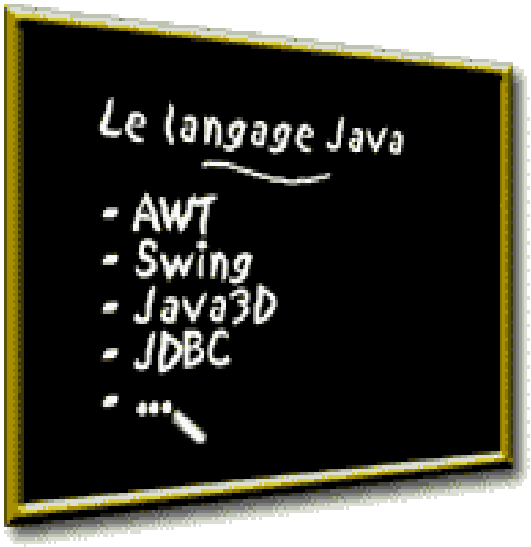
Architecture modèle-vue-contrôleur (MVC)



- Architecture logicielle (design pattern) qui permet de séparer :
  - les objets applicatifs(Model)
  - leur représentation (View)
  - les interactions qu'ils subissent (Controler)
- Utilisée pour la programmation de client/serveur, d'interface graphique
- Généralisation du pattern Observer/observable vu en algo avancé

Architecture MVC en général

- Schéma de programmation qui permet de séparer une application en 3 parties :

Côté applicatif

- Le modèle contient la logique de l'application

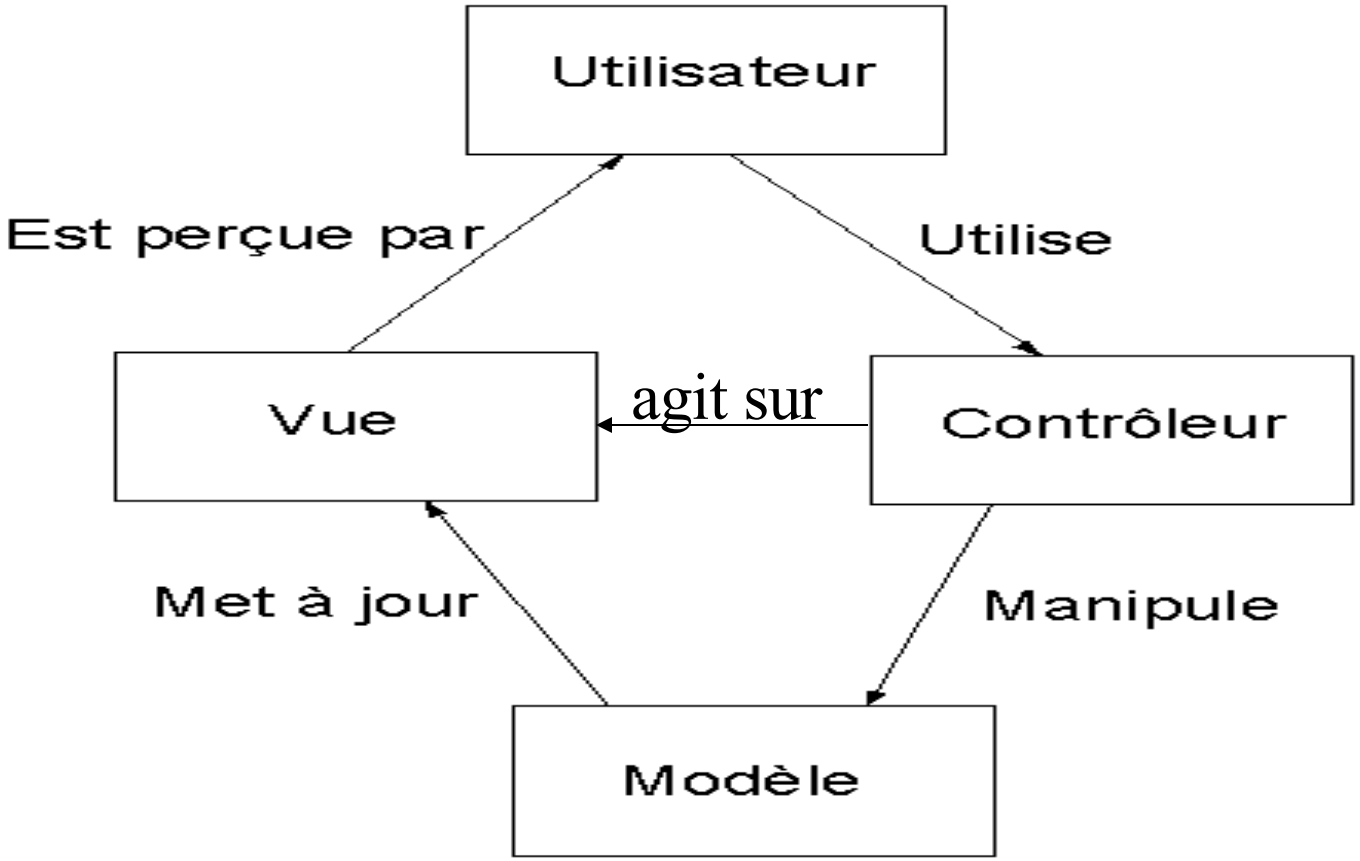
Côté visuel

- Les vues affichent à l'utilisateur des informations sur le modèle

Côté événementiel

- Le contrôleur agit sur demande de l'utilisateur et effectue les actions nécessaires sur le modèle.

Aperçu


Utilisateur
Est perçue par
Utilise
Vue
agit sur
Contrôleur
Met à jour
Manipule
Modèle

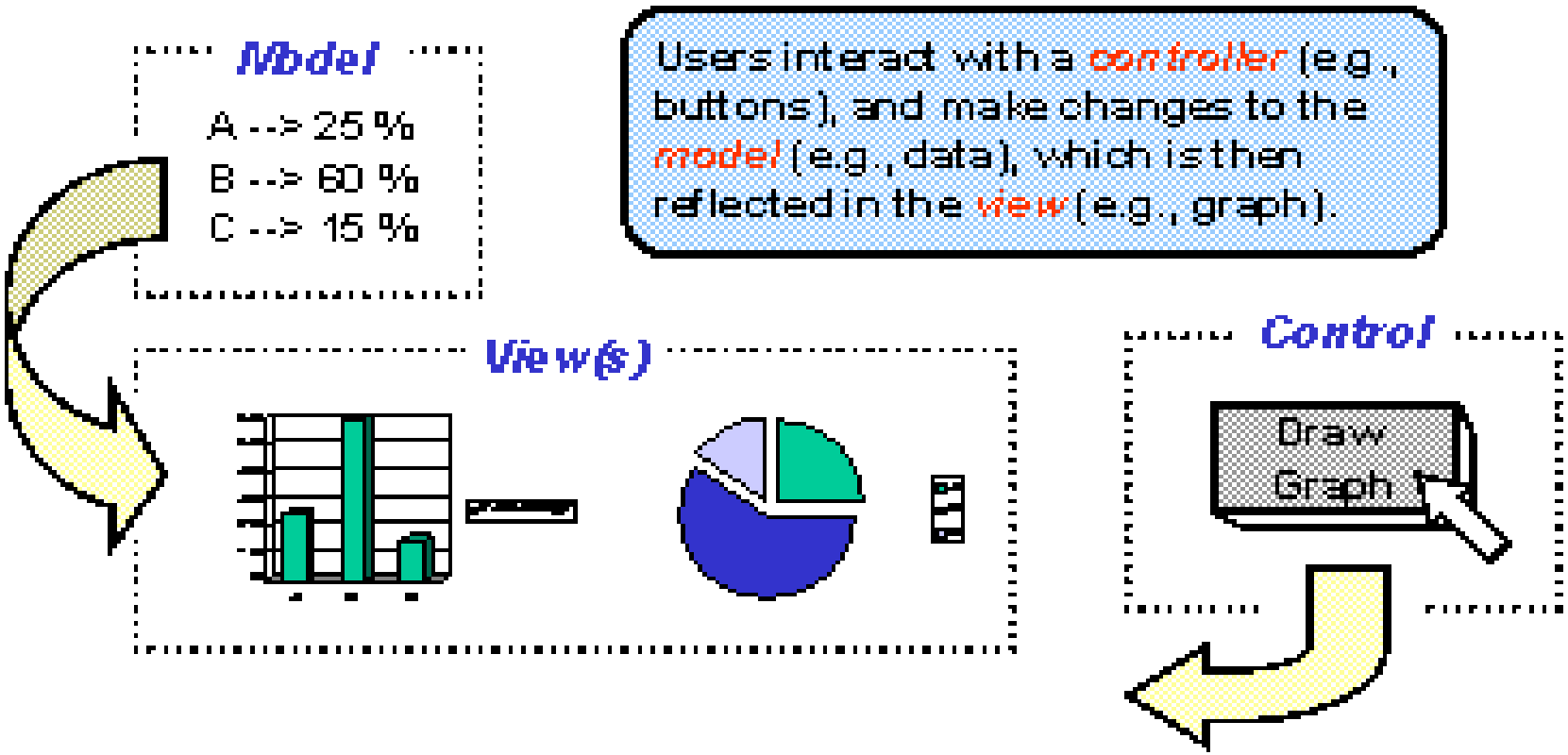
architecture MVC : exemple1

- Document texte (modèle : chaîne de caractères html) peut avoir deux vues :
  - une vue WYSIWYG (What You See is What you Get) vue par un navigateur
  - une vue brute : code html source avec balises vue par un éditeur de textes
- Toute modification sur le modèle se répercute dans les deux vues

