

# Introduction à Java Foundation Class et Swing

# Présentation

Java Foundation Class (JFC) est un ensemble regroupant des ajouts à Java 1.1 qui sont désormais intégrés au JDK1.2 (Java 2 Platform).

JFC comprend :

- les composants Swing. C'est l'objet de ce cours. Ce sont des composants avancés complètement écrits en langage Java et ne faisant donc pas appel aux composants de la plate-forme (composants ou appels natifs).
- Java 2D. Utilisation de classes Graphics 2D amenant des manipulations complexes de la couleur, la manipulation simple des transformation affines (rotation, homothétie, ...), traitement des textures, ...
- L'accessibilité : la manipulation simple des ordinateurs pour les personnes handicapés moteurs
- Le « drag and drop » : glisser-déposer entre application quelconque (pas forcément Java) sur une plate-forme.

Lorsque le JDK1.2 est installé, on trouve une démonstration de l'ensemble des composants Swing dans le répertoire `INSTAL_JDK1_2\demo\jfc\SwingSet` et on lance :

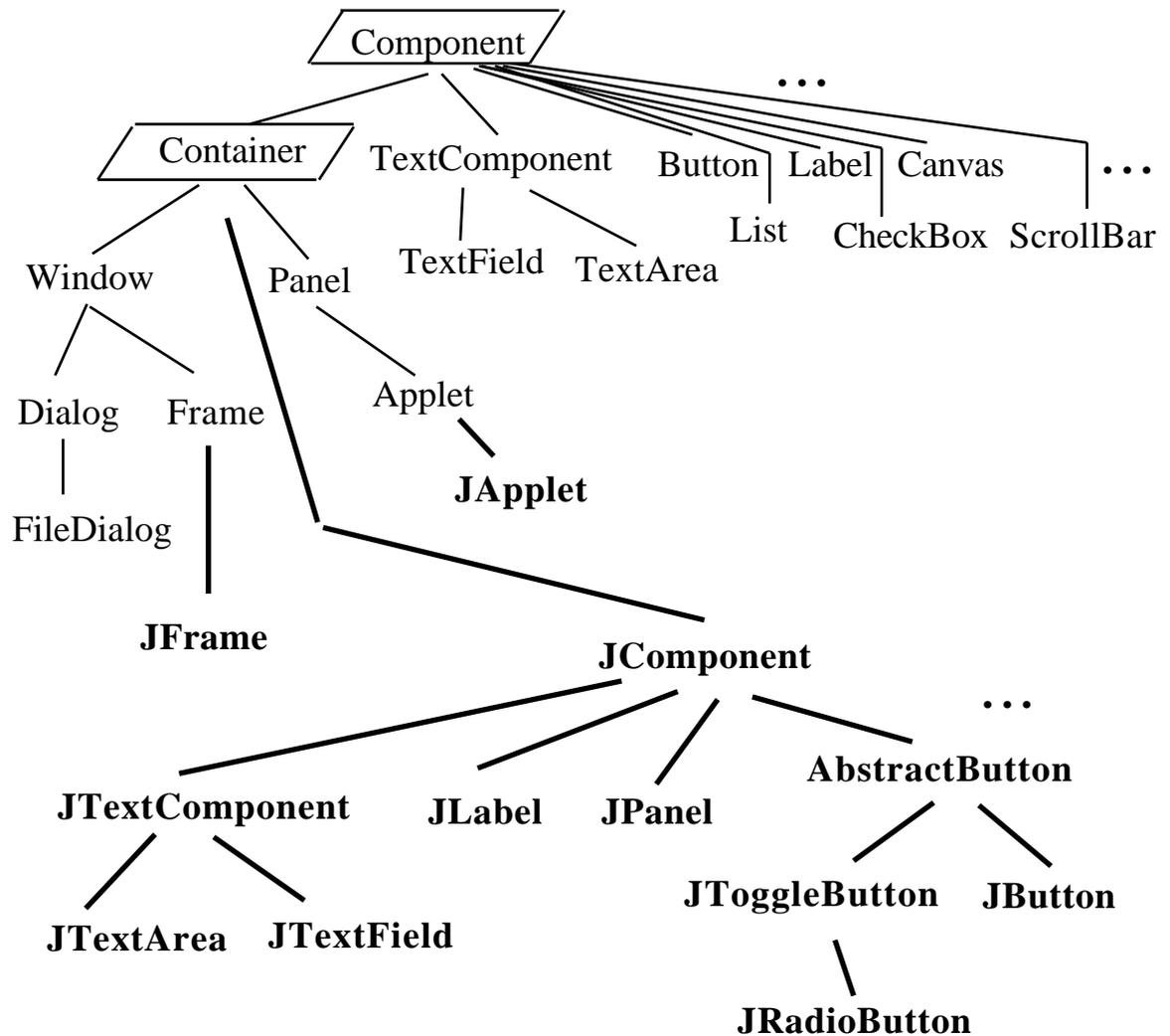
```
java SwingSet
```

ou

```
appletviewer SwingSetApplet.html
```

# Les classes Swing

Les classes Swing s'appuient sur l'arborescence AWT. Les principales classes Swing sont :



## Remarques générales

En Java 1.2, les classes Swing se trouvent dans le paquetage `javax.swing` ou ses sous-paquetages. Il est fortement déconseillé de mélanger des composants Swing et AWT dans une interface graphique.

# Java Plug In

Pour utiliser Swing avec les applets, il faut installer le Java plug-in.

C'est un plug in à mettre dans un navigateur Web (Netscape Navigator ou Internet Explorer) pour utiliser la MV de SUN au lieu de celle du navigateur.

Gros avantages :

les MV de SUN sont à jour vis à vis des versions courantes de Java.

Java plug-in à chercher à :

<http://www.javasoft.com/products/plugin/>

Disponible aussi pour Linux voir à

<http://www.blackdown.org/activator/index.html>

anciennement appelé Java Activator.

Technique disponible depuis le 30 Avril 1998.

gratuit.

Lors d'une première connexion demande à charger le plug-in

Le plug-in pour Java 1.2 fait 7,434,288 bytes !!

Avec ce plug-in on peut alors :

- exécuter des applets 100% pure Java de la version Java courante.

# Java plug in : le fonctionnement

Il faut utiliser une autre balise que la balise `APPLET` pour utiliser la MV de SUN.

Ce sont les balises :

`OBJECT` dans le cas de Explorer.

`EMBED` dans le cas de Netscape.

Il faut donc changer les pages HTML. Un outil est proposé par SUN : HTML Converter (pour 300 Ko environ en .zip pour Java 1.2) voir documentation à :

<http://java.sun.com/products/plugin/1.1.1/docs/tags.html>

# De APPLET à OBJECT

Pour Internet Explorer.

