueObjet extends JFrame {
    public JTableBasiqueAvecModeleStatiqueObjet() {
        super();

        setTitle("JTable avec modèle statique et des objets");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JTable tableau = new JTable(new ModeleStatiqueObjet());

        getContentPane().add(new JScrollPane(tableau), BorderLayout.CENTER);

        pack();
    }

    public static void main(String[] args) {
        new JTableBasiqueAvecModeleStatiqueObjet().setVisible(true);
    }
}
```

Rien ne change au niveau du rendu. Dans le chapitre suivant, on va rendre dynamique notre modèle en permettant l'ajout et le retrait d'ami.

4 - Ajouter/Supprimer des lignes

On va maintenant passer à quelque chose qui va rendre notre opération un peu plus intéressante et surtout rendre notre modèle indispensable. C'est-à-dire donner à l'utilisateur la possibilité de modifier le contenu du tableau. Pour le moment, on va se contenter d'ajouter et de supprimer des lignes. On verra au chapitre 6 comment modifier des valeurs dans le tableau.

La première chose à faire est donc de rendre notre modèle dynamique. Pour cela, on va donc ajouter des méthodes addAmi et removeAmi. Pour avertir le JTable qu'il y a eu des modifications sur le modèle, il faut appeler les méthodes fireXXX qui sont définies dans AbstractTableModel. En plus de cela, il faut bien évidemment utiliser une structure de données qui soit dynamique. Le tableau n'est pas du tout adapté. On va donc utiliser cette fois une ArrayList. Voici donc ce que pourrait donner notre modèle dynamique :

```
public class ModeleDynamiqueObjet extends AbstractTableModel {
    private final List<Ami> amis = new ArrayList<Ami>();

    private final String[] entetes = {"Prénom", "Nom", "Couleur favorite", "Homme", "Sport"};

    public ModeleDynamiqueObjet() {
        super();

        amis.add(new Ami("Johnathan", "Sykes", Color.red, true, Sport.TENNIS));
        amis.add(new Ami("Nicolas", "Van de Kampf", Color.black, true, Sport.FOOTBALL));
        amis.add(new Ami("Damien", "Cuthbert", Color.cyan, true, Sport.RIEN));
        amis.add(new Ami("Corinne", "Valance", Color.blue, false, Sport.NATATION));
        amis.add(new Ami("Emilie", "Schrödinger", Color.magenta, false, Sport.FOOTBALL));
        amis.add(new Ami("Delphine", "Duke", Color.yellow, false, Sport.TENNIS));
        amis.add(new Ami("Eric", "Trump", Color.pink, true, Sport.FOOTBALL));
    }

    public int getRowCount() {
        return amis.size();
    }

    public int getColumnCount() {
        return entetes.length;
    }

    public String getColumnName(int columnIndex) {
        return entetes[columnIndex];
    }

    public Object getValueAt(int rowIndex, int columnIndex) {
        switch(columnIndex){
            case 0:
                return amis.get(rowIndex).getPrenom();
            case 1:
                return amis.get(rowIndex).getNom();
            case 2:
                return amis.get(rowIndex).getCouleur();
            case 3:
                return amis.get(rowIndex).isHomme();
            case 4:
                return amis.get(rowIndex).getSport();
            default:
                return null; //Ne devrait jamais arriver
        }
    }

    public void addAmi(Ami ami) {
        amis.add(ami);

        fireTableRowsInserted(amis.size() -1, amis.size() -1);
    }

    public void removeAmi(int rowIndex) {
        amis.remove(rowIndex);
```

```
            fireTableRowsDeleted(rowIndex, rowIndex);
        }
    }
```

Rien de bien compliqué donc. Pour la méthode add(), on ajoute le nouvel Ami dans la liste ensuite de quoi on prévient la JTable qu'un nouvel élément a été inséré. Pour la méthode remove() le principe est le même, on commence par supprimer l'élément de la liste et enfin on prévient le tableau qu'un élément a été supprimé. On va ajouter deux actions dans notre interface graphique. La première va ajouter un ami (pour simplifier, cela va toujours rajouter le même objet, alors qu'en réalité, il faudrait proposer à l'utilisateur de configurer le nouvel ami) et la seconde va supprimer le ou les éléments sélectionnés. Voici ce que ça va nous donner :