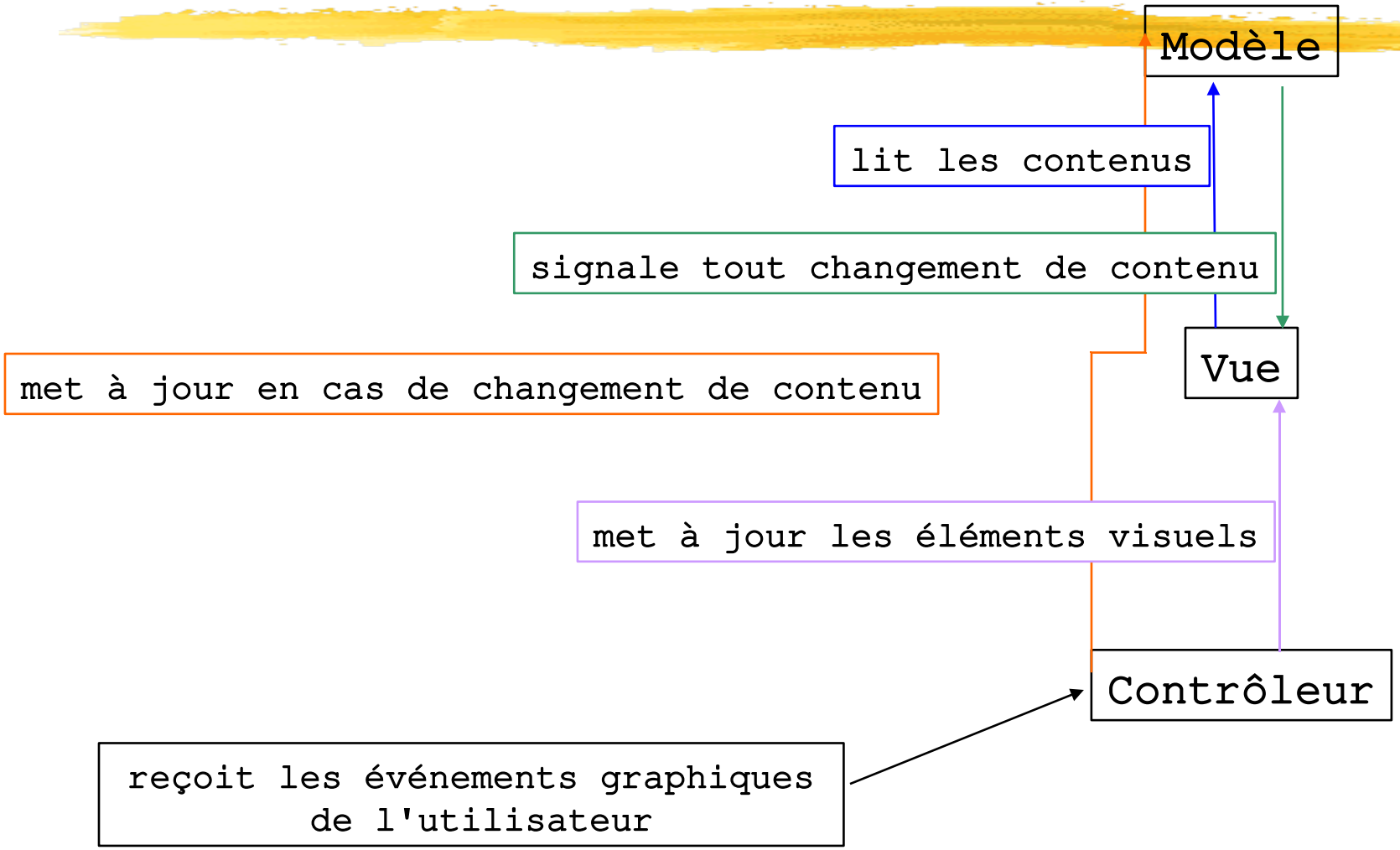
architecture MVC : exemple2

- Application de calcul du budget d'une société
  - le modèle : les données et le procédures de calcul pour le budget prévisionnel de la société
  - les vues : une représentation sous forme de tableau, une sous forme graphique, une vue pour la direction, une pour la DRH,...
  - le contrôleur : des éléments de l'interface pour faire varier les paramètres du calcul

Exemple 3 : différentes vues sur un ensemble de pourcentages



Interactions entre le modèle, la vue et le contrôleur



Rôle des 3 éléments : le contrôleur

- Reçoit les événements de l'interface utilisateur
- Les traduit :
  - en changement dans la vue s'ils agissent sur le côté visuel
    - exemple : clic dans l'ascenseur de la fenêtre
  - en changement dans le modèle s'ils agissent sur le contenu du modèle

    exemple : on réajuste le camembert, on modifie le diagramme en bâtons en agissant dans la vue

Rôle des 3 éléments : le modèle

- Il peut être modifié sur ordre du contrôleur
- Il signale à ses vues tout changement de contenu en leur envoyant un événement qui leur spécifie de se mettre à jour
- Mais il ignore :
  - comment il est affiché
  - qui lui a notifié un changement d'état

Rôle des 3 éléments : la vue

- La vue se met à jour dès qu'elle reçoit un ordre de notification
  - du contrôleur
  - du modèle
- Quand la notification vient du modèle, elle va consulter le modèle pour se réafficher de manière adéquate

Intérêts de l'architecture MVC

- **Indépendance** entre :
  - la représentation logique d'une application (modèle)
  - la représentation visuelle qu'on en donne (vue)
  - les actions que l 'utilisateur effectue (contrôleur)

⇒ **Séparation** claire entre les données du programme et l'interface graphique affichant ces données

⇒ **Possibilités de vues différentes d'un même modèle**

L'application peut montrer l'état du modèle de différentes façons, avec différentes interfaces utilisateurs

Intérêts de l'architecture MVC

- Cette indépendance favorise le développement et la maintenance des applications :
  - Modularité dans la conception
    - vue et contrôleur peuvent être développés indépendamment du modèle (pourvu qu'une interface entre les deux soit définie)
  - Meilleure répartition des tâches
    - développeurs du modèle/développeurs de l'interface ont des compétences différentes
      - développeurs du modèle : connaissance métier
      - développeurs de l ’interface : connaissance des besoins utilisateurs, souci d ’ergonomie...

Architecture MVC et composants swing

- Chaque composant swing s'appuie sur une architecture MVC en l'adaptant : la vue et le contrôleur sont combinés en un même objet
- Pour beaucoup de composants, cette architecture n'est pas exploitée (JButton,…)
- Pour d'autres, elle est très importante : JList, JTable
- L'architecture MVC fonctionne en utilisant la notion d'événement

Fonctionnement du MVC avec la notion d'événement notifié aux vues

- Le modèle doit connaître les vues qui dépendent de lui

- **Dès que le modèle change,**
  - il crée un événement correspondant à ce changement
  - il envoie cet événement à toutes ses vues
  - ses vues, à la notification de l'événement, réagissent en conséquence, en mettant à jour leur apparence visuelle.

