Interface Graphique en Java 1.6

MVC et listes

Sébastien Paumier

Le MVC

- MVC=Modèle Vues Contrôleurs
- architecture permettant de séparer proprement la gestion des données et l'affichage des données:
  - LE modèle gère les données
  - les vues les affichent
  - les contrôleurs demandent au modèle de modifier les données

Le modèle

- défini par une interface indiquant ce qu'on peut faire
- exemple: **`ListModel`**

```
void addListDataListener(ListDataListener l)
Object getElementAt(int index)
int getSize()
void removeListDataListener(ListDataListener l)
```

- donc, un modèle de liste permet d'avoir la taille, un élément donné et des listeners pour savoir quand ça change

Le modèle

- l'interface définit ce dont les vues peuvent avoir besoin, pas ce qui concerne la mise à jour des données:
  - pas de **`add`** dans **`ListModel`**
  - ce sera le rôle d'une implémentation comme **`DefaultListModel`**
- toujours bien séparer l'interface de l'implémentation, pour éviter de forcer la main sur la façon de gérer les données

Le modèle

- le `xxxListener` doit fournir les méthodes informant des changements possibles
- exemple: `ListDataListener`

```
public void contentsChanged(ListDataEvent e)
public void intervalAdded(ListDataEvent e)
public void intervalRemoved(ListDataEvent e)
```

- le modèle doit appeler les `fire...` correspondant quand c'est nécessaire (cf. cours précédent)

Le modèle

- le `xxxEvent` doit fournir les renseignements nécessaires sur le changement qui a eu lieu
- exemple: `ListDataEvent`

  `int getIndex0()`: début de l'intervalle concerné (bord inclus)

  `int getIndex1()`: fin de l'intervalle concerné (bord inclus)

  `int getType()`: type d'événement (modification, ajout, suppression)

Les vues

- une vue est un mode de représentation des données
- souvent un objet graphique, mais ça pourrait être autre chose:
  - contenu d'un fichier mappé
  - image
  - pages web
  - etc

Les vues

- une vue a pour but de montrer les données que son modèle lui fournit
- pour cela, elle met un listener sur son modèle pour savoir quand elle doit se mettre à jour
- fait implicitement pour les objets usuels de Swing, quand on passe un modèle au constructeur d'une vue:

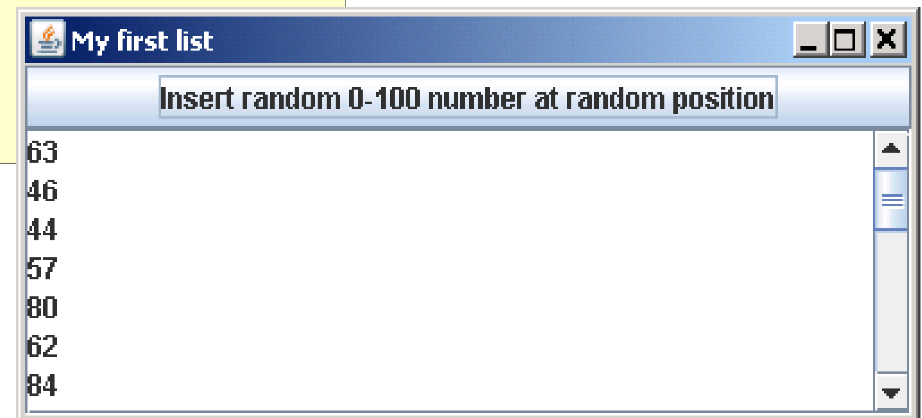
```
final DefaultListModel model=new DefaultListModel();
JList list=new JList(model);
```