Une balise comme :

```
<APPLET code="XYZApp.class" codebase="html/" width="200"
height="200">
<PARAM NAME="model"
VALUE="models/HyaluronicAcid.xyz">
</APPLET>
```

doit être transformée en :

```
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-
00805F499D93"
width="200" height="200"
codebase="http://java.sun.com/products/plugin/1.1/jinstall-11-
win32.cab#Version=1,1,0,0">
<PARAM NAME="code" VALUE="XYZApp.class">
<PARAM NAME="codebase" VALUE="html/">
<PARAM NAME="type" VALUE="application/x-java-
applet;version=1.1">
<PARAM NAME="model"
VALUE="models/HyaluronicAcid.xyz">
</OBJECT>
```

## remarque

La balise OBJECT est une balise HTML normalisé.

classid est l'identificateur de classe du Java plug-in. Sa valeur est celle ci dessus. Les balises code (et codebase) ne sont pas directement conservées du passage de APPLET à OBJECT : dans OBJECT on utilise la sous-balise PARAM.

# De APPLET à OBJECT (fin)

L'attribut `codebase` de la balise `OBJECT` a une autre signification : elle indique l'URL de chargement du plug-in Java au cas où ce plug-in ne se trouverait pas sur la machine locale.

le champ `type` de `PARAM` indique au Java plug-in le type de programme Java à exécuter i.e. une applet ou un bean. C'est un type MIME.

remarque

les 3 types de `PARAM` `code`, `codebase` et `type` ne doivent pas apparaître dans les `PARAM` de la balise `APPLET`.

# De APPLET à EMBED

Pour Netscape Navigator.

La balise :

```
<APPLET code="XYZApp.class" codebase="html/" width="200"
height="200">
<PARAM NAME="model"
VALUE="models/HyaluronicAcid.xyz">
</APPLET>
```

doit être transformée en :

```
<EMBED type="application/x-java-applet;version=1.1"
width="200" height="200" code="XYZApp.class"
codebase="html/" model="models/HyaluronicAcid.xyz"
pluginspage="http://java.sun.com/products/plugin/1.1/plugin-
install.html">
<NOEMBED>
</NOEMBED>
</EMBED>
```

Contrairement à OBJECT toutes les informations sont des attributs de la balise EMBED => plus de balise PARAM.

Entre autre, les balises PARAM de APPLET

(par exemple

```
<PARAM NAME="model "
```

```
VALUE="models/HyaluronicAcid.xyz">)
```

deviennent des attributs de EMBED (par exemple

```
model="models/HyaluronicAcid.xyz").
```

type indique le type de programme Java à exécuter : applet ou bean.

pluginspage est utilisé si le Java plug-in n'est pas déjà chargé. Sa valeur indique une URL de chargement de ce Java plug-in.

# De APPLET à EMBED (suite)

Entre `<NOEMBED>` et `</NOEMBED>`, est mis un texte de remplacement au cas où le browser ne supporte pas la balise `EMBED`.

## Java plug-in dans MSIE et NN

Finalement, une balise classique comme :

```
<APPLET code="XYZApp.class" codebase="html/"
align="baseline" width="200" height="200">
<PARAM NAME="model"
VALUE="models/HyaluronicAcid.xyz">

No JDK 1.1 support for APPLET!!
</APPLET>
```

devient, pour être lue par Java plug-in dans MSIE et NN :

# Java plug-in dans MSIE et NN (suite)

```
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93" width="200" height="200" align="baseline"
codebase="http://java.sun.com/products/plugin/1.1/jinstall-11-
win32.cab#Version=1,1,0,0">
  <PARAM NAME="code" VALUE="XYZApp.class">
  <PARAM NAME="codebase" VALUE="html/">
  <PARAM NAME="type" VALUE="application/x-java-
  applet;version=1.1">
  <PARAM NAME="model"
  VALUE="models/HyaluronicAcid.xyz">
    <COMMENT>
    <EMBED type="application/x-java-applet;version=1.1"
width="200" height="200" align="baseline"
code="XYZApp.class"
codebase="html/" model="models/HyaluronicAcid.xyz"
pluginspage="http://java.sun.com/products/plugin/1.1/plu
gin-install.html">
    <NOEMBED>
    </COMMENT>
      No JDK 1.1 support for APPLET!!
  </NOEMBED></EMBED>
</OBJECT>
```

Les balises `<COMMENT>` et `</COMMENT>` ne sont comprises que par MSIE et ce navigateur ignore le contenu entre ces 2 balises. On obtient alors code OBJECT correct pour MSIE. NN ne comprend pas les balises OBJECT et COMMENT et lit le résultat recherché :

```
  <EMBED type="application/x-java-applet;version=1.1"
width="200" height="200" align="baseline"
code="XYZApp.class"
codebase="html/" model="models/HyaluronicAcid.xyz"
pluginspage="http://java.sun.com/products/plugin/1.1/plu
gin-install.html">
  <NOEMBED>
  No JDK 1.1 support for APPLET!!
</NOEMBED></EMBED>
```

# Java plug-in : quelques critiques

Il s'installe assez bien mais est parfois bogué (impossible d'avoir des renseignements en cliquant sur l'icône Programmes | Java Plug in sous Windows NT)

appletviewer ne sait pas lire un fichier transformé par HTML Converter (balise `<embed>` non connue)

Il faut gérer deux familles de fichiers pour les browsers utilisant Java plug in (et donc `<embed>` et `<object>`) et les browsers utilisant `<applet>`.

# Les composants "haut niveau"

Les composants dit de haut niveau sont :

- les fenêtres fille de la fenêtre fond d'écran, objets des classes `JFrame`, `JWindow`, `JDialog`
- ainsi que les objets de la classe `JApplet`

Pour ces objets conteneurs, on n'ajoute pas leurs contenus directement dans ces objets (contrairement à AWT). On passe par l'intermédiaire de leur « `ContentPane` » qui est récupéré par la méthode `getContentPane()`. Par exemple :

```
|| JFrame maFrame = ... ;  
|| JButton monJButton= ... ;  
||  
|| maFrame.getContentPane().add(monJButton,  
|| BorderLayout.NORTH) ;
```

# Les ajouts Swing

Par rapport à AWT, Swing propose des améliorations notoires sur certains composants (JLabel, JButton, ...) par rapport à Label, Button, ...) et de nouveaux composants (bulle d'aide, onglet, ...)