Implémentation en java des modèles

- Tous les composants swing reposent sur le même principe:
  - Les modèles sont des interfaces `XXModel`
  - Des classes abstraites `AbstractXXModel` implémentent l 'interface en fournissant le code de certaines méthodes
    - de gestion du modèle (modification,…),
    - d'ajouts de vue
    - des méthodes de notification d'événements
  - Le modèle mémorise l'ensemble de ses vues

Implémentation en java des vues/contrôleurs

- Les vues sont interfaces ZZListener correspondant à des écouteurs d'événements
- Les événements correspondant à des changements dans le modèle sont des YYEvent
- Dès qu'un tel événement est notifié à une vue, elle réagit en déclenchant la méthode appropriée
- Les vues doivent connaître le modèle

vue sur un modèle = composant qui s'est déclaré comme écouteur d'événements de changements sur le modèle

En général : comment réaliser un MVC en java

- Définir un modèle
- Définir le composant qui sera une vue sur ce modèle :
  - Il doit se déclarer écouteur des événements de changement dans le modèle en implémentant l'interface adéquate
  - Il doit mémoriser le modèle en variable membre
  - Il doit donner un corps à chaque méthode de l'interface
  - Il doit se faire ajouter à la liste des écouteurs du modèle

Un exemple de MVC

Permettant de gérer un ensemble simple d'éléments

MVC pour un ensemble d ’éléments

- MVC
  - Modèle : interface ListModel, classe abstraite AbstractListModel et classe DefaultListModel
  - Vue : interface ListDataListener concernée par les changements sur le modèle : ajout/suppression d'éléments de la liste
  - Evénement transmis du modèle aux vues : ListDataEvent

Le modèle (1)

- Par exemple, une instance de la classe DefaultListModel ou d'une sous-classe que vous créerez

- Il mémorise l'ensemble des données et fournit des méthodes qui permettent de manipuler ces données

  void addElement(Object obj)
  boolean contains(Object elem)
  Object get(int index)
  boolean removeElement(Object obj)
  Object remove (int index)
  int size()

Noms des méthodes repris de la classe Vector

Le modèle (2)

- **Il mémorise la liste des vues** (ListDataListener) et fournit des méthodes d'ajout/suppression de tels écouteurs

```
void addListDataListener(ListDataListener
listener);
void removeListDataListener(ListDataListener
listener);
```

- **Il fournit des méthodes de notification d'événements à ses vues qu'il adresse à ses vues quand il se modifie**

```
    void fireContentsChanged(Object source, int
index0, int index1)
    void fireIntervalAdded(Object source, int index0,
int index1)
    void fireIntervalRemoved(Object source, int
index0, int index1)
```

La vue sur un modèle de type ensemble d'objets

- doit implémenter l'interface ListDataListener
- Doit donner un corps aux 3 méthodes :
  - void contentsChanged(ListDataEvent e)
    - en réponse à fireContentsChanged déclenchée dans le modèle ;
    - reçu quand le contenu de la liste a changé
  - void intervalAdded(ListDataEvent e)
    - en réponse à la méthode fireIntervalAdded déclenchée dans le modèle;
    - reçu quand un intervalle (index0,index1) a été inséré dans le modèle
  - void intervalRemoved(ListDataEvent e)
    - en réponse à fireIntervalRemoved;
    - reçue quand l'intervalle a été enlevé

L'événement de notification de changement envoyé par le modèle à ses vues

- Evénement de la classe ListDataEvent à exploiter par la vue pour savoir ce qui a changé dans le modèle.
- ListDataEvent posséde les méthodes

```
public int getIndex0()
public int getIndex1()
public int getType()
```

  - renvoie un mnémonique pour le type de l'événement (défini en constante de classe)

```
CONTENTS_CHANGED
INTERVAL_ADDED
INTERVAL_REMOVED
```

Comment se passe l'interaction entre le modèle et ses vues ?

Quand le modèle (l'ensemble d'objets) change parce qu'on lui a ajouté ou retiré un (ou plusieurs) élément(s),

Le modèle

- construit un événement de type ListDataEvent prenant en compte le type de modifications
- notifie les écouteurs ListDataListener présents dans sa liste du changement. Pour cela, il déclenche la méthode correspondante, fireYY.

La vue

- déclenche la méthode correspondante
- Et peut utiliser l'événement qu'elle a en paramètre

Composant graphique encapsulant ce MVC : la classe JList

- C'est un composant graphique de swing qui affiche une liste
- Les données à afficher sont présentes dans un modèle sous-jacent (ListModel)
- JList est une classe "Wrapper"
  - Elle écoute les événements reçus du modèle
  - Elle y réagit « toute seule »

⇒ Toute modification du modèle se répercutera dans la liste sans rien avoir à déclarer, ni faire de particulier

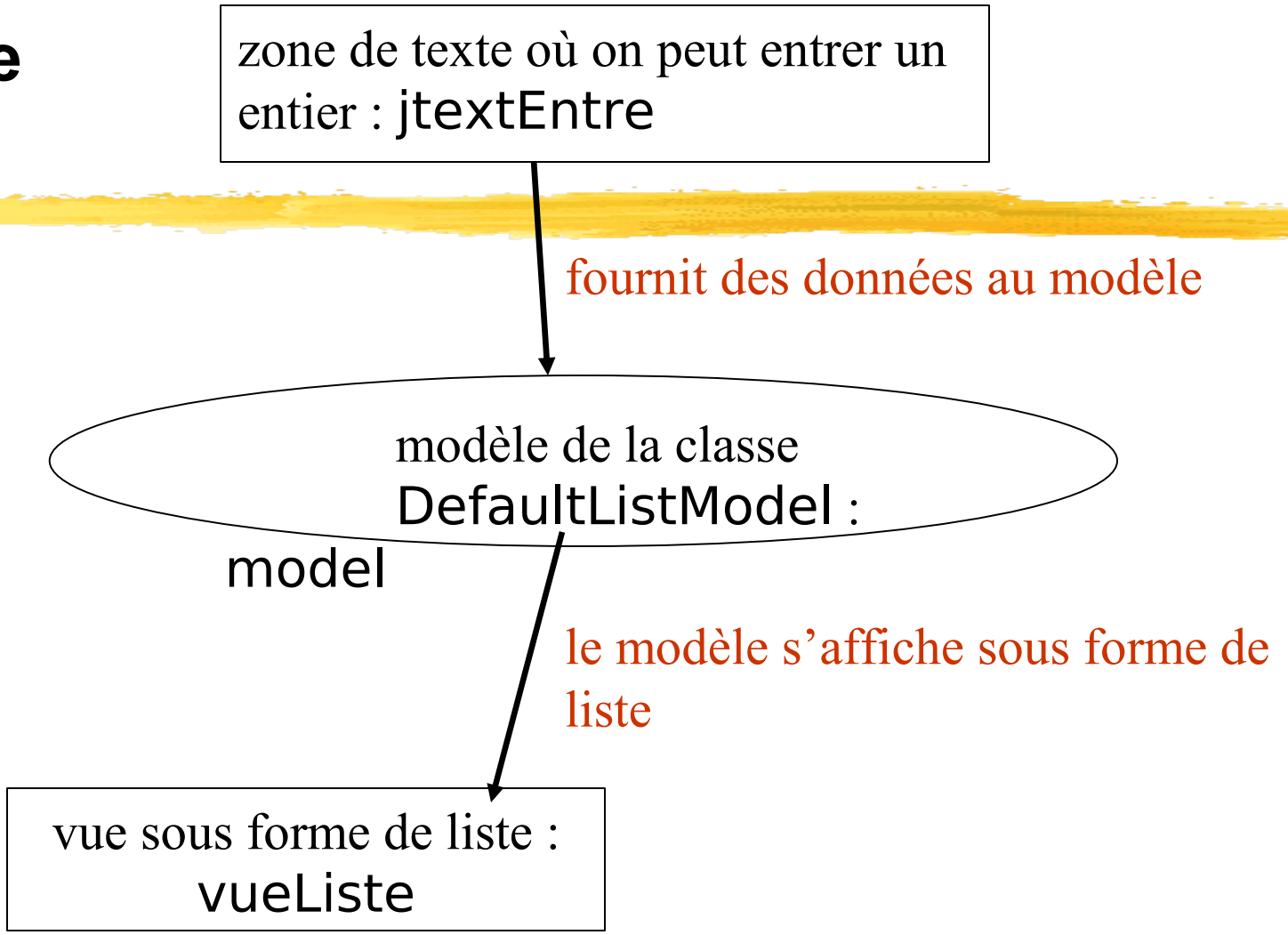
⇒ Transparent pour l'utilisateur

Une application : entrer un ensemble de nombres un par un dans une zone de texte et les afficher

Démo

- Modèle (ensemble de nombres entiers) à créer et mémoriser : instance de la classe `DefaultListModel`
- Vue sur ce modèle : une JList dans le panneau central qui affiche cet ensemble
- On peut ajouter des entiers à la liste en les entrant dans une zone de texte située dans le panneau nord
- La vue doit se mettre à jour automatiquement

Analyse


zone de texte où on peut entrer un entier : jtextEntre
fournit des données au modèle
modèle de la classe DefaultListModel : model
le modèle s'affiche sous forme de liste
vue sous forme de liste : vueListe

Analyse : suite

- **vueListe** : instance de JList possédant comme modèle : model.