Les vues

- exemple: liste de nombres avec insertion

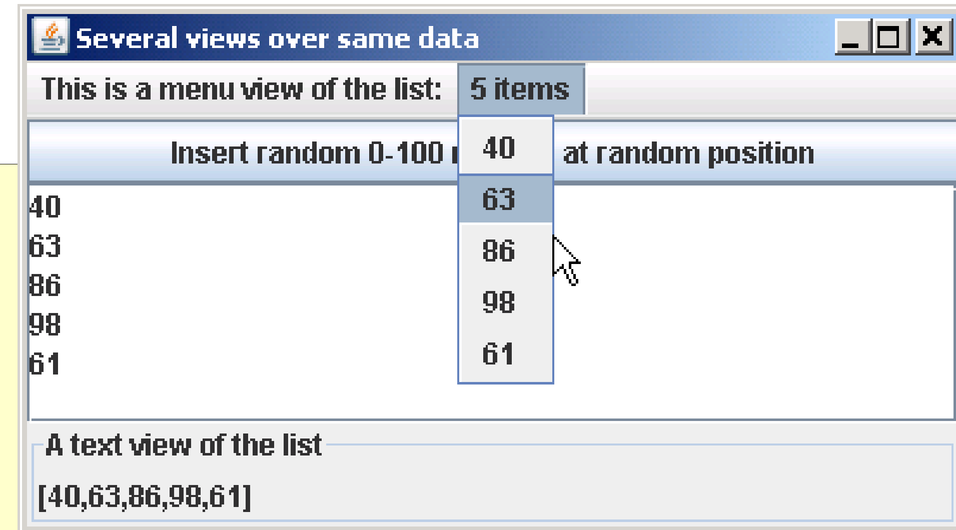
```
final DefaultListModel model=new DefaultListModel();
JList list=new JList(model);
JButton b=new JButton("Insert random 0-100 number at random position");
b.addActionListener(new ActionListener() {
    Random random=new Random();
    @Override
    public void actionPerformed(ActionEvent e) {
        int index=random.nextInt(model.getSize()+1);
        int value=random.nextInt(101);
        model.add(index,value);
    }
});
```

on est dans la thread Swing, pas de problème



Les vues

- on peut ajouter d'autres vues sur les mêmes données:




```java
final JMenu menu=new JMenu("0 item");
model.addListDataListener(new ListDataListener() {
    @Override
    public void contentsChanged(ListDataEvent e) {
        /* Nothing to do because we only add data */
    }

    @Override
    public void intervalAdded(ListDataEvent e) {
        int n=model.getSize();
        menu.setText(n+" item"+(n>1?"s":""));
        for (int i=e.getIndex0();i<=e.getIndex1();i++) {
            menu.add(new JMenuItem(model.getElementAt(i).toString()),i);
        }
    }

    @Override
    public void intervalRemoved(ListDataEvent e) {
        /* Nothing to do because we only add data */
    }
});
```

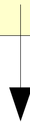
Les contrôleurs

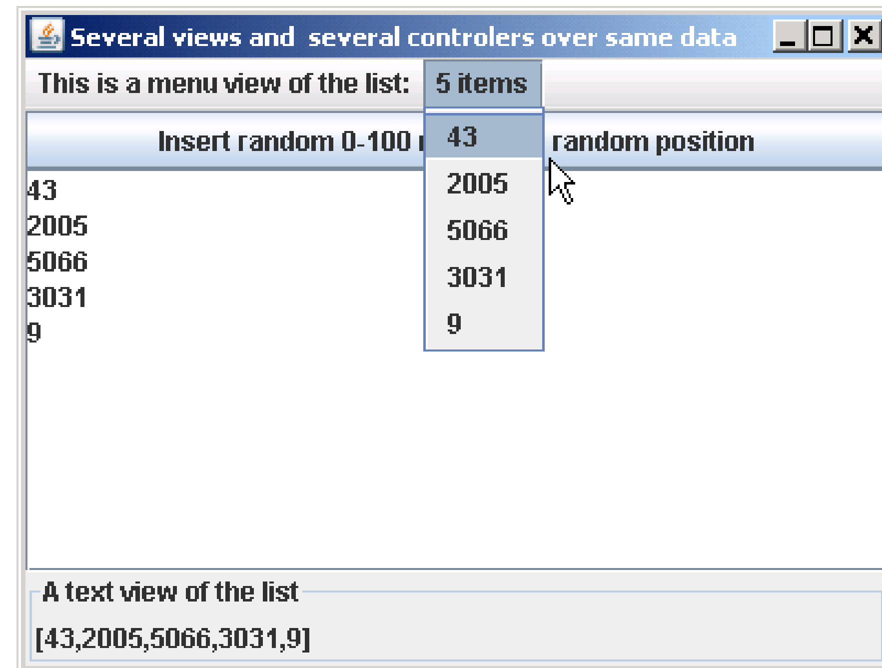
- contrôleur="chose" qui demande au modèle de modifier les données
- dans l'exemple précédent, c'était le bouton
- on peut avoir plusieurs contrôleurs
- ils doivent **toujours** poster leurs demandes au modèle dans la thread Swing!
- exemple: `Timer` qui incrémente chaque seconde de 1000 un élément tiré au hasard

