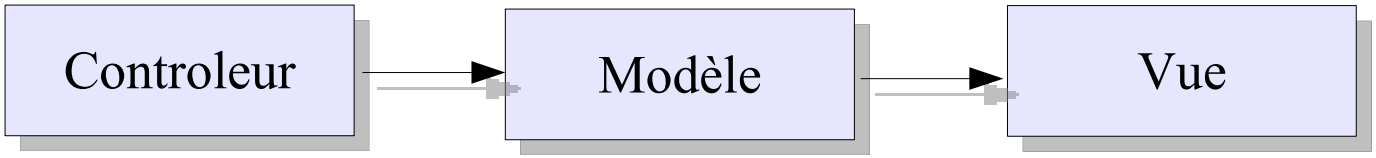
Modèle Vue Controleur

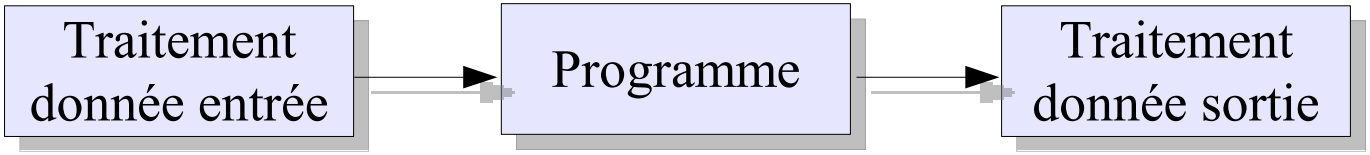
- Le *Design Pattern* MVC
- MVC de Swing
- Evènement et MVC
- Les *Renderer*

Pourquoi le MVC ?

- Décomposer une interface graphique en trois parties distinctes

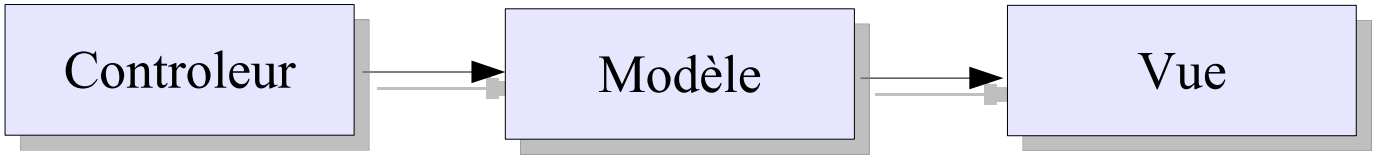


- Schéma calqué sur le traitement classique d'un programme



C'est quoi le MVC ?

- Les trois parties du MVC sont :



- **Contrôleur** : agit sur demande de l'utilisateur et modifie le modèle.
- **Modèle** : contient les données du programme sous une forme compréhensible par la vue, notifie la vue des changements.
- **Vue** : composant graphique affichant le modèle.

Avantage du MVC

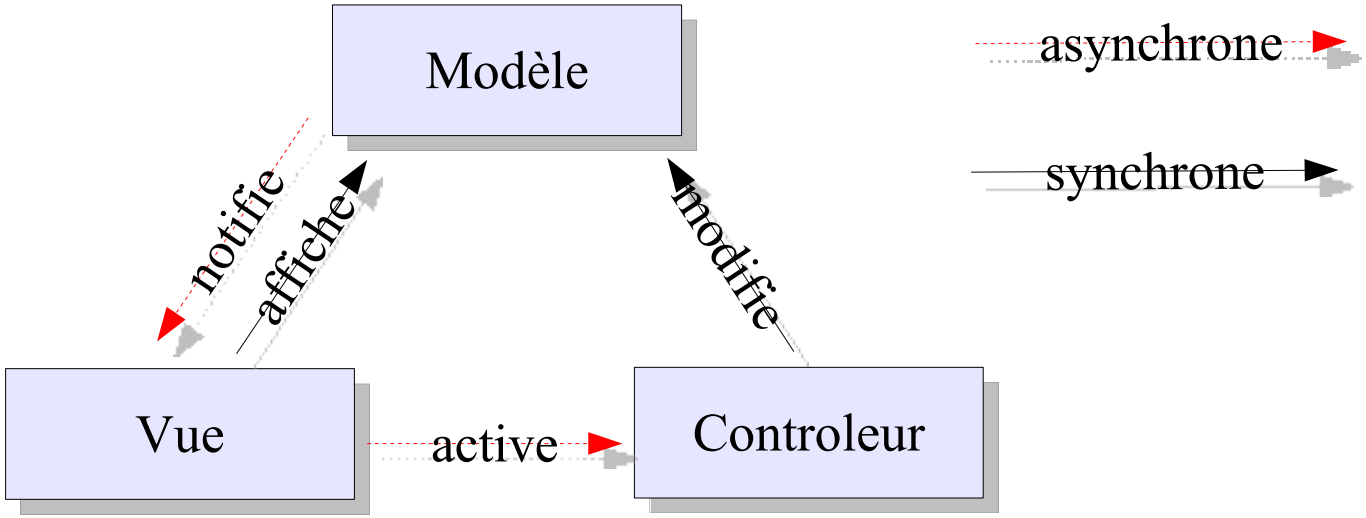
- **Séparation** claire entre les données du programme et la partie graphique affichant les données.
  - Le modèle montre les données à la vue.
  - La vue est le composant graphique.

