Cours 3 : Architecture MVC et composants

FIP/Mise à niveau

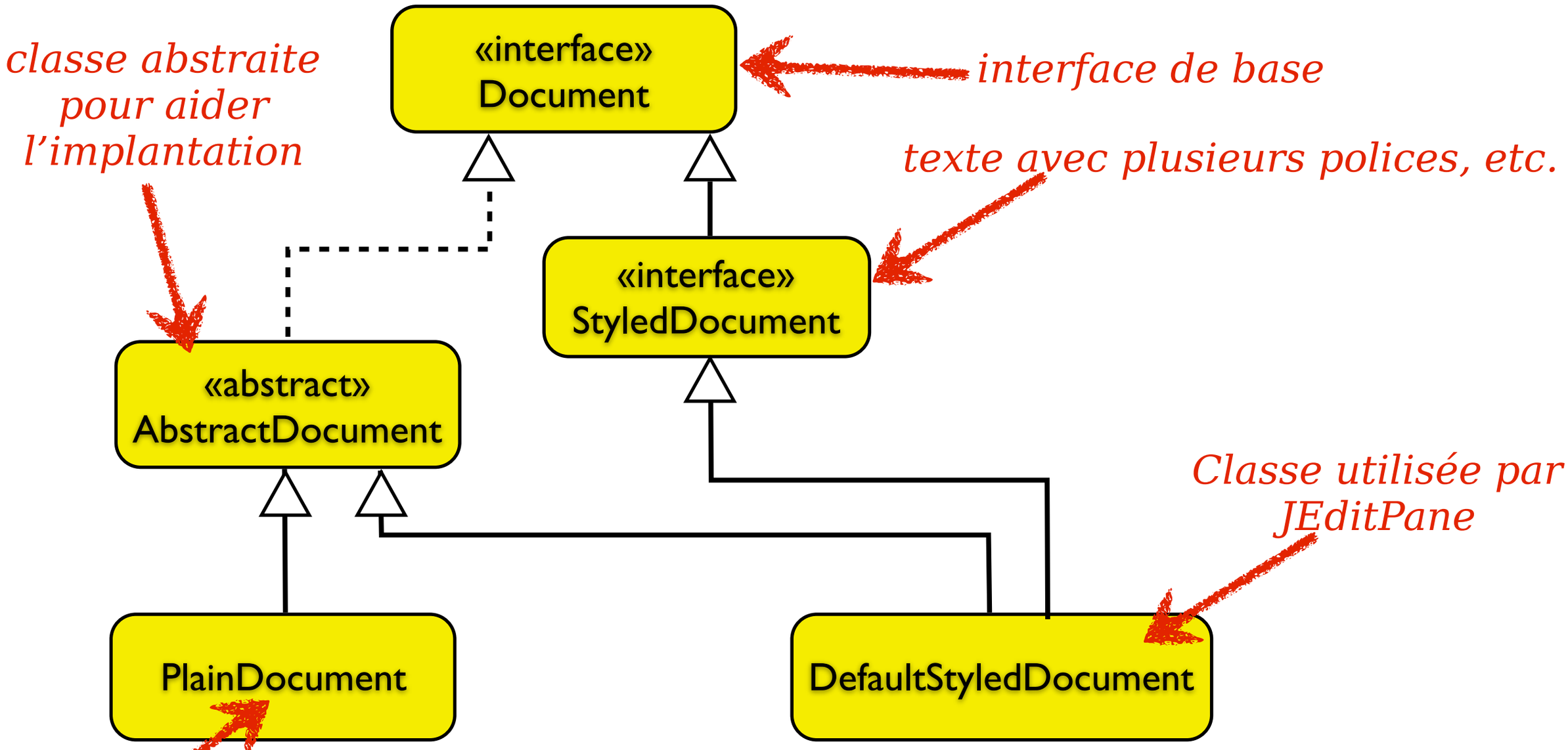
S. Rosmorduc

Études de quelques composants

Composants textuels

- JTextField, JTextArea, JFormatedTextField, JPasswordField, JEditorPane
- leur modèle implémente l'interface Document
  - méthodes getDocument(), setDocument()
- On peut attacher au document un DocumentListener pour être au courant des modifications dans le document

Document (modèle textuel)



En pratique...

- On peut déjà faire des choses sans créer son propre Document:
  - on récupère le document sur le JTextField avec
    - Document doc= textField.getDocument()
    - et on lui attache un documentListener.

DocumentListener

```
public interface DocumentListener {
    /**
     * prévient d'une insertion de texte.
     */
    void insertUpdate(DocumentEvent e);
    /**
     * prévient d'une suppression de texte.
     */
    void removeUpdate(DocumentEvent e)
    /**
     * prévient d'une modification des styles (utilisé uniquement par JEditPane).
     */
    void changedUpdate(DocumentEvent e);
}
```

Exemple

La classe demo a deux champs textes, dont l'un sert à saisir un mot de passe («secret»). On veut afficher «correct» dès que celui-ci est bon.

```
class MonDocListener implements DocumentListener {
    private Demo demo;
    public MonDocListener(Demo demo) {
        this.demo = demo;
    }
    public void insertUpdate(DocumentEvent e) {
        verifierMotDePasse();
    }
    public void removeUpdate(DocumentEvent e) {
        verifierMotDePasse();
    }
    public void changedUpdate(DocumentEvent e) {
        // Jamais appelée
    }
    private void verifierMotDePasse() {
        if ("secret".equals(demo.getField().getText())) {
            demo.getAffichageField().setText("correct");
        } else {
            demo.getAffichageField().setText("incorrect");
        }
    }
}
```