Les contrôleurs

- nécessite une modification de la vue menu pour tenir compte des changements:

```
@Override
public void contentsChanged(ListDataEvent e) {
    for (int i=e.getIndex0();i<=e.getIndex1();i++) {
        JMenuItem item=(JMenuItem) menu.getItem(i);
        item.setText(model.getElementAt(i).toString());
    }
}
```

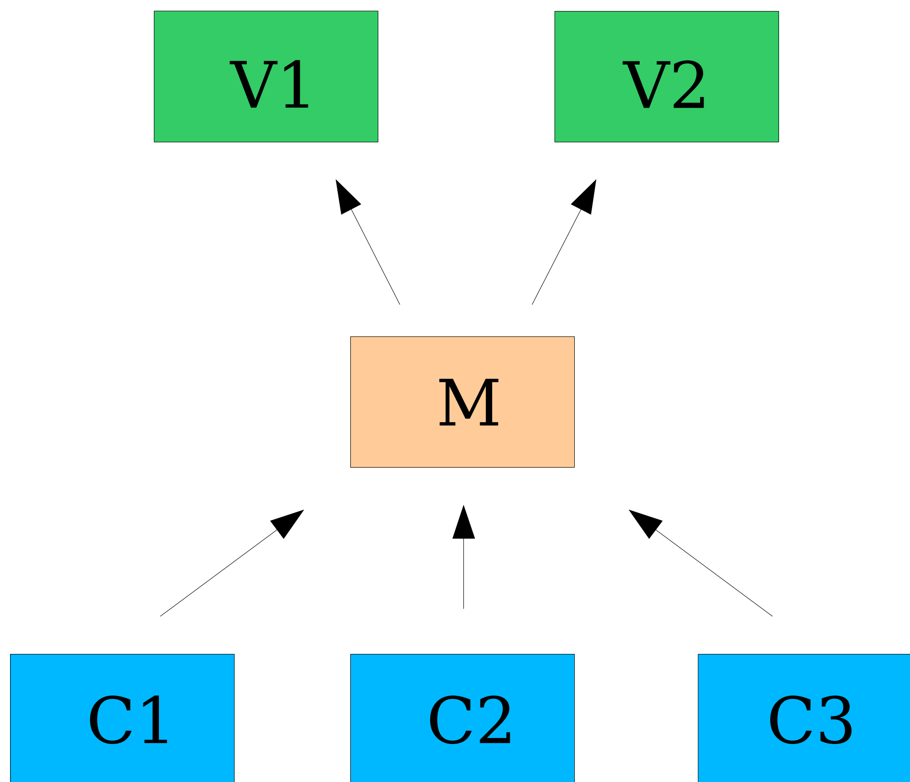




le **Timer** n'ajoute ni ne retire d'élément: il en modifie un existant; c'est donc **contentChanged** qui est concernée

En résumé

- le MVC, c'est facile:



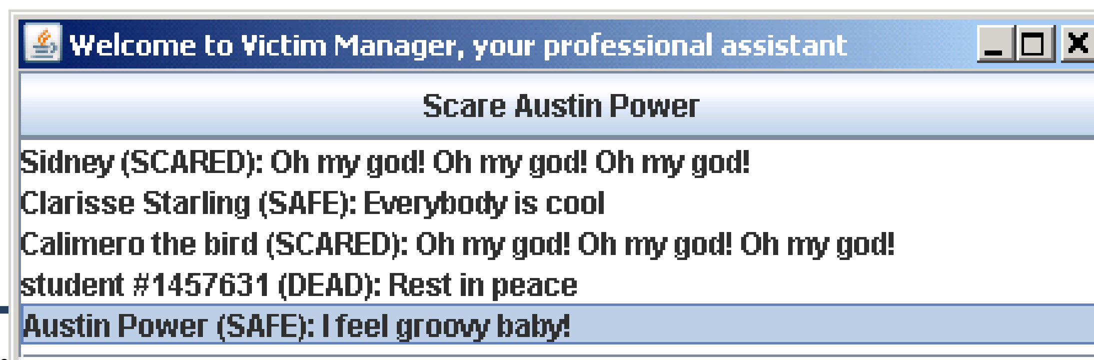
les vues se rafraîchissent en redemandant les valeurs nécessaires au modèle

