IUT Lyon 1

l'excellence technologique

Programmation d'IHM- Cours 3
Architecture MVC, JList

V. DESLANDRES

veronique.deslandres@univ-lyon1.fr

Sommaire

Architecture MVC
Modèle, Vue, Contrôle

Introduction

- Pour visualiser et manipuler un gros volume d'informations : composants spécifiques
- Les informations peuvent être présentées sous forme de **tableau**, de **liste**, **d'arbre** ou de **graphe**
- L'API Java Swing propose plusieurs composants pour visualiser les informations :
  - *JTable, JList, JTree, JGraph,…*

MVC: Principes de base

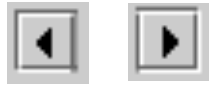
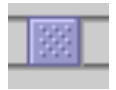
- Le modèle **d'architecture** MVC (Model View Controller) est à la base de nombreux systèmes de visualisation graphiques
- Principe de Base: **séparation des rôles**
  - Le ***modèle*** est l'élément principal du composant, il contient les **données**
  - Les ***vues*** du composant sont **des visualisations des données d**u modèle : une vue s'abonne à un modèle, et se met à jour quand les données du modèle évoluent
  - Le ***contrôleur*** assure la synchronisation entre modèle et vues (**traitement**)
- La Java Swing repose sur l'architecture M-VC
  - (càd que Vue et Contrôleur sont souvent dans le même composant graphique, séparés du Modèle)

MVC exemple (JSlider)

Quelles sont les données associées à un *slider* ?

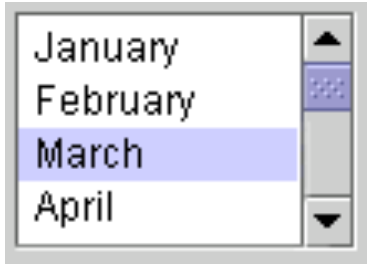
- *Modèle* :
  - valeur minimale = 0
  - valeur courante = 15
  - valeur maximale = 100

- *Vue* :

- *Contrôleur* :
  - Traiter les clics de souris sur les boutons terminaux 
  - Gérer les *drags* de souris sur l'ascenseur 

Modèles MVC des composants SWING

- Il existe en SWING des composants génériques pour les **modèles** des données
- `JList` :
  - classe **`ListModel`** pour les données
  - classe **`ListSelectionModel`** pour gérer les sélections

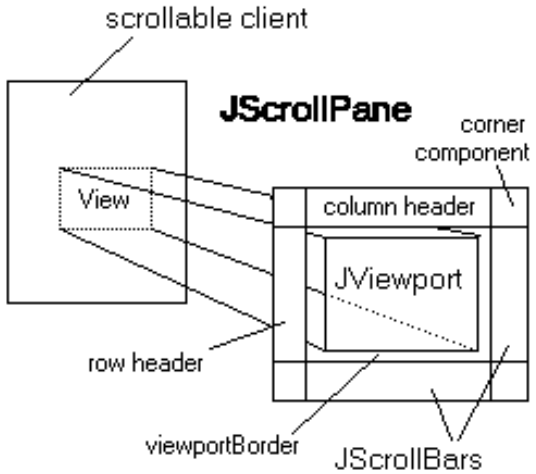


- `JTable` :
  - classe **`TableModel`** pour les données
  - classe **`TableColumnModel`** pour définir les colonnes
  - classe **`ListSelectionModel`** pour gérer les sélections

Host	User	Password	Last Modified
Biocca Games	Freddy	!#asf6Awwzb	Mar 16, 2006
zabble	ichabod	Tazb!34$fZ	Mar 6, 2006
Sun Developer	fraz@hotmail.com	AasW541!fbZ	Feb 22, 2006
Heirloom Seeds	shams@gmail.com	bkz[ADF78!	Jul 29, 2005
Pacific Zoo Shop	seal@hotmail.com	vbAf124%z	Feb 22, 2006