```
public class JTableBasiqueAvecModeleDynamiqueObjet extends JFrame {
    private ModeleDynamiqueObjet modele = new ModeleDynamiqueObjet();
    private JTable tableau;

    public JTableBasiqueAvecModeleDynamiqueObjet() {
        super();

        setTitle("JTable avec modèle dynamique");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        tableau = new JTable(modele);

        getContentPane().add(new JScrollPane(tableau), BorderLayout.CENTER);

        JPanel boutons = new JPanel();

        boutons.add(new JButton(new AddAction()));
        boutons.add(new JButton(new RemoveAction()));

        getContentPane().add(boutons, BorderLayout.SOUTH);

        pack();
    }

    public static void main(String[] args) {
        new JTableBasiqueAvecModeleDynamiqueObjet().setVisible(true);
    }

    private class AddAction extends AbstractAction {
        private AddAction() {
            super("Ajouter");
        }

        public void actionPerformed(ActionEvent e) {
            modele.addAmi(new Ami("Megan", "Sami", Color.green, false, Sport.NATATION));
        }
    }

    private class RemoveAction extends AbstractAction {
        private RemoveAction() {
            super("Supprimmer");
        }

        public void actionPerformed(ActionEvent e) {
            int[] selection = tableau.getSelectedRows();

            for(int i = selection.length - 1; i >= 0; i--){
                modele.removeAmi(selection[i]);
            }
        }
    }
}
```

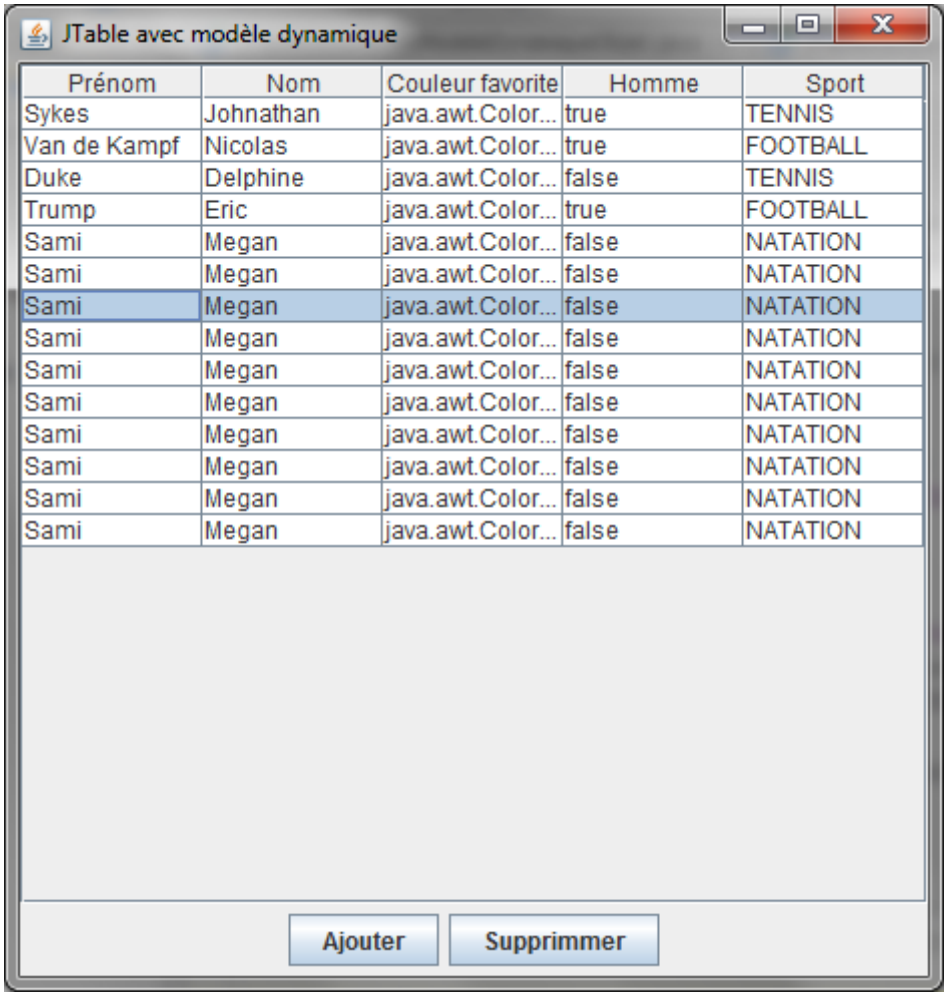
L'action pour ajouter un ami ne fait rien de bien spécial et est plutôt triviale. Par contre, il y a quelques petites choses à dire sur l'action de suppression. Tout d'abord, il faut savoir qu'une JTable peut fonctionner selon plusieurs modes de

sélection qui sont configurables via la méthode setSelectionMode. Le paramètre mode est une des valeurs suivantes venant de ListSelectionModel :

- SINGLE_SELECTION : permet de sélectionner une seule ligne.
- SINGLE_INTERVAL_SELECTION : permet de sélectionner un intervalle de ligne.
- MULTIPLE_INTERVAL_SELECTION : permet de sélectionner de multiples intervalles. C'est la valeur par défaut.

Il faut donc comprendre que le tableau de lignes renvoyé par la méthode getSelectedRows() peut retourner plusieurs intervalles. Les résultats sont retournés dans l'ordre ascendant. Il nous faut donc les supprimer depuis la fin, sinon on fausserait les résultats.

Cela va nous donner l'affichage suivant :



JTable avec un modèle dynamique

Comme vous pouvez le constater, on vient de construire un tableau tout à fait dynamique sans grands problèmes. Dans le prochain chapitre, on va maintenant résoudre le problème des colonnes Couleur et Homme qui ne sont pas très pratiques en l'état.

5 - L'affichage des cellules

On va maintenant passer à la personnalisation de l'affichage des différentes cellules. Voici ce qu'on va effectuer comme changements :

- Afficher la couleur au lieu du toString() de Color.
- Afficher l'image du sexe au lieu d'un booléen.
- Afficher le nom d'un ami en gras.

Pour cela, il va falloir commencer par spécifier dans le modèle à quelle classe correspond chacune colonne. On ne peut configurer des *renderers* que par colonne. A la suite de quoi, on configurera les *renderers* pour chaque classe de colonne au niveau de la JTable. Voici donc la première chose à faire. Il suffit de redéfinir la méthode getColumnClass() dans notre modèle :