## JLabel

On peut construire facilement un JLabel comportant une image gif ou jpg (accompagné d'un texte ou non).

```
|| Icon image = new ImageIcon("tarde.jpg");  
|| JLabel labelImage = new JLabel(image);  
||
```

Le constructeur le plus complet de JLabel est :

```
public JLabel(String text, Icon icon,  
horizontalAlignment)
```

et il existe d'autres constructeurs avec des arguments en moins.

## JTabbedPane

Les onglets n'existent pas en AWT. On crée une boîte à onglets par le constructeur `public JTabbedPane(int tabPlacement)` qui indique où placer les titres des onglets, ou bien par `public JTabbedPane()` qui par défaut les placent en haut.

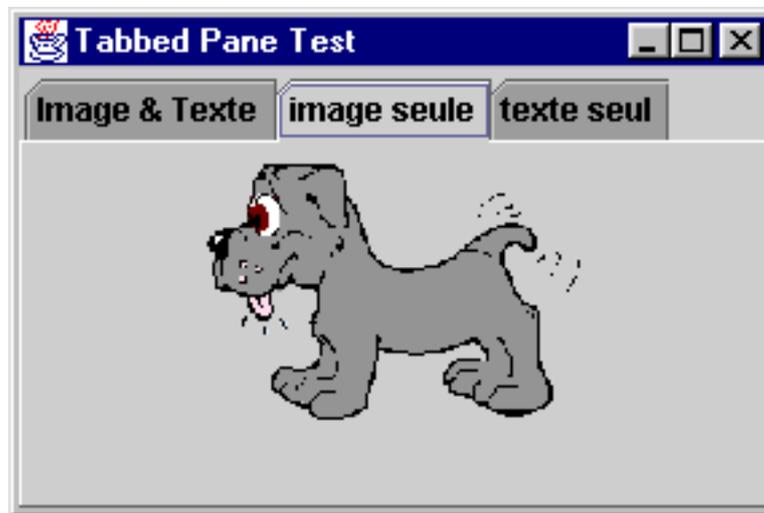
On ajoute les composants (souvent des JPanel) dans la boîte par :

```
public void addTab(String title, Icon  
icon, Component component, String tip)
```

ou des méthodes de même nom avec moins d'argument.

# JLabel et JTabbedPane

Voici un programme qui utilise ces composants :



```
import java.awt.*;
import java.util.*;
import java.awt.event.*;
import javax.swing.*;

public class TabTest extends JPanel {
    private JTabbedPane jtp;
    private JLabel labelImage;
    private JPanel panneau1 = new JPanel();
    private JPanel panneau2 = new JPanel();
    private JPanel panneau3 = new JPanel();
```

```
public TabTest() {
    setLayout(new BorderLayout());
    jtp = new JTabbedPane();

    Icon image = new ImageIcon("clouds.jpg");
    labellImage = new JLabel("label avec texte et
image", image, SwingConstants.CENTER);
    panneau1.add(labellImage);

    Icon image2 = new ImageIcon("dog.gif");
    panneau2.add(new JLabel(image2));

    panneau3.add(new JLabel("JLabel avec du
texte seulement"));

    jtp.addTab("Image & Texte", panneau1);
    jtp.addTab("image seule", panneau2);
    jtp.addTab("texte seul", panneau3);

    add(jtp, BorderLayout.CENTER);
}

public static void main(String args[]) {
    JFrame jf = new JFrame("Tabbed Pane Test");
    TabTest tt = new TabTest();
    jf.getContentPane().add(tt,
BorderLayout.CENTER);
    jf.setSize(600,300);
    jf.setVisible(true);
}
}
```

---

# Look and Feel

Swing propose plusieurs aspects et utilisation de l'interface graphique : Look and Feel (L&F).

Le Look and Feel de Java est appelé Metal ou Java L&F : c'est le L&F par défaut. On peut aussi avoir les L&F Motif, Windows et Macintosh (contrôlé par des droits) et même se créer son propre L&F.

On peut connaître les divers L&F utilisables sur une machine à l'aide de la méthode statique

```
UIManager.getInstalledLookAndFeels();
```

qui retourne un tableau de

```
UIManager.LookAndFeelInfo
```

# Look and Feel (suite)

Par exemple le programme :

```
import javax.swing.*;

public class ListPlafs {
    public static void main (String args[]) {
        UIManager.LookAndFeelInfo plaf[] =
        UIManager.getInstalledLookAndFeels();
        for (int i=0, n=plaf.length; i<n; i++) {
            System.out.println("Nom : " +
            plaf[i].getName());
            System.out.println("Nom de la
            classe : " + plaf[i].getClassName());
        }
        System.exit(0);
    }
}
```

**retourne :**

Nom : Metal

Nom de la classe :

javax.swing.plaf.metal.MetalLookAndFeel

Nom : CDE/Motif

Nom de la classe :

com.sun.java.swing.plaf.motif.MotifLookAndFeel

Nom : Windows

Nom de la classe :

com.sun.java.swing.plaf.windows.WindowsLookAndFeel

# Look and Feel (suite)

On positionne le L&F par la méthode statique

```
public static void setLookAndFeel(String  
className)throws ClassNotFoundException,  
InstantiationException,  
IllegalAccessException,  
UnsupportedLookAndFeelException
```

de la classe UIManager.

Puis il faut indiquer la racine de l'arborescence qui prend en compte ce L&F par :

```
SwingUtilities.updateComponentTreeUI(racine);
```

en général cette racine est la fenêtre principale.

# Divers L&F

Les différents L&F de la plate-forme  
Windows



sont obtenus par le programme :

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/**
 * Cette classe illustre le changement dynamique de
 * L&F
 */
public class LnF extends JPanel {
    private JButton jb = new JButton("A Button");
    private String [] cbitems = {"One", "Two", "Three"};
    private JComboBox jcb = new
    JComboBox(cbitems);
    private JTextField jtf = new JTextField("Hello", 14);
    private JRadioButton jrbHost = new
    JRadioButton("Windows", false);
    private JRadioButton jrbCross = new
    JRadioButton("CrossPlatform", true);
    private JRadioButton jrbMotif = new
    JRadioButton("Motif", true);
    private ButtonGroup bg = new ButtonGroup();
```

```
public LnF() {
    bg.add(jrbHost);
    bg.add(jrbCross);
    bg.add(jrbMotif);
    add(jb);
    add(jcb);
    add(jtf);
    add(jrbHost);
    add(jrbCross);
    add(jrbMotif);

    jrbMotif.addActionListener(
        new ActionListener() {
            public void actionPerformed(ActionEvent ev) {
                // installe le L&F Motif
                try {
                    UIManager.setLookAndFeel(
                        "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
                } catch (Exception e) {}
            }
        }
    );
}
```

**[www.Mcours.com](http://www.Mcours.com)**  
Site N°1 des Cours et Exercices Email: [contact@mcours.com](mailto:contact@mcours.com)