  => utilisation du constructeur : public JList (ListModel m)

- **jtextEntre** : instance de JTextField devant prévenir le modèle si l'utilisateur entre un nouveau nombre

=> lui associer un écouteur de classe ActionListener réalisant ce travail (l'événement ActionEvent est construit quand l'utilisateur tapera sur 'retour chariot' dans la zone de texte).

```
public class FenEnsembleMVC extends JFrame
   {private JTextField jTextEntre;
    private DefaultListModel model;

   public FenEnsembleMVC(String titre)    {
        super(titre);
        model=new DefaultListModel();
        this.initialise();
        ...
   }

pùblic void initialise()    {
   ....
   }
```

```
public JPanel creePanelNord()        {
    JPanel pan=new JPanel();
    pan.add(new JLabel("Donnez un nombre"));
    jTextEntre=new JTextField("",20);
    pan.add(jTextEntre);
    jTextEntre.addActionListener(new
AjouteNombreEcouteur());
    return(pan);
}

public JSplitPane creePanelCentre() {
    JSplitPane pan=new
JSplitPane(JSplitPane.HORIZONTAL_SPLIT);
    pan.add(new JLabel("Liste des nombres");

    JList jListe =new JList(model);
    JScrollPane listeScrol=new  JScrollPane(jListe);
    pan.add(listeScrol);
    return pan;
 }
```

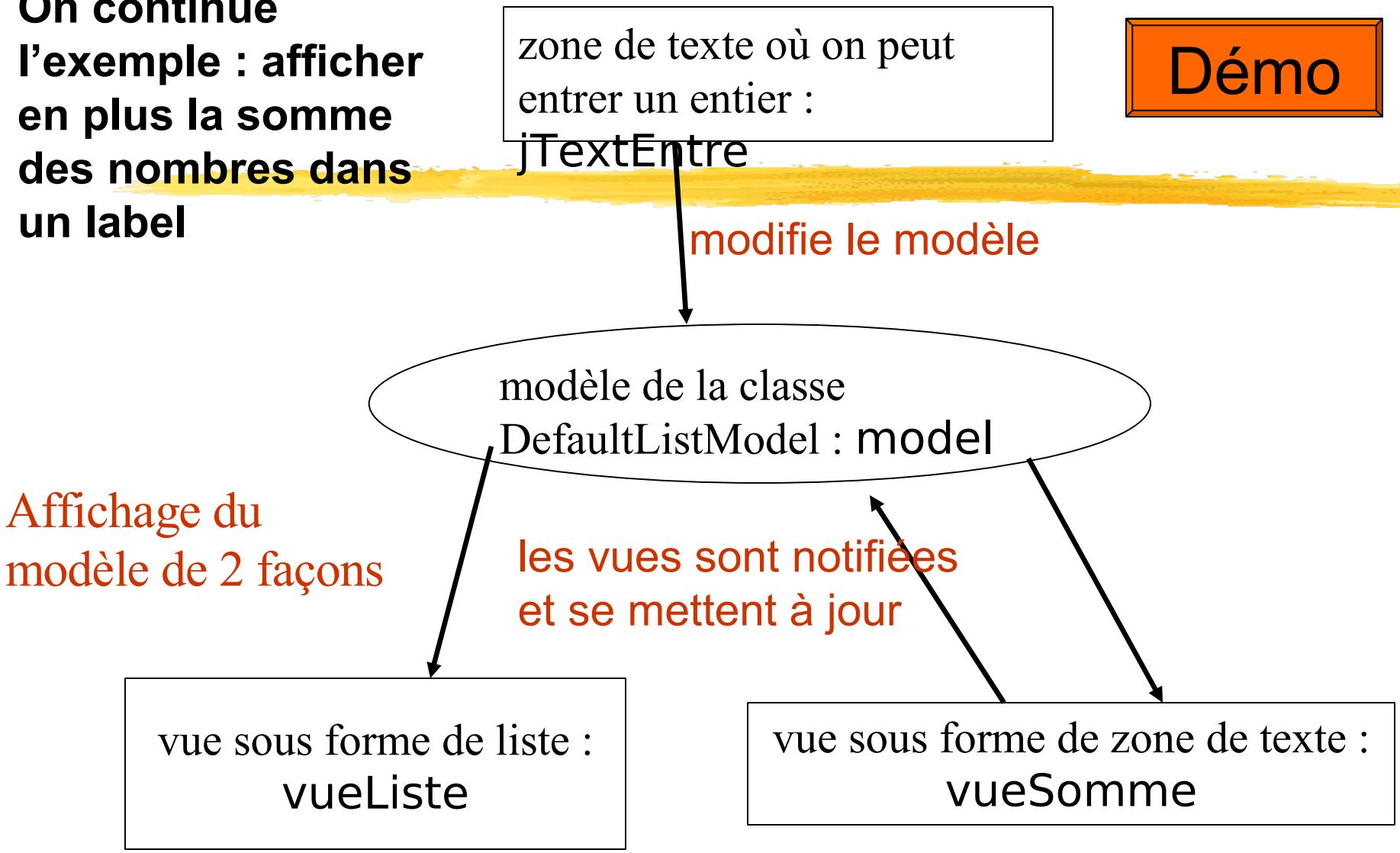
inner-classe

```
class AjouteNombreEcouteur  implements ActionListener
{
    public void actionPerformed(ActionEvent e)   {

      String s=jTextEntre.getText();
      try  {

          int i=Integer.parseInt(s);
            Integer oi= new Integer(i);
          model.addElement(oi);
            }
      catch (NumberFormatException en) {}


      jTextEntre.setText("");
}
```

Remarques

- Le modèle se modifie ici par la méthode addElement.
- Attention : Dans un DefaultListModel, on ne met que des instances d'Object
  - donc impossible d 'y ranger des int.
  - On y rangera des Integer construits à partir des int après avoir vérifié que les chaînes entrées sont dans le bon format
- Toute modification du modèle se répercute automatiquement dans la JList sans que le programmeur fasse quelque chose
- car JList est une classe "wrapper" qui gère seule les interactions modèle/vue

On continue l'exemple : afficher en plus la somme des nombres dans un label



Analyse : suite

- **vueSomme** : - instance de JLabel possédant comme modèle : model.
  - est une vue sur ce modèle, donc doit y être lié, de façon à modifier son affichage dès qu'un changement dans le modèle a lieu.
- => déclarer une sous-classe spécifique de JLabel qui :
  - a comme modèle le même modèle que la JList : model
  - implémente l'interface ListDataListener : …
  - s'enregistre auprès du modèle comme listener des événements de type ListDataEvent...

Ce qu ’on ajoute : la création du panel Sud

```
public JSplitPane creePanelSud()  {

        JSplitPane pan=new
JSplitPane(JSplitPane.HORIZONTAL_SPLIT);

        pan.add(new JLabel("somme"));

        SommeVue vueSomme=new
SommeVue(model);

        pan.add(vueSomme);
        return pan;
}
```

Ce qu ’on ajoute : la création d ’une vue

```
class SommeVue extends JLabel implements
ListDataListener  {

private DefaultListModel model;

   public SommeVue(DefaultListModel mod)   {

         model=mod;
         model. addListDataListener(this);
         }


public void contentsChanged(ListDataEvent event)   {
         // ne sera jamais déclenchée ici
}
   public void intervalRemoved(ListDataEvent event)   {
         //ne sera jamais déclenchée  ici
}
```

La seule méthode qui sera déclenchée : intervalAdded

1ère solution : la somme est entièrement recalculée

```
public void intervalAdded(ListDataEvent
event)  {
      int s = this.calculeSomme() ;
      this.setText(""+s);
      }
private int calculeSomme()  {
      int taille=model.getSize();
      int s = 0;
      int val;
      for (int i=0; i<taille ; i++)
         {String snum= (String)model.elementAt(i);
                  val= Integer.parseInt(snum);
                  s=s+val;
         }
```

La seule méthode qui sera déclenchée : intervalAdded

2ème solution : la somme est calculée à partir de son ancienne valeur

```
public void intervalAdded(ListDataEvent
   event)  {
     int indAjout=e.getIndex0();
   Integer oi= (Integer) model.elementAt(indAjout);

   String ssom=this.getText();
   int somAnc=Integer.parseInt(ssom);

   int som=somAnc+oi.intValue();

   this.setText(som+"");
      }
```