- Evite le stockage des données à plusieurs endroits (syndrome Visual Basic).

- Possibilité pour le modèle d'informer la vue des **modifications incrémentalement**.
  (par exemple, ajout d'une ligne dans une table).

Le MVC de swing

- Pour un modèle, il peut y avoir plusieurs vues et plusieurs contrôleurs.

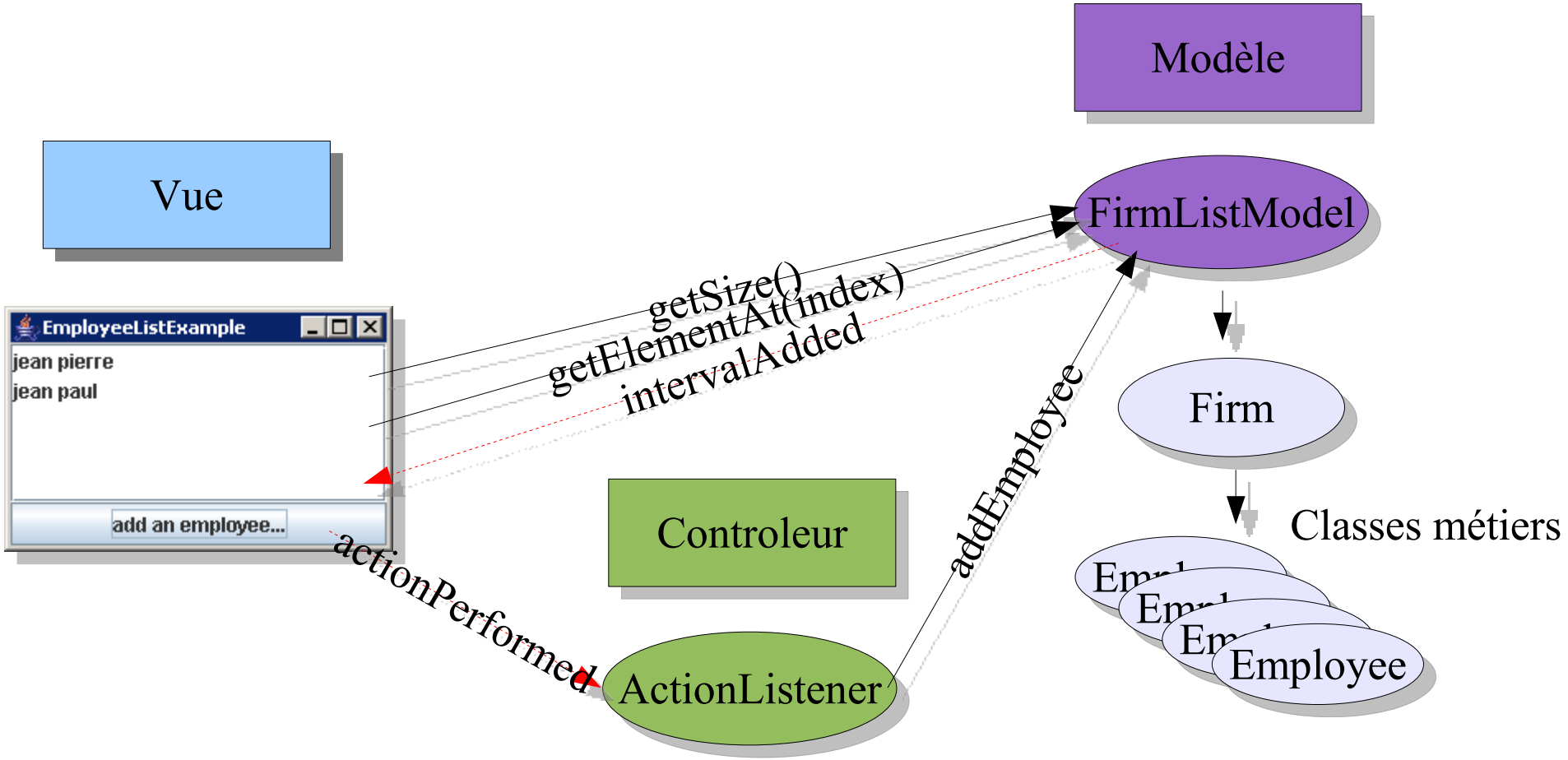


- La notification des changements d'état
  - entre le modèle et les vues et
  - entre la vue et les contrôleurs

  se font de façon **asynchrone** (par *listener*)

Un exemple en Swing

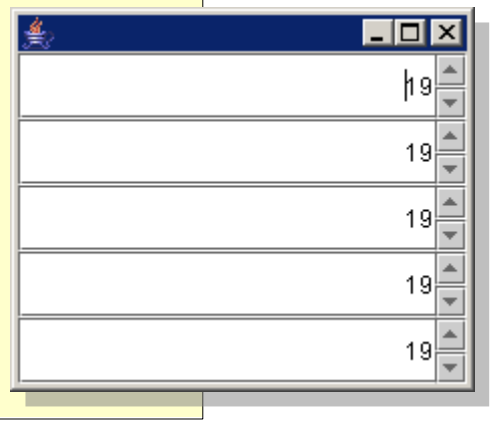
- Afficher une liste d'employée



Relation entre les vues et le modèle

- Pour un modèle, il peut y avoir plusieurs vues.