```
@Override
public Class getColumnClass(int columnIndex){
 switch(columnIndex){
  case 2:
   return Color.class;
  case 3:
   return Boolean.class;
  default:
   return Object.class;
 }
}
```

À noter que ceci n'est pas vraiment indispensable car cela est automatiquement fait par **AbstractTableModel**. Mais je trouve personnellement cela plus clair ainsi.

On va maintenant créer nos *renderers*. Un *renderer* est simplement une classe implémentant **TableCellRenderer** qui est une interface ne contenant qu'une seule méthode retournant un composant Swing. En pratique, on hérite généralement de **DefaultCellRenderer** qui représente un JLabel comme renderer.

Il faut éviter dans la mesure du possible de renvoyer un nouvel objet dans un *renderer* si on a beaucoup d'éléments dans notre JTable. Cela voudrait dire qu'il faudrait créer un objet pour chaque ligne et à chaque fois qu'on redessine la JTable, ce qui peut dégrader très fortement les performances. C'est pourquoi on garde un même objet qu'on modifie pour chaque cellule.

On va donc créer notre premier *renderer* qui va simplement modifier le *background* du JLabel avec la couleur favorite de l'élément courant :

```
public class ColorCellRenderer extends DefaultTableCellRenderer {
    @Override
    public Component getTableCellRendererComponent(JTable table, Object value, boolean
 isSelected, boolean hasFocus, int row, int column) {
        super.getTableCellRendererComponent(table, value, isSelected, hasFocus, row, column);

        Color color = (Color) value;

        setText("");
        setBackground(color);

        return this;
    }
}
```

C'est donc extrêmement simple d'implémenter ce *renderer*. Il suffit de récupérer la couleur de l'ami et de la mettre en *background* de notre JLabel. On peut donc passer au suivant. Cette fois, on va afficher une image pour le sexe de la personne (il faut aussi penser à modifier l'en-tête de la colonne pour mettre Sexe au lieu d'homme).