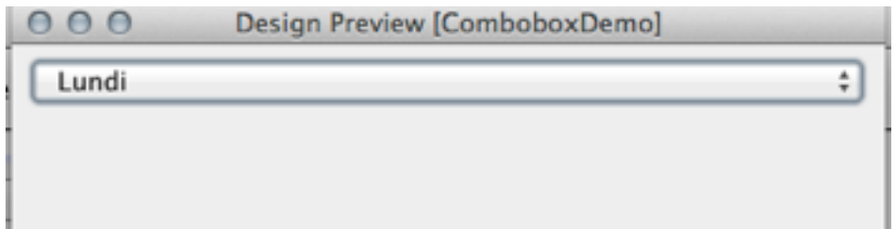
Exemple... mise en place

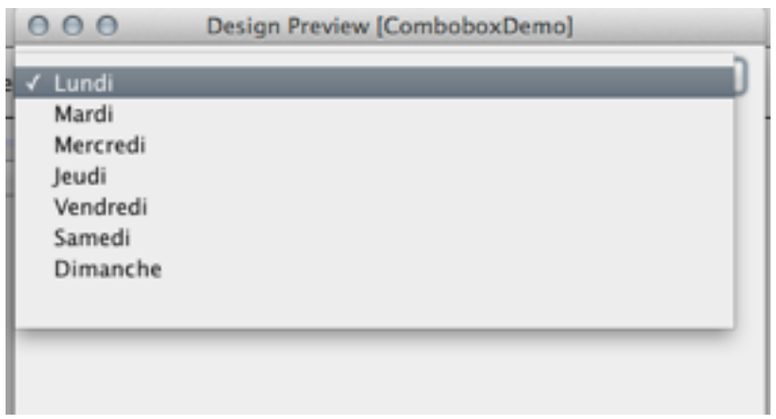
....
Demo demo= new Demo();
demo.getField().**getDocument().addDocumentListener(**
    new MonDocListener(demo));

- Utile par exemple pour déclencher des recherches quand un champ texte est rempli

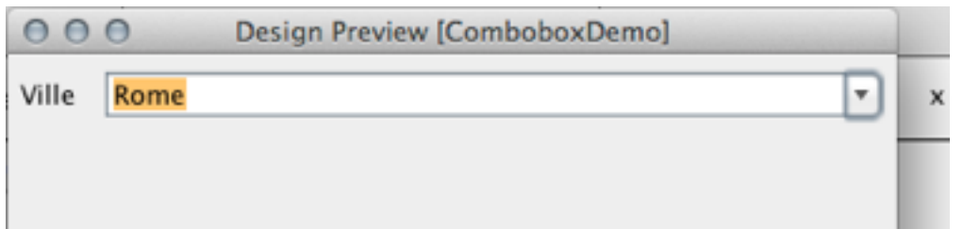
JCombobox

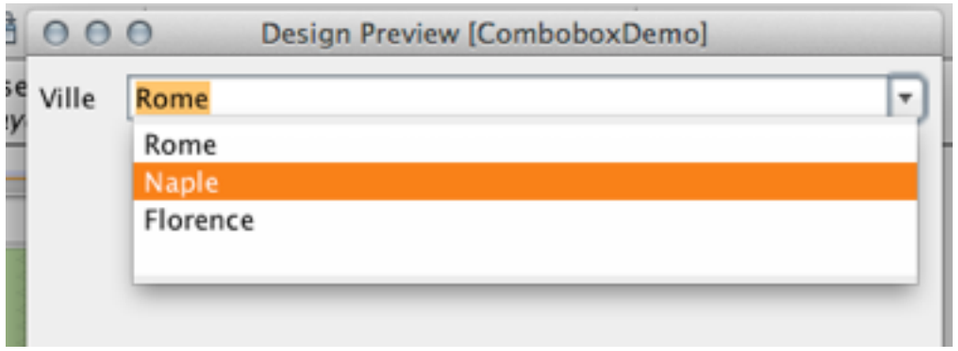
non éditable

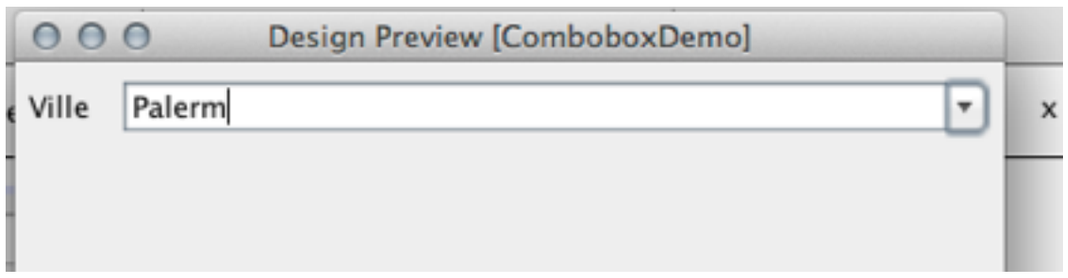




éditable







On peut presque tout changer

JCombobox : modèle

- Interface ComboBoxModel (type générique depuis jdk 1.7! avant : **Object** au lieu de T)
- Implantation par défaut : DefaultComboboxModel
- *Les entrées du modèle sont par défaut visualisées en utilisant toString().*

JCombobox (Constructeurs)

- JComboBox() : construit une combobox avec un DefaultComboboxModel
- JComboBox(ComboBoxModel)
- JComboBox(Object[]) : construit une combobox avec un DefaultComboboxModel qui contient les objets du tableau.
- JComboBox(Vector) JComboBox(Object[]) : construit une combobox avec un DefaultComboboxModel qui contient les objets du Vector (équivalent ArrayList).