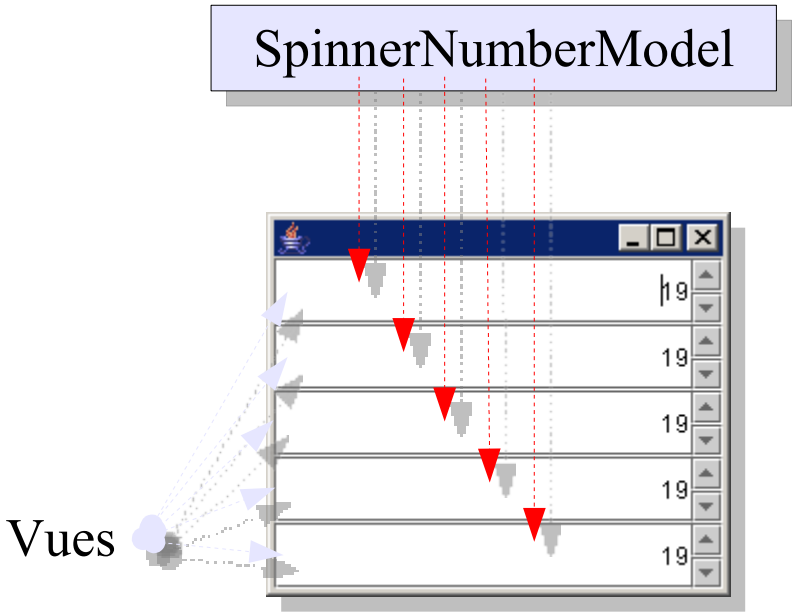
```
SpinnerNumberModel model=new SpinnerNumberModel(0,0,100,1);
JPanel panel=new JPanel(null);
BoxLayout layout=new BoxLayout(panel,BoxLayout.Y_AXIS);
panel.setLayout(layout);
for(int i=0;i<5;i++) {
  JSpinner spinner=new JSpinner(model);
  panel.add(spinner);
}
JFrame frame=new JFrame();
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setContentPane(panel);
frame.setSize(400,300);
frame.setVisible(true);
```



- Lors de la création de **la vue**, celle-ci s'**enregistre** en tant que **listener du modèle**.

Relation entre les vues et le modèle (2)

- Ici chaque vue partage le même modèle.



- Chaque **vue reçoit un évènement** (notification) indiquant que le modèle a changé.
- Les vues peuvent alors demander au modèle les valeurs ayant changées

Implanter un modèle

- Un modèle pour Swing est une **interface** que l'on peut implanter pour mettre des données sous une certaine forme.
- Exemple :
  - **`ListModel`** pour les listes,
  - **`TreeModel`** pour les arbres,
  - **`TableModel`** pour les tableaux (feuille de calcul),
  - **`ButtonModel`** pour les boutons,
  - etc.
- Il existe de plus, pour chaque modèle, une classe implantant le modèle par défaut. Par exemple, **DefaultListModel** pour **ListModel**

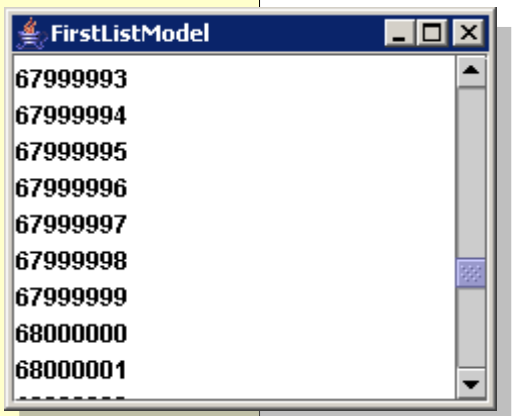
L'exemple du modèle de liste

- Le modèle de liste permet de voir n'importe quelle classes métiers sous forme de liste.
- L'interface **`ListModel`** est décomposée en deux groupes de méthodes :
  - Les méthodes exposant les objets métiers :
    - int getSize()
      renvoie le nombre d'objet de la liste
    - Object getElementAt(int index)
      renvoie un élement à un index donné
  - Les méthodes gérant les vues sous forme de listener :
    - addListDataListener(ListDataListener l)
      enregistre une vue pour quelle soit avertie des changements
    - removeListDataListener(ListDataListener l)
      dés-enregistre une vue

Implanter un modèle de liste

- Voici un modèle simple affichant 100 000 000 valeurs.

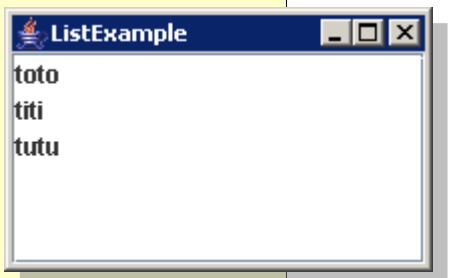
```
public class FirstListModel implements ListModel {
  public int getSize() {
    return 100000000;
  }
  public Integer getElementAt(int index) {
    return index; // boxing
  }
  public void addListDataListener(ListDataListener l) {
  } // ne rien faire
  public void removeListDataListener(ListDataListener l) {
  } // ici aussi
  public static void main(String[] args) {
    ListModel model=new FirstListModel();
    JList list=new JList(model);
    list.setPrototypeCellValue(100000000);

    JFrame frame=new JFrame("FirstListModel");
    frame.setContentPane(new JScrollPane(list));
    frame.setSize(300,200);
    frame.setVisible(true);
  }
}
```



Implanter un modèle de liste stockant des données

- Voici un modèle simple de list utilisant l'interface des collections **java.util.List**.