```
public class SexeCellRenderer extends DefaultTableCellRenderer {
    private Icon manImage;
    private Icon womanImage;

    public SexeCellRenderer() {
        super();

        manImage = new ImageIcon("man.png");
        womanImage = new ImageIcon("woman.png");
    }

    @Override
    public Component getTableCellRendererComponent(JTable table, Object value, boolean
 isSelected, boolean hasFocus, int row, int column) {
        super.getTableCellRendererComponent(table, value, isSelected, hasFocus, row, column);

        Boolean homme = (Boolean)value;

        setText("");

        if(homme){
            setIcon(manImage);
        } else {
            setIcon(womanImage);
        }

        return this;
    }
}
```

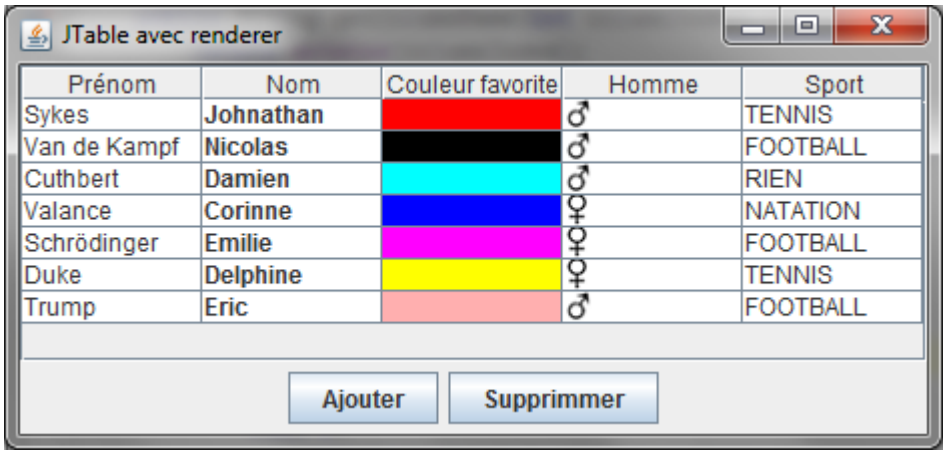
On commence donc par charger les images dans le constructeur ensuite de quoi dans la méthode de rendu, en fonction du sexe de la personne, on affiche la bonne image. On peut donc passer au dernier *renderer* :

```
public class BoldCellRenderer extends DefaultTableCellRenderer {
    @Override
    public Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected,
 boolean hasFocus, int row, int column) {
        super.getTableCellRendererComponent(table, value, isSelected, hasFocus, row, column);

        setFont(getFont().deriveFont(Font.BOLD));

        return this;
    }
}
```

Rien à dire de ce côté, c'est plus que trivial. On va maintenant configurer le tout dans notre tableau :

```
tableau.setDefaultRenderer(Boolean.class, new SexeCellRenderer());
tableau.setDefaultRenderer(Color.class, new ColorCellRenderer());
tableau.getColumnModel().getColumn(1).setCellRenderer(new BoldCellRenderer());
```

Pour les deux premiers, on peut directement les lier à une classe de colonne, mais pour le *renderer* qui met en gras le texte, on ne peut pas le lier à String puisqu'on veut seulement le mettre sur une colonne et non les deux. Ceci va nous donner l'affichage suivant :



JTable avec des renderers personnalisés

Cette fois on a déjà quelque chose de beaucoup plus intéressant visuellement. Vous pouvez faire des *renderers* beaucoup plus évolués avec d'autres composants qu'un JLabel comme des JPanel ou même pourquoi pas une JTable.

Dans le prochain chapitre, on va permettre la modification directe des valeurs de la JTable.

6 - Permettre de modifier les cellules

On va maintenant rendre notre tableau éditable pour que l'utilisateur puisse modifier un ami. Pour cela, il va falloir commencer par rendre notre modèle éditable. Pour cela, il faut implémenter la méthode **isCellEditable(int row, int column)** qui va indiquer quelles sont les cellules éditables. Dans notre tableau, toutes les cellules seront éditables. Ensuite, il faut prendre en compte les modifications. Pour cela, il faut implémenter la méthode **setValueAt(Object value, int column, int row)** qui est automatiquement appelée lorsque l'utilisateur valide sa modification. En plus de cela, on va également modifier notre méthode **getColumnClass** pour ajouter la classe Sport pour la colonne 4 puisqu'on voudra également le modifier. Voici ce que vont donner ces méthodes dans notre modèle éditable :

```
@Override
public boolean isCellEditable(int rowIndex, int columnIndex) {
    return true; //Toutes les cellules éditables
}