le modèle prévient les vues des changements avec les méthodes `fire...`

les contrôleurs demandent des modifications des données au modèle

Un exemple complet

- projet: un Victim Manager pour le syndicat des tueurs en série
- besoins client:
  - tenir à jour une liste de victimes, définies par un nom et un état (*safe, scared, dead*)
  - on doit pouvoir modifier l'état d'une victime (*safe* → *scared* ou *scared* → *dead*)



Les données

- les données seront représentées par le type **`Victim`**:

```java
private final String name;
private State state;

public Victim(String name,State state) {
    this.name=name;
    this.state=state;
}

public State getState() { return state; }
public String getName() { return name; }

/* VALUES saves defensive array copies */
private static final State[] VALUES = State.values();

/* Only classes of the current package will be able to invoke this method */
void upgrade() {
    if (state == State.DEAD) throw new IllegalStateException("Cannot upgrade a dead victim !");
    state = VALUES[state.ordinal() + 1];
}

@Override public String toString() {
    return name + " (" + state + "): " + getComment();
}

public String getComment() {
    return state.getComment(name.hashCode());
}
```

Les données

- l'état est défini avec une jolie **enum**, afin de gérer de beaux commentaires:

```
public enum State {

    SAFE("smile.png", "Life is beautiful", "The world is great", "Everybody is cool",
            "I feel groovy baby!"),
    SCARED("scared.png", "Oh my god! Oh my god! Oh my god!", "You'll never get me!",
            "I'm scared!", "Please save me!"),

    DEAD("rip.png", "Rest in peace", "dead", "horribly murdered");

    private final ImageIcon icon;
    private final String[] comment;

    private State(String iconName, String... comment) {
        this.icon = new ImageIcon(State.class.getResource(iconName));
        this.comment = comment;
    }

    public ImageIcon getIcon() {
        return icon;
    }

    public String getComment(int n) {
        return comment[Math.abs(n) % comment.length];
    }
}
```

Le modèle

- il nous faut un modèle de liste permettant l'ajout et la modification de victimes:
  - **VictimListModel** qui va hériter de **AbstractListModel** (pas de **DefaultListModel**, car on aurait plein de méthodes dont on ne veut pas !)
- on va gérer les données avec une **ArrayList<Victim>**:

```
ArrayList<Victim> victims=new ArrayList<Victim>();
```

Le modèle

- on ajoute et obtient les victimes normalement:

```java
@Override
public Victim getElementAt(int index) {
    return victims.get(index);
}

@Override
public int getSize() {
    return victims.size();
}

public void addVictim(Victim v) {
    int size=getSize();
    victims.add(v);
    fireIntervalAdded(this,size,size);
}
```

le modèle prévient de l'ajout

Le modèle

- comme on a restreint la visibilité de **`upgrade`**, on met **`Victim`** et **`VictimListModel`** dans un même sous-package, pour que seul le modèle puisse modifier les données
- avantage: il est au courant de tous les changements et peut donc faire tous les **`fireContentsChanged`** nécessaires proprement

```
public void upgradeVictim(int index) {
    Victim v=victims.get(index);
    v.upgrade();
    fireContentsChanged(this,index,index);
}
```

Un contrôleur

- pour modifier l'état d'une victime de la liste, on utilise un bouton qui agit sur la victime sélectionnée

```
final JButton b=new JButton("Choose a victim");
b.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        int n=list.getSelectedIndex();
        if (n==-1) {
            /* Should not occur, because the button is supposed to
             * be disabled in that case */
            return;
        }
        model.upgradeVictim(n);
    }
});
```

on demande au modèle de modifier la victime au lieu de le faire soi-même

Mise à jour du contrôleur

- pour modifier l'état du bouton en fonction de la victime sélectionnée, on utilise un **ListSelectionListener**:

```
list.addListSelectionListener(new ListSelectionListener() {
    @Override public void valueChanged(ListSelectionEvent e) {
        int selected=list.getSelectedIndex();
        updateButton(b,selected,model);
    }
});
```

