Compléments sur les interfaces graphiques en Java

Université de Nice - Sophia Antipolis

Version 0.9.9 – 6/10/07

Richard Grin

- Ce support de cours étudie des compléments sur Swing :
  - compléments divers : vérification des entrées, timers, renderers
  - introduction aux composants complexes de Swing : arbres, tables
  - traitements sur les pages HTML
- Ces compléments ne font qu'effleurer les sujets, trop vastes pour être approfondis dans ce cours

Bibliographie

- Le tutoriel de Sun sur Swing : http://java.sun.com/docs/books/tutorial/
- De nombreuses figures de ce cours sont extraites de ce tutoriel

Vérification des entrées

Utilisation

- La classe **`javax.swing.InputVerifier`** facilite le traitement des saisies invalides de l'utilisateur : l'utilisateur ne passe au champ suivant qu'après vérification de la validité de ce qu'il a saisi
- On définit une classe fille de **`InputVerifier`** en redéfinissant la méthode **`verify`** pour qu'elle renvoie **`true`** si et seulement si la saisie est valide
- On indique que le composant est vérifié avec la méthode **`setInputVerifier`**

Exemple

```
  JTextField tf =
    new JTextField (10);
  tf.setInputVerifier(new PassVerifier());
. . .
class PassVerifier extends InputVerifier {
  public boolean verify(JComponent input) {
    JTextField tf = (JTextField) input;
    String pass = tf.getText();
    if (pass.equals("pass")) return true;
    else return false;
  }
}
```

Condition de validité

JFormattedTextField

- Cette classe fille de **`JTextField`** a été introduite par le SDK 1.4
- Elle permet de donner un format pour la saisie des données (et aussi éventuellement pour leur affichage)
- De très nombreuses possibilités sont offertes au programmeur ; par exemple, pour dire ce qui se passe si la valeur saisie ne correspond pas au format

Timer

Définition

- Un timer est une sorte de réveil que l'on remonte et qui « sonne » après un certain délai
- Un timer est associé à un (ou plusieurs) **`ActionListener`** qui reçoit un **`ActionEvent`** quand le timer « sonne »
- Un timer est représenté par la classe **`javax.Timer`**

Comportement des timers

- Un timer peut sonner une seule fois ou à intervalles réguliers
- On peut indiquer le délai avant la 1ère sonnerie et un autre délai pour l'intervalle régulier

Timers et threads

- Le timer s'exécute à chacun de ses réveils dans le thread de répartition des événements et il en est donc ainsi des méthodes actionPerformed() appelées par le timer
  - ces méthodes peuvent donc manipuler des éléments graphiques
  - mais elles doivent s'exécuter rapidement pour ne pas figer l'interface graphique

Exemple

```
Timer chaqueSeconde =
  new Timer(1000, new ActionListener1());
Timer uneFois =
  new Timer(5000, new ActionListener2());
Timer delaiInitial =
  new Timer(1000, new ActionListener3());
uneFois.setRepeats(false);
delaiInitial.setInitialDelay(3000);
chaqueSeconde.start();
uneFois.start();
delaiInitial.start();
```

Arrêt et démarrage d’un timer

- **`stop()`** stoppe l’envoi des **`ActionEvent`**
- **`start()`** reprend l’envoi des **`ActionEvent`**
- **`restart()`** reprend tout depuis le début comme si le timer venait d’être démarré pour la 1ère fois

`java.util.Timer`

- La version JDK 1.3 a ajouté les classes **`Timer`** et **`TimerTask`** dans le paquetage **`java.util`** pour lancer une tâche à un moment donné ou à des intervalles répétés
- Au contraire du timer de swing, les tâches ne sont pas nécessairement associées à l’interface graphique et ne sont donc pas déposées dans la file d’attente des événements

Renderer

Les *renderers*

- De nombreux composants complexes de Swing utilisent un « renderer » pour afficher les éléments dont il est composé :
  - les listes (interface **`ListCellRenderer`**)
  - les arbres (interface **`TreeCellrenderer`**)
  - les tables (interface **`TableCellRenderer`**)
  - et d'autres…

Exemple de renderer

- Exemple tiré du livre Core Java qui affiche une liste de polices de caractères en utilisant la police elle-même
- Pour installer le renderer :

```
fontList.setCellRenderer(
              new FontCellRenderer());
```