```
jrbHost.addActionListener(  
    new ActionListener() {  
        public void actionPerformed(ActionEvent ev) {  
            // installe le L&F de la plate-forme i.e. Windows  
            try {  
                UIManager.setLookAndFeel(  
                    UIManager.getSystemLookAndFeelClassName(  
));  
  
                SwingUtilities.updateComponentTreeUI(LnF.this  
);  
                } catch (Exception e) {}  
            }  
        }  
    });  
jrbCross.addActionListener(  
    new ActionListener() {  
        public void actionPerformed(ActionEvent ev) {  
            // installe le L&F Metal  
            UIManager.setLookAndFeel(  
                UIManager.getCrossPlatformLookAndFeelClass  
Name());  
            SwingUtilities.updateComponentTreeUI(LnF.this  
);  
            } catch (Exception e) {}  
        }  
    });  
}  
public static void main(String args[]) {  
    JFrame jf = new JFrame("LnF");  
    LnF lnf = new LnF();  
    jf.getContentPane().add(lnf);  
    jf.pack();  
    jf.setVisible(true);  
}
```

# L&F compléments

On peut aussi récupérer le L&F Metal par la méthode de statique de la classe UIManager.

```
public static String  
getCrossPlatformLookAndFeelClassName()
```

et le L&F de la plate-forme par la méthode statique de la classe UIManager :

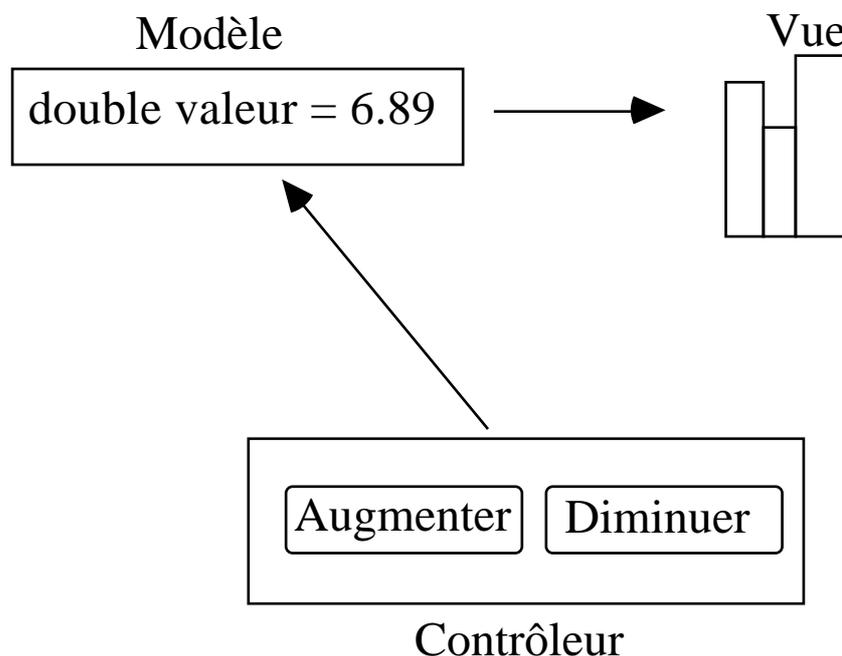
```
public static String  
getSystemLookAndFeelClassName()
```

# Le modèle MVC

Swing utilise fondamentalement une architecture d'interface homme-machine inspiré de Smalltalk : l'architecture Model-View-Controller.

Cette architecture est composée de trois parties :

- le modèle qui est la partie décrivant les données à afficher
- la vue qui est la représentation graphique de ces données
- le contrôleur qui est la partie qui traite des interactions du composant avec l'utilisateur.



---

# Intérêts de l'architecture MVC

- meilleure modularité
- possibilité d'associer plusieurs vues distinctes à un même modèle (histogramme, courbes, camembert, valeur flottante, ...)
- possibilité de changer l'implémentation d'un modèle (optimisation, réorganisation, ...) sans rien changer aux vues et aux contrôleurs.

# Implémenter le modèle MVC

On s'appuie sur une technique qu'on retrouve dans d'autres parties du langage Java (Java Beans, ...) et qui utilise le mécanisme des événements par délégation (Java 1.1 et suivant).

Le modèle permet d'inscrire des objets qui seront écouteurs de ce modèle. Lorsque les données changent le modèle envoie des renseignements sous forme d'un objet événement à ces (ses !!) écouteurs en leur demandant d'exécuter une méthode convenue.

Dans le modèle MVC, ce sont :

- les contrôleurs qui demandent à changer une donnée du modèle
- les vues qui sont les écouteurs du modèle et qui sont avertis du changement des données.

Pour cela, les vues doivent s'enregistrer auprès du modèle. Celui-ci possède des méthodes appropriées (`addXXXListener()`).



---

# Implémenter le modèle MVC (suite)

Les étapes pour construire cet ensemble sont :

1°) Définir une classe événement qui contiendra les informations à envoyer du modèle vers ses vues. L'événement qui sera généré est parfois appelé l'événement source. Le générateur de cet événement est appelé parfois l'"objet source de l'événement".

2°) définir une méthode convenue que devront lancer les vues lorsqu'elles seront informées d'un événement. Mettre cette méthode dans une interface que devront implémenter les vues.

# Implémenter le modèle MVC (suite)

La classe événement à définir est souvent une sous classe de `java.util.EventObject` qui contient déjà des méthodes appropriées pour cette architecture. Par exemple retourner l'objet source de l'événement par la méthode :

```
public Object getSource()
```

On a donc :

```
public class XXXEvent extends  
java.util.EventObject { ... }
```

La méthode à lancer peut être de nom quelconque mais les conventions proposent qu'elle soit de la forme :

```
public void nomQuelconque(XXXEvent e)  
{...}
```

De même il est convenu que cette méthode soit dans une interface de la forme :

```
public interface XXXListener {  
    public void nomQuelconque(XXXEvent e)  
        {...}  
}
```

# Implémenter le modèle MVC (suite)

Le modèle qui enregistre, retire, diffuse les événements aux différents listeners propose par convention :

```
public MonModele {  
    Vector lesListener = new Vector() ;  
    public void addXXXListener(...){ ...}  
    public void removeXXXListener(...){ ...}  
    public void fireXXXEvent(...){ ...}  
}
```

Plus précisément, ces méthodes sont :