Le composant `JScrollPane`

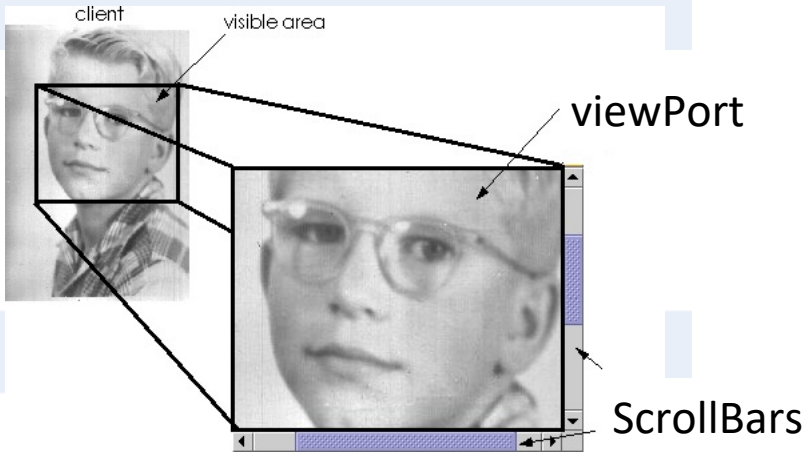
Barres de défilement horizontale et verticale, quand besoin



Fenêtre avec des ascenseurs : `JScrollPane`

- On définit une instance de la classe `JScrollPane` à laquelle on envoie le composant à afficher, ici une liste :

```
JScrollPane jsp = new JScrollPane(liste);
```



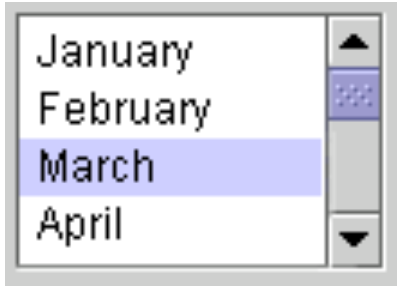
- Ou bien, en 2 temps :

```
JScrollPane jsp = new JScrollPane();
jsp.getViewport().setView(liste);
```

- C'est le `JScrollPane` qu'on ajoute au panneau de la fenêtre :

```
getContentPane().add(jsp);
```

Le composant JList



Caractéristiques de base

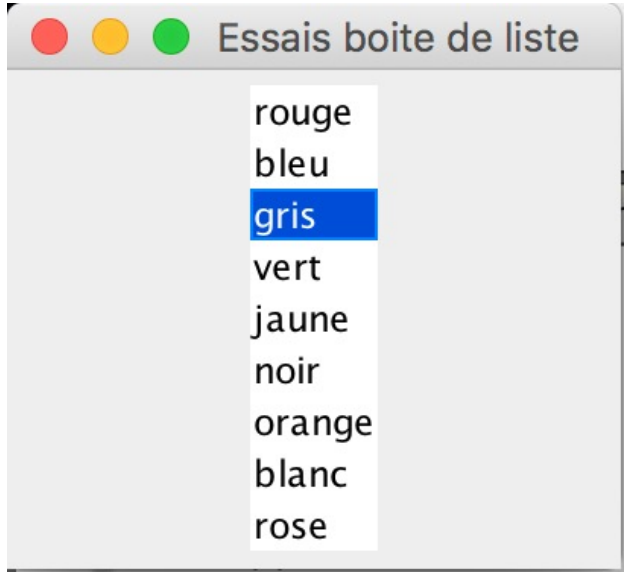
- Une **JList** est une présentation des données sous forme de liste
  - Affichage d'une liste d'items :
- 2 types de **JList**
  - Liste **statique** : sélectionner des éléments
  - Liste **dynamique** : la liste des items peut évoluer
- Modalité de sélection : **simple** ou **multiple**

Si une **seule valeur** doit être sélectionnée, **ComboBox** préférable

Initialisation d'une liste

- Constructeur simple (avec modèle implicite)

```
String[] couleurs = {"rouge", "bleu", "gris", "vert",
                     "jaune", "noir","orange","blanc","rose" } ;
JList liste = new JList(couleurs) ;
```



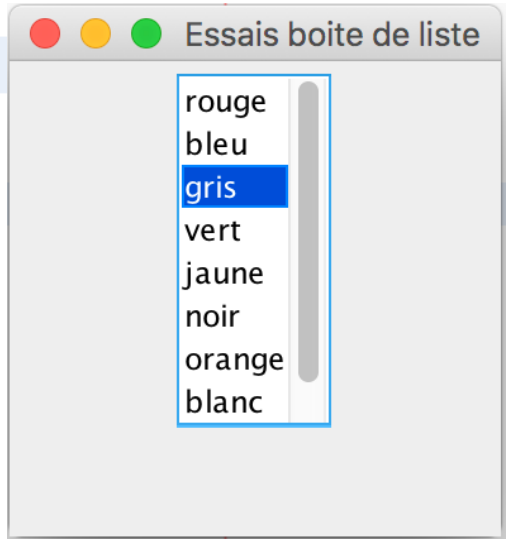
- Définir une pré-sélection d'un élément
  - Les indexes démarrent à 0
  - Exemple: sélection de l'élément de rang 2

```
liste.setSelectedIndex(2);
```