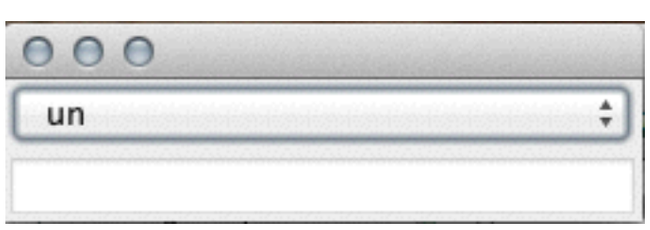
JCombobox (méthodes utiles)

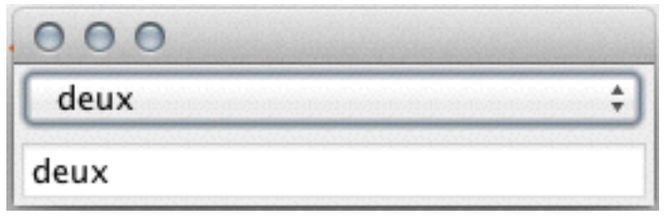
- addActionListener(ActionListener) : l'action listener est appelé après une édition, ou après une sélection. Alternative : il existe aussi une interface ItemListener (plus complexe, voir tutoriel Swing)
- getItemCount() , getItemAt(int) : permet de lister les entrées de liste (mais on peut utiliser le modèle)
- getSelectedItem() / setSelectedItem(Object) : renvoie ou fixe l'item sélectionné ; getSelectedIndex() renvoie sa position dans la liste.
- getModel()/setModel(ComboboxModel) : gère le modèle.
- setEditable(boolean) : l'utilisateur peut-il saisir de nouvelles entrées ?

Exemple

```
public class SimpleCBDemo {
  private JComboBox comboBox;
  private JTextField textField;

  public SimpleCBDemo() {
   String[] tab= {"un", "deux", "trois"};
   comboBox= new JComboBox(tab);
   textField= new JTextField(20);
   comboBox.addActionListener(new RecopierActionListener(this));
   mettreEnPage();
  }

  public void recopier() {
   String texte= (String) comboBox.getSelectedItem();
   textField.setText(texte);
  }
  ....
}

class RecopierActionListener implements ActionListener {
  SimpleCBDemo simpleCBDemo;
  public RecopierActionListener(SimpleCBDemo simpleCBDemo) {
   this.simpleCBDemo = simpleCBDemo;
  }
  public void actionPerformed(ActionEvent arg0) {
   simpleCBDemo.recopier();
  }}
```

Exemple





DefaultComboBoxModel

- Représente l'état de la combobox : liste des choix possibles, et item sélectionné
- méthodes
  - getSelectedItem() ; setSelectedItem(Object): gestion de la sélection
- getSize() ; getElementAt(int) : liste des items
- addElement(Object); insertElementAt(int i) ; removeElementAt(int i); removeElement(Object); removeAllElements() : modification de la liste des items
- lors de grosses modifications, on remplace souvent l'intégralité du modèle plutôt que de modifier son contenu.

Manipulation du modèle à travers la Combobox

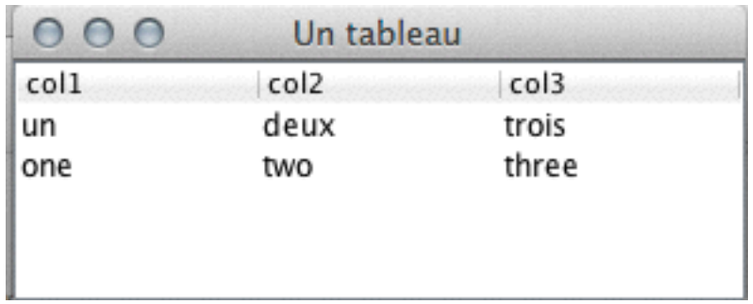
- Plus simple, mais passe par l'interface graphique
- La classe JCombobox fournit les méthodes:
  - addItem(Object item)
  - insertItemAt(Object item, int pos)
  - removeItem(Object item)
  - removeItemAt(int pos)
  - removeAllItems()

Pour aller plus loin...