Exemple de renderer

```
class FontCellRenderer implements ListCellRenderer {
  public Component getListCellrendererComponent(
    final JList list, final Object value,
    final int index, final boolean isSelected,
    final boolean cellHasFocus) {
      return new JPanel() {
        public void paintComponent(Graphics g) {
          Font font = (Font)value;
          String text = font.getFamily();
          FontMetrics fm = g.getFontMetrics(font);
          g.setColor(isSelected
            ? list.getSelectionBackground()
            : list.getBackground());
          g.fillRect(0, 0, getWidth(), getHeight());
```

"**`final`**" pour la classe interne

Exemple de renderer (suite)

```
        g.setColor(isSelected
          ? list.getSelectionForeGround()
          : list.getForeGround());
        g.setFont();
        g.drawString(text, 0, fm.getAscent());
      }
      public Dimension getPreferredSize() {
        Font font = (Font)value;
        String text = font.getFamily();
        Graphics g = getGraphics();
        FontMetrics fm = g.getFontMetrics(font);
        return new Dimension(fm.stringWidth(text),
                             fm.getHeight());
      }
    }; // fin classe fille de JPanel
} // fin classe FontCellRenderer
```

A

`JLabel` comme *renderer*

- Si l'effet graphique ne concerne que l'affichage de texte avec une fonte donnée, une image et un changement de couleur, on peut simplifier en renvoyant un `JLabel` comme *renderer*

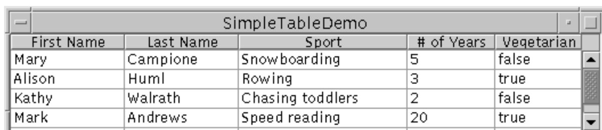
Exemple de `JLabel` comme *renderer*

```
class FontCellRenderer implements ListCellRenderer {
  public Component getListCellrendererComponent(
    JList list, Object value, int index,
    boolean isSelected, boolean cellHasFocus) {
      JLabel label = new Jlabel();
      Font font = (Font)value;
      label.setText(font.getFamily());
      label.setFont(font);
      label.setOpaque(true);
      label.setBackGround(isSelected
        ? list.getSelectionBackground()
        : list.getBackground());
      return label;
  }
}
```


`JTable`

Utilité

- Représenter des données sous forme tabulaire :



- Les lignes et les colonnes sont identifiées par un nombre entier (en commençant à 0)

Objets utilisés pour le fonctionnement de `JTable`

- Un modèle de données qui représente les données dans la `JTable`
- Un modèle de colonnes qui représente les colonnes de la table
- Un (ou plusieurs) *renderer* pour afficher les données du modèle
- Un (ou plusieurs) éditeur de cellule pour saisir les modifications des données par l'utilisateur

Tables simples

- Les tables les plus simples sont construites avec les constructeurs qui prennent en paramètres 2 tableaux d'`Object` ou 2 `Vector` (1 pour les noms des colonnes et 1 pour les données)
- On n'a pas à créer une classe pour le modèle de données
- La table utilise les tableaux ou vecteurs passés en paramètres comme modèles (ils sont donc modifiés si les données de la table le sont)

Exemple de table simple

```
Object[][] data =
    {{"a11", "a12"}, {"a21", "a22"}};
String[] nomsColonnes = {"Col1", "Col2"};
JTable table =
    new JTable(data, nomsColonnes);
JScrollPane sp = new JScrollPane(table);
// Si on veut définir la taille de la zone
// d'affichage :
table.setPreferredScrollableViewportSize(
    new Dimension(500, 70));
```


Table en dehors d'un ScrollPane

- Le plus souvent les tables sont dans un `ScrollPane` (voir exemple précédent)
- Sinon, il faut faire afficher explicitement les en-têtes de colonnes ; par exemple :

```
container.setLayout(new BorderLayout());
container.add(table.getTableHeader(),
              BorderLayout.NORTH);
container.add(table, BorderLayout.CENTER);
```


Constructeurs de JTable

- `JTable()` : crée une table avec un modèle de données de type `DefaultTableModel` (utilise un `Vector` de `Vector` pour ranger les données) et un modèle de colonnes de type `DefaultTableColumnModel`
- `JTable(int ligne, int colonne)` : des lignes et des colonnes dont les cellules ne contiennent rien (`null`)