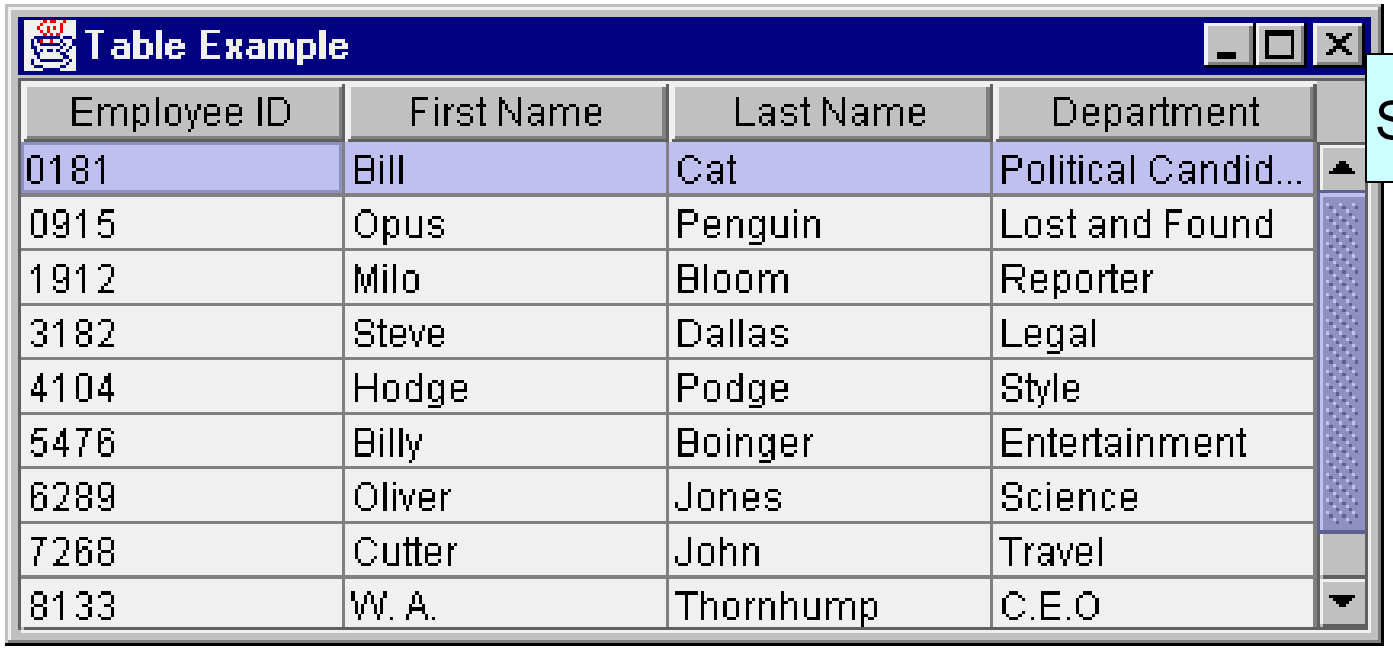
Résumé : pour créer un MVC avec un ensemble d'objets comme modèle

- Créer un modèle, instance de la classe DefaultListModel, et le mémoriser
- Faire que chaque composant qui veut être prévenu des changements de contenu dans le modèle :
  - mémorise le modèle créé en 1.
  - implémente l'interface **ListDataListener** et donne un corps aux méthodes correspondantes pour se mettre à jour
  - s'ajoute à la liste d'écouteurs maintenue par le modèle (emploi de la méthode **addListDataListener**)

Nouvel exemple de MVC

Permettant de gérer des données « tabulaires » (à 2 dimensions : lignes/colonnes)

Données « tabulaires » : exemple



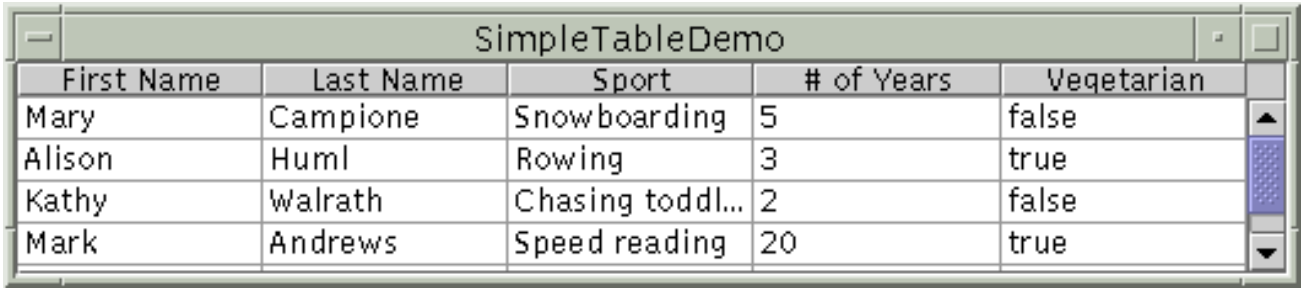
Structure de la table

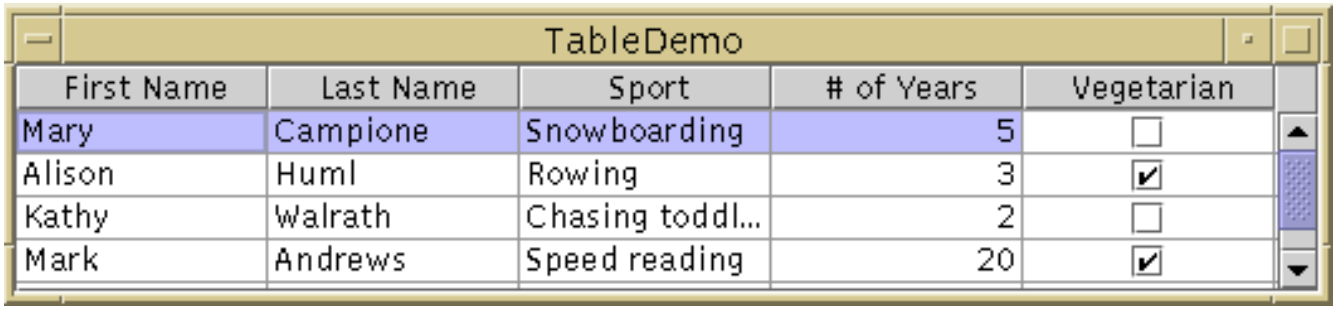
Données de la table

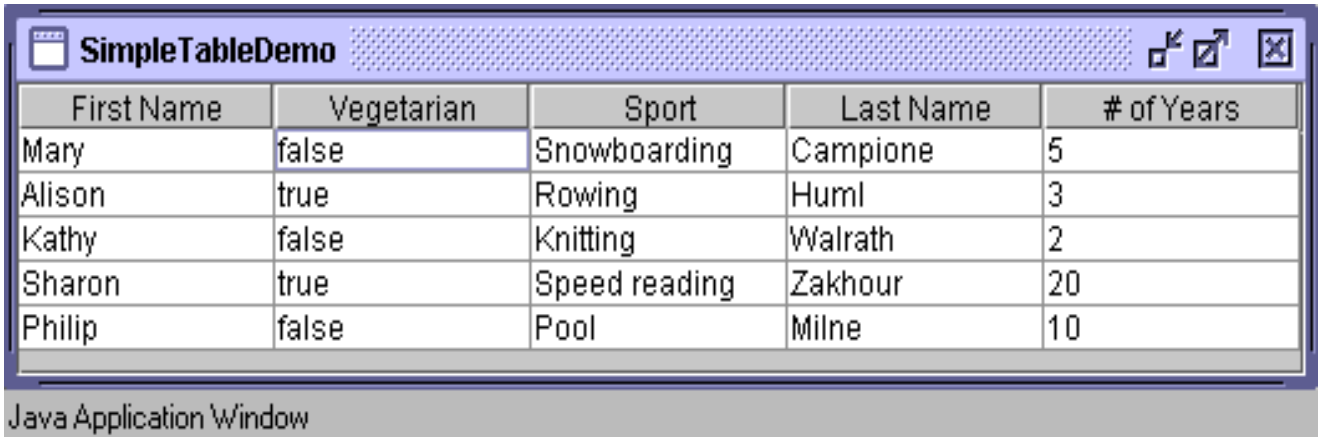
Qu'est-ce qu'une table ?

- Une table possède :
  - une structure : noms d'attributs et leurs types
    - On parle aussi de schéma de la table ou de méta-données
  - des données : liste de nuplets ou tuples ou rows
  - Chaque nuplet est un ensemble de valeurs données aux attributs

Les données « tabulaires » peuvent se visualiser différemment : plusieurs vue sur le même modèle







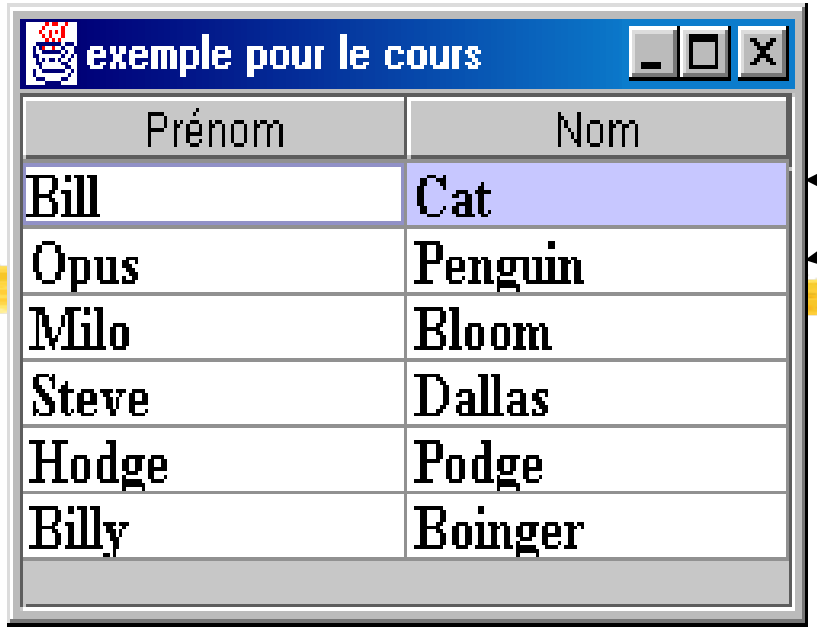
Implantation du MVC pour les tables

- MVC
  - Modèle : interface TableModel et classe AbstractTableModel et DefaultTableModel
  - Vue : interface TableModelListener concernée par les changements sur le modèle
  - Evénement transmis du modèle aux vues : TableDataEvent