@Override
public void setValueAt(Object aValue, int rowIndex, int columnIndex) {
    if(aValue != null){
        Ami ami = amis.get(rowIndex);

        switch(columnIndex){
            case 0:
                ami.setPrenom((String)aValue);
                break;
            case 1:
                ami.setNom((String)aValue);
                break;
            case 2:
                ami.setCouleur((Color)aValue);
                break;
            case 3:
                ami.setHomme((Boolean)aValue);
                break;
            case 4:
                ami.setSport((Sport)aValue);
                break;
        }
    }
}
@Override
public Class getColumnClass(int columnIndex){
    switch(columnIndex){
        case 2:
            return Color.class;
        case 3:
            return Boolean.class;
        case 4 :
            return Sport.class;
        default:
            return Object.class;
    }
}
```

La première méthode retourne true car toutes les méthodes sont éditables. La deuxième récupère l'Ami modifié et en fonction de la colonne modifie la bonne propriété de l'Ami.

Maintenant notre modèle est modifiable, mais cela ne va pas fonctionner tout seul car JTable ne sait pas comment éditer des couleurs ou des "Sport" par défaut. Il va donc falloir utiliser un nouveau concept, celui des TableCellEditor. Un editor est en fait simplement un objet permettant de gérer l'édition d'une cellule, un peu sur le même principe que le renderer. Par défaut, JTable gère déjà l'édition de tous les champs configurés en tant que Object sous forme d'un JTextField ainsi que les Boolean sous forme de cases à cocher. Donc, dans notre cas, cela marchera directement pour le nom et le prénom de l'ami, mais ne marchera pas pour le reste. On ne peut pas éditer une couleur ou une énumération sous forme de texte. Pour ce qui est du sexe, ça va fonctionner, mais pas complètement à cause de notre renderer qui n'est pas cohérent avec l'editor. Il nous faudra donc créer 3 editors.

On va commencer par le plus simple, pour l'énumération. On va utiliser une simple liste déroulante. Dans ce cas, c'est plus que simple, car il existe la classe DefaultCellEditor qui a un constructeur prenant un JComboBox, on va donc en profiter :

```
public class SportCellEditor extends DefaultCellEditor {
    public SportCellEditor() {
        super(new JComboBox(Sport.values()));
    }
}
```

On peut donc voir que créer un éditeur pour un type énuméré est extrêmement simple.

On va maintenant créer un éditeur pour notre couleur. On pourrait utiliser un champ texte avec la valeur hexadécimale de la couleur ou encore trois champs texte avec chacun une composante RGB, mais ce ne serait pas très pratique alors qu'on a un composant de choix de couleur dans Swing, le JColorChooser. Par contre, on ne peut pas l'utiliser comme editor. Il faut en fait utiliser un bouton comme editor qui va ouvrir le JColorChooser. Cela va nous montrer comment faire un editor évolué et nous montrer les différents concepts inhérents à ces editors. Voici donc un editor permettant d'utiliser le JColorChooser de Swing :

```
public class ColorCellEditor extends AbstractCellEditor implements TableCellEditor, ActionListener {
    private Color couleur;
    private JButton bouton;
    private JColorChooser colorChooser;
    private JDialog dialog;

    public ColorCellEditor() {
        super();

        bouton = new JButton();
        bouton.setActionCommand("change");
        bouton.addActionListener(this);
        bouton.setBorderPainted(false);

        colorChooser = new JColorChooser();
        dialog = JColorChooser.createDialog(bouton, "Pick a Color", true, colorChooser, this, null);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if ("change".equals(e.getActionCommand())) {
            bouton.setBackground(couleur);
            colorChooser.setColor(couleur);
            dialog.setVisible(true);

            fireEditingStopped();
        } else {
            couleur = colorChooser.getColor();
        }
    }