Constructeurs de JTable

- `JTable(Object[][] données, Object[] descriptionColonnes)` : les données sont rangées dans un tableau
- `JTable(Vector données, Vector descriptionColonnes)` : les données sont dans un `Vector` de `Vector`

Constructeurs de JTable

- `JTable(TableModel modèleDonnées)` : crée une table avec un modèle de données
- `JTable(TableModel modèleDonnées, TableColumnModel modèleColonnes)` : le modèle pour les colonnes n'est pas nécessairement un `DefaultTableColumnModel`
- `JTable(TableModel modèleDonnées, TableColumnModel modèleColonnes, ListSelectionModel modèleSélection)` : le modèle de sélection n'est pas nécessairement un `DefaultListSelectionModel`

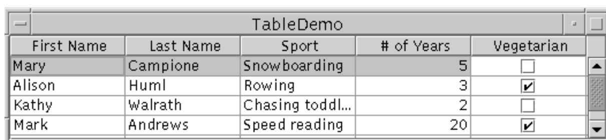
Propriétés des tables simples

- Toutes leurs cellules sont éditables
- Tous les types de données sont affichés sous forme de chaînes de caractères
- Toutes les données sont dans un tableau ou un **`Vector`**, ce qui ne convient pas si, par exemple, les données viennent d'une base de données

Tables avec modèle de données explicite

- Pour avoir plus de souplesse, on peut utiliser les autres constructeurs de **`JTable`** qui prennent en paramètre un modèle explicite
- Avec le modèle on peut indiquer le type (et le nom) de chaque colonne, indiquer les cellules qui sont modifiables et avoir plus de souplesse dans la manière d'obtenir les données affichées par la table

Table avec modèle de données


TableDemo

First Name	Last Name	Sport	# of Years	Vegetarian
Mary	Campione	Snowboarding	5	☐
Alison	Huml	Rowing	3	☑
Kathy	Walrath	Chasing toddl...	2	☐
Mark	Andrews	Speed reading	20	☑

On a indiqué que le type de cette colonne est numérique ; les données sont justifiées à droite

On a indiqué que cette colonne contient des booléens

Modèle de données

- Les données de la table sont fournies par un objet à part
- Cet objet est une instance d'une classe qui implémente l'interface **`TableModel`**

Méthodes de **`TableModel`**

- Accéder aux données du modèle ou les modifier : {**`get|set`**}**`valueAt`**
- Indiquer si une cellule est modifiable : **`isCellEditable`**
- Décrire les colonnes : **`getColumnCount`**, **`getColumnName`**, **`getColumnClass`**
- Nombre de lignes : **`getRowCount`**
- Ajouter ou enlever des écouteurs : {**`add|remove`**}**`TableModelListener`**

Modèles de données

- **`AbstractTableModel`** implémente l'interface **`TableModel`**
- Elle est abstraite car elle n'implémentant que les méthodes associées aux écouteurs
- On l'utilise le plus souvent si on veut un modèle de données explicite

AbstractTableModel

- Pour avoir un modèle en héritant de **`AbstractTableModel`** il suffit de définir les 3 méthodes suivantes :
  - **`int getRowCount()`**
  - **`int getColumnCount()`**
  - **`Object getValueAt(int ligne, int colonne)`**

AbstractTableModel

- Le plus souvent il faut aussi redéfinir les méthodes qui décrivent les colonnes :
  - **`String getColumnName()`** ; par défaut, elles s'appellent A, B, etc.
  - **`Class getColumnClass()`** ; par défaut, cette méthode renvoie **`Object`** et aucun format particulier n'est utilisé pour l'affichage (voir transparent suivant)

AbstractTableModel

- Format pour les colonnes suivant les classes :
  - **`Boolean`** : boîte à cocher (**`CheckBox`**)
  - **`Number`** : cadré à droite dans un **`JLabel`**
  - **`Double`**, **`Float`** : cadré à droite et utilisation de **`NumberFormat`** pour « localiser »
  - **`Date`** : **`JLabel`** et utilisation de **`DateFormat`**
  - **`ImageIcon`**, **`Icon`** : centré dans un **`JLabel`**
  - **`Object`** : ce que renvoie **`toString()`** cadré à gauche dans un **`JLabel`**