- Dans le JCombobox, par exemple :
  - on peut créer son propre modèle (par exemple lié à une base de données)
  - on peut remplacer le système de rendu des éléments (Renderer) : ListCellRenderer
  - on peut remplacer l'éditeur (l'objet chargé de prendre en charge la saisie d'une nouvelle entrée) : ComboBoxEditor
- pour la classe JTable, l'écriture de modèles est souvent très utile

JTable

- Affiche un tableau à deux dimensions
- a priori, un nombre (relativement) fixe de colonnes, et un nombre de lignes variables
- utilise un TableModel




```
JFrame frame= new JFrame("Un tableau");
JTable table= new JTable(0, 3);
DefaultTableModel defaultModel= (DefaultTableModel)
                                                table.getModel();
defaultModel.setColumnIdentifiers(new String [] {
                                              col1", "col2", "col3"});
defaultModel.addRow(new String[] {"un", "deux", "trois"});
defaultModel.addRow(new String[] {"one", "two", "three"});
```

TableModel

- Interface qui décrit le contenu d'une table
- En pratique, on étend souvent AbstractTableModel (fournit certaines des méthodes)

TableModel en lecture seule

- trois informations nécessaires:
  - nombre de lignes
    - getRowCount()
  - nombre de colonnes
    - getColumnCount()
  - contenu d'une case
    - getValueAt(int rowIndex, int ColumnIndex)

Exemple : table de personnes

- Une personne:
  - un identifiant numérique (invariable)
  - un nom
  - un prénom

Modèle

```
public class PersonnesTableModelReadOnly extends AbstractTableModel {
    private List<Personne> personnes = new ArrayList<>();
    public PersonnesTableModelReadOnly(Collection<Personne> personnes ) {
        this.personnes= new ArrayList<>(personnes);
    }
   public int getRowCount() {
        return personnes.size();
    }
   public int getColumnCount() {
        return 3;
    }
   public Object getValueAt(int rowIndex, int columnIndex) {
        Personne p = personnes.get(rowIndex);
        switch (columnIndex) {
            case 0:
                return p.getId();
            case 1:
                return p.getNom();
            case 2:
                return p.getPrenom();
            default:
                return "";
        }
    }
}
```

Mise en place

```
public class DemoTablePersonnes {
    JFrame frame= new JFrame("Un tableau");
    JTable table= new JTable();
    public DemoTablePersonnes() {
        table.setModel(new PersonnesTableModelReadOnly(Personnes.getList()));
        frame.add(new JScrollPane(table));
        frame.setVisible(true);
        frame.pack();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
   public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> new DemoTablePersonnes());
    }
}
```

Tout TableModel

- Listeners
  - void addTableModelListener(TableModelListener l)
  - void removeTableModelListener(TableModelListener l)
- méta data
  - Class<?> getColumnClass(int columnIndex)
  - String getColumnName(int columnIndex)
- contenu
  - int getColumnCount()
  - int getRowCount()
  - Object getValueAt(int rowIndex, int columnIndex)
- modification
  - boolean isCellEditable(int rowIndex, int columnIndex)
  - void setValueAt(Object aValue, int rowIndex, int columnIndex)

Créer un modèle modifiable

- AbstractTableModel fournit des méthodes pour prévenir les listeners des modifications
  - void fireTableCellUpdated(int row, int column)

    contenu d'une case modifiée
  - void fireTableRowsDeleted(int firstRow, int lastRow)
  - void fireTableRowsInserted(int firstRow, int lastRow)
  - void fireTableRowsUpdated(int firstRow, int lastRow)

    *les coordonnées sont **inclusives**.*
  - void fireTableStructureChanged()

    structure modifiée (exemple: une colonne en plus)

permettre l'édition des champs

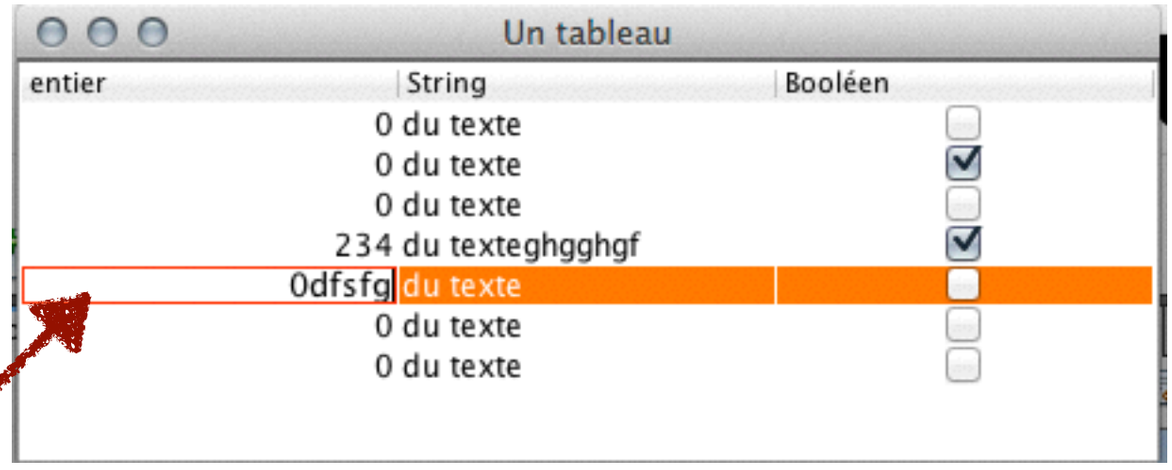
```
public class PersonnesTableModel extends AbstractTableModel {

    public boolean isCellEditable(int rowIndex, int columnIndex) {
         return columnIndex >= 1; // dans notre exemple, col. 0 non éditable
    }

    public void setValueAt(Object aValue, int rowIndex, int columnIndex) {
        Personne p = personnes.get(rowIndex);
        switch (columnIndex) {
            case 1:
                p.setNom((String)aValue);
                break;
            case 2:
                p.setPrenom((String)aValue);
                break;
        }
        fireTableCellUpdated(rowIndex, columnIndex);
    }
```