Affichage d'une liste

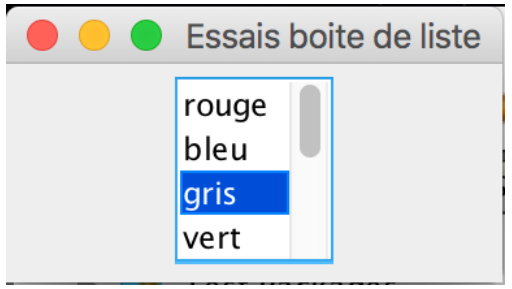
- Ajout d'une barre de défilement à une liste
  - Par défaut, la liste affichera **8 valeurs** avec une présentation verticale
  - La barre de défilement **n'apparaît pas** si la liste comporte moins de valeurs

```
JScrollPane jsp = new JScrollPane(liste);
getContentPane().add(jsp); // ajouter le jsp au content pane
```



- Choisir le **nombre d'items** à afficher avec barre de défilement
  - Exemple: afficher seulement 4 valeurs à la fois

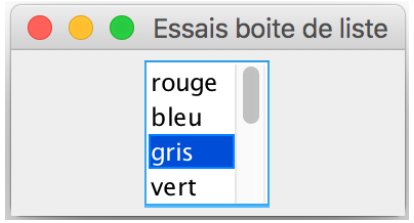
```
liste.setVisibleRowCount(4);
```



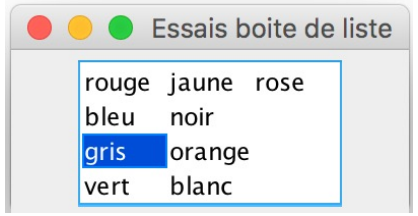
Mode d'affichage des items d'une liste

- Méthode: `liste.setLayoutOrientation(orientation);`
- 3 modes: VERTICAL VERTICAL_WRAP HORIZONTAL_WRAP

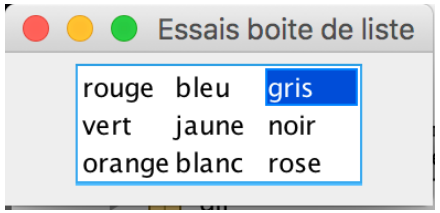
```
liste.setLayoutOrientation(JList.VERTICAL);
```
OU `liste.setLayoutOrientation(0);` (par défaut)




```
liste.setLayoutOrientation(JList.VERTICAL_WRAP);
```
OU `liste.setLayoutOrientation(1);`

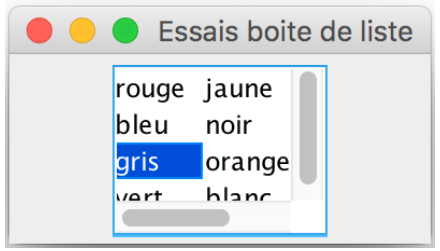



```
liste.setLayoutOrientation(JList.HORIZONTAL_WRAP);
```
OU `liste.setLayoutOrientation(2);`

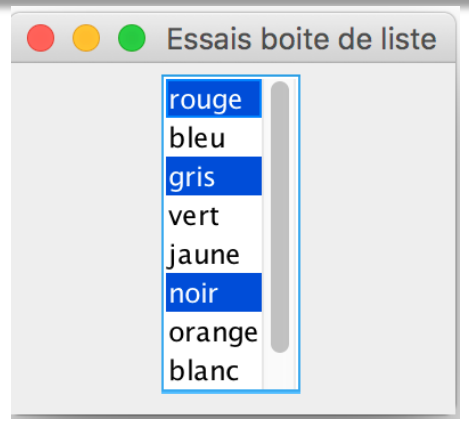


- Afficher avec une dimension et des barres de défilement

```
jsp.setPreferredSize(new Dimension(100, 80));
```



Modes de sélection des items d'une liste



- Méthode: `liste.setSelectionMode(mode);`
- 3 modes: SINGLE_SELECTION SINGLE_INTERVAL_SELECTION MULTIPLE_INTERVAL_SELECTION