AbstractTableModel

- Si des cellules sont modifiables, il faut aussi redéfinir la méthode (par défaut, elle renvoie **`false`** pour toutes les cellules) :
  - **`boolean isEditable(int ligne, int colonne)`**

setValueAt

- La méthode **`setValueAt`** qui permet de modifier les valeurs du modèle est implémentée dans **`AbstractTableModel`**, mais elle est vide
- Quand les données de la table sont modifiables, il faut donc redéfinir cette méthode
- Sinon, même les modifications de l'utilisateur avec un éditeur de cellules, ne sont pas prises en compte par la table (les éditeurs appellent **`setValueAt`** pour modifier la table)

setValueAt

- Dans cette méthode, on doit le plus souvent avertir les observateurs du modèle de la table qu'une donnée a été modifiée
- En effet, il faut au moins avertir la **`JTable`** elle-même qui écoute son modèle, sinon elle n'affichera pas les modifications par la suite
- Pour cela, la méthode doit appeler la méthode **`void fireTableCellUpdated(int ligne, int colonne)`** définie dans la classe **`AbstractTableModel`**

Avertir les écouteurs

- Les méthodes suivantes de la classe **`AbstractTableModel`** avertissent les observateurs des modifications sur le modèle :
  - **`fireTableCellUpdated(int ligne, int colonne)`**
  - **`fireTableRowsDeleted(ligne, colonne)`**
  - **`fireTableRowsInserted(ligne, colonne)`**
  - **`fireTableChanged(TableModelEvent e)`**
  - **`fireTableDataChanged()`**
  - **`fireTableStructuredChanged()`**

Exemple de **`setValueAt`**

```
public void setValueAt(Object valeur,
                       int ligne,
                       int colonne) {
  ((ArrayList)lignes).get(ligne).set(colonne,
                                     valeur);
  fireTableCellUpdated(ligne, colonne);
}
```


Ajouter un écouteur

```
table.getModel().addTableModelListener(
    new EcouteurModeleTable());
```

- La table construite avec un modèle de données écoute automatiquement son modèle
- On peut aussi ajouter d'autres écouteurs avec la méthode **`addTableModelListener`**

Écouteurs des modifications d'une table

- Ils implémentent **`TableModelListener`** qui a une seule méthode **`tableChanged(TableModelEvent e)`**
- On peut interroger **`e`** par **`getFirstRow`**, **`getLastRow`**, **`getColumn`**, **`getType`** (type de l'événement : **`INSERT`** et **`DELETE`** pour insertion et suppression de lignes ou colonnes, **`UPDATE`** pour une modification de données)

Exemple d'écouteur de table

```
class EcouteurModeleTable
    implements TableModelListener {
  public void tableChanged(TableModelEvent e) {
    int row = evt.getFirstRow();
    int column = evt.getColumn();
    Object data =
      ((TableModel)e.getSource()).
          getValueAt(row,column);
    traiter(data);
  }
  private void traiter(Object o) { . . . };
}
```


`DefaultTableModel`

- **`DefaultTableModel`** hérite de **`AbstractTableModel`** ; elle utilise un **`Vector`** de **`Vector`** pour ranger les données
- Si les données viennent d'une base de données, **`DefaultTableModel`** ne convient pas et il faut hériter de **`AbstractTableModel`**
- **`JTable`** offre maintenant beaucoup de possibilités de configuration, donc le plus souvent on héritera directement **`AbstractTableModel`** plutôt que de **`DefaultTableModel`**

Renderers

- Les « *renderers* » sont associés aux colonnes
- Par défaut, ils dépendent des types des données des colonnes
- On peut aussi associer son propre renderer à une colonne

Exemple de renderer

- Un combobox comme renderer

Éditeur de cellule

- Par défaut, on a un éditeur par type de données
- On peut associer son propre éditeur à une classe des données de la table
- On peut aussi en associer un à une colonne de la table par la méthode **`setCellEditor`** de la classe **`javax.swing.table.TableColumn`** ; par exemple,

```
table.getColumnModel().getColumn(2).
setCellEditor(editeur);
```