```
public void addXXXListener(XXXListener l){  
    lesListener.addElement(l) ;  
}  
  
public void removeXXXListener(XXXListener l){  
    lesListener.removeElement(l) ;  
}
```

Avertir l'ensemble des listeners est obtenu par un code dont le trame est :

```
public void fireXXXEvent(Info i){  
    XXXEvent evt = new XXXEvent(this, i) ;  
    /* this est le modele */  
    Enumeration e = lesListeners.elements() ;  
    while (e.hasMoreElements()){  
        ((XXXListener)e.nextElement()).nomQuelconque(evt) ;  
    }  
}
```

## Remarques sur MVC

En pratique, ce ne sont pas les vues qui sont des listeners mais des classes internes aux vues (des classes `XXXDataListener`) ou parfois des classes anonymes construites « à la volée ».

Les modèles sont souvent définis à partir d'une interface Java puis sont implémentés à partir de cette interface. Les vues et les contrôleurs manipulent les interfaces de ces modèles. Ainsi les vues et les contrôleurs ne sont pas liés à un modèle particulier mais à un ensemble de modèles qui implémentent le même interface.

Ces mécanismes étant classiques en Java (cf. Java Beans), il existe des classes et interfaces toutes dans le paquetage `java.beans` qui donnent ce genre de code :

- `PropertyChangeEvent` (classe qui hérite de `java.util.EventObject`),
- `PropertyChangeListener` (interface qui définit une méthode à lancer par les listeners lors d'un changement de « propriété »)
- `PropertyChangeSupport` (classe dont il faut hériter proposant les enregistrements, les retraits des listeners et le lancement de la méthode définie dans `PropertyChangeListener`).

# Les composants Swing et MVC

On va étudier les composants « complexes » de Swing : JList, JTable, JTree.

Les composants List et Table utilisent une arborescence de classes et d'interfaces de la forme :

```
-public interface XXXModel {...}
-public abstract class AbstractXXXModel
  implements XXXModel, Serializable
-public class DefaultXXXModel extends
  AbstractXXXModel { ... }
  où XXX vaut List ou Table.
```

Certains de ces composants sont assez riches pour occuper à eux seuls un ou plusieurs paquetages !! C'est le cas avec les paquetages `javax.swing.table` et `javax.swing.tree`.

Les JXXX sont des vues. Pour les construire on utilise souvent les constructeurs `JXXX(leModele)`.



# Cellules

Les éléments d'une `JTable` ainsi que les nœud d'un `JTree` sont appelés des cellules (`cell`). Ces cellules sont similaires aux éléments d'un `JList` ou d'une `JComboBox` et sont dessinées par des « `renderer` ». Ce `renderer` est une « machine » (`factory`) pour fabriquer des `Component` (i.e. des cellules).

En fait l'objet construit est un `Component` ou plutôt un objet d'une sous classe, par défaut un `JLabel` construit à partir de la valeur de la cellule sur lequel est lancé `toString()`, et les composants `JXXX` affichent des `Component`.

# Le composant JList

Une `JList` modélise comme une `java.awt.List`, une liste. Contrairement à une `List`, une `JList` n'a pas de barre de défilement par défaut. Aussi on met souvent une `JList` dans une `JScrollPane` :

```
|| JList maJList = ... ;  
|| JScrollPane maJSP = new JScrollPane(maJList) ;  
||
```

`JList` utilise l'arborescence : interface `ListModel` implémentée par la classe abstraite `AbstractListModel` dont dérive la classe concrète `DefaultListModel`.

## L'interface `ListModel`

contient le minimum de déclarations de méthodes.

Ce qu'il faut pour gérer des vues (listener d'événements)

```
public void  
addListDataListener(ListDataListener l) ;
```

et

```
public void  
removeListDataListener(ListDataListener  
l) ;
```

Récupérer un élément donné dans la liste ainsi que le nombre d'éléments de cette liste :

```
public Object getElementAt(int index) ;  
public int getSize() ;
```

# Le composant `JList` (suite)

## La classe `AbstractListModel`

Cette classe abstraite donne un corps pour les méthodes de traitement des événements ci dessus `add/remove..Listener()` et `fire..()`.

## La classe `DefaultListModel`

Cette classe concrète donne une implantation d'un modèle de liste à l'aide de `Vector` ainsi que des méthodes de gestion de liste associées.

## JList (suite)

Le code habituel qu'on écrit est :

- construire une classe concrète modèle de liste à partir de la classe `DefaultListModel` qui implante beaucoup de fonctionnalités nécessaires pour les listes.
- construite une `JList` à l'aide de ce modèle.

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;

public class EssaiJList {
    // declaration du tableau de donnees
    private static String [] data = {
        "Et un", "Et deux", "Et trois", "Zéro" };
    public static void main(String args[]) {
        JFrame jf1 = new JFrame("Essai JList");
        DefaultListModel dlm = new
DefaultListModel();
        // remplir le modele de donnees
        for (int i = 0; i < data.length; i++)
            dlm.addElement(data[i]);
        JList jl1 = new JList(dlm) ;
        jf1.getContentPane().add(jl1,
BorderLayout.CENTER);
        // Créer un controleur et le mettre dans l'IHM
        jf1.getContentPane().add(new
ListController(dlm), BorderLayout.SOUTH);
        jf1.pack();
        jf1.setVisible(true); }
}
```

# Le Contrôleur

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Cette classe est un JTextField qui est un
 * controleur pour un modele de liste.
 * Il ajoute des items a ce modele
 * lorsque l'utilisateur appuie sur " Enter ".
 */
public class ListController extends JTextField {
    private final DefaultListModel model;

    public ListController(DefaultListModel lm) {
        model = lm;
        // Ce controleur est son propre ActionListener
        // Lorsque l'utilisateur appuie sur Entrer,
        // la chaine (String) est ajoutee au modele.
        // Ceci est effectuee par une classe anonyme
        // qui implemente l'interface ActionListener.
        addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent ev)
                {
                    // On utilise la methode getActionCommand()
                    // de l'ActionEvent
                    // pour recuperer le texte du JTextField
                    // et l'ajoute au modele.
                    model.addElement(ev.getActionCommand());
                }
            }
        );
    }
}
```

# Le composant JTable

JTable modélise un tableau c'est à dire une structure présentant des lignes et des colonnes, bref un tableur. Les cellules de ce tableur sont des Component.

C'est un composant complexe et beaucoup de classes dont il dépend sont dans le paquetage `javax.swing.table`.

En général on met une JTable dans une JScrollPane. Si ce n'est pas le cas les entêtes des colonnes n'apparaissent pas.

Les données d'un JTable sont dans le modèle `javax.swing.table.TableModel`.

Plus précisément, on a l'architecture :

```
-public interface TableModel {...}
-public abstract class
  AbstractTableModel implements
  TableModel, Serializable
-public class DefaultTableModel
  extends AbstractTableModel { ... }
```

Voici un exemple créant une JTable de 10 lignes et 10 colonnes d'Integer.