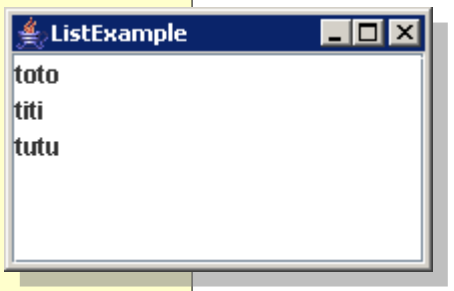
```
import java.util.List;
import javax.swing.AbstractListModel;

public class MyListModel<T> implements ListModel {
  public MyListModel(List<T> list) {
    this.list=list;
  }
  public int getSize() {
    return list.size();
  }
  public T getElementAt(int index) {
    return list.get(index);
  }
  public void addListDataListener(ListDataListener l) {
  } // ne rien faire
  public void removeListDataListener(ListDataListener l) {
  } // ici aussi
  private final List<T> list;
}
```



Implanter un modèle de liste stockant des données (2)

- On passe lors de la construction du modèle de liste la liste des arguments de ligne de commande

```
public static void main(String[] args) {
  List<String> arguments=Arrays.asList(args);
  ListModel model=new MyListModel<String>(arguments);

  JList list=new Jlist(model);

  JFrame frame=new JFrame("MyListModel");
  frame.setContentPane(new JScrollPane(list));
  frame.setSize(400,300);
  frame.setVisible(true);
}
```



Implanter un modèle de liste à l'envers

- Voici un modèle affichant la liste à l'envers.

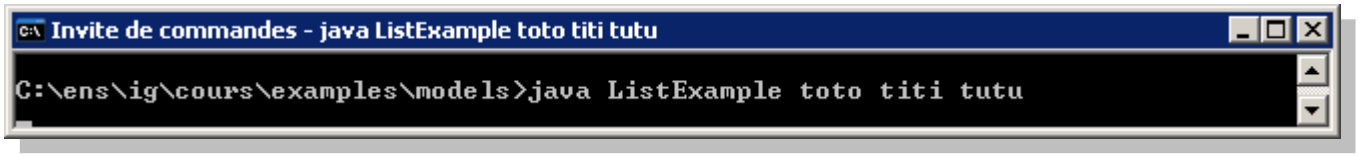
```
import java.util.List;
import javax.swing.AbstractListModel;

class ReverseListModel<T> implements ListModel {
  public ReverseListModel(List<T> list) {
    this.list=list;
  }
  public int getSize() {
    return list.size();
  }
  public T getElementAt(int index) {
    return list.get(getSize()-index-1);
  }
  ... // manque la gestion des listeners
  private final List<T> list;
}
```

- **getElementAt**() renvoie les objets en calculant l'index miroir

Implanter un modèle de liste à l'envers (2)

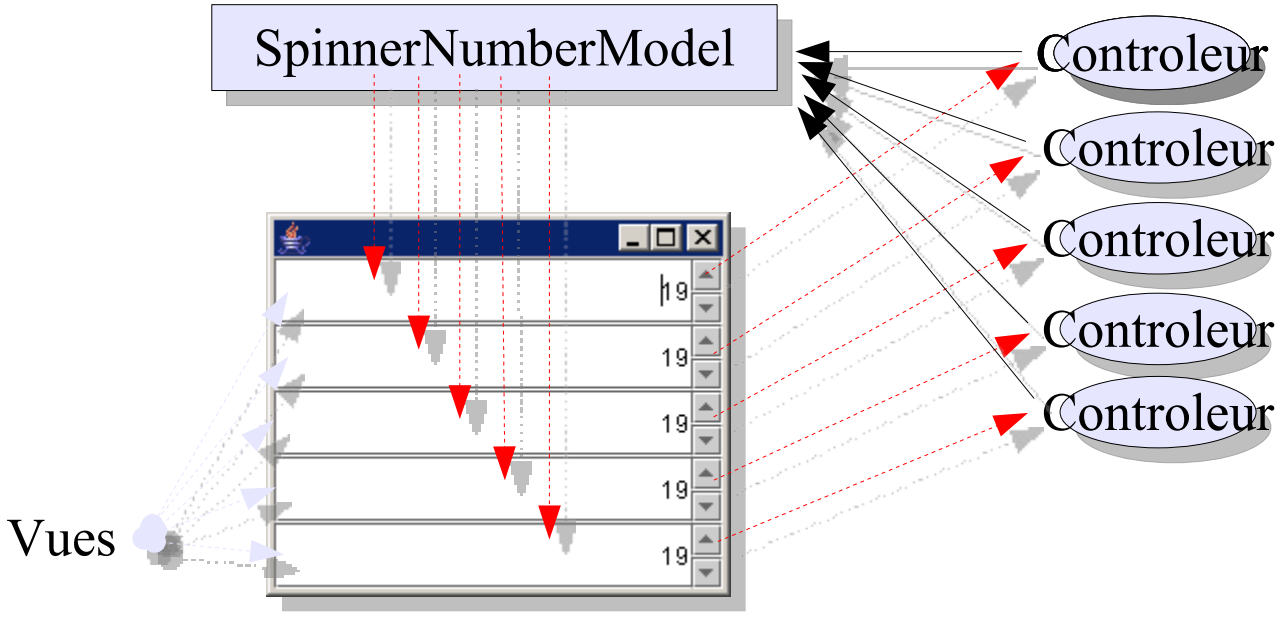
- La méthode main affichant une vue pour chaque modèle.

```
public static void main(String[] args) {
  ArrayList<String> list=new ArrayList<String>();
  Collections.addAll(list,args);

  JList view=new JList(new MyListModel<String>(list));
  JList view2=new JList(new ReverseListModel<String>(list));

  JFrame frame=new JFrame();
  frame.setContentPane(
    new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
      new JScrollPane(view),
      new JScrollPane(view2)));
  frame.pack();
  frame.setVisible(true);
}
```