- ici, le contrôleur dépend de la vue

Mise à jour du contrôleur

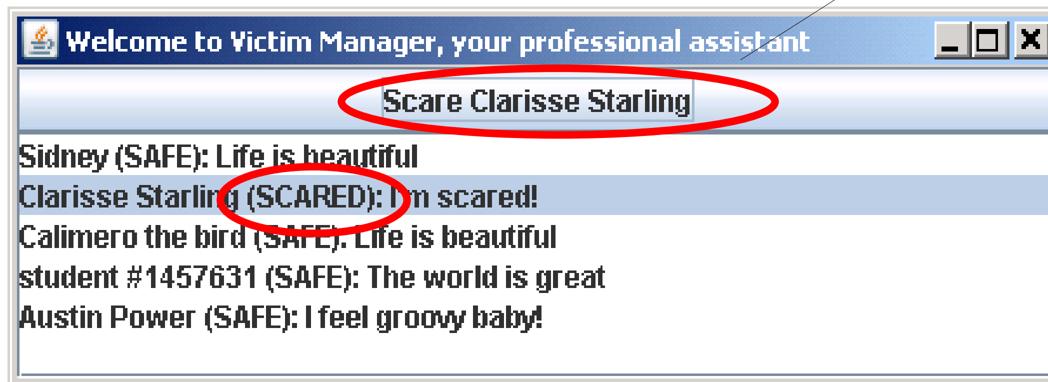
- mise à jour du bouton:

```
static void updateButton(JButton b,int selected,VictimListModel model) {
    if (selected == -1) {
        b.setText("Choose a victim");
        b.setEnabled(false);
        return;
    }
    Victim v = model.getElementAt(selected);
    Victim.State s = v.getState();
    switch (s) {
    case SAFE:
        b.setEnabled(true);
        b.setText("Scare " + v.getName());
        break;
    case SCARED:
        b.setEnabled(true);
        b.setText("Kill " + v.getName());
        break;
    case DEAD:
        b.setText(v.getName() + ": done");
        b.setEnabled(false);
        break;
    }
}
```

Rafraîchissement du contrôleur

- problème: quand on modifie une victime, le bouton n'est pas rafraîchi

devrait proposer "Kill"



- pourquoi ? parce que le bouton se rafraîchit en fonction de la sélection et qu'elle n'a pas changé

Rafraîchissement du contrôleur

- solution: écouter le modèle pour savoir quand la cellule sélectionnée change

```
/* We listen to the model in order to test if the content of the
 * selected cell has changed */
model.addListDataListener(new ListDataListener() {
    @Override
    public void contentsChanged(ListDataEvent e) {
        int selected = list.getSelectedIndex();
        if (selected == e.getIndex0()) {
            updateButton(b,selected,model);
        }
    }

    @Override
    public void intervalAdded(ListDataEvent e) {
        /* Does not change the selected cell */
    }

    @Override
    public void intervalRemoved(ListDataEvent e) {
        /* If the selected cell has been removed, then the ListSelectionListener
         * will already inform the button that it must refresh */
    }
});
```

Le renderer

- en Swing, une vue utilise un unique composant pour dessiner ses éléments: le *renderer*
- le renderer est mis à jour pour chaque élément à afficher, puis il est placé au bon endroit et dessiné
- il ne réagit pas aux événements!
- pour une **`JList`**, c'est par défaut un **`DefaultListCellRenderer`** qui hérite de **`JLabel`**

Le renderer

- le renderer d'une liste invoque **`setText`** avec ce que retourne **`toString`** pour chaque élément
- on peut redéfinir le renderer avec:
  - **`setCellRenderer(ListCellRenderer r)`**
- un **`ListCellRenderer`** doit renvoyer un **`Component`**:

```
public Component getListCellRendererComponent(JList list,
     Object value, int index, boolean isSelected,
     boolean cellHasFocus)
```

Le renderer

- on pourrait donc utiliser autre chose qu'un `JLabel`
- dans notre exemple, on veut ajouter une image et de la couleur
- le `DefaultListCellRenderer` héritant de `JLabel`, il nous suffit donc de le personnaliser