```
TableModel dataModel = new AbstractTableModel() {
    public int getColumnCount() { return 10; }
    public int getRowCount() { return 10;}
    public Object getValueAt(int row, int col) { return new
Integer(row*col); }
};
JTable table = new JTable(dataModel);
JScrollPane scrollpane = new JScrollPane(table);
```

---

# Les colonnes dans une JTable

On dispose aussi d'un modèle pour les colonnes spécifié par l'interface `TableColumnModel` implémenté par la classe concrète `DefaultTableColumnModel` qui indique le nombre de colonnes, les colonnes elles mêmes, propose le mode de sélection (une seule cellule, une seule ligne ou colonnes, plusieurs lignes ou colonnes contigües ou pas).

Les colonnes sont repérées par des indices commençant à 0 ou des noms, typées i.e. les cellules sont des objets de classe. Le `renderer` par défaut fabrique un `JLabel` à partir de cet objet sur lequel a été lancé la méthode `toString()`. Les cellules d'une même colonne doivent être des objets d'une même classe ou de classes sous classes d'une classe commune.

# L'interface TableModel

Cette interface contient les déclarations et indique la sémantique des méthodes que doivent implanter un modèle de JTable.

Ce qu'il faut pour gérer des listener d'événements lorsque le modèle change.

```
public void  
add/removeTableModelListener(ListDataListe  
ner l) ;
```

```
Class getColumnClass(int columnIndex)  
qui retourne la classe de base commune à  
tous les éléments de la colonne.
```

```
int getColumnCount() (resp int  
getRowCount()) retourne le nombre de  
colonnes (resp de lignes) de l'objet
```

```
String getName(int columnIndex)  
retourne le nom de la colonne d'indice  
columnIndex
```

```
Object getValueAt(int rowIndex, int  
columnIndex) retourne l'objet de la cellule  
(rowIndex, columnIndex)
```

```
boolean isCellEditable(int rowIndex, int  
columnIndex) retourne true si la cellule  
indiquée est éditable par l'utilisateur.
```

```
void setValueAt(Object aValue, int  
rowIndex, int columnIndex) positionne  
l'objet de la cellule (rowIndex, columnIndex)
```

# La classe DefaultTableModel

propose plusieurs constructeurs pour créer un modèle de table.

```
DefaultTableModel()  
DefaultTableModel(int numRows, int  
numColumns)  
DefaultTableModel(Object[][] data,  
Object[] columnNames)  
DefaultTableModel(Vector columnNames,  
int numRows)  
DefaultTableModel(Vector data, Vector  
columnNames)
```

...

## Construction d'une table revisitée

```
public class MonTableModel extends DefaultTableModel {  
    public MonTableModel() {  
    }  
    public MonTableModel(Object [][] data, Object [] headings) {  
        super(data, headings);  
    }  
    public MonTableModel(Vector [][] data, Vector [] headings) {  
        super(data, headings);  
    }  
}
```

puis

```
Object [][] donnees = {  
    {"Pierre", Boolean.FALSE},  
    {"Paul", Boolean.FALSE},  
    {"Jacques", Boolean.TRUE}  
};  
String [] enTete = {  
    "Nom", "Est Inscrit ?"  
};  
  
MonTableModel dtm = new MonTableModel(donnees,  
enTete);  
JTable jt = new JTable(dtm);  
JScrollPane jsp = new JScrollPane(jt);
```

# Mode d'affichage des cellules

Par défaut les cellules sont des `JLabel` qui affichent le texte provenant de la méthode `toString()` de l'objet de la cellule.

Si l'objet est de classe `Boolean`, la cellule est une `JCheckBox`.

On peut changer ce comportement en définissant son propre `CellRenderer`.



# Construire son renderer

Une `JTable` possède un `TableCellRenderer` utilisé pour toute la `JTable` sauf si ce renderer est redéfini pour une colonne.

Pour construire un renderer pour une `JTable`, on définit une classe qui implémente l'interface `TableCellRenderer` et on associe un objet de cette classe à la `JTable` à l'aide de la méthode `public void setDefaultRenderer(Class columnClass, TableCellRenderer renderer)`.

L'interface `TableCellRenderer` contient la seule déclaration :

```
public Component  
getTableCellRendererComponent(JTable  
table, Object value, boolean  
isSelected, boolean hasFocus, int row,  
int column) qui doit retourner un  
Component.
```

`table` est la `JTable` qui appelle son renderer.

`value` est la valeur initiale de la cellule à dessiner.

`isSelected` vaut `true` si le dessin doit être fait seulement lorsque la cellule est sélectionnée.

`row` est la ligne de cette cellule (-1) si c'est un en tête.

`column` est la colonne de la cellule.

# Exemple de renderer

Le renderer est fixé pour une colonne donnée. On récupère une colonne à partir du `TableColumn` lui même récupéré à partir du `TableColumnModel` de la `JTable`.

```
|| JTable jt = new JTable(...);  
|| TableColumnModel tcm = jt.getColumnModel();  
|| TableColumn tc = tcm.getColumn(2);  
|| ColorRenderer cr = new ColorRenderer();  
|| tc.setCellRenderer(cr);  
||
```

alors que le renderer est :

```
|| public class ColorRenderer implements TableCellRenderer {  
||     private JLabel composant= new JLabel("");  
  
||     public ColorRenderer()  
||     {  
||         // Les JLabels sont transparents par default.  
||         // il faut donc les rendre opaque  
||         composant.setOpaque(true);  
||     }  
  
||     /**  
||     * La méthode de construction d'un copmposant  
||     * i.e. le travail d'un renderer (factory).  
||     * Elle retourne un Component qui est l'affichage de la cellule  
||     * Ici c'est un JLabel avec la couleur appropriée.  
||     */  
||     public Component getTableCellRendererComponent(JTable  
|| t,  
||     Object value, boolean selected, boolean focussed, int r, int c)  
||     {  
||         if (value instanceof Color) {  
||             composant.setBackground((Color)value);  
||         }  
||         return composant;  
||     }  
|| }  
||
```

# Le composant JTree

modélise une arborescence.

utilise l'interface `TreeModel` et la classe `DefaultTreeModel` (il n'y a pas de `AbstractTreeModel`).

## Construire un arbre

On définit tout d'abord les nœuds comme instance de la classe

`DefaultMutableTreeNode`. En général on

utilise le constructeur :