Relation entre les controleurs et le modèle

- Un **contrôleur** s'**enregistre sur la vue** (ou sur un composant à coté de la vue) pour recevoir les évènements (notification) lorsqu'un utilisateur souhaite modifier celle-ci.



Modifier une liste dynamiquement

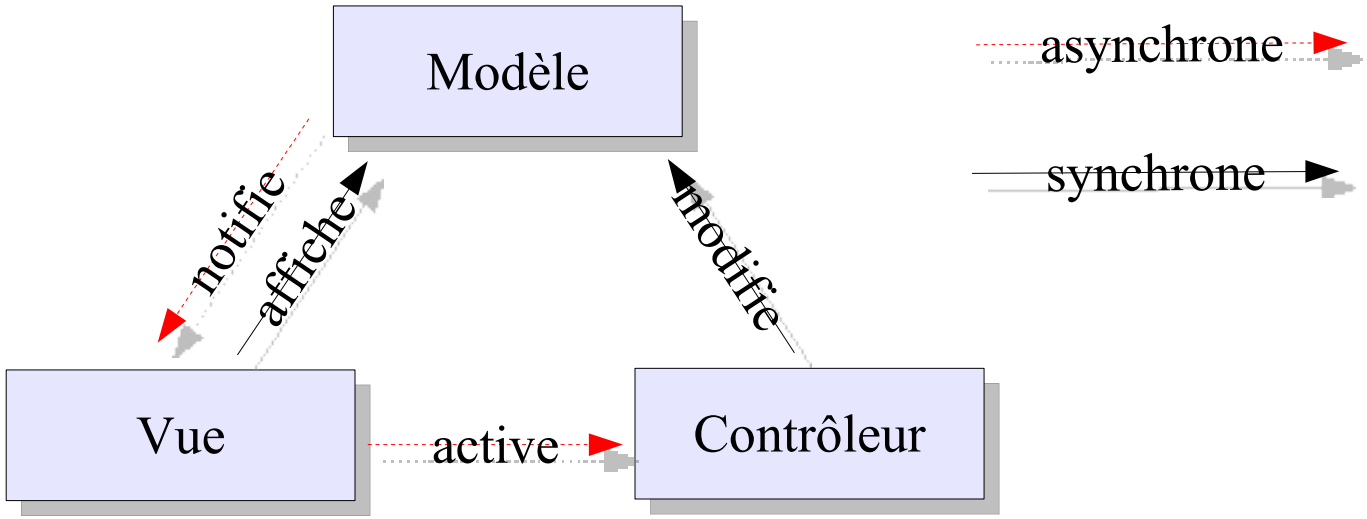
- On souhaite pouvoir ajouter dynamiquement des valeurs

```
public static void main(String[] args) {
  final ArrayList<String> list=new ArrayList(Arrays.asList(args));
  ListModel model=new MyListModel<String>(list);
  JList view=new JList(model);
  // controleur
  JButton button=new JButton("add \"hello\"");
  button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      list.add("hello"); // modifie la liste
    }
  });
  JFrame frame=new JFrame("ListExample (qui marche pas !)");
  frame.getContentPane().add(new JScrollPane(view));
  frame.getContentPane().add(button,BorderLayout.SOUTH);
  ...
}
```

- La vue **n'est pas raffraichie !!**

Modification du modèle

- Lorsque le contrôleur modifie les données metiers **à travers** le modèle, le modèle notifie alors la modification aux vues.



- La notification de la modification est fait sous forme d'un évènement indiquant uniquement ce qui a été modifié

Modification du modèle (2)

- Un modèle doit donc implanter un mécanisme de listener pour que les vues s'enregistre pour être averties de toutes modifications

- Pour le modèle de liste **`ListModel`**, il faut implanter les méthodes :
  - addListDataListener(ListDataListener l)
  - removeListDataListener(ListDataListener l)

- De plus pour chaque modification sur le modèle, il faut pour l'ensemble des listeners enregistrés transmettre un évènement indiquant le changement.