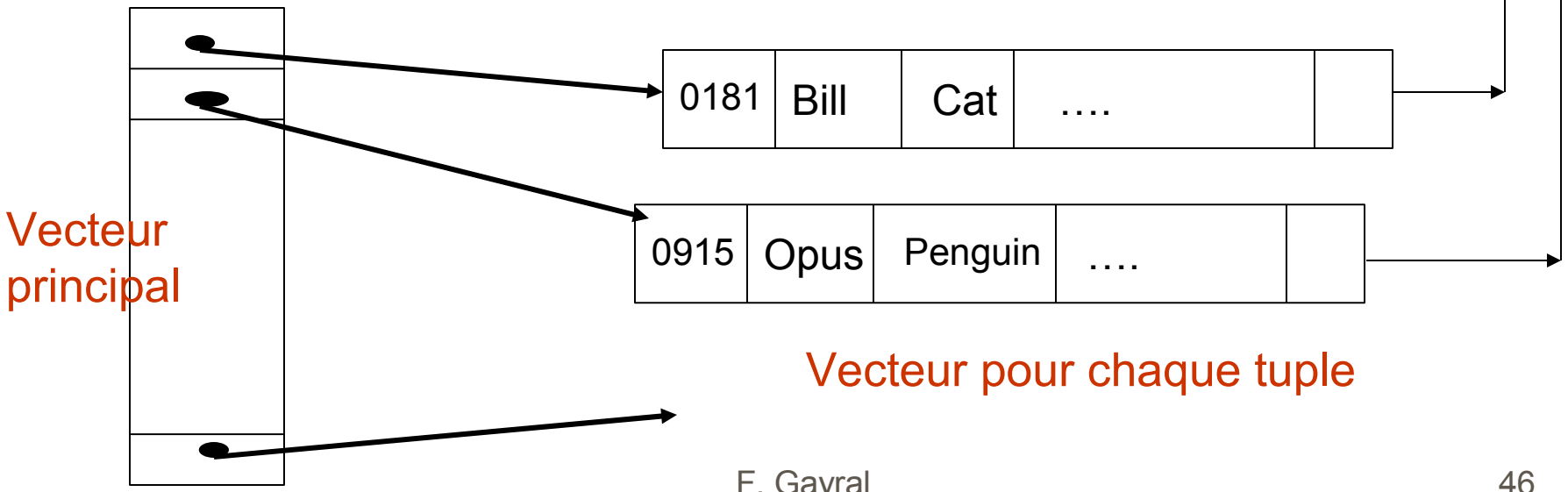
Le modèle (1)

- Par exemple, une instance de la classe DefaultTableModel ou d'une sous-classe que vous créerez
- Il mémorise les données sous la forme
  - d 'un tableau de lignes
  - ou d 'un vecteur de lignes
- Il mémorise les méta-données sous la forme d 'un tableau ou d 'un vecteur

Le modèle : la classe DefaultTableModel



- **Il mémorise les données de la table** sous forme d ’un vecteur contenant des vecteurs



Comment construire un modèle ? constructeurs de la classe DefaultTableModel

- Pour construire un modèle de table, toujours fournir :
  - une structure pour le schéma de la table
  - une structure pour les données

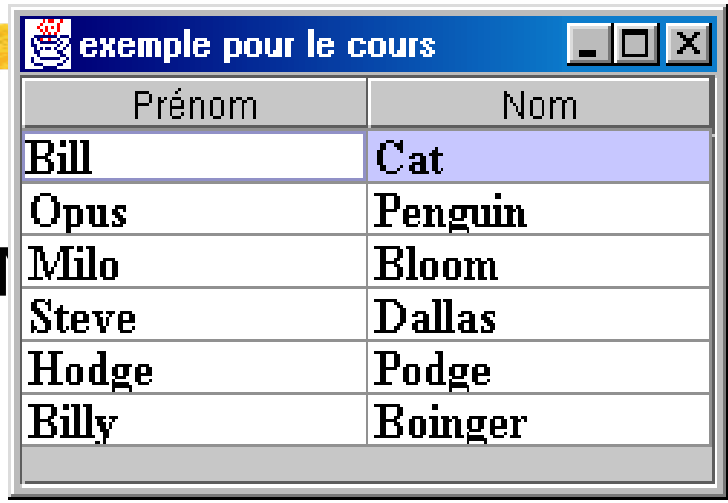
- Soit sous forme de deux tableaux

  DefaultTableModel(Object[][] data,Object[] columnNames)

- Soit sous forme de deux vecteurs

  DefaultTableModel(Vector data, Vector columnNames)

Exemple de création d'un modèle de table de nom/prénom




```
String nomsCol[] = { "Prénom", "N

String rows[][] = {       {"Bill", "Cat"}, {"Opus",
   "Penguin"},
   {"Milo","Bloom"},{"Steve","Dallas"},...
              {"Hodge","Podge"},{"Billy", "Boinger"}
                  };
DefaultTableModel model=new
   DefaultTableModel(rows, nomsCol);
```

Représentation des données

- Il existe un vecteur principal qui contient des tuples
- Un tuple est une instance de Vector
- Chaque tuple est repéré par un indice : sa position dans le vecteur principal du modèle
- Chaque donnée est repérée par deux indices
  - un indice de ligne : qui repère le tuple dans le vecteur principal
  - un indice de colonne : qui repère la position de la donnée dans le tuple

Le modèle DefaultTableModel : méthodes d'accès et de manipulation des données du modèle

Vector getDataVector()
int getRowCount()

void addRow(Vector rowData)
void insertRow(int iRow,Vector rowData)
void removeRow(int iRow)

Object getValueAt(int iRow,int jColumn)
void setValueAt(Object aValue, int iRow, int jColumn)

Le modèle DefaultTableModel : méthodes d'accès et de manipulation des méta-données

Méthodes concernant la structure de la table

- Accès

  int getColumnCount()

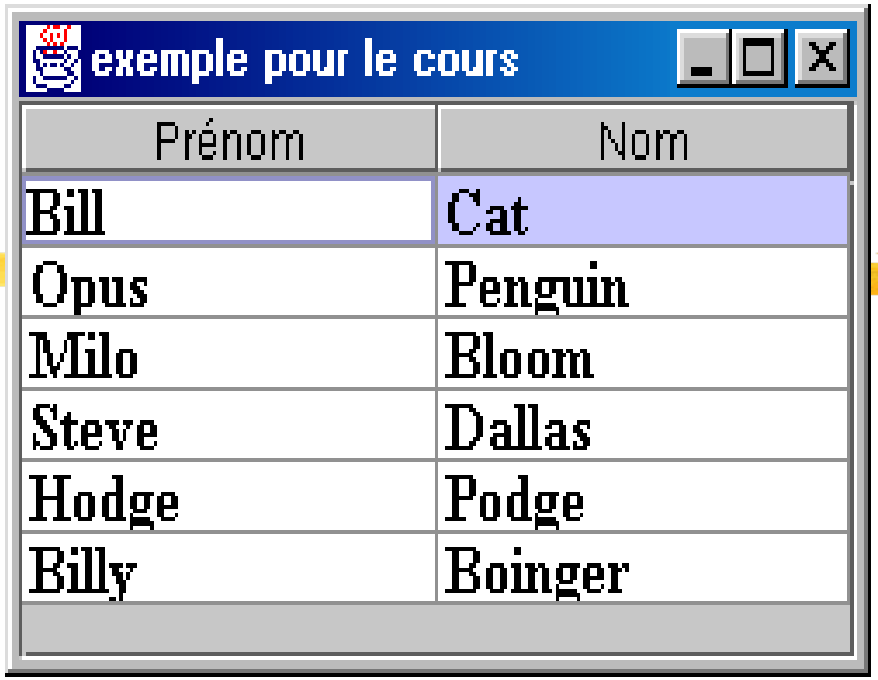
  String getColumnName(int column)

- Modification de la structure

void addColumn(Object columnName, Vector columnData)

void addColumn(Object columnName)

Exemple suite




```
int
    nbTuple=model.getRowCo
    unt();
String
    s=model.getColumnName
    (0);
String
    pren=model.getValueAt(3,
```

Le modèle DefaultTableModel : relation avec ses vues

- **Ajout/suppression de vues** (TableModelListener)

  void addTableModelListener(TableModelListener l)

  void removeTableModelListener(TableModelListener l)

- **Mécanismes de notification d'événement**

  void fireTableChanged(TableModelEvent e)

Les vues sur un modèle
DefaultTableModel : interface TableModelListener

- Implémentent l'interface TableModelListener :
  - écouteur d'événement de changements de données ou méta-données dans le modèle
- L'interface possède une unique méthode :
  - void tableChanged(TableModelEvent e)
    - en réponse à fireTableDataChanged déclenchée dans le modèle ; reçu quand les données ou méta-données ont été modifiées.

