

## Le composant JTable

*Nous disposons, pour notre application 'eVente', d'une bonne partie des couches 'Métier' et 'Data Access'. Il nous reste à développer l'une ou l'autre fonctionnalités de la couche 'Présentation'. Nous utiliserons une interface riche de type 'Swing' pour offrir les fonctionnalités du rôle 'Administrateur' qui aura à s'occuper du catalogue et des clients. Nous veillerons à développer des composants graphiques réutilisables pour faciliter la cohérence de la présentation et faciliter la maintenance. Ces composants utiliserons un certain nombre de composants Swing bien adaptés à nos besoins, nous commencerons par découvrir le composant JTable et en dériver un composant plus adapté à nos besoins.*

### 1 EXEMPLE SIMPLISTE

Dans cet exemple, nous développons un composant permettant d'afficher et d'éditer un ensemble de données 'conditionnées' sous forme de tableau d'objets à deux dimensions.

```
public class MonHoraire extends JPanel{
    public MonHoraire() {
        setLayout(new BorderLayout());
        Object[][] data={
            {"8h15-10h15", "", "", "", "", ""},
            {"10h30-12h30", "", "", "", "", ""},
            {"", "", "", "", "", ""},
            {"13h45-15h45", "", "", "", "", ""},
            {"16h00-18h00", "", "", "", "", ""}
        };
        String[] titreColonnes={"Heure", "Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi"};
        JTable horaire=new JTable(data, titreColonnes);
        add(new JScrollPane(horaire));
    }

    public static void main(String args[]){
        JFrame maFrame=new JFrame();
        JPanel jP=new JPanel();
        MonHoraire hor=new MonHoraire();
        maFrame.add(hor, BorderLayout.CENTER);
        maFrame.setVisible(true);
    }
}
```

Il est possible de personnaliser l'affichage et le comportement de la JTable au travers de méthodes de la table et de ses composants: cf. <http://java.sun.com/j2se/1.4.2/docs/api/javaw/swing/JTable.html>

## 2 EXEMPLE UTILISANT UN MODÈLE

Les JTable mettent en oeuvre de manière interne le DP MVC. Il nous sera utile de développer des composants adaptés à nos besoins utilisant les Jtable. Nous allons produire un composant réutilisable d'affichage de liste pour les produits, clients, .... Pour cela nous devons définir un modèle de table dérivé d'AbstractTableModel. Nous devons passer au constructeur de la JTable les données, les titres des colonnes et le nom des méthodes des objets présentés.

MonModele<T> reprendra le modèle des données à présenter c'est à dire une Collection de ProduitDto, de ClientDto, ...

### 1 Le modèle

```
public class MonModele<T> extends AbstractTableModel {
    Vector<T> data ;
    String[] titresColonnes ;
    String[] methodNames;

    public MonModele(Collection<T> data, String[] titresColonnes, String[] methodNames) {
        this.data=new Vector<T>(data);
        this.titresColonnes=titresColonnes ;
        this.methodNames=methodNames;
    }
}
```

Mais nous aurons aussi à implémenter au minimum les méthodes promises par l'interface TableModel: public int getRowCount(), public int getColumnCount(), public Object getValueAt(int rowIndex, int columnIndex)

```
public int getRowCount() {
    return data.size();
}
public int getColumnCount() {
    return titresColonnes.length;
}
public Object getValueAt(int rowIndex, int columnIndex) {
    try {
        T o = data.get(rowIndex);
        Method m = o.getClass().getMethod(methodNames[columnIndex]);
        return m.invoke(o);
    } catch (Exception ex) {
        // sans objet
        ex.printStackTrace();
        return null;
    }
}
```

pour pouvoir afficher les noms de colonnes nous devons prévoir :

```
public String getColumnName(int col){return titresColonnes[col]; }
```

## 2 MaJTable

Pour définir un Panel reprenant ce type de JTable, prévoyons :

```
public class MaJTable<T> extends JPanel {
    JTable maTable ;
    MonModele<T> monModele;

    protected MaJTable() {
        setLayout(new BorderLayout());
        maTable = new JTable();
        add(new JScrollPane(maTable));
    }

    protected void setModel(MonModele<T> monModele) {
        this.monModele = monModele;
        maTable.setModel(monModele);
    }

    public void setColumnWidth(int[] largeurs){
        int i=0;
        while (i<largeurs.length && i<maTable.getColumnCount()){
            maTable.getColumn(maTable.getColumnModel().getColumn(i)).setPreferredWidth(largeurs[i]);
            i++;
        }
    }
}
```