Modification du modèle (2)

- **ListDataListener** est une interface définissant :
  - void contentsChanged(ListDataEvent e)
    le contenue d'un interval de lignes a changé
  - void intervalAdded(ListDataEvent e)
    un interval de lignes a été ajouté
  - void intervalRemoved(ListDataEvent e)
    un interval de lignes a été retiré
- **ListeDateEvent** définie :
  - une source (le modèle par convention)
  - Un type (CONTENT_CHANGED, CONTENT_ADDED, CONTENT_REMOVED)
  - Deux index définissant un interval [index1,index2]

Modification du modèle (4)

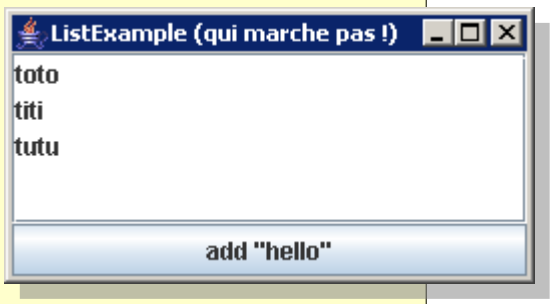
- Exemple de gestion des listeners sur le modèle

```
public class MyListModel implements ListModel {
  ...
  public void addListDataListener(ListDataListener l) {
    listeners.add(l);
  }
  public void removeListDataListener(ListDataListener l) {
    listeners.remove(l);
  }
  public void add(T t) {
    list.add(t);
    int row=list.size()-1;
    ListDataEvent event=
      new ListDataEvent(this,ListDataEvent.INTERVAL_ADDED,row,row);
    for(int i=listeners.size();--i>=0;)
      listeners.get(i).intervalAdded(event);
  }
  private final ArrayList<ListDataListener> listeners=
    new ArrayList<ListDataListener>();
}
```

Modification du modèle (5)

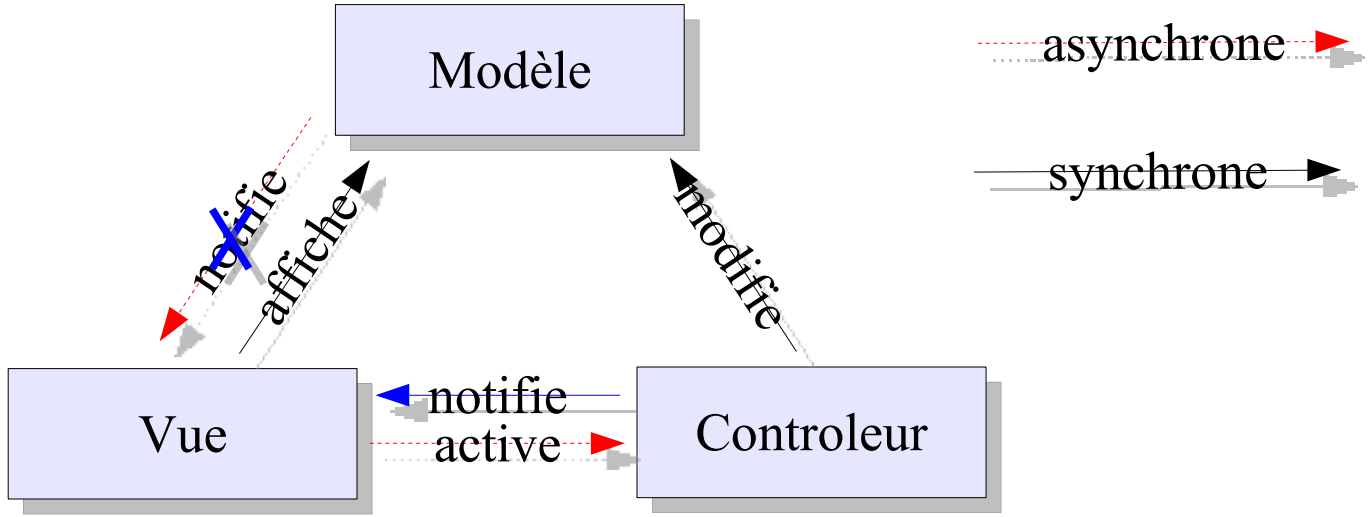
- Le contrôleur appele alors les méthodes de modification sur le modèle

```
public static void main(String[] args) {
  ArrayList<String> list=new ArrayList(Arrays.asList(args));
  final ListModel model=new MyListModel<String>(list);
  JList view=new JList(model);
  // controleur
  JButton button=new JButton("add \"hello\"");
  button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      model.add("hello"); // modifie le modèle
    }
  });
  JFrame frame=new JFrame("ListExample (qui marche)");
  frame.getContentPane().add(new JScrollPane(view));
  frame.getContentPane().add(button,BorderLayout.SOUTH);
  ...
}
```



Relation contrôleur/vue

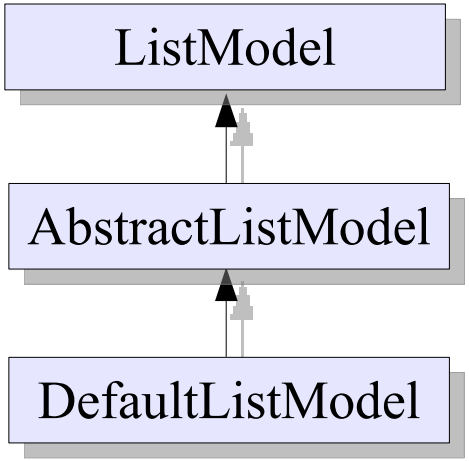
- Pourquoi le controleur (qui initie l'action) n'indique pas directement aux vues la modification ?



- Car il ne sait pas ce qui a changé de façon fine
- Car il peut y avoir plusieurs vues et contrôleurs

La gestion de la notification des changements

- Pour la pluspart des modèles, la gestion des listeners correspondant aux vues existe déjà.