```
public DefaultMutableTreeNode(Object  
userObject, boolean allowsChildren) qui  
crée un nœud sans parent et sans enfant,  
initialisé avec l'objet indiqué. Si
```

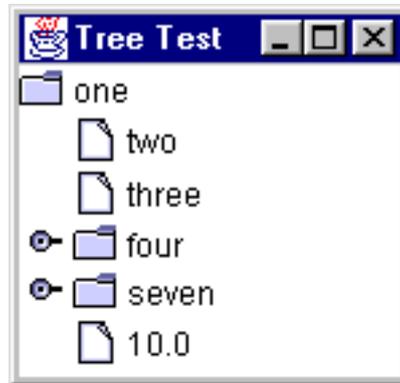
`allowsChildren` vaut `false` ce nœud restera une feuille.

Par la suite, lorsque tous les nœuds ont été créés comme objet `DefaultMutableTreeNode`, ils sont assemblés pour construire un arbre à l'aide de la méthode `add()` de

`DefaultMutableTreeNode` et l'arbre final est construit à l'aide de `JTree(TreeNode root)`.

# Exemple de JTree

Pour faire afficher :



il suffit d'écrire :

```
import java.awt.*;
import javax.swing.*;
import javax.swing.tree.*;

public class TreeTest {
    private static Object [] nodeNames = { "one",
"two", "three", "four", "five", "six", "seven",
new Integer(8), new Integer(9), new Float(10) };

    private static boolean [] leaf = { false, true, true,
false, true, true, false, true, true, true };

    public static void main(String args[]) {
        JFrame jf = new JFrame("Tree Test");
        DefaultMutableTreeNode [] nodes = new
DefaultMutableTreeNode[10];
        for (int i = 0; i < nodes.length; i++) {
            nodes[i] = new
DefaultMutableTreeNode(nodeNames[i], !leaf[i]);
        }
    }
}
```

```
nodes[0].add(nodes[1]);
nodes[0].add(nodes[2]);
nodes[0].add(nodes[3]);
nodes[0].add(nodes[6]);
nodes[0].add(nodes[9]);
nodes[3].add(nodes[4]);
nodes[3].add(nodes[5]);
nodes[6].add(nodes[7]);
nodes[6].add(nodes[8]);
JTree jt = new JTree(nodes[0]);
jf.getContentPane().add(jt,
BorderLayout.CENTER);
jf.pack();
jf.setVisible(true);
}
```



---

# L'interface `TreeModel`

Les principales manipulations pour arbre sont déjà déclarées dans cette interface.

On trouve :

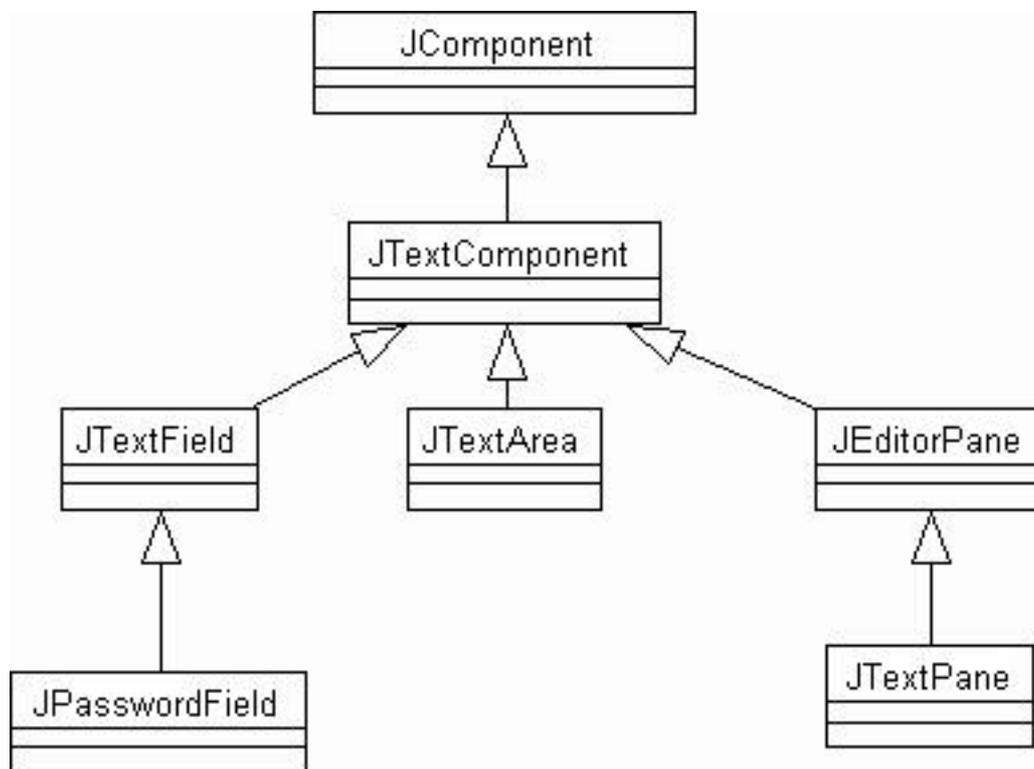
```
void  
add/removeTreeModelListener(TreeModelL  
istener l)  
Object getChild(Object parent, int  
index) retourne le indexième fils de  
parent.  
int getChildCount(Object parent)  
int getIndexOfChild(Object parent,  
Object child)  
Object getRoot()  
boolean isLeaf(Object node)
```

# Les composants Swing pour le texte

Swing propose 5 classes pour traiter le texte. Ces classes dérivent de la classe `JTextComponent` (qui dérive de `JComponent`).

Deux de ces classes remplacent les équivalents AWT : `JTextField` et `JTextArea`. Swing propose une classe pour des entrées « masquées » :

`JPasswordField`. Dans ces 3 classes, l'affichage du texte est constitué d'une seule police et d'une seule couleur.



---

# Les composants Swing pour le texte

Il existe deux autres classes pour le traitement avancé de « texte ».

`JEditorPane` est un traitement de texte pouvant afficher du texte formaté avec de nombreux attributs (différentes couleurs, polices, souligné, gras, ...).

`JTextPane` permet de définir sa propre grammaire de documents et ses styles associés alors que `JTextEditor` est spécialisé pour les textes de types MIME `text/plain`, `text/html`, `text/rtf`.

Dans ces deux classes, les données peuvent être des `Component Swing` ou des images.

# Le composant JTextPane

Ce composant permet d'éditer du texte formaté : différentes polices, styles, couleurs, .... Un tel texte est appelé un document.

Plus précisément un JTextPane peut contenir des images (Icon) ou des Component Swing.

La plupart des classes et interfaces utilisées se trouve dans le paquetage `javax.swing.text`.

Cette classe dérive de JEditorPane qui contient des fonctionnalités avancées : affichage de texte HTML ou rtf.

Le modèle associé à un JTextPane est l'interface StyledDocument (i.e. document contenant des styles) implanté par la classe concrète DefaultStyledDocument.

Un JTextPane peut contenir des Component.

# Les styles

Un style est un ensemble d'attributs à appliquer à une partie d'un document.

L'ensemble des styles d'un document est un `StyleContext`.

Les styles sont hiérarchisés et un style sur une suite de caractères écrase le style sur le paragraphe où se trouve ces caractères.

La hiérarchie des interfaces pour la manipulation des styles est : `AttributeSet` -> `MutableAttributeSet` -> `Style`.

Un style est créé grâce à la méthode `addStyle()` de la classe `StyleContext`.

Enfin `StyleContext` est à la fois une machine (factory) et un conteneur de styles.

Le contenu des styles est construit à l'aide des méthodes statiques (`setBold()`, `setFontFamily()`, ...) de la classe `StyleConstants`.