Type et noms des colonnes

```
public class PersonnesTableModel extends AbstractTableModel {

    public Class<?> getColumnClass(int columnIndex) {
        Class<?>[] columnClasses = {
            Integer.class, String.class, String.class
        };
        return columnClasses[columnIndex];
    }

    public String getColumnName(int column) {
        String[] titres = {"id", "nom", "prénom"};
        return titres[column];
    }
```

Classe des colonnes et éditeur

- Fixer la classe de la colonne permet à JTable de choisir
  - un éditeur adapté pour en modifier le contenu
  - une visualisation adaptée



erreur de saisie détectée

Ajout/suppression de lignes

- Dans le modèle:

```
    public void ajouterPersonne(Personne p) {
        int ligne= personnes.size();
        personnes.add(p);
        fireTableRowsInserted(ligne, ligne);
    }

    public void supprimerPersonne(int ligne) {
        personnes.remove(ligne);
        fireTableRowsDeleted(ligne, ligne);
    }
```

Ajout/suppression de lignes

- Dans le programme principal

```
public class DemoTablePersonnes {
   private JButton addPersonneButton= new JButton("+");
   private JButton removePersonneButton= new JButton("-");
   private PersonnesTableModel model;
public DemoTablePersonnes() {
    // Bug fix:
    table.putClientProperty("terminateEditOnFocusLost", Boolean.TRUE);
    …
    addPersonneButton.addActionListener(e-> addPersonne());
    removePersonneButton.addActionListener(e-> enleverPersonne());
    … }
 public void addPersonne() {
   Personne p= new Personne(100, "(nom)", "(prenom)");
    model.ajouterPersonne(p);
 }
 public void enleverPersonne() {
    int ligne= table.getSelectedRow();
     if (ligne != -1)
         model.supprimerPersonne(ligne);
 }
```

(petit détail ennuyeux)

```
table.putClientProperty("terminateEditOnFocusLost", Boolean.TRUE);
```

- la jtable tient à jour un éditeur qui est réutilisé pour les cases d'une colonne
- en cas de suppression d'une ligne, l'éditeur n'est pas prévenu par défaut (!)
- elle est déclenchée par la ligne ci-dessus
- voir

  http://bugs.java.com/bugdatabase/view_bug.do?bug_id=6349059
  http://bugs.java.com/bugdatabase/view_bug.do?bug_id=4709394
- Pour la petite histoire : la correction du bug n'est pas faite par défaut, par crainte de casser la compatibilité descendante.

Architecture d'application graphique

Le MVP (modèle, vue, présentateur)

- Plus simple que MVC
- Moins adapté si on a beaucoup de vues *du même modèle*
- pratique pour les formulaires

Modèle/Vue/ Présentateur

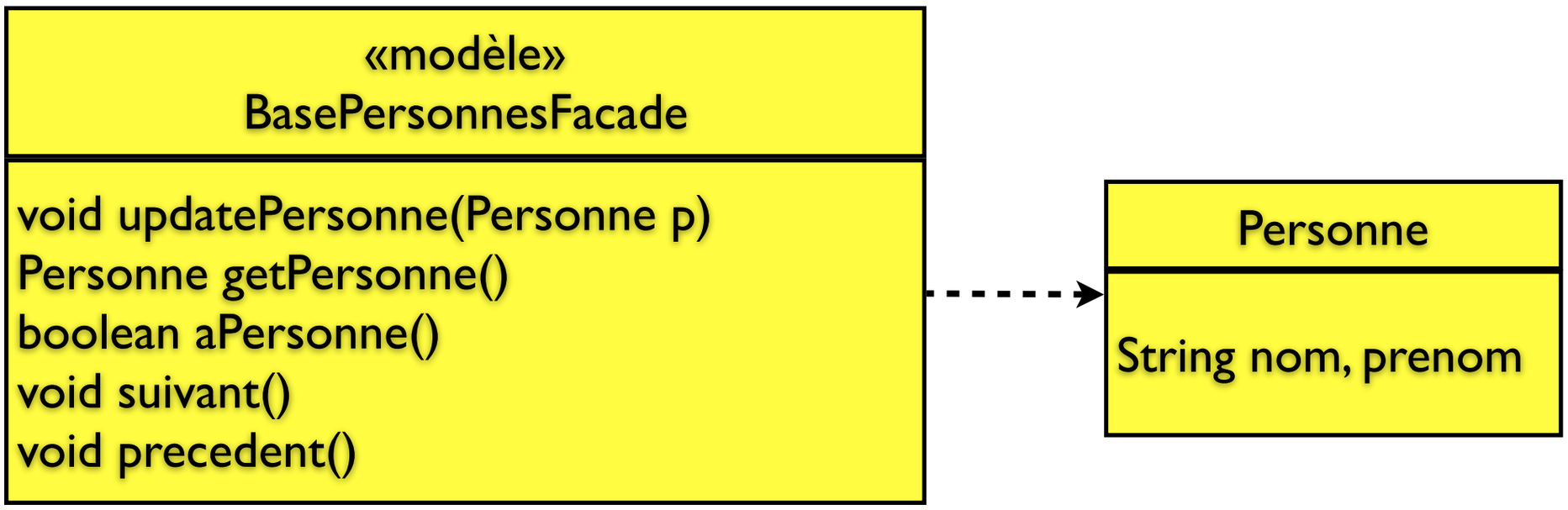
- Le modèle décrit les données
- La vue les affiche
- Le présentateur reçoit les événements, les traite (en modifiant le modèle), **et met à jour les vues**
- Différence avec MVC: c'est le présentateur qui rafraîchit les objets graphiques