    @Override
    public Object getCellEditorValue() {
        return couleur;
    }

    @Override
    public Component getTableCellEditorComponent(JTable table, Object value, boolean isSelected, int
 row, int column) {
        couleur = (Color)value;

        return bouton;
    }
}
```

Cette fois, on peut tout de suite voir que ça se complique un peu. Le TableCellEditor en lui-même est un JButton. La méthode **getTableCellEditorComponent** retourne donc le composant faisant l'édition. La méthode

getCellEditorValue retourne la valeur entrée dans l'editor, dans notre, cas, il s'agit de la couleur modifiée (ou laissée telle quelle) du JColorChooser. On appelle la méthode fireEditingStopped() pour avertir le JTable qu'on a terminé l'édition et qu'il faut afficher à nouveau le renderer.

On va s'occuper du dernier de nos editors, celui pour le sexe. Encore une fois, on aurait plusieurs solutions. On pourrait afficher une liste déroulante avec les deux choix, des boutons radios, une case à cocher ou même un JTextField pourquoi pas. Dans notre cas, on va faire très simple, un simple bouton qui change la valeur à chaque clic :

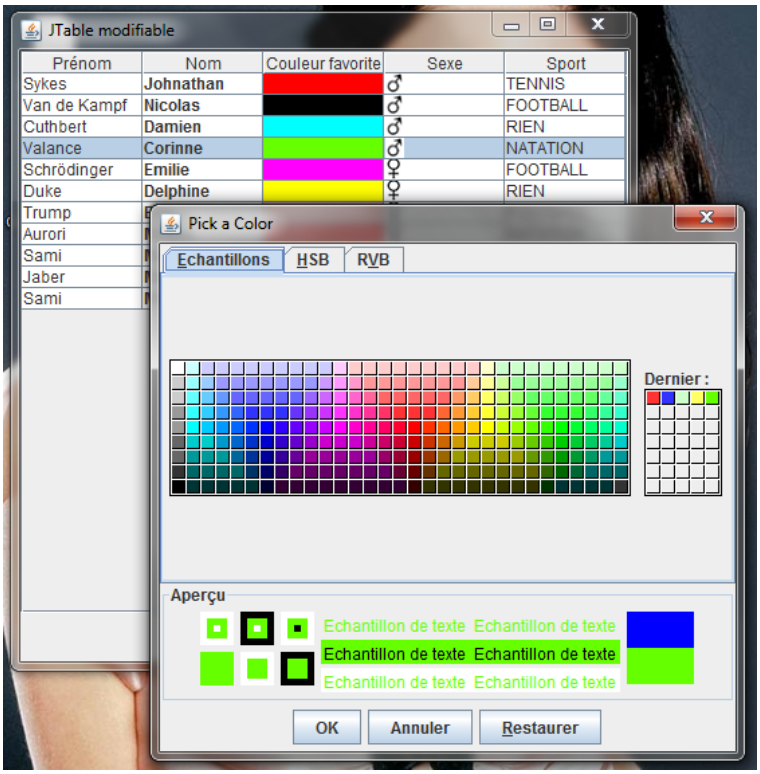
```
public class SexeCellEditor extends AbstractCellEditor implements TableCellEditor, ActionListener {
    private boolean sexe;
    private JButton bouton;

    public SexeCellEditor() {
        super();

        bouton = new JButton();
        bouton.addActionListener(this);
        bouton.setBorderPainted(false);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        sexe ^= true;

        fireEditingStopped();
    }

    @Override
    public Object getCellEditorValue() {
        return sexe;
    }

    @Override
    public Component getTableCellEditorComponent(JTable table, Object value, boolean isSelected, int
 row, int column) {
        sexe = (Boolean)value;

        return bouton;
    }
}
```

On garde le même principe que pour le choix des couleurs sauf que cette fois c'est encore plus simple, il suffit de faire une inversion du booléen et de le retourner.

Enfin, on configure notre JTable pour qu'elle prenne en compte nos renderers :

```
tableau.setDefaultEditor(Sport.class, new SportCellEditor());
tableau.setDefaultEditor(Color.class, new ColorCellEditor());
tableau.setDefaultEditor(Boolean.class, new SexeCellEditor());
```

Le principe est le même que pour les renderers, on a un editor par classe de colonne. Voilà ce que donne l'édition d'une couleur :



Edition des cellules du JTable

Voilà, on a maintenant un tableau complètement éditable et fonctionnel. Dans les prochains chapitres, on va fignoler le tout en permettant de trier des colonnes et de filtrer le contenu du tableau.

7 - Trier le contenu

On va maintenant rendre notre tableau triable par colonne. Cela permet à un utiliser de trier le contenu du tableau en fonction de la colonne en appuyant sur le titre d'une colonne. Cela est fait au moyen d'un objet RowSorter. Le JTable possède une méthode permettant d'activer un sorter par défaut :