Le renderer

- on appelle **super.get...** pour garder le comportement par défaut d'une cellule de liste (sélection):

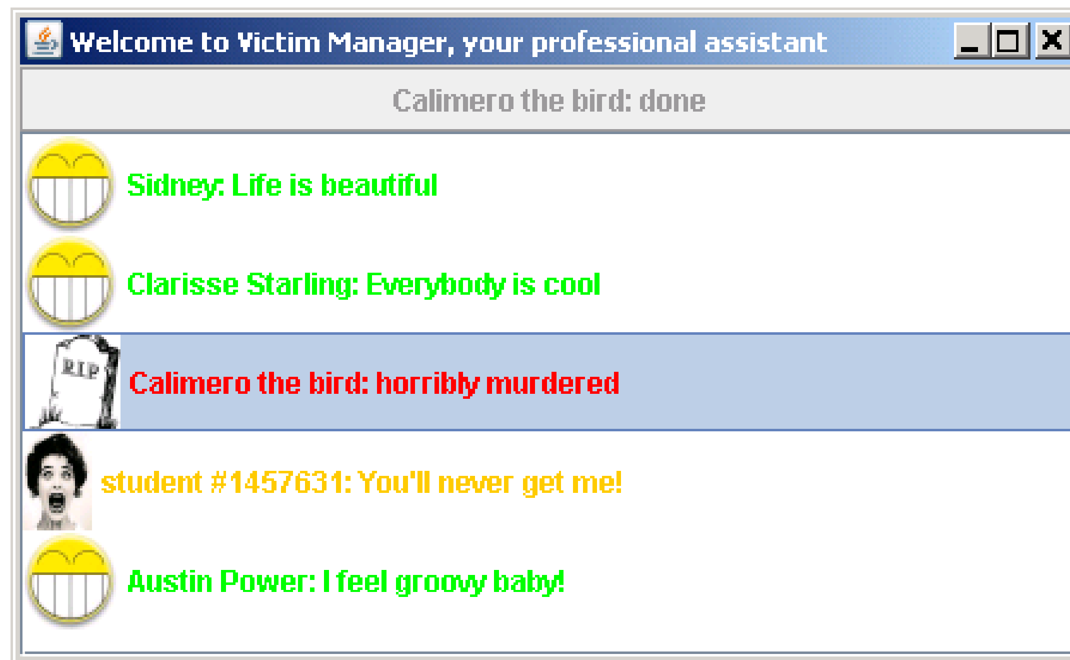
```
list.setCellRenderer(new DefaultListCellRenderer() {
    @Override
    public Component getListCellRendererComponent(JList list, Object value, int index,
                    boolean isSelected, boolean cellHasFocus) {
        /* We call super so that the default behavior (highlight cells
         * when selected) will remain */
        Victim v = (Victim) value;
        super.getListCellRendererComponent(list, v.getName() + ": " + v.getComment(), index,
                    isSelected, cellHasFocus);
        setIcon(v.getState().getIcon());
        switch (v.getState()) {
        case SAFE: setForeground(Color.GREEN); break;
        case SCARED: setForeground(Color.ORANGE); b
        case DEAD: setForeground(Color.RED); break;
        }
        return this;
    }
});
```

on passe notre propre chaîne au lieu de **Victim.toString()**

on associe une image à chaque état

Résultat

- l'aspect de notre liste est maintenant personnalisé:



Un renderer d'arbre

- autre exemple: rendu des noeuds particulier quand on a une classe avec un **main**:

```
JTree tree = new JTree(treeModel);
tree.setCellRenderer(new DefaultTreeCellRenderer() {
    @Override
    public Component getTreeCellRendererComponent(JTree tree, Object value,
                    boolean sel, boolean expanded, boolean leaf, int row, boolean hasFocus) {
        super.getTreeCellRendererComponent(tree, value, sel, expanded, leaf, row, hasFocus);
        DefaultMutableTreeNode node = (DefaultMutableTreeNode) value;
        Object userObject = node.getUserObject();
        if (userObject instanceof Class<?>) {
            Class<?> clazz = (Class<?>) userObject;
            setText(clazz.getSimpleName());
            if (null != getMainMethod(clazz)) {
                setIcon(runIcon);
            }
        }
        return this;
    }
});
```