`DefaultCellEditor`

- La classe **`DefaultCellEditor`** a des constructeurs pour avoir des éditeurs qui sont des **`JTextField`**, **`JComboBox`** ou **`JCheckBox`**
- Par exemple,

```
editeur =
  new DefaultCellEditor(comboBox);
```

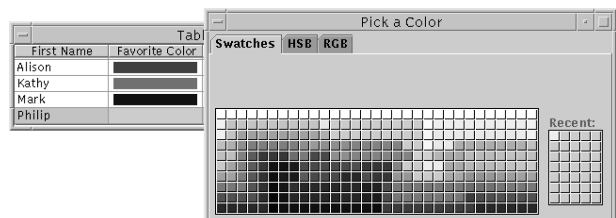
- On peut aussi construire d'autres types d'éditeur en implémentant directement l'interface **`TableCellEditor`**

`DefaultCellEditor`

- La classe **`DefaultCellEditor`** fait un appel à la méthode **`setValueAt`** du modèle de données de la table

Exemple d'éditeur

- Un éditeur pour une colonne qui contient des couleurs favorites



Sélection par l'utilisateur

- Par défaut l'utilisateur peut sélectionner une ou plusieurs lignes (pas nécessairement contiguës) en cliquant dessus
- On peut changer ce mode de sélection par la méthode **`setSelectionMode`**
- On peut enregistrer des écouteurs de sélection qui réagiront à une nouvelle sélection

Sélection de lignes ou de colonnes

```
// On ne permet la sélection que d'1 ligne
table.setSelectionMode(
    ListSelectionModel.SINGLE_SELECTION);
...
// Récupère le modèle pour la sélection
ListSelectionModel rowSM =
  table.getSelectionModel();
// Pour y ajouter un écouteur
rowSM.addListSelectionListener(
  new EcouteurSelection());
```


Écouteur de sélection d'une ligne

```
class EcouteurSelection
  implements ListSelectionListener {
  public void valueChanged(ListSelectionEvent e) {
    if (e.getValueIsAdjusting()) return;
    ListSelectionModel lsm =
      (ListSelectionModel)e.getSource();
    if (lsm.isSelectionEmpty()) {
     ... // aucune sélection
    }
    else {
      int nLigne = lsm.getMinSelectionIndex();
      ... // Traitement pour la sélection
   }
  }
}
```


Variante : sélection de plusieurs lignes (pas toutes contiguës)

```
. . .
else {
  int minIndex = lsm.getMinSelectionIndex();
  int maxIndex = lsm.getMaxSelectionIndex();
  for (int i = minIndex; i <= maxIndex; i++) {
    if (lsm.isSelectedIndex(i)) {
      // Ligne i est sélectionnée
      . . .
    }
  }
}
```


Sélectionner par programmation

- Les méthodes **`set{Row|Column}SelectionInterval`** permettent de sélectionner des lignes ou colonnes contiguës par programmation
- On peut ajouter ou enlever des lignes ou colonnes contiguë à la sélection par les méthodes **`add{Row|Column}SelectionInterval`** et **`remove{Row|Column}SelectionInterval`**

Donner une largeur à une colonne

- On utilise explicitement le modèle pour les colonnes :

```
TableColumn col =
  table.getColumnModel().getColumn(3);
col.setPreferredWidth(100);
```

Exemple (1)

```
// Créer la table et l'ajouter à la fenêtre
JTable table =
  new JTable(new MonModele());
table.setPreferredScrollableViewportSize(
  new Dimension(500, 70));
JScrollPane scrollPane =
  new JScrollPane(table);
add(scrollpane);
```


Exemple (2 ; modèle de données)

```
class MonModele extends AbstractTableModel {
  private String[] nomsColonnes =
    {"Nom", "Prénom", "Age"};
  // Un élement de la liste contient une ligne
  private ArrayList lignes =
    new ArrayList();
  public int getColumnCount() {
    return nomsColonnes.length;
  }
  public int getRowCount() {
    return lignes.length;
  }
```


Exemple (3 ; modèle de données)

```
  public String getColumnName(int col) {
    return nomsColonnes[col];
  }
  public Object getValueAt(int ligne,
                           int col) {
    return
     ((ArrayList)lignes.get(ligne)).get(col);
  }
  public Class getColumnClass(int c) {
    // Dépend de la table
    . . .
  }
```