OU
```
liste.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
liste.setSelectionMode(0);
```

OU
```
liste.setSelectionMode(ListSelectionModel.SINGLE_INTERVAL_SELECTION);
liste.setSelectionMode(1);
```

OU
```
liste.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
liste.setSelectionMode(2);
```
(par défaut)

Accès aux éléments du modèle

- Récupérer l'élément à la position i

Les indices commencent à 0

```
//récupérer le modèle de la liste
ListModel myModel=liste.getModel();
String premierelement=myModel.getElementAt(0).toString();
System.out.println("Le premier élément est: "+premierelement);
```

- Récupérer tous les éléments d'une liste par une boucle

```
ListModel myModel=liste.getModel();
int size = myModel.getSize();
for (int i = 0 ; i < size ; i++) {
  Object elem = myModel.getElementAt(i);
  System.out.println(elem);
}
```

Trouver par la **position** → modèle

Accès aux éléments sélectionnés

- Liste à sélection simple : récupérer l'élément sélectionné
  - Méthode: `Object valeur=liste.getSelectedValue();`

```
String ch = (String) liste.getSelectedValue();
System.out.println("Action Liste - La valeur sélectionnée: "+ch) ;
```

- Listes à sélection multiple : récupérer tous les éléments sélectionnés
  - Méthode: `List<type> valeurs = liste.getSelectedValuesList();`

```
System.out.println("Action Liste - Les valeurs selectionnees :");
List<String> valeurs = liste.getSelectedValuesList();
for (int i = 0; i<valeurs.size(); i++)
        System.out.println(valeurs.get(i)) ;
```

Trouver par la **sélection** → liste

Accès aux positions des items sélectionnés de la liste

- Liste à sélection simple : récupérer la position de la 1ère valeur sélectionnée par l'utilisateur
  - Méthode: **`public int getSelectedIndex();`**

```
int index = liste.getSelectedIndex();
System.out.println("Action Liste - Index de la valeur sélectionnée: "+index) ;
```

- Listes à sélection multiple : récupérer les positions de toutes les valeurs sélectionnées
  - Méthode: **`public int[] getSelectedIndices();`**

```
System.out.println("Action Liste - Les indexes des valeurs selectionnees :");
int[] indexes = liste.getSelectedIndices();
for (int i = 0; i<indexes.length; i++)
        System.out.println(indexes[i]) ;
```

Evénements générés

Une liste ne génère pas d'événement de type `ActionEvent`

- Les événements générés par une liste sont des **événements de sélection**
  - de type : `ListSelectionEvent`
- Implémentation de l'interface: `ListSelectionListener`
- L'interface ne comporte qu'une seule méthode :

```
public void valueChanged(ListSelectionEvent e)
```

Méthodes de `ListSelectionEvent`

- `Object getSource()`: objet source de l'événement (héritée de *EventObject*)
- `int getFirstIndex()`: index du 1er item dont la valeur de sélection a changé
- `int getLastIndex()`: index du dernier item dont la valeur de sélection a changé
- boolean `getValueIsAdjusting()` →

Spécificité des événements générés par une JList

- `ListSelectionEvent` est généré :
  - Lors de l'appui sur le bouton de la souris
  - Lors du relâchement du bouton

- Les traitements **pourraient ainsi être générés 2 fois**
- Pour pallier cette redondance, il existe la méthode :

```
public boolean getValueIsAdjusting();
```

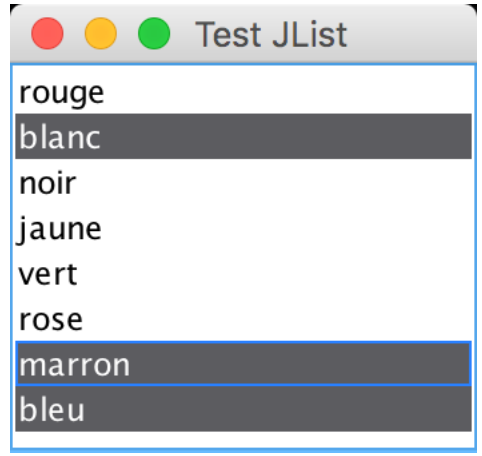
```
public void valueChanged (ListSelectionEvent e) {
    if ( !e.getValueIsAdjusting()) {
        // accès aux informations sélectionnées et traitement
    }
}
```

```
public class JListStatiqueTest extends JFrame implements ListSelectionListener {

    private JList liste ;
    static final String[] couleurs = {"rouge", "blanc", "noir", "jaune", "vert",
        "rose", "marron", "bleu"};


    public JListStatiqueTest(String t) {
        super (t);

        // definition d'une liste
        liste = new JList(couleurs);
        liste.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
        this.getContentPane().setLayout(new BorderLayout());

        liste.addListSelectionListener(this);
        JScrollPane panneau = new JScrollPane(liste);
        liste.setSelectedIndex(1);

        this.getContentPane().add(panneau, BorderLayout.CENTER);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
```