```
StyleContext sc = new StyleContext();  
  
// on cree un style de base  
Style normal = sc.addStyle("Normal", null);  
  
// un style dérivé du style de base  
Style titre = sc.addStyle("Titre", normal);  
  
// On définit le contenu de chaque style :  
StyleConstants.setFontSize(titre, 32);  
StyleConstants.setBold(titre, true);
```

# Éditeur Swing qui affiche de l'HTML

```
import javax.swing.event.*;
import java.awt.*;
import java.io.*;

public class EditorPaneSample {
    public static void main(String args[]) throws
    IOException {
        JFrame frame = new
    JFrame("EditorPane HTML");
        Container content =
    frame.getContentPane();

        JEditorPane editor = new
    JEditorPane("file:///JeanMarc/Java/index.html");
        editor.setEditable(false);

        JScrollPane scrollPane = new
    JScrollPane(editor);
        content.add(scrollPane);

        frame.setSize(640, 480);
        frame.setVisible(true);
    }
}
```

---

# Le Copier/Couper/Coller

En fait cela ne fait pas parti de Swing mais des JFC.

Java permet de manipuler le presse papier (Clipboard) de la machine et de faire ainsi des "Copier/Couper/Coller" entre une application quelconque et un programme Java à condition que la donnée transférée "convienne" pour les 2 programmes.

Le "type" de cette donnée à transférer est un ensemble de type MIME et est représenté par des objets de la classe

`java.awt.datatransfer.DataFlavor.`

# Déposer dans le presse papier

Il suffit :

- de récupérer le presse papier par :

```
Clipboard c =  
getToolkit().getSystemClipboard();
```

lancer sur un Component.

- créer un objet *obj* d'une classe implémentant l'interface *Transferable* : ce sera la donnée à déposer dans le presse-papier.

- déposer la donnée dans le presse papier par :

```
c.setContents(obj, proprietairePP).
```

*proprietairePP* est désormais le propriétaire du presse papier. C'est un objet d'une classe qui implémente l'interface *ClipboardOwner*. Cet objet sera informé que le presse papier a changé de propriétaire (i.e. une autre application a déposé une donnée dans le presse-papier) par le lancement de la méthode

```
public void lostOwnership(Clipboard c,  
Transferable t) qui est déclarée dans  
l'interface ClipboardOwner.
```

# Récupérer du presse papier

Il suffit :

- de récupérer le presse papier par :

```
Clipboard c =  
getToolkit().getSystemClipboard();
```

lancer sur un Component.

- de récupérer la donnée du presse papier  
par:

```
Transferable t = c.getContents(this);
```

(le paramètre de cette méthode n'est pas  
utilisé pour l'instant).

- récupérer les différents types possibles  
(DataFlavor) pour la donnée du presse  
papier par `t.getTransferDataFlavors()`.

On obtient un tableau et de ce tableau, on  
peut avoir la donnée pour chaque

DataFlavor par :

```
Object o =
```

```
t.getTransferData(flavors[i]).
```

# Programme

## Copier/Couper/Coller

```
import java.awt.*;
import java.awt.event.*;
import java.awt.datatransfer.*;
import java.io.*;
import javax.swing.*;

public class Clip extends JPanel implements
ClipboardOwner {
    JTextArea text = new JTextArea();
    JButton cutButton = new JButton("Cut");
    JButton copyButton = new JButton("Copy");
    JButton pasteButton = new JButton("Paste");

    public Clip() {
        setLayout(new BorderLayout());
        JScrollPane jsp = new JScrollPane(text);
        add(jsp, BorderLayout.CENTER);
        add(cutButton, BorderLayout.NORTH);
        add(copyButton, BorderLayout.EAST);
        add(pasteButton, BorderLayout.SOUTH);
    }
}
```

```
pasteButton.addActionListener(new
ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Paste!");
        Clipboard c =
getToolkit().getSystemClipboard();
        Transferable t = c.getContents(this);
        DataFlavor [] flavors =
t.getTransferDataFlavors();
        for (int i = 0; i < flavors.length; i++) {
            try {
                Object o = t.getTransferData(flavors[i]);
                System.out.println("Flavor " + i + " gives "
+ o.getClass().getName());
                if (o instanceof String) {
                    text.insert((String)o,
text.getCaretPosition());
                }
            } catch (Exception ex) { ex.printStackTrace();
            }
        }
    }
});

cutButton.addActionListener(new
ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Cut!");
        Clipboard c =
getToolkit().getSystemClipboard();
        StringSelection ss = new
StringSelection(text.getSelectedText());
        text.replaceRange("",
text.getSelectionStart(), text.getSelectionEnd());
        c.setContents(ss, Clip.this);
    }
});
```

```
        copyButton.addActionListener(new
ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Copy!");
        Clipboard c =
getToolkit().getSystemClipboard();
        StringSelection ss = new
StringSelection(text.getSelectedText());
        c.setContents(ss, Clip.this);
    }
});
}

public void lostOwnership(Clipboard c,
Transferable t) {
    System.out.println("Lost clipboard");
}

public static void main(String args[]) {
    JFrame f = new JFrame("Clipboard Test");
    Clip c = new Clip();
    f.getContentPane().add(c,
BorderLayout.CENTER);
    f.setSize(300, 200);
    f.setVisible(true);
}
}
```

---

# Bibliographie

John Zukowski's Definition Guide to Swing  
for Java 2 ; John Zukowski ed Apress ISBN  
1-893115-02-X

Tutorial Swing en ligne à :

<http://java.sun.com/docs/books/tutorial/ui/swing/index.html>

