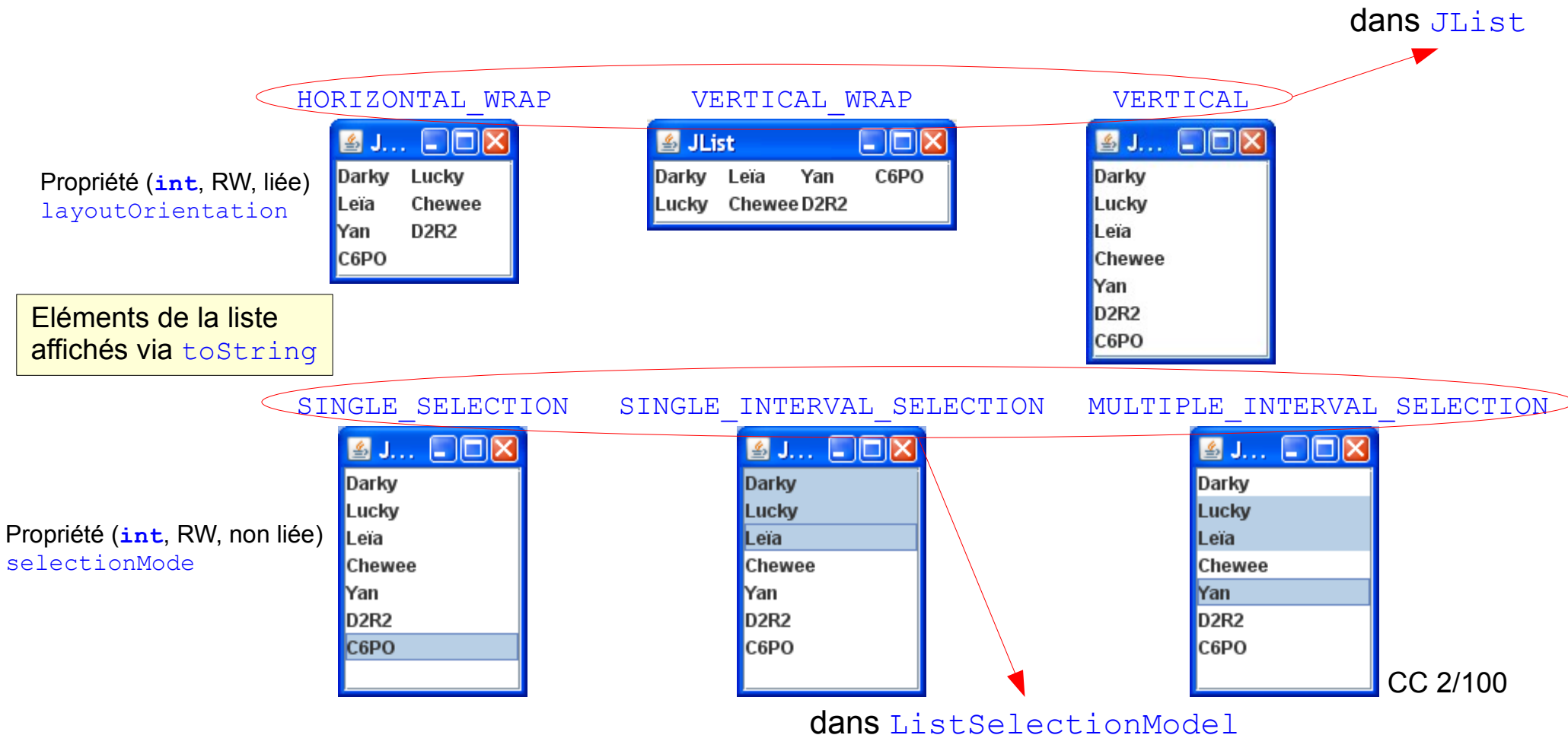


Séance 7

- Composants complexes
 - JList
 - *JTable*
 - *JTree*

Présentation de `JList`

- Affichage d'un groupe d'éléments avec sélection simple ou multiple (contiguë ou non)



Architecture MVC

- Modèles

- Données : `ListModel`
→ `ListDataListener`
- Sélection : `ListSelectionModel`
→ `ListSelectionListener`

- Vue et Contrôleur standards

UI-delegate

- `ListUI` (`BasicListUI`)
 - Observe son `ListModel` et son `ListSelectionModel`
 - Agit sur son `ListSelectionModel` à l'aide de `FocusListener`, `KeyListener` et `MouseListener` internes

réunion de `MouseListener`
et `MouseMotionListener`

Mise en place d'une JList

- `JList()`
 - modèle vide
- `JList(Object[] listData)`
 - modèle pré-initialisé, non mutable
- `JList(Vector<?> listData)`
 - modèle pré-initialisé, non mutable
- `JList(ListModel listData)`
 - modèle (éventuellement mutable) non **null**