La fenêtre est son propre écouteur

Ajouter un écouteur à la liste

Exemple de JList (statique) avec gestion des événements



Exemple JList statique (suite)

```
@Override
public void valueChanged(ListSelectionEvent e) {

    if (e.getValueIsAdjusting())
        System.out.println("=> "+e.getSource());
    else {
        // affiche les items sélectionnés
    System.out.println("**** affichage des éléments sélectionnés :");
    List<String> valeurs;
        valeurs = liste.getSelectedValuesList();
    for (int i = 0; i < valeurs.size(); i++)
        System.out.println(valeurs.get(i));
    }


}
```

```
public static void main (String[] args) {

    JListTest f1 = new JListTest("Test JList");
    f1.setSize(200, 300);
    f1.setVisible(true);
}
```

```
run:
**** affichage des éléments sélectionnés :
blanc
=> javax.swing.JList[,
0,0,171x142,alignmentX=0.0,alignmentY=0.0,border=,flags=50332008,maximumSize=,minimumSize=
,preferredSize=,fixedCellHeight=-1,fixedCellWidth=-1,horizontalScrollIncrement=-1,selectio
nBackground=com.apple.laf.AquaImageFactory$SystemColorProxy[r=92,g=92,b=96],selectionForeg
round=com.apple.laf.AquaImageFactory$SystemColorProxy[r=255,g=255,b=255],visibleRowCount=8
,layoutOrientation=0]
**** affichage des éléments sélectionnés :
blanc
bleu
```

Listes dynamiques (modifiables)

- Quand on crée la liste en lui envoyant un vecteur d'objets, Java crée implicitement un **`DefaultListModel`** mais il est **non modifiable :**
  - On ne peut ni **ajouter**, ni **supprimer** les items de la liste

- Quand on veut pouvoir modifier les items de la liste :
  - Il faut créer le modèle **explicitement** avec **`DefaultListModel`**

Listes dynamiques

- Création du modèle:

```
DefaultListModel monModele = new DefaultListModel();
```

- Création de la liste:

```
JList liste = new JList(monModele);
```

- Ajout d'un élément **à la fin** de la liste :

```
monModele.addElement(element);
```

- Ajout d'un élément **à la position i** :

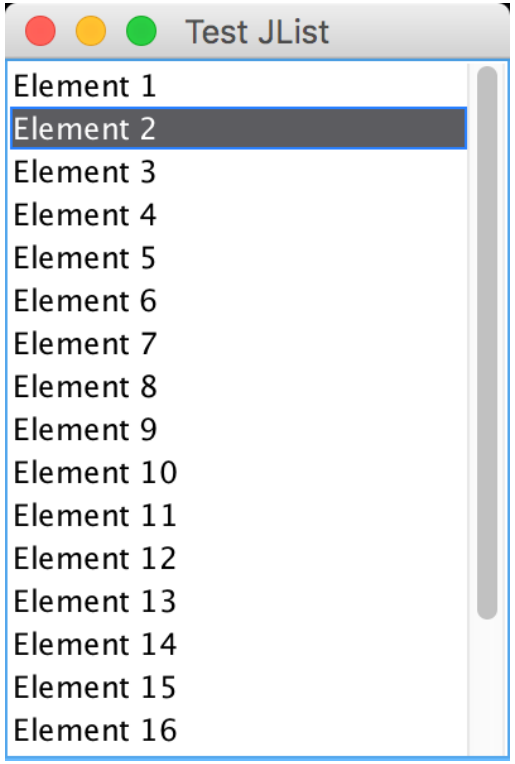
```
monModele.add(i,element);
```

- Supprimer un élément (la 1ère occurrence de l'élément) :

```
monModele.removeElement(element);
```