Le MVC

- (déjà vu)
- facile à mettre en œuvre au niveau des composants
- plus délicat (mais possible) à utiliser pour la totalité des données d'une application
- En pratique, il peut arriver qu'on mélange les styles dans une même application

MVP sur un formulaire

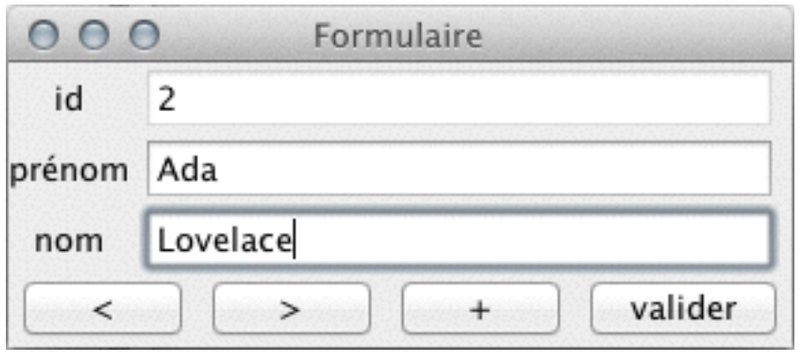
- Exemple: une application qui parcourt une base de données de personnes et permet de modifier les données associées



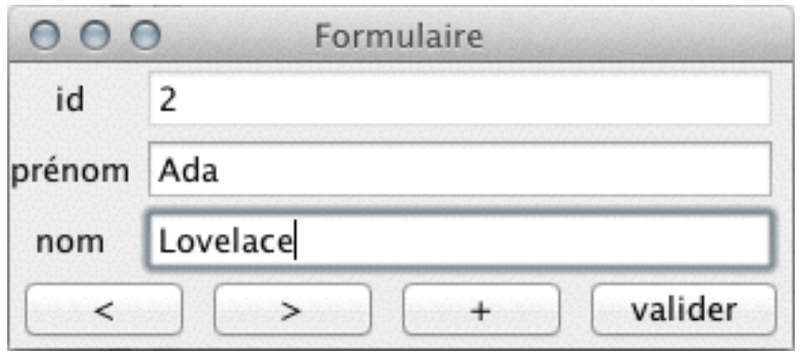
- Note: pour simplifier l'architecture, les objets Personne sont des *valeurs*. Pour mettre à jour les données d'une personne dans le modèle, il faut appeler updatePersonne().

- Le modèle est ici un **modèle de l'application** (avec la notion de navigation dans la liste)

Exemple MVP

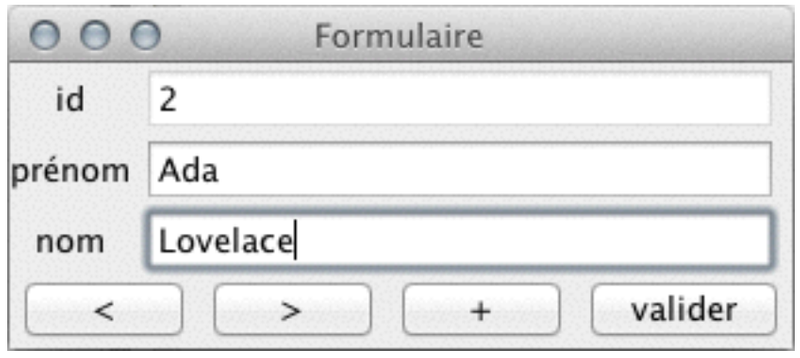


MVP



- On presse sur le bouton «suivant»
- le présentateur
  - appelle la méthode «suivant» du modèle
  - récupère la personne correspondante
  - copie les données de la personne dans les champs du formulaire

MVP



- On presse sur le bouton «mettre à jour»
- Le présentateur
  - récupère les données nom et prénom dans les champs du formulaire
  - appelle la méthode updatePersonne() du modèle pour mettre à jour celui-ci.

Vue...

```
public class JPersonneFormulaire {
  private JPanel panel= new JPanel();
  private JButton suivantButton= new JButton(">");
  private JButton precedentButton= new JButton("<");
  private JButton ajouterButton= new JButton("+");
  private JButton validerButton= new JButton("valider");
  private JTextField idField= new JTextField(20);
  private JTextField nomField= new JTextField(20);
  private JTextField prenomField= new JTextField(20);

  public JPersonneFormulaire() {
   idField.setEditable(false);
   mettreEnPage();
  }
  ... getters...
}
```

```
public class PersonnesFacade {
  // Invariant pour cet exemple :
  //il y a toujours une personne à la position "position".
  private ArrayList<Personne> personnes= new ArrayList<Personne>();
  private int position= 0;
  public PersonnesFacade() {
   personnes.add(new Personne(1, "", ""));
  }
  public Personne getPersonne() {
   return personnes.get(position);

  }
```

```
/**
 * Ajoute une entrée après l'entrée courante...
 */
public void ajouter() {
 personnes.add(position+1,
   new Personne(personnes.size() + 1, "", ""));
 suivant();
}

public void suivant() {
 if (position < personnes.size() -1 )
    position++;
}
```