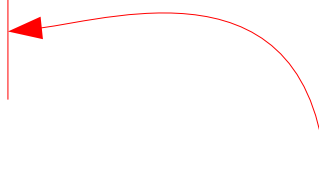
Gestion des données

- Propriété `model` (`ListModel`, RW, liée)
- **interface** `ListModel`
 - **void** {add|remove}`ListDataListener` (LDL)
 - **int** `getSize()`
 - `Object` `getElementAt(int)`
 - Doc Java → fonctionne comme un `Vector` (indices variant de 0 à `getSize() - 1`)
- Modification des données
 - uniquement à partir du modèle
 - notification aux `ListDataListeners`
 - ! le *UI-delegate* observe son modèle avec un `ListDataListener`

- AbstractListModel

- méthodes concrètes :

- {add|remove}ListDataListener
 - fireInterval{Added|Removed}
 - fireContentsChanged
 - get[ListData]Listeners



à utiliser pour notifier les
ListDataListeners

- seules méthodes abstraites :

- getSize
 - getElementAt

- /! pas de modification prévue

- DefaultListModel

- **extends** AbstractListModel
 - implémentation des modèles mutables

Mutabilité des données

- Modèles non mutables

- `JList()`
- `JList(Object[])`
- `JList(Vector<?>)`

```
public JList(final Object[] listData) {  
    this(new AbstractListModel() {  
        public int getSize() { return listData.length; }  
        public Object getElementAt(int i) {  
            return listData[i];  
        }  
    });  
}
```

- Modèles mutables

- `JList(ListModel)`

`DefaultListModel`



Modèle de données personnalisé

- doit obligatoirement :
 - implanter `ListModel`
 - notifier les `ListDataListeners`
- peut éventuellement :
 - dériver `AbstractListModel`
 - ou dériver `DefaultListModel`

dériver ces classes est conseillé car
la gestion des `ListDataEvent` est facilitée
par la présence des méthodes `fire*`

ListDataListener

- **void** `intervalAdded(ListDataEvent e)`
 - les éléments entre `e.getIndex0()` et `e.getIndex1()` viennent d'être insérés
- **void** `intervalRemoved(ListDataEvent e)`
 - les éléments entre `e.getIndex0()` et `e.getIndex1()` viennent d'être supprimés
- **void** `contentsChanged(ListDataEvent e)`
 - il y a eu des modifications entre `e.getIndex0()` et `e.getIndex1()`

`e.getIndex0()` : plus petit index de la zone modifiée
`e.getIndex1()` : plus grand index de la zone modifiée

Défilement des éléments

- /! une `JList` ne gère pas le défilement

```
JScrollPane jsp = new JScrollPane(list);  
ou  
JScrollPane jsp = new JScrollPane();  
jsp.setViewportViewView(list);
```

mais elle est `Scrollable` !

- Propriété intéressante associée au défilement
 - `visibleRowCount` (**int**, RW, liée)
 - nombre de lignes à afficher
- Méthode en rapport avec le défilement
 - **void** `ensureIndexIsVisible(int)`
 - impose l'affichage d'une ligne particulière

Clics multiples

- /!\ une `JList` ne gère que le simple clic (+ Shift ou Ctrl)
 - clics multiples → ajout d'un `MouseListener`

```
public void mouseClicked(MouseEvent e) {  
    if (e.getClickCount() == 2) {  
        int index = list.locationToIndex(e.getPoint());  
        System.out.println("Double clic sur le " + index + "-ième élément");  
    }  
}
```

... avec `indexToLocation`, deux méthodes bien pratiques

Sélection des éléments

- Modèle de sélection → sous-type de `ListSelectionModel`
- Propriété liée (RW) : `selectionModel`
- /!\ toute méthode du modèle de sélection peut être exécutée à partir de la `JList`
 - Ex : `list.getSelectedIndex()`
→ `list.getSelectionModel().getSelectedIndex()`

- **void** setSelectionInterval(**int** index0, **int** index1)
 - sélectionne de index0 à index1 (inclus)
- **void** clearSelection()
 - désélectionne la sélection courante
- **void** addSelectionInterval(**int** index0, **int** index1)
 - ajoute à la sélection courante les éléments entre index0 et index1 (inclus)
- **void** removeSelectionInterval(**int** index0, **int** index1)
 - supprime de la sélection courante les éléments entre index0 et index1 (inclus)