- Pour les listes, La classe abstraite **`AbstractListModel`** implante add/removeListDataListener() et fourni des méthodes fire* permettant de notifier tous les listeners enregistrés
  - addListDataListener(ListDataListener l)
  - removeListDataListener(ListDataListener l)
  - ListDataListener[] getListDataListeners()
  - fireContentsChanged(Object source, int index0, int index1)
  - fireIntervalAdded(Object source, int index0, int index1)
  - fireIntervalRemoved(Object source, int index0, int index1)

*Les méthodes fire**

- Le nom et la signature exacte des méthodes **fire*** dépend du type de modèles
- Les méthodes **fire*** permettent de lancer un évènement qui sera distribué à l'ensemble des listeners enregistrés.
- Ces méthodes ont une visibilité **protected** pour éviter qu'une classe autre qu'un implantant le modèle ne puisse notifier les vues.

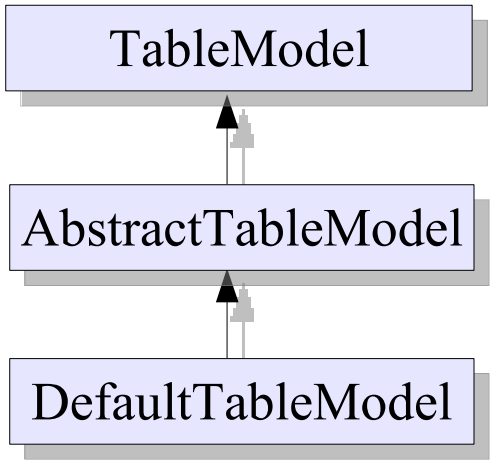
En utilisant AbstractListModel

- Lors d'une modification de la liste, le modèle doit mettre à jour les données et signaler la modification aux vues.

```
public class MyListModel<T> extends AbstractListModel {
  public MyListModel(List<T> list) {
    this.list=list;
  }
  public int getSize() {
    return list.size();
  }
  public T getElementAt(int index) {
    return list.get(index);
  }
  public void add(T t) {
    int index=list.size();
    list.add(t);
    fireIntervalAdded(this, index, index);
  }
  private final List<T> list;
}
```

Les modèles par défaut

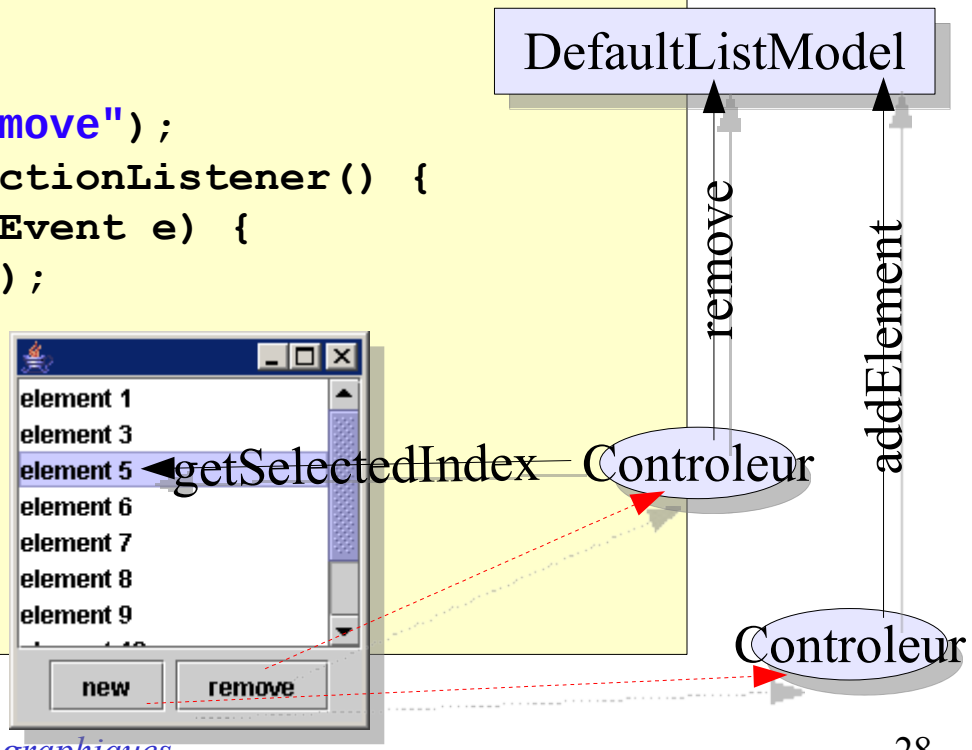
- Chaque interface correspondant à un modèle de swing possède une implantation par défaut
- Chaque implantation propose pour un type de modèle une implantation la plus usuel du stockage des classes métiers. Toute modification génére une notification.
- Exemple :
  - pour ListModel, DefaultListModel stocke les classes métiers dans un Vector (bof!)
  - pour TreeModel, DefaultTreeModel stocke les classes métiers sous forme d'un arbre
  - Pour TableModel, DefaultTableModel stocke les classes métiers sous forme de Vector de Vector (beurk!)