## 3 MaJTableProduit

Pour mettre en oeuvre ce composant, nous allons en dériver un composant spécifique pour les produits MaJTableProduit<sup>1</sup> :

```
public class MaJTableProduit extends MaJTable<ProduitDto> {
    private Collection<ProduitDto> data;
    private String[] titres = {"id", "Libellé", "Marque", "Prix", "Promotion", "Rayon", "Dispo"};
    private String[] methodes = {"getId", "getLibelle", "getMarqueLib", "getPrix", "getPromotion",
    "getRayonLib", "isDisponible"};
    private int[] largeurs = {15, 300, 100, 20, 20, 80, 20};

    public MaJTableProduit() {super();}

    public MaJTableProduit(Collection<ProduitDto> data) {
        super();
        setData(data);
    }

    public void setPresentation(String[] titres, String[] methodes, int[] largeurs ) {
        this.titres=titres;
        this.methodes=methodes;
        this.largeurs=largeurs;
        if (data!=null) { setData(data); }
    }

    public void setData(Collection<ProduitDto> col){
        this.data=col;
        MonModele<ProduitDto> modele=new MonModele<ProduitDto>(col,titres,methodes);
        setModel(modele);
        setColumnWidth(largeurs);
    }
}
```

<sup>1</sup> Votre classe ProduitDTO doit posséder les méthodes getMarqueLib() et getRayonLib()

## 4 Test

Ajoutons le main suivant pour pouvoir tester:

```
public static void main(String[] args){
    JFrame maFrame=new JFrame();
    JPanel jP=new JPanel();
    Collection<ProduitDto> produits=null;
    try {
        produits = AdminFacade.getAllProduits();
    } catch (EventeBusinessException ex) {
        ex.printStackTrace();
        // en pratique, à traiter
    }
    MaJTableProduit maTable=new MaJTableProduit(produits);
    maFrame.add(maTable);
    maFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    maFrame.pack();
    maFrame.setVisible(true);
}
```

## 5 Connaître l'élément sélectionné

De nombreux ajouts doivent être réalisés pour offrir les fonctionnalités nécessaires à notre application. Pour permettre de récupérer un objet de la liste, nous aurons à ajouter au modèle la méthode suivante :

```
public T getObject(int ind){
    return data.get(ind);
}
```

et à MaJTable :

```
public int getRowCount(){
    return maTable.getRowCount();
}
public int getSelectedRow(){
    return maTable.getSelectedRow();
}

public T getSelectedObject(){
    if (maTable.getSelectedRow() < 0) return null;
    else return monModele.getObject(maTable.getSelectedRow());
}
```

## 6 Gestion des événements.

Plusieurs événements sont à gérer :

- sélection d'un objet(clic), ouverture d'un objet(double clic),...
- modification de la liste présentée, ...
- ...

Illustrons avec le changement de sélection. Dans MaJTable nous aurons à ajouter un attribut :

```
private T objSelected=null;
```

et une méthode (qui devra être appelée du constructeur) :

```
private void gereEvtSelections() {
    maTable.getSelectionModel().addListSelectionListener(new ListSelectionListener() {
        public void valueChanged(ListSelectionEvent e) {
            //ne prend pas en compte l'événement de préparation à la sélection
            if (e.getValueIsAdjusting()) {
                return;
            }
            ListSelectionModel lsm = (ListSelectionModel) e.getSource();
            if (lsm.isSelectionEmpty()) {
                firePropertyChange("Unselected", objSelected, null);
                objSelected = null;
            } else {
                int selectedRow = lsm.getMinSelectionIndex();
                firePropertyChange("Selected", objSelected, getSelectedObject());
                objSelected = getSelectedObject();
            }
        }
    });
}
```

Testons en ajoutant ceci dans le main() :

```
maTable.addPropertyChangeListener(new PropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent propertyChangeEvent) {
        System.out.println("Event : " + propertyChangeEvent.getPropertyName() + " - "
            + propertyChangeEvent.getNewValue() );
    }
});
```

### 3 CONCLUSION

---

Définir MaJTable peut sembler lourd mais nous offre un composant générique que nous utiliserons pour les différents types d'objets à lister tout en offrant une homogénéité d'apparence et de comportement dans l'interface.

Ainsi définie, MaJTable offre le minimum de fonctionnalités dont nous aurons besoin. Beaucoup d'autres fonctionnalités peuvent être prévues et seront peut être nécessaires suite à l'apparition de nouveaux besoins. Entre autres, nous pouvons veiller à interdire le réordonnement de colonnes, permettre à l'utilisateur de réaliser un réordonnement des lignes, interdire la sélection multiple qui n'a pas d'objet dans notre cadre, ... Ajoutons notamment dans les constructeurs de MaJTable:

```
maTable.setDragEnabled(false);
maTable.getTableHeader().setReorderingAllowed(false);
maTable.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
```

Il est évident que nous avons encore du travail avant de publier notre composant : prévoir la JavaDoc, écrire le BeanInfo, ...

### 4 EXERCICE

---

Nous avons créé un composant MaJTableProduit. Nous pouvons imaginer réaliser des composants analogue pour Client, Commande, ...

**Exercice** : Mettez ce composant en œuvre pour lister des Clients. Attention, pour ce faire, il faudra ajouter à AdminFacade les méthodes getClientById(int id), getCliSelectionnes(ClientSel sel), getClientByIdent(String ident).