- Supprimer l'élément à la position i:

```
monModele.remove(i);
```



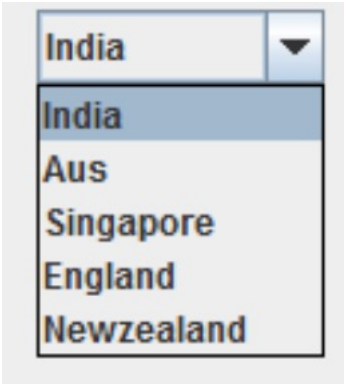

```
lm = new DefaultListModel();
// definition d'une liste
for (int i=0; i<20; i++) {
        lm.addElement("Element "+(i+1));
}
liste = new JList(lm);
```

Exemple avec création de modèle

```
static JList liste;
static DefaultListModel monModele;
TestJListModel(){
    //Utilisation d'un modèle des données (par défaut)
    monModele = new DefaultListModel();
    //Construction de la liste
    monModele.addElement("rouge");
    monModele.addElement("gris");
    monModele.addElement("bleu");
    monModele.addElement("bleu");

    liste = new JList(monModele);

    Container contenu = getContentPane();
    contenu.setLayout(new FlowLayout());

    JScrollPane jsp = new JScrollPane(liste);
    contenu.add(jsp);
}
```

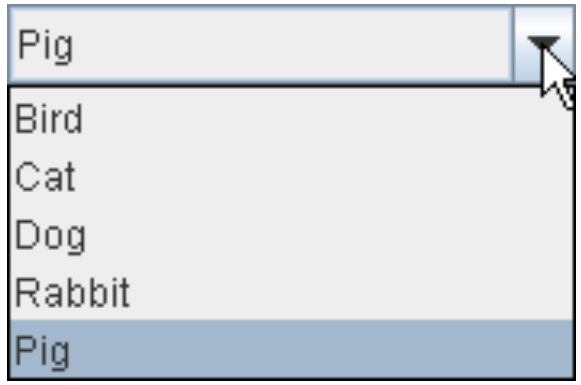
```
//Ajout d'un élément à une position donnée
monModele.add(1,"jaune");
//Ajout d'un élément à la fin de la liste
monModele.addElement("rose");
//Suppression d'un élément d'un index donné
monModele.remove(0);
//Supprimer l'élément sélectionné
int index = liste.getSelectedIndex();
monModele.remove(index);
//Suppression d'un item donné
//Supprimer la 1ère occurrence de « bleu »  (boolean)
//Retourne vrai si « bleu » était un item de la liste, faux sinon
monModele.removeElement("bleu");
//Pour supprimer toutes les occurrences :
boolean  suppr = false;
do
    suppr = monModele.removeElement("bleu");
while (suppr);
```

ComboBox



JComboBox : les caractéristiques de base

- **Une ComboBox** est un composant très souvent utilisé : il permet d'éviter les erreurs de saisie
  - Par rapport à un `JTextField` par ex.
- Il permet aux utilisateurs de **choisir une des options** proposées.
- Lorsque l'utilisateur clique sur la **ComboBox**, une liste d'options à sélectionner apparaît et il choisit un item.

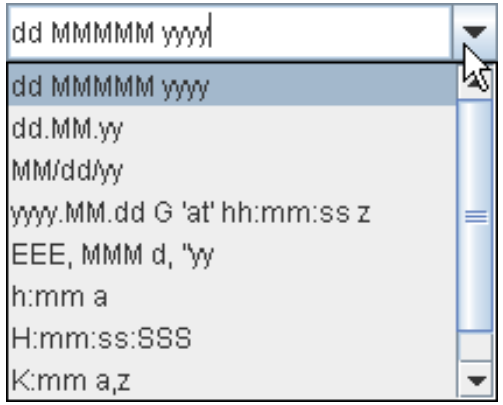


4 constructeurs :

- **JComboBox()** crée une ComboBox avec un modèle par défaut pour les items.
- **JComboBox(ComboBoxModel<E> unModele**) Crée une ComboBox avec les items fournis par le ComboBoxModel.
- **JComboBox(E[] *tabItems*)** Crée une ComboBox qui contient les éléments du tableau *tabItems*
- **JComboBox(Vector<E> *vectorItems*)** Crée une ComboBox qui contient les éléments du Vector *vectorItem*

JComboBox : les caractéristiques avancées

- Une JComboBox peut être **éditable,** comme ici :
  - ou **non modifiable**



- Pour gérer les actions de l'utilisateur sur une JComboBox :
  - les interfaces **ActionListener, ChangeListener** ou **ItemListener** peuvent être utilisées
- Une méthode **getSelectedItem()** permet de récupérer l'élément sélectionné
- Une méthode **setEditable()** peut être utilisée pour activer ou désactiver la partie saisie du texte
  - (sans que ça permette de trouver l'item qui s'en rapproche*)
  - Permet d'ajouter des items à la liste