ListSelectionEvent

ListSelectionListener

- **Sélection** → `ListSelectionEvent`
 - client humain (souris/clavier)
 - programmeur (opérations de sélection)
- **Enregistrer un** `ListSelectionListener`
 - sur une `JList` (recommandé)
 - sur un `ListSelectionModel`
 - différence → source des événements

- `ListSelectionListener` :
 - **void** `valueChanged(ListSelectionEvent)`
- `ListSelectionEvent` :
 - `Object getSource()`
 - la `JList` ou son `ListSelectionModel`
 - **int** `getFirstIndex()`
 - premier index de la sélection
 - **int** `getLastIndex()`
 - dernier index de la sélection
 - **boolean** `getValueIsAdjusting()`
 - **true** au début d'une opération de sélection, **false** dès qu'elle est terminée

Rendu graphique des éléments

- Affichage des éléments → utilise un *renderer*
- *renderer* = instance de `ListCellRenderer`

```
Component getListCellRendererComponent(  
    JList list,    // la liste cliente  
    Object value, // l'élément à afficher  
    int index,    // la position de l'élément  
    boolean isSelected, // élmt sélectionné ?  
    boolean cellHasFocus // élmt a le focus ?  
)
```

- Principe
 - la liste veut dessiner un élément → elle demande à son *renderer* de le faire pour elle

Utilisation d'un *renderer*

- Association à la liste :
 - `void setCellRenderer(ListCellRenderer)`
- L'action de dessiner s'effectue en interne ainsi
 - *renderer* configuré pour l'un des éléments
 - `Component c = r.getListCellRendererComponent(...)`
 - *renderer* positionné
 - `c.setBounds(...)`
 - *renderer* peint (dessine un élément de la liste)
 - `c.paint(...)`

DefaultListCellRenderer

- Implante `ListCellRenderer`
- Étend `JLabel` :
 - sait afficher
 - des images
 - du texte
 - tout objet (*via* `toString`)
 - redéfinition de certaines méthodes
 - optimisation de l'affichage du label
 - court-circuite `repaint/revalidate`

```

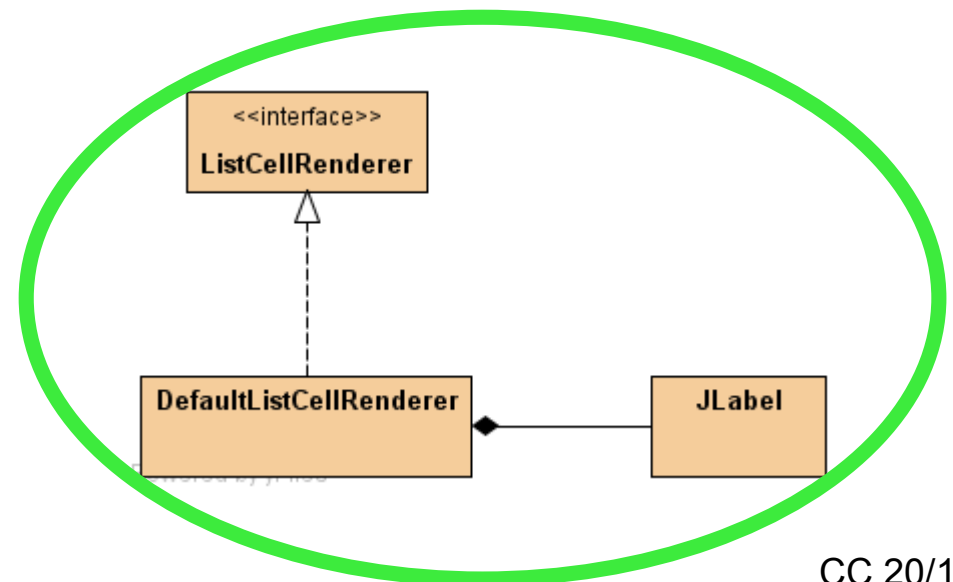
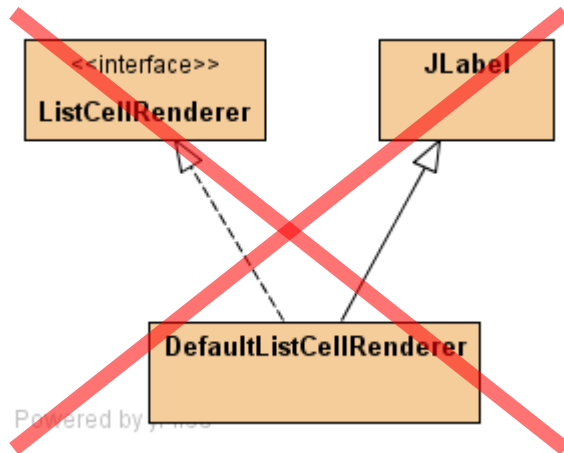
public class DefaultListCellRenderer extends JLabel
    implements ListCellRenderer, Serializable {
    ...
    public Component getListCellRendererComponent(
        JList list,
        Object value,
        int index,
        boolean isSelected,
        boolean cellHasFocus) {

        Color bg = ...;
        Color fg = ...;
        if (isSelected) {
            setBackground(bg == null ? list.getSelectionBackground() : bg);
            setForeground(fg == null ? list.getSelectionForeground() : fg);
        } else {
            setBackground(list.getBackground());
            setForeground(list.getForeground());
        }
        if (value instanceof Icon) {
            setIcon((Icon) value);
            setText("");
        } else {
            setIcon(null);
            setText((value == null) ? "" : value.toString());
        }
        setEnabled(list.isEnabled());
        setFont(list.getFont());
        setBorder(...);
        return this;
    }
}