Modèle

```
public class PersonnesFacade {
  ...

  public void precedent() {
   if (position > 0)
      position--;
  }

  public void mettreAJour(Personne nouveau) {
   // On peut directement utiliser "nouveau",
     // car les objets personnes sont immuables.
   // pas de risque qu'on le modifie derrière notre dos.
   personnes.set(position, nouveau);
  }

}
```

Modèle

Présentateur

```
public class PersonnesPresentateur {
  private JPersonneFormulaire vue;
  private PersonnesFacade modele;

  public PersonnesPresentateur(JPersonneFormulaire vue,
                               PersonnesFacade modele) {
   this.vue = vue;
   this.modele = modele;
   activer();
   charger();
}
/**
 * Copie les données de la  personne "courante" vers
 * le formulaire
 */
public void charger() {
 Personne p = modele.getPersonne();
 vue.getIdField().setText("" + p.getId());
 vue.getNomField().setText(p.getNom());
 vue.getPrenomField().setText(p.getPrenom());
}
```

Présentateur

```
public class PersonnesPresentateur {
  ........
  private void activer() {
   vue.getAjouterButton().addActionListener(
      EventHandler.create(ActionListener.class, this, "ajouter"));
   vue.getPrecedentButton().addActionListener(
      EventHandler.create(ActionListener.class, this, "precedent"));
   vue.getSuivantButton().addActionListener(
      EventHandler.create(ActionListener.class, this, "suivant"));
 vue.getValiderButton().addActionListener(
      EventHandler.create(ActionListener.class, this, "valider"));
}

/**
 * Passe à la personne suivante (si possible).
 */
public void suivant() {
 modele.suivant(); // on modifie le modèle...
 charger(); // on met à jour l'affichage
}
```

Présentateur

```
public class PersonnesPresentateur {
  ........


  /**
   * Met à jour la personne courante.
   */
  public void valider() {
     // récupération des données depuis la vue
  String pr=   vue.getPrenomField().getText()
  String n=   vue.getNomField().getText(),
  Personne ancien = modele.getPersonne();
  Personne nouveau =
     new Personne(ancien.getId(), n, pr);
     // modification des données dans le modèle
  modele.mettreAJour(nouveau);
     // mise à jour de la vue
  charger();  }

}
```

MVC «pur»

- Toute modification du modèle déclenche un événement
- les champs du formulaire ont des modèles (de type Document) qui «écoutent» eux-même ces événements et se mettent à jour automatiquement
- Attention cependant : les données du formulaire ne sont pas toujours les données du modèle. Par exemple, tant que je n'ai pas validé ma saisie, les modifications ne sont pas prises en compte.

Un peu plus de patterns

pattern commande

- **Problème** : on veut représenter explicitement une action dans l'application, pour pouvoir la manipuler:
  - la lier à plusieurs composants graphiques
  - l'activer/la désactiver
  - pouvoir éventuellement gérer un historique des action (fonction undo/redo)

Pattern Commande

- Solution : réifier l'action (de *res*, «chose» en latin).
- En clair: représenter l'action par un objet.
- Deux principales variantes:
  - on représente une action avec ses données contextuelles associées (fonction undo) : chaque exécution de l'action crée une instance.
  - on représente l'action «en général» (menu, etc...) : l'action est représentée par un seul objet. Cas de l'interface Action.

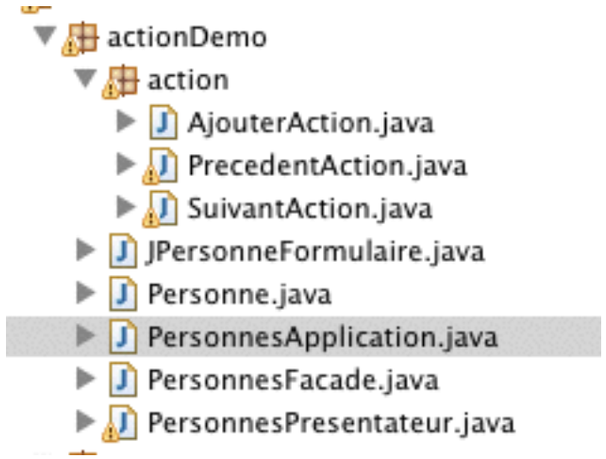
L'interface Action et la classe AbstractAction

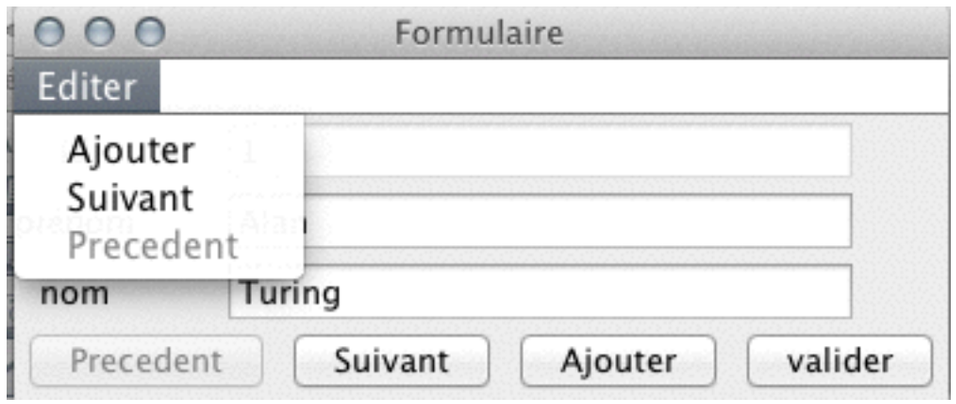
- Une action est un actionlistener qui a de l'ambition :
- elle peut être activée/désactivée (isEnabled; setEnabled)
- elle a des propriétés (par exemple pour lui ajouter des raccourcis, des icônes, des toolstips)
- On utilise généralement AbstractAction pour travailler.