* *Pour faire de l'auto-complétion avec une Combox, cf* http://www.orbital-computer.de/JComboBox/

JComboBox : caractéristiques avancées (2)

- Souvent on utilise un **ComboBoxModel** pour stocker les données à afficher de la comboBox
  - Cela permet de charger dynamiquement la liste des éléments de la ComboBox
    - Par ex. quand une comboBox dépend d'une **autre variable** (Master / Detail)
- (Pour aller plus loin : on peut aussi préférer un **MutableComboBoxModel**
  - Quand on veut pouvoir modifier dynamiquement la liste des items : ajouter, modifier, supprimer des éléments, etc.)

- On peut récupérer la **partie éditeur de la ComboBox** avec :

```
JTextComponent editor = (JTextComponent) comboBox.getEditor().getEditorComponent();
```

Et ainsi placer des écouteurs sur le texte, par ex. pour permettre l'auto-complétion.

Pour aller plus loin…

ComboBox vs. List, modifier le bord, imposer une taille d'affichage

JComboBox

JList

Caractéristiques de la JList

- Avec une **`JList`**, l'utilisateur peut choisir entre une **sélection unique ou multiple** des éléments de la liste :
  - **C'est la principale différence avec la ComboBox**

- Une **`JList`** ayant moins de 8 éléments ne possède pas de barre de défilement. Il faut passer par un **`JScrollPane`** si on souhaite en avoir ;
- La méthode **getSelectedIndex()** renvoie l'index du 1er élément sélectionné ou -1 si aucun élément n'est sélectionné et la méthode **getSelectedIndexes()** renvoie un tableau avec l'index de chaque élément sélectionné. Le tableau est vide si aucun élément n'est sélectionné ;
- La méthode **getSelectedValue()** renvoie le 1er élément sélectionné ou *null* si aucun élément n'est sélectionné ;

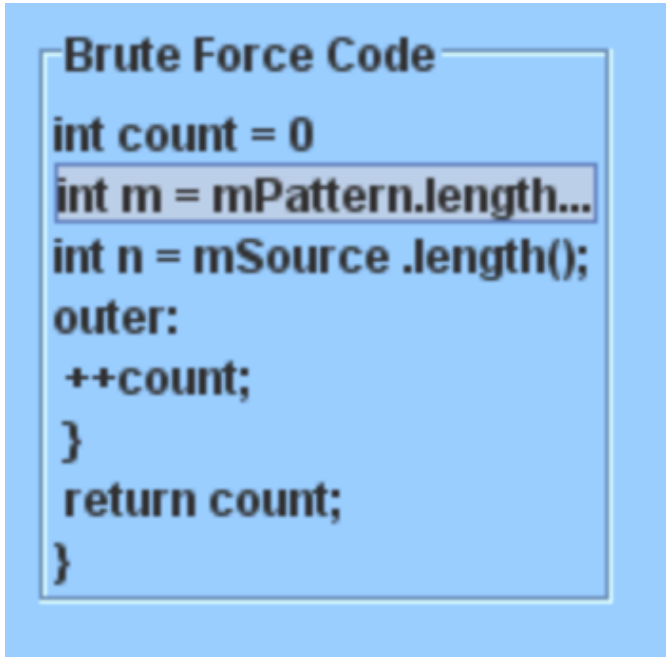
- Une classe **DefaultListModel** fournit une implémentation simple d'un modèle de liste, qui peut être utilisé pour gérer les éléments affichés par une **`JList`**.

Modifier le bord de la JList

Il est possible de modifier le bord d'une JList avec :

```
Border bo = BorderFactory.createEtchedBorder();
maList.setBorder(BorderFactory.createTitledBorder(bo, "Le
titre"));
```

Pour par exemple, afficher des lignes de code sélectionnable :



Taille d'affichage de la JList

- Nativement, le composant **JList** parcourt tous les éléments pour choisir *la taille à afficher* de la liste
  - Ça peut être pénalisant en cas de longue liste…
- On peut **définir soi-même** la taille d'affichage de la liste avec la méthode `setPrototypeCellValue()` :

```
JList<String> bigDataList =
                        new JList<String>(bigData);
/* On donne ici la cellule qui a la taille la plus grande. La
MVJ l'utilise pour calculer la valeur des propriétés
fixedCellWidth et fixedCellHeight de la liste */

bigDataList.setPrototypeCellValue("Index 1234567890");
```