L'événement de notification du modèle vers ses vues : TableModelEvent

- Il encapsule les informations sur les changements qui ont eu lieu sur le modèle
  - changement de la valeur d'une ou plusieurs cellules.
  - ajout/suppression de lignes.
  - changement du nombre de colonnes
- Il possède les méthodes

int getColumn() int getFirstRow()

int getLastRow() int getType()

– mnémonique pour le type de l'événement (défini en constante de classe de TableModelEvent) : INSERT, UPDATE, DELETE

L'événement de notification du modèle vers ses vues : TableModelEvent

- La vue peut exploiter le paramètre TableModelEvent que lui fournit la méthode

  void tableChanged(TableModelEvent e)

  pour savoir ce qui a changé dans le modèle

- Exemple

```
public void tableChanged(TableModelEvent e) {
if (e.getType()== TableModelEvent.UPDATE) {
        int row = e.getFirstRow();
          String columnName =
model.getColumnName(column);
          Object data = model.getValueAt(row,
column);
```

Comment se passe l'interaction entre le modèle et ses vues ?

Quand le modèle change parce qu'on lui a ajouté, retiré ou modifié un (ou plusieurs) élément(s) ou méta-éléments,

Le modèle

- construit un événement de type `TableModelEvent` prenant en compte le type de modifications
- notifie ses vues (écouteurs TableModelListener) du changement. Pour cela, il déclenche la méthode, `fireTableChanged` .

La vue

- dès qu ’elle est notifiée, elle exécute la méthode correspondante :

```
void tableChanged(TableModelEvent e)
```

- et doit se mettre à jour en exploitant l'événement qu'elle a en paramètre

Un canevas de code pour une vue sur un DefaultTableModel

```
class ExempleVue extends .... implements
   TableModelListener   {
   private DefaultTableModel model ;
public ExempleVue (DefaultTableModel mod)   {
      model=mod;
     model.addTableModelListener(this);
   }
  public void tableChanged(TableModelEvent e) {
    switch (e.getType())      {
    case TableModelEvent.UPDATE : ....
    case TableModelEvent.DELETE : ....
    case TableModelEvent.INSERT : ....
```

Composant graphique encapsulant ce MVC : La classe JTable

- C'est un composant graphique fourni qui affiche une table avec ses méta-données et ses données présentes dans un modèle sous-jacent de type `TableModel`
- JTable est une classe "Wrapper"
  - Elle écoute les événements reçus du modèle
  - Elle y réagit « toute seule »

⇒ Toute modification du modèle se répercutera dans la liste sans rien avoir à déclarer, ni faire de particulier

⇒ Transparent pour l'utilisateur

Détails sur JTable

- La classe JTable possède un modèle qui lui est associé grâce au constructeur

  JTable(TableModel dm)

  appelé après construction du modèle
- Exemple

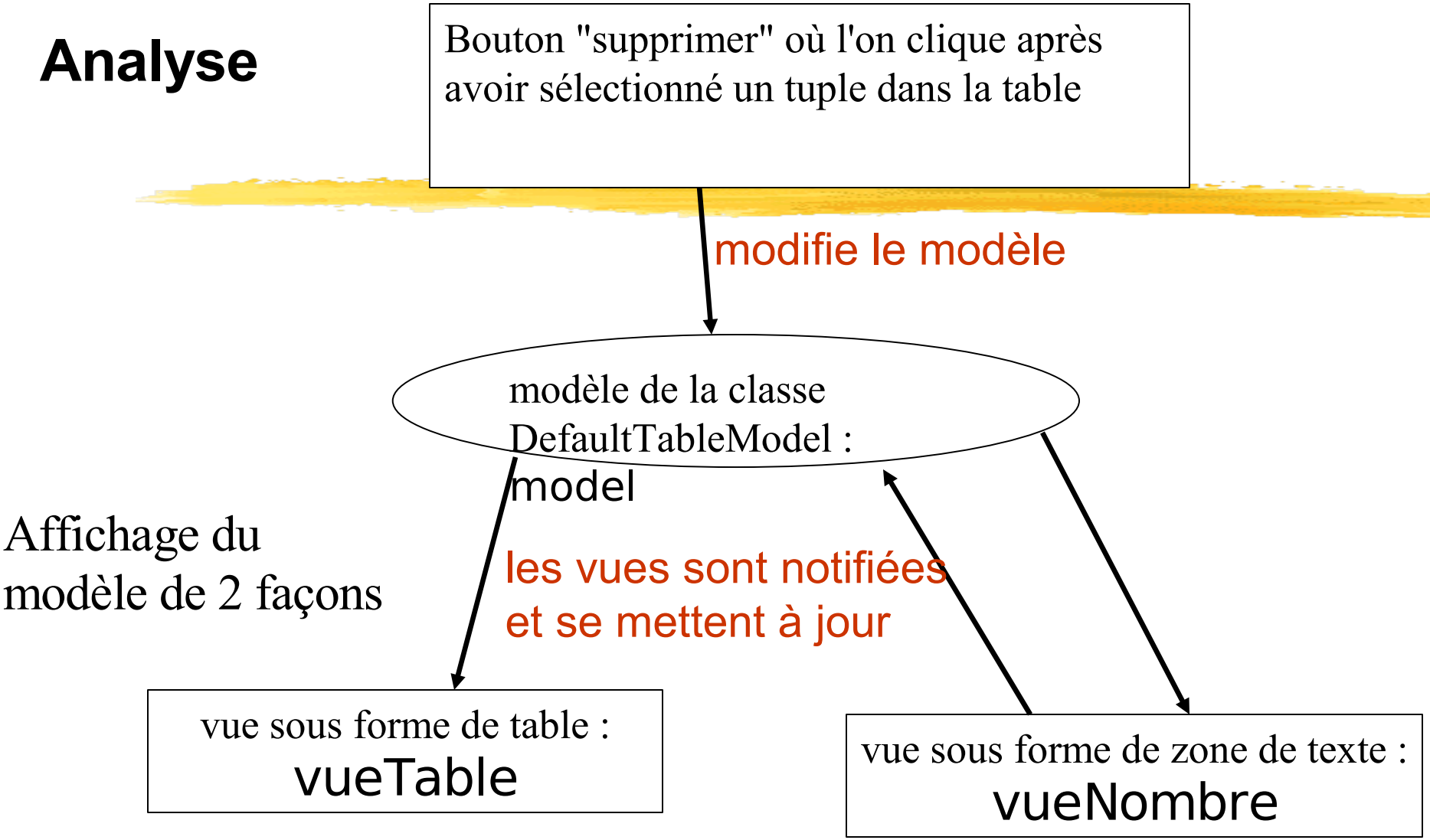
```
String columnNames[] = { "Prénom", "Nom"};
 String rows[][] = { {"Bill", "Cat"},{"Opus", "Penguin"},... }
;
DefaultTableModel model = new DefaultTableModel(rows,
columnNames) ;
JTable table = new JTable(model);
 JScrollPane scrollpane = new JScrollPane(table);
```

Une application

Démo

- modèle : un ensemble de noms prénoms
- 2 vues : - une qui affiche leur nombre
  - une qui affiche l'ensemble sous forme de table
- On peut supprimer un tuple au modèle en le sélectionnant dans la table : les 2 vues doivent se mettre à jour automatiquement

Analyse


Bouton "supprimer" où l'on clique après avoir sélectionné un tuple dans la table
modifie le modèle
modèle de la classe DefaultTableModel : model
Affichage du modèle de 2 façons
les vues sont notifiées et se mettent à jour
vue sous forme de table : vueTable
vue sous forme de zone de texte : vueNombre

Analyse : suite

- **vueTable** : - instance de JTable possédant comme modèle : model.

  - la gestion MVC est gérée par la classe "wrapper" JTable

- **vueNombre** : - instance de JLabel est une vue sur le modèle : model.