```

Remarque

- Devrait déléguer la gestion de l'affichage à un label plutôt que de dériver `JLabel`
 - un *renderer* ne doit pas être utilisé en tant que `JLabel` mais en tant que `ListCellRenderer`



Définition d'un *renderer*

```
public class ItalicRenderer implements ListCellRenderer {
    private Font italic;
    private Font origin;
    private DefaultListCellRenderer delegate;

    public ItalicRenderer() {
        delegate = new DefaultListCellRenderer();
        origin = delegate.getFont();
        italic = new Font(origin.getName(), Font.ITALIC, origin.getSize());
    }

    public Component getListCellRendererComponent(
        JList list,
        Object value,
        int index,
        boolean isSelected,
        boolean cellHasFocus) {
        Component result = delegate.getListCellRendererComponent(
            list, value, index,
            isSelected, cellHasFocus
        );
        if (isSelected) {
            result.setFont(italic);
        } else {
            result.setFont(origin);
        }
        return result;
    }
}
```

Présentation de JTable

- `javax.swing.JTable`
 - affichage et édition d'une séquence de données structurées en champs (les lignes)



tuto Swing à lire d'abord,
puis revenir au diaporama
et lire les diapos 22-59

- Gestion des données stockées dans les cellules
 - `TableModel`
- Gestion des données par colonne
 - `TableColumnModel` et `TableColumn`
- Entêtes de colonnes
 - `JTableHeader`
- Sélection des cellules
 - `ListSelectionModel`
- Rendu visuel des cellules
 - `TableCellRenderer`
- Édition des cellules
 - `TableCellEditor`

Mise en place

- Table avec zéro cellule
 - `JTable()`
- Table avec $r \times c$ cellules vides
 - `JTable(int r, int c)`
- Table définie par un tableau de tableaux ou un vecteur de vecteurs de données
 - `JTable(Object[][] rowCells, Object[] colNames)`
 - `JTable(Vector rowCells, Vector colNams)`
- Table définie par un modèle de données qcq
 - `JTable(TableModel tm)`
 - `JTable(TableModel tm, TableColumnModel tcm)`
 - `JTable(TableModel tm, TableColumnModel tcm, ListSelectionModel lsm)`

Modèle de données

- Interface `TableModel` :
 - **void** `addTableModelListener(TML)`
 - **void** `removeTableModelListener(TML)`
 - **int** `getColumnCount()`
 - **int** `getRowCount()`
 - `String` `getColumnName(int)`
 - `Class<?>` `getColumnClass(int)`
 - `Object` `getValueAt(int, int)`
 - **void** `setValueAt(Object, int, int)`
 - **boolean** `isCellEditable(int, int)`

Changements d'état du modèle

- `TableModelListener`
 - `void tableChanged(TableModelEvent e)`
- `TableModelEvent`
 - `int getColumn()`
 - si égal à `ALL_COLUMNS`, alors toutes les colonnes ont changé, sinon c'est l'indice de la colonne modifiée
 - `int getFirstRow()`
 - si égal à `HEADER_ROW`, alors c'est l'en-tête qui a changé, sinon c'est l'indice de la première ligne modifiée
 - `int getLastRow()`
 - indice de la dernière ligne modifiée
 - `int getType()`
 - `INSERT` ou `UPDATE` ou `DELETE`