Les énumérations

Les énumerations

- Existe depuis le JDK Java 5 (Tiger)
- Mot-clefs : **enum ou Enum**
- Les énumérations sont un **ensemble de constantes liées**
- Plus propres que les constantes du langage Java
  - Vérifie les valeurs lors de la **compilation**
- Il existe le type *enum* et des classes *Enum*
  - Une **enum** est un *type dont les champs* sont des constantes fixes
  - **Enum** est un **type de classe** qui hérite de `java.lang.Enum`

Les énumerations simples

- Ce sont des types particuliers, définissant une liste de constantes
- Pour représenter le jour de la semaine :

```
public enum JoursSemaine {
    LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE
}
```

- Utilisation :

```
JoursSemaine jour = JoursSemaine.LUNDI;
…
if (jour == JoursSemaine.SAMEDI) {
    // ...
}
```

```
switch (jour) {
    case DIMANCHE:
    break;

    case LUNDI:
    break;

    // ...
}
```

Les variables de type **enum** sont implicitement **comparables** et **sérialisables**

Récupérer une constante en String

- 2 méthodes possibles
  - `toString()` : méthode *overridable*
  - `name()` : méthode `final()` (on ne peut pas la redéfinir en cas d'héritage)

- Illustration :

```
enum Size { SMALL, MEDIUM, LARGE, EXTRALARGE }
System.out.println( "String value of SMALL is " + Size.SMALL.toString());
System.out.println( "String value of MEDIUM is also " + Size.MEDIUM.name());
```

String value of SMALL is SMALL

String value of MEDIUM is also MEDIUM

Les Classes Enumeration

- On met **enum** au lieu de **class**
- Ce sont des types **enum** transformés en classe, càd. qu'on peut leur ajouter :
  - Des méthodes
  - L'implémentation d'interfaces
- Les classes **enum** sont *final* par défaut : on ne peut pas étendre une classe **enum**.
- Quand on crée une classe **enum**, le compilateur crée des objets pour chaque constante de l'énumération.
- Les constantes sont toutes définies en *public static final* par défaut.

  ➔ Cf illustration ci-après

Classe Enumeration : un exemple

- On veut représenter les continents et leur superficie

```
public enum Continent
```

- 2 méthodes:
  - getArea() : donne la superficie
  - getCoverage() : renvoie le ratio de la Région par rapport à la superficie globale

Nom	Superficie (km ^{2})
Amérique du Nord	24.490.000
Amérique du Sud	17.840.000
Antarctique	13.720.000
Asie	43.810.000
Europe	10.400.000
Afrique	30.370.000
Océanie	9.010.000

```
public enum Continent {
    NORTHAMERICA (24490000),
    SOUTHAMERICA (17840000),
    ANTARCTICA (13720000),
    ASIA (43810000),
    EUROPE (10400000),
    AFRICA (30370000),
    OCEANIA (9010000);

    private final int area;
    private static final int TOTALAREA = 149640000;
```

La variable numérique est définie après les données : ici, c'est la superficie en m^2 du continent (un entier)

NOTE: le compilateur créer une liste des valeurs que l'on récupère avec la méthode **values()** :

```
for (Continent c: Continent.values() ){
        System.out.println("continent: " + c);
}
```

Cela affiche les valeurs dans l'ordre de leur définition.

Le constructeur de la classe énumération est ici :

```
    private Continent (int area) {
        this.area = area;
    }

    public int getArea() {
        return area;
    }

    public double getCoverage() {
        return area / (double) TOTALAREA * 100;
    }
}
```

Utilisation :

```
Continent c = Continent.EUROPE;

System.out.printf ("%s continent couvre %.2f%% de la
surface mondiale\n" , c, c.getCoverage() );
```

Affichage *: « EUROPE couvre 6,95% de la surface mondiale »*

Classe Enum implémentant une interface

```
interface Pizza {
        public void displaySize();
}
public enum Size implements Pizza {
        SMALL, MEDIUM, LARGE, EXTRALARGE;
        public void displaySize() {
                System.out.println("The size is " + this);
        }
}
class Main {
        public static void main(String[] args) {
                Size.MEDIUM.displaySize();
        }
}
```

Sortie :

The size is MEDIUM