```
tableau.setAutoCreateRowSorter(true);
```

Cela va trier les colonnes de classe String de façon alphabétique en fonction de la Locale courante (langue et pays), les colonnes d'une classe implémentant l'interface Comparable en fonction de leur comparaison et les autres colonnes de façon alphabétique sur leur valeur toString().

Dans la plupart des cas, c'est suffisant, mais on peut tout de même personnaliser le sorter. On peut évidemment créer notre propre RowSorter, mais il est plus pratique d'utiliser la classe TableRowSorter et de la personnaliser pour effectuer des changements plutôt que de redéfinir une nouvelle implémentation ce qui peut s'avérer assez lourd. Voici une façon de faire permettant de personnaliser le sorter :

```
TableRowSorter<TableModel> sorter = new TableRowSorter<TableModel>(tableau.getModel());
tableau.setRowSorter(sorter);
```

Une première chose qu'on peut faire est de spécifier une colonne comme non triable via la méthode setSortable :

```
sorter.setSortable(2, false);
```

Ce code spécifie que la colonne "Couleur" n'est pas triable. Ensuite, on peut également indiquer au sorter s'il faut retrier le tableau après une mise à jour des données dans le tableau :

```
sorter.setSortsOnUpdates(true);
```

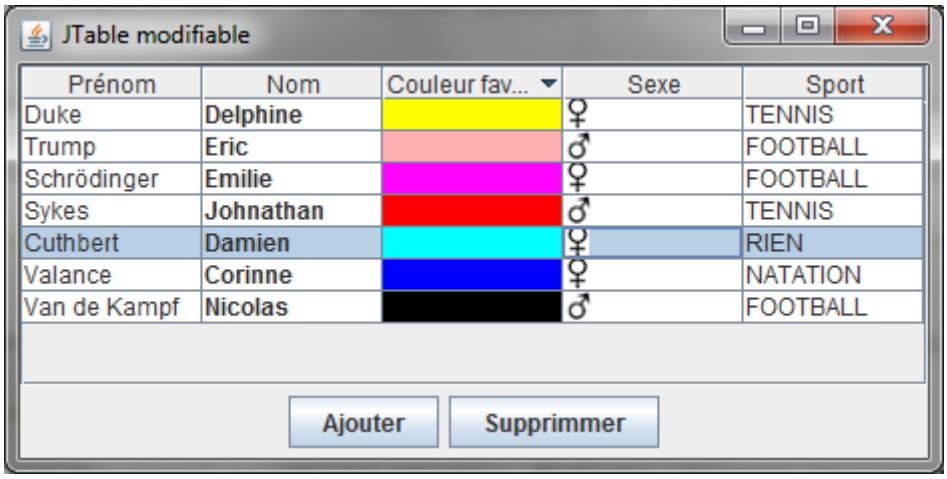
Ce code indique au sorter qu'il faut retrier le tableau après chaque modification des données. Une autre chose intéressante est qu'on peut spécifier notre propre comparateur pour une colonne. Dans notre cas, c'est ce qu'il faut pour la colonne Couleur, car Color n'implémentant pas Comparable et est donc trié via sa valeur de toString() ce qui n'est pas très intéressant. On va donc trier cette colonne en fonction du niveau de bleu dans la couleur. On commence donc par créer un Comparator de Color :

```
public class ColorComparator implements Comparator<Color> {
    @Override
    public int compare(Color c1, Color c2) {
        return new Integer(c1.getBlue()).compareTo(c2.getBlue());
    }
}
```

Ensuite de quoi, on spécifie que la colonne 2 doit utiliser ce nouveau comparator :

```
sorter.setComparator(2, new ColorComparator());
```

Voilà ce que ça donnerait avec un tri sur la colonne Couleur :



JTable triée sur la colonne Couleur

Il est donc très facile de trier un tableau.

Maintenant, lorsqu'une JTable est triable, cela va poser un problème pour la suppression de lignes. Vous pouvez essayer avec le code courant si vous triez puis tentez de supprimer des lignes, vous allez voir que les lignes supprimées ne sont pas les bonnes. A quoi est-ce dû ?

Tout simplement parce que les index retournés par les méthodes pour récupérer la sélection (getSelectedRows() par exemple) retournent l'index visuel. Dans le cas d'une JTable non triable, cet index correspond également à l'index du modèle, mais ce n'est plus le cas avec un tableau trié. On peut néanmoins très facilement résoudre ce problème en utilisant la méthode convertRowIndexToModel de la classe RowSorter. On va donc recoder la méthode RemoveAction avec cette nouvelle méthode :