Détail des méthodes `fireTable*` de `AbstractTableModel`

rafraîchit tout	<code>fireTableDataChanged()</code>	<code>new TableModelEvent(this, 0, Integer.MAX_VALUE, TME.ALL_COLUMNS, TME.UPDATED)</code>	toutes les données du modèle ont changé
rafraîchit le strict nécessaire	<code>fireTableCellUpdated(int row, int column)</code>	<code>new TableModelEvent(this, row, row, column, TME.UPDATED)</code>	quelques données du modèle ont changé
	<code>fireTableRowsDeleted(int firstRow, int lastRow)</code>	<code>new TableModelEvent(this, firstRow, lastRow, TME.ALL_COLUMNS, TME.DELETED)</code>	
	<code>fireTableRowsInserted(int firstRow, int lastRow)</code>	<code>new TableModelEvent(this, firstRow, lastRow, TME.ALL_COLUMNS, TME.INSERTED)</code>	
	<code>fireTableRowsUpdated(int firstRow, int lastRow)</code>	<code>new TableModelEvent(this, firstRow, lastRow, TME.ALL_COLUMNS, TME.UPDATED)</code>	
recrée toutes les colonnes	<code>fireTableStructureChanged()</code>	<code>new TableModelEvent(this, TME.HEADER_ROW, TME.HEADER_ROW, TME.ALL_COLUMNS, TME.UPDATED)</code>	la structure du modèle a changé

Implantation partielle du modèle

- `AbstractTableModel`, méthodes concrètes :
 - `getColumnName` retourne 'A', 'B', etc.
 - `getColumnClass` retourne `Object.class`
 - `isCellEditable` retourne **false**
 - `setValueAt` ne fait rien
 - `{add|remove}TableModelListener` gère la liste d'écouteurs du modèle de données
- et 3 méthodes abstraites :
 - `getRowCount`, `getColumnCount`, `getValueAt`

Exemple : un modèle simple non éditable

```
class SimpleROTableModel extends AbstractTableModel {
    private Object[][] data;
    public SimpleROTableModel(Object[][] data) {
        if (data == null) {
            throw new AssertionError();
        }
        this.data = data;
    }
    public int getRowCount() {
        return data.length;
    }
    public int getColumnCount() {
        return data.length == 0 ? 0 : data[0].length;
    }
    public Object getValueAt(int row, int col) {
        return data[row][col];
    }
}
```

Exemple : un modèle éditable à colonnes personnalisées

```
class SimpleRTableModel extends AbstractTableModel {
    private Object[][] data;
    private Object[] colNames;
    public SimpleRTableModel(Object[][] data, Object[] colNames) {
        // ...
        this.data = data;
        this.colNames = colNames;
    }

    public int getRowCount() { return data.length; }
    public int getColumnCount() { return colNames.length; }
    public Object getValueAt(int row, int col) { return data[row][col]; }

    @Override public String getColumnName(int column) {
        return colNames[column].toString();
    }
    @Override public boolean isCellEditable(int row, int column) {
        return true;
    }
    @Override public void setValueAt(Object value, int row, int col) {
        data[row][col] = value;
        fireTableCellUpdated(row, col);
    }
}
```

Implantation standard du modèle

- Classe `DefaultTableModel`
 - utilise un `Vector` de `Vectors`
- Ajout des données
 - `addColumn` : ajout d'une colonne (évent. vide) et de son nom, en dernière position
 - `addRow` : ajout d'une ligne
 - `insertRow` : insère une ligne
 - `setValueAt` : modifie une cellule
 - `setDataVector` : change la structure complète (cellules et noms des colonnes)

Gestion des colonnes graphiques

- Classe `TableColumn`
 - colonne graphique
 - gestion de taille, rendu graphique, éditeur de cellules, position dans le modèle, ...
- Principales propriétés :
 - `headerValue` (`Object`, RW, liée)
 - `identifier` (`Object`, RW, liée)
 - `maxWidth` (**`int`**, RW, liée)
 - `minWidth` (**`int`**, RW, liée)
 - `modelIndex` (**`int`**, RW, liée)
 - `preferredWidth` (**`int`**, RW, liée)
 - `resizable` (**`boolean`**, RW, liée)
 - `width` (**`int`**, RW, liée)

Modèle des colonnes graphiques

- Interface `TableColumnModel`, modèle de gestion des colonnes pour :
 - la sélection
 - le déplacement
 - la correspondance entre position dans la `JTable` et position dans le modèle de données
- Implantation standard
 - `DefaultTableColumnModel`

- Un `DefaultTableColumnModel` est automatiquement créé sur la base des cellules du modèle :
 - lors de la création d'une `JTable`
 - lors