- => déclarer une sous-classe spécifique de JLabel qui :

  - a comme modèle le même modèle : model
  - implémente l'interface TableDataListener : …
  - s'enregistre auprès du modèle comme listener des événements de type TableDataEvent.
  - donne un corps à ….

```
public class FenTable extends JFrame  {

private DefaultTableModel model;
private JTable table;

public FenTable(String titre, int w, int h) {
  super(titre);
  this.creeModele();
  this.initialise();
  this.setSize (w, h);
  this.show();
  }

public void creeModele() {

    String columnNames[] = { "Prénom", "Nom"};

    String rows[][] = { {"Bill", "Cat"},{"Opus", "Penguin"},... };

    model = new DefaultTableModel(rows, columnNames) ;
  }
```

```
public void initialise()  {
  Container c=this.getContentPane();
  c.setLayout(new BorderLayout());
   c.add(this.creeTableCentre(),"Center");
   c.add(this.creePanelSud(),"South");}

 public JScrollPane creeTableCentre() {
    table = new JTable (model);


 table.setSelectionMode(ListSelectionModel.SINGLE_SELEC
 TION) ;
    JScrollPane scrollPane = new JScrollPane (table);
    return scrollPane;
 }
public JPanel creePanelEst() {
  JPanel pEast=new JPanel();
  pEast.add(new JLabel("nombre de tuples"));

  NombreRowsVue vueNombre= new NombreRowsVue(model)
  pEast.add(vueNombre);
  return pEast;
 }
```

```
public JPanel creePanelSud() {
  JPanel pSud=new JPanel();
  JButton butSup=new JButton("supprime");
  pSud.add(butSup);
  butInsere.addActionListener(new SupprimeListener());
  return pSud;        }
```

```
 class SupprimeListener implements ActionListener  {
       public void actionPerformed(ActionEvent e) {

          int i=table.getSelectedRow();

          model.removeRow(i);
 }
 }
```

inner-classe

```
class NombreRowsVue extends JLabel implements TableMod
   private DefaultTableModel model;

   public NombreRowsVue(DefaultTableModel mod)   {
      model=mod;
      model.addTableModelListener(this);
    }

   public void tableChanged(TableModelEvent event) {

      int nombre=model.getRowCount();

      this.setText( nombre+"");

}  // fin de tableChanged
}  // fin de NombreRowsVue
```

Utilisation des JTable pour récupérer des résultats de requêtes à une base de données

Contexte

Programme client : une interface graphique pour une application qui interagit avec une base de données

L ’interaction se fait avec jdbc

Contexte

- Interaction avec une base de données via jdbc, les classes Connexion et Statement
  - Requête à la base
  - Récupération du résultats via un ResultSet
- Le résultat d'une requête de sélection (ResultSet) a :
  - des méta-données (via ResultSetMetaData)
  - des données (un curseur qui permet de parcourir la table)

⇒ va être mémorisé dans un modèle de table : instance de DefaultTableModel

⇒ va pouvoir être affiché dans une JTable

Construction d'un modèle de table passage d'un ResultSet à un DefaultTableModel

- Construire un DefaultTableModel à partir d'un ResultSet rs obtenu par une requête
- En trois temps :
  - (1) Création d'un modèle « vide »

```
DefaultTableModel model = new
    DefaultTableModel();
```

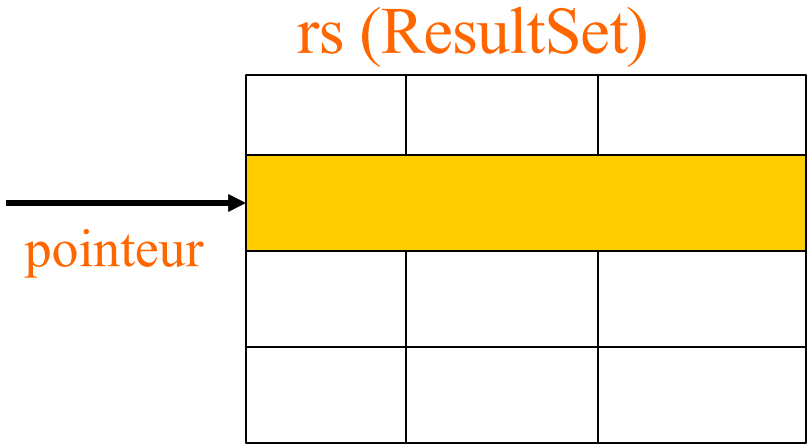
  - (2) Construction des méta-données (noms des colonnes)
  - (2) Construction des données (les lignes)

Passage d'un ResultSet (rs) à un DefaultTableModel : (1) Construction des méta-données

```
ResultSetMetaData rsmd = rs.getMetaData();
int i;
int numCols = rsmd.getColumnCount();
// création des colonnes
for (i=1; i<=numCols; i++) {
        String nomCol = rsmd.getColumnLabel(i );
      model.addColumn(nomCol));
      }
```

Passage d 'un ResultSet à un DefaultTableModel : (2) Construction des données : principe

A partir de chaque ligne extraite d'un ResultSet :

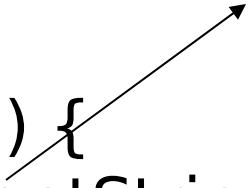


Créer un vecteur row à ajouter au modèle :

```
Vector row = new Vector;
...
model.addRow(row)
```

(2) Construction des données : code

```
while (rs.next ()) {
// ....
        Vector row= new Vector();
        for (i=1; i<=numCols; i++) {
                row.addElement( rs.getString(i) );
                }
        model.addRow(row);
     }  // fin du while
```



Et ensuite... On fait ce qu 'on veut avec le modèle, la JTable,...

- Pour créer une vue sous forme de JTable :

```
JTable table=new JTable(model);
```

- On l 'ajoute à une zone de travail

```
model =....
JTable table=new JTable(model);
JScrollPane scrollPane = new JScrollPane (table);
Container c=this.getContentPane();
c.add(scrollPane, «  Center  »);
```

Autres Classes liées à une JTable

- **Autre MVC sur les modifications de colonnes**
  - **TableColumnModel** : modèle pour les colonnes de la table
  - **TableColumnModelEvent** : événement d'ajout de colonne, de suppression ou de déplacement
  - **TableColumnModelListener, TableColumn,...**
- **Autre MVC**
  - **TableSelectionModel** : modèle de sélection des cellules
  - ...
- **JTableHeader** : composant graphique gérant les titres des colonnes.
- **TableCellRenderer** : interface de rendu des cellules
- **TableCellEditor** : l'éditeur de cellules

Détecter les lignes sélectionnées par l 'utilisateur dans la JTable

- Une JTable possède un modèle de sélection pour ses lignes
  ListSelectionModel
- L 'événement généré par la sélection dans une table est de type
  ListSelectionEvent
- L ' écouteur d 'événement de sélection d 'une ligne est de type
  ListSelectionListener qui a une seule méthode :
  public void valueChanged(ListSelectionEvent e)

Extrait de code

Démo

```
DefaultTableModel model=….
JTable table=new JTable(model);
ListSelectionModel rowSM = table.getSelectionModel();
rowSM. addListSelectionListener(new
MonListSelectionListener()););
```

```
class MonListSelectionListener implements
    ListSelectionListener {
    public void valueChanged(ListSelectionEvent e)          {
    if (e.getValueIsAdjusting())
                    return;
     // faire le nécessaire avec ce qui a été sélectionné dans la
    table
    ListSelectionModel lsm =
    (ListSelectionModel)e.getSource();
    int iSelect = lsm.getMinSelectionIndex();
    jta.append("---------- ligne sélectionnée "+ iSelect
    +"----------\n");
```

```
public class TableDemo extends JFrame {
    private DefaultTableModel model;
    private JTextArea jta;

     public TableDemo() {
       super();
       this.initModel() ;
       this.initComposants();
       ...      }

     private void initComposants()   {
       Container c=this.getContentPane();
       JSplitPane splitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT
       c.add(splitPane, BorderLayout.CENTER);
       JTable table = new JTable(model);
       ListSelectionListener rowSM= table.getSelectionModel();
       rowSM.addListSelectionListener(new MonListSelectionListener()
       JScrollPane scrollPane = new JScrollPane(table);
       splitPane.add(scrollPane);
```

```
        jta = new JTextArea();
        jta.setEditable(false);
        JScrollPane jtaPane = new JScrollPane(jta,
                    ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,

ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
        splitPane.add(jtaPane);
      }

       public void  initModel()  {
        String[] columnNames = ....
        Object[][] data = ...;
        model=new DefaultTableModel(data,columnNames);
       }
```

```
class MonListSelectionListener implements ListSelectionListener {


public void valueChanged(ListSelectionEvent e){
        if (e.getValueIsAdjusting())
           return;

        ListSelectionModel lsm = (ListSelectionModel)e.getSource();
                if (lsm.isSelectionEmpty()) {
                jta.append("Rien n'est sélectionné\n");
        }
        else {
           int selectedRow = lsm.getMinSelectionIndex();
           jta.append("---------- ligne sélectionnée "+selectedRow+"-----
           for (int j=0; j< model.getColumnCount();j++)
              jta.append(model.getValueAt(selectedRow,j)+"    ");
           jta.append("\n");
        }
```