Controleurs utilisant un modèle par défaut

- Ici, la vue possède plusieurs contrôleurs.

```
final DefaultListModel model=new DefaultListModel();
final JList list=new JList(model);
JButton newButton=new JButton("new");
newButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    model.addElement("element "+count++);
  }
});
JButton removeButton=new JButton("remove");
removeButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    int index=list.getSelectedIndex();
    if (index!=-1)
      model.remove(index);
  }
});
JPanel panel=new JPanel();
panel.add(newButton);
panel.add(removeButton);
```



Le Renderer

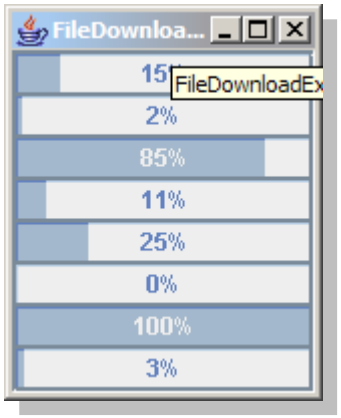
- En Swing, les vues utilise un **seul** objet graphique pour afficher l'ensemble des valeurs du modèle.

- Le *renderer* est un objet implantant une interface différente pour chaque vue (`ListCellRenderer`, `TableCellRenderer`, etc.) qui possède une méthode (`getListCellRenderer()`, etc.) qui retourne le composant graphique définissant l'aspect graphique des cellules.

- Ce composant est successivement placé à chaque endroit où les données doivent apparaitre.

Le renderer

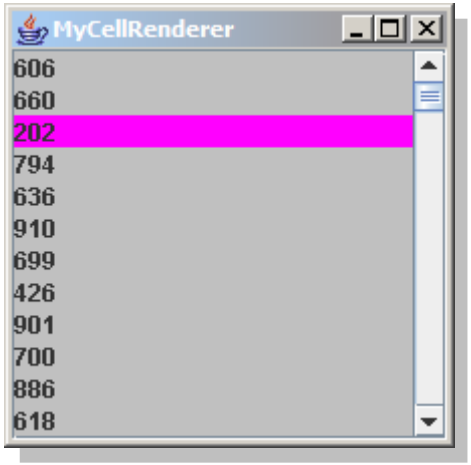
- Le renderer est responsable :
  - De renvoyer le composant de rendu
  - De paramétrer le composant en fonction de l'objet à rendre
- En règle générale, le renderer initialise
  - la couleur de fond du composant en fonction de la sélection
  - Le bord du composant en fonction du focus clavier
  - Le texte en fonction de l'objet à rendre (toString())
- Exemple en utilisant d'une liste avec un renderer renvoyant un JProgressBar comme composant de rendu



L'exemple du renderer d'une liste (2)

- La méthode **`setCellRenderer()`** sur une **`JList`** permet de changer le gestionnaire de rendu.

```
public static void main(String[] args) {
  Integer[] ints=new Integer[1000];
  for(int i=0;i<ints.length;i++)
    ints[i]=Integer.valueOf(i);
  Collections.shuffle(Arrays.asList(ints));

  JList list=new JList(ints);
  list.setCellRenderer(new MyCellRenderer());

  JFrame frame=new JFrame("MyCellRenderer");
  frame.setContentPane(new JScrollPane(list));
  frame.pack();
  frame.setVisible(true);
}
```

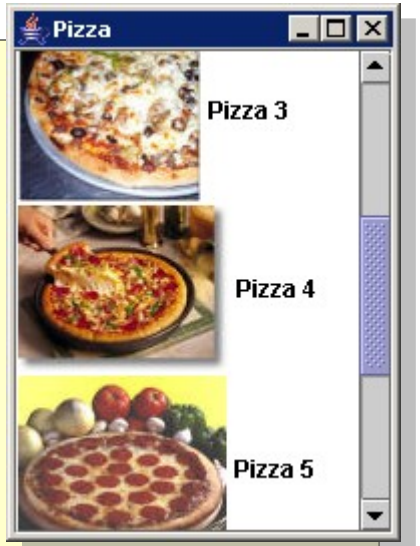


*Default*CellRenderer*

- Chaque interface *renderer* possèdent une implantation par défaut (**ListCellRenderer** -> **DefaultListCellRenderer**)
- Celle-ci rédéfinie certaines méthodes internes à Swing pour accélérer l'affichage.
- L'implantation par défaut hérite de **`JLabel`** et implante l'interface du *renderer* (erreur de *design*).
- La méthode **`get*CellRenderer()`** renvoie **this** comme composant graphique.

DefaultListCellRenderer

- Le **DefaultListCellRenderer** hérite d'un **JLabel**.




```
public JList createList() {
  JList list=new JList(new AbstractListModel() {
    public int getSize() {
      return icons.length;
    }
    public String getElementAt(int index) {
      return "Pizza "+(index+1);
    }
  });
  list.setVisibleRowCount(5);
  list.setCellRenderer(new DefaultListCellRenderer() {
    public Component getListCellRendererComponent(JList list,Object value,int index,
      boolean isSelected,boolean cellHasFocus) {

      super.getListCellRendererComponent(list,value,index,isSelected,cellHasFocus);
      setIcon(icons[index]);
      return this;
    }
  });
  return list;
}
final Icon[] icons=getPizzaIcons();
```

En résumé

- Le *design pattern* ***MVC*** permet de séparer la partie métier du logiciel de la partie interface graphique.
- Il peut y avoir :
  - Plusieurs vues pour un même modèle.
  - Plusieurs controleurs pour une même vue.
- Le modèle est différent à chaque type de vue, **`ListModel`** pour les **`JList`**, etc.
- Les changements sur le modèle sont notifiés (évènements) après un enregistrement au préalable.