Exemple (4 ; modèle de données)

```
public boolean isCellEditable(int ligne,
                              int col) {
  // Seul l'âge est modifiable
  if (col < 2) {
    return false;
  }
  else {
    return true;
  }
}
```


Exemple (5 ; modèle de données)

```
  public void setValueAt(Object valeur,
                         int ligne,
                         int col) {
    ((ArrayList)lignes.get(ligne)).
       set(col, valeur);
    fireTableCellUpdated(ligne, col);
  }
}
```


Traitements « avancé »

- Si on veut traiter les données avant de les afficher, le plus simple est d'utiliser des modèles intermédiaires qui observent les modèles de base
- Le tutorial Java de Sun fournit 2 classes (`TableMap` et `TableSorter`) qui facilitent ces traitements, en particulier le tri des données suivant une colonne (utiliser la dernière version de février 2004)

Trier les données d'une colonne

```
// On utilise la classe TableSorter
// fournie par le tutoriel Java de Sun
TableSorter trieur =
  new TableSorter(new MonTableModel());
trieur.setTableHeader(
         table.getTableHeader());
JTable table = new JTable(trieur);
```

Attention, les données du modèle de base ne sont pas triées.

Trier et filtrer les données – JDK 6

- A partir du JDK 6, le tri et le filtrage des tables sont inclus dans le JDK
- Pour trier : `javax.swing.RowSorter`
- Pour filtrer (sélectionner des lignes) : `javax.swing.RowFilter`
- Dans les 2 cas les données du modèle ne sont pas modifiées ; c'est la vue de ce modèle qui est modifiée

Exemple de tri (début)

```
TableModel model =
new DefaultTableModel(rows, columns) {
  public Class getColumnClass(int column) {
    Class returnValue;
    if (column >= 0
        && column < getColumnCount()) {
      returnValue =
      getValueAt(0, column).getClass();
    }
    else {
      returnValue = Object.class;
    }
    return returnValue;
  } // fin getColumnClass
}; // fin classe interne anonyme
```

Pour que le tri s'effectue suivant la classe de la colonne (supposée implémenter `Comparable`)

Exemple de tri (suite)

```
JTable table = new JTable(modele);
RowSorter<TableModel> sorter =
  new TableRowSorter<TableModel>(modele);
table.setRowSorter(sorter);
```


Types de filtres

- Plusieurs types de filtres sont déjà fournis dans le JDK
- Ils sont renvoyés par les méthodes `static` suivantes de la classe `RowFilter` :
  - `regexFilter` filtre les chaînes de caractères suivant une expression régulière
  - `dateFilter` filtre les dates
  - `numberFilter` filtre les nombres
  - `andFilter`, `orFilter` et `notFilter`

Exemple de filtrage

```
JTable table = new JTable(modele);
RowSorter<TableModel> sorter =
  new TableRowSorter<TableModel>(modele);
table.setRowSorter(sorter);
...
sorter.setRowFilter(RowFilter
                    .regexFilter(texte));
```

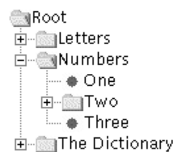
Imprimer

- Le JDK 5 a ajouté plusieurs méthodes `print` pour faciliter l'impression d'une `JTable`
- La méthode `getPrintable` peut être utilisée pour faire des impressions un peu plus complexes

`JTree`

Utilité

- Un arbre permet de représenter des données sous une forme hiérarchisée :



Fonctionnement des arbres

- Un arbre n'est qu'une vue de données mais ne contient aucune donnée
- Il contient une donnée par ligne
- Chaque donnée s'appelle un nœud
- Un nœud peut avoir des fils
- Un arbre a un et un seul nœud racine
- Par défaut, seul le nœud racine est affiché et déployé (avec ses fils non déployés)

Objet associé au nœud

- On peut associer un objet à un nœud, d'une autre classe que la classe `String`
- Le nœud est affiché avec la méthode `toString()` de la classe de l'objet
- On peut utiliser un *renderer* pour représenter les nœuds