```
private class RemoveAction extends AbstractAction {
    private RemoveAction() {
        super("Supprimmer");
    }

    public void actionPerformed(ActionEvent e) {
        int[] selection = tableau.getSelectedRows();
        int[] modelIndexes = new int[selection.length];

        for(int i = 0; i < selection.length; i++){
            modelIndexes[i] = tableau.getRowSorter().convertRowIndexToModel(selection[i]);
        }

        Arrays.sort(modelIndexes);

        for(int i = modelIndexes.length - 1; i >= 0; i--){
            modele.removeAmi(modelIndexes[i]);
        }
    }
}
```

On commence donc par récupérer les index au niveau de la vue, puis on les convertit au niveau modèle. Ensuite de quoi, il faut les trier pour supprimer les éléments depuis la fin. Vous verrez que cette fois, la suppression d'éléments fonctionne parfaitement.

Au prochain (et dernier) chapitre, on va étendre ce comportement en permettant de filtrer le contenu du tableau.

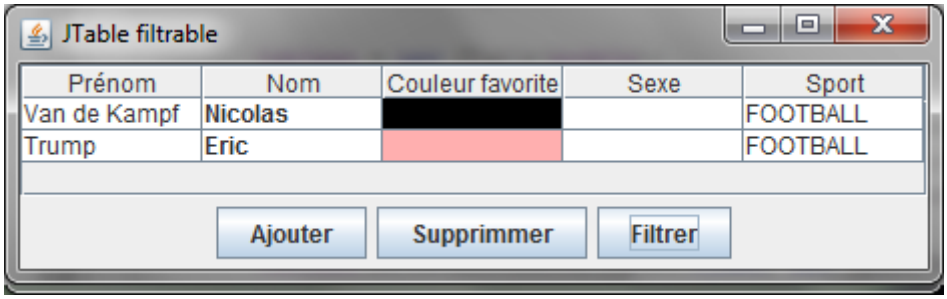
8 - Filtrer le contenu

En plus d'effectuer un tri sur les cellules, la classe RowSorter permet également de filtrer le contenu du tableau. On peut utiliser pour cela la méthode setRowFilter() qui prend en paramètre un objet RowFilter. RowFilter possède plusieurs méthodes statiques permettant de créer des filtres. Notamment des méthodes permettant d'effectuer des opérations "and" ou "or" sur des filtres. En plus de cela, on a également une méthode statique permettant d'effectuer un filtrage par regex sur une des colonnes.

On va ajouter un bouton permettant d'effectuer un filtre sur les colonnes "nom" et "prénom" :

```
private class FilterAction extends AbstractAction {
    private FilterAction() {
        super("Filtrer");
    }

    public void actionPerformed(ActionEvent e) {
        String regex = JOptionPane.showInputDialog("Regex de filtre : ");

        sorter.setRowFilter(RowFilter.regexFilter(regex, 0, 1));
    }
}
```

Il est donc extrêmement facile d'effectuer un filtre simple sur une ou plusieurs colonnes. Voici le résultat pour un filtre avec "mp" :



JTable dont le contenu est filtré avec la regex mp

Comme vous pouvez le constater, il est facile de filtrer le contenu avec les méthodes statiques de RowFilter. Pour plus de souplesse, on peut également étendre la classe RowFilter qui possède une seule méthode abstraite, include(Entry entry), qui permet d'indiquer si une ligne doit être incluse dans le tableau.

9 - Conclusion

Voilà, nous avons maintenant traité tous les différents aspects de la création et la manipulation de tableaux (JTable) avec Swing. J'espère que ce tutoriel vous permettra de maîtriser ce composant qui n'est, une fois les différents concepts compris, pas si compliqué à utiliser.

N'hésitez pas à commenter cet article sur le sujet lié sur le forum : .

Voici une archive ZIP contenant l'intégralité des sources de cet article : **Fichiers sources de cet article**.

9.1 - Remerciements

Un grand merci à **jacques_jean** et **Wachter** pour leurs corrections orthographiques.