AbstractAction

- Constructeurs:
  - AbstractAction()
  - AbstractAction(String name)
  - AbstractAction(String name, Icon icon)
- permet de dire quelle texte/icône seront attachés aux menus, boutons... qui utilisent l'action.

Utilisation





Actions...

```
@SuppressWarnings("serial")
public class AjouterAction extends AbstractAction {
  public static final String ID = "AJOUTER";
  PersonnesPresentateur presentateur;

  public AjouterAction(PersonnesPresentateur presentateur) {
     // Label de l'action (boutons, menus...)
   super("Ajouter");
   this.presentateur= presentateur;
      // Propriété : raccourci clavier.
   putValue(ACCELERATOR_KEY, KeyStroke.getKeyStroke("control a"));
  }

  @Override
  public void actionPerformed(ActionEvent ev) {
   presentateur.ajouter();
  }
}
```

Actions

```
public class PersonnesPresentateur {
  private JPersonneFormulaire vue;
  private PersonnesFacade modele;
  private HashMap<String, Action> actMap =
                                      new HashMap<String, Action>();

  public PersonnesPresentateur(JPersonneFormulaire vue,
                               PersonnesFacade modele) {
   this.vue = vue;
   this.modele = modele;
   creerActions(); activer(); charger();
  }

  private void creerActions() {
   actMap.put(AjouterAction.ID, new AjouterAction(this));
   actMap.put(SuivantAction.ID, new SuivantAction(this));
   actMap.put(PrecedentAction.ID, new PrecedentAction(this));
   ...
  }
  private void activer() {
    vue.getAjouterButton().setAction(actMap.get(AjouterAction.ID));

vue.getPrecedentButton().setAction(actMap.get(PrecedentAction.ID));
    vue.getSuivantButton().setAction(actMap.get(SuivantAction.ID));
    ...
  }
```

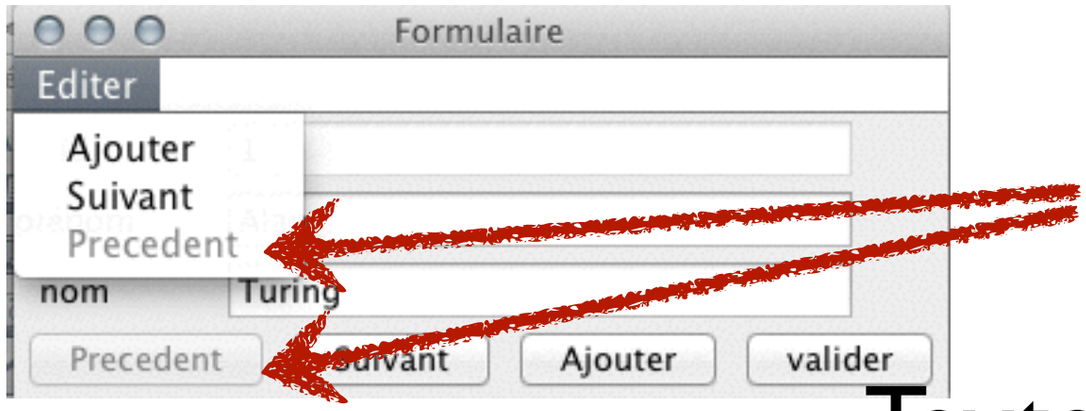
Actions

```
public class PersonnesPresentateur {
  private JPersonneFormulaire vue;
  private PersonnesFacade modele;
  private HashMap<String, Action> actMap =
                                    new HashMap<String, Action>();

  ....
  /**
   * Charge la personne "courante".
   */
public void charger() {
 Personne p = modele.getPersonne();
 vue.getIdField().setText("" + p.getId());
 vue.getNomField().setText(p.getNom());
 vue.getPrenomField().setText(p.getPrenom());
 mettreAjourActions();
}

private void mettreAjourActions() {
 actMap.get(SuivantAction.ID).setEnabled(modele.aSuivant());
 actMap.get(PrecedentAction.ID).setEnabled(modele.aPrecedent());

}
```

Actions *(et menus)*

```
public class PersonnesApplication {
  private PersonnesFacade modele;
  private JPersonneFormulaire vue;
  private PersonnesPresentateur presentateur;
  private JFrame frame;

  public PersonnesApplication() {
   ....
   frame= new JFrame("Formulaire");
   frame.add(vue.getPanel());
   creerMenu();
   frame.pack();
   frame.setVisible(true);
   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }

  private void creerMenu() {
   JMenuBar menuBar= new JMenuBar();
   JMenu menu= new JMenu("Editer");
   menu.add(presentateur.getAction(AjouterAction.ID));
   menu.add(presentateur.getAction(SuivantAction.ID));
   menu.add(presentateur.getAction(PrecedentAction.ID));

   menuBar.add(menu);
   frame.setJMenuBar(menuBar);
  }
```

Actions (et menus)



même objet action, même état

Texte