Création d'un arbre

- Pour créer un arbre, on crée le nœud racine :

```
DefaultMutableTreeNode top =
  new DefaultMutableTreeNode("Racine");
```

- On ajoute ensuite des nœuds fils à ce nœud racine ou à n'importe quel nœud :

```
DefaultMutableTreeNode n1, n2;
DefaultMutableTreeNode n2;
n1 = new DefaultMutableTreeNode("Letters");
top.add(n1);
n1 = new DefaultMutableTreeNode("Numbers");
top.add(n1);
n2 = new DefaultMutableTreeNode("One");
. . .
```

"User Object" associé au noeud

Création d'un arbre (2)

- On crée ensuite l'arbre ; le plus souvent on met l'arbre dans un **`JScrollPane`** :

```
JTree arbre = new JTree(top);
JScrollPane vueArbre = new JScrollPane(tree);
```

- On indique le mode de sélection des nœuds :

```
tree.getSelectionModel().setSelectionMode(
       TreeSelectionModel.SINGLE_TREE_SELECTION);
```

- On ajoute ensuite souvent un écouteur pour réagir aux sélections de nœuds par l'utilisateur

`TreeSelectionListener`

```
tree.addTreeSelectionListener(
  new TreeSelectionListener() {
    public void valueChanged(TreeSelectionEvent e) {
      DefaultMutableTreeNode node =
        (DefaultMutableTreeNode)
        tree.getLastSelectedPathComponent();
      if (node == null) return;
      Object nodeInfo = node.getUserObject();
      if (node.isLeaf())
        processFeuille(book.bookURL);
      else
        processNonFeuille(helpURL);
    }
  });
```


Traitements sur pages HTML

Parser la page HTML

```
// Crée le reader de la page
URL url = new URI(urlString).toURL();
URLConnection conn = url.openConnection();
Reader reader = new
  InputStreamReader(conn.getInputStream());
// Crée le kit et le document HTML
EditorKit kit = new HTMLEditorKit();
HTMLDocument doc =
  (HTMLDocument)kit.createDefaultDocument();
kit.read(reader, doc, 0);
```


Traitements liés à la page lue

- La classe **`HTMLDocument`** contient des méthodes pour
  - parcourir les tags de la page
  - modifier la page HTML lue
  - sauvegarder la page modifiée
- Avec ces méthodes on peut faire des traitements qui modifient la page ou non

Parcourir les tags

- On peut parcourir les tags qui ont été trouvés par le kit avec la méthode

```
HTMLDocument.Iterator
     getIterator(HTML.Tag t)
```

- La classe **`HTML.Tag`** contient des constantes pour tous les tags de HTML ; par exemple, **`HTML.Tag.A`** pour le tag <A>
- Ne marche pas pour les tags qui ajoutent une structure au document tels que <TABLE> ou <TR>

Exemple de parcours des tags

```
HTMLDocument.Iterator it =
  doc.getIterator(HTML.Tag.A);
while (it.isValid()) {
  SimpleAttributeSet s =
    (SimpleAttributeSet) it.getAttributes();
  String link = (String)
    s.getAttribute(HTML.Attribute.HREF);
  if (link != null) {
    System.out.println(link);
  }
  it.next();
}
```

Affiche les adresses des liens de la page

Afficher la valeur d'un tag

- On aurait aussi pu faire afficher le texte inclus dans le tag, à la place de l'attribut HREF du tag :

```
int start = it.getStartOffset();
int end = it.getEndOffset();
String valeur =
  doc.getText(start, end - start + 1);
System.out.println("Valeur : "
                   + valeur);
```


Traitement spéciaux sur la page

- Pour d'autres traitements, il est nécessaire d'hériter de la classe **HTMLDocument** pour redéfinir des méthodes
- Par exemple, si on veut conserver à part tous les textes de la page dans un fichier non HTML, on remplace la création du document HTML de l'exemple précédent par le code du transparent suivant

Exemple

```
HTMLDocument doc = new HTMLDocument() {
  public HTMLEditorKit.ParserCallback
         getReader(int pos) {
    return
      new HTMLEditorKit.ParserCallback() {
        public void handleText(char[] data,
                               int pos) {
          buf.append(data);
          buf.append('\n');
        }
      };
  } // getReader
};
```


Afficher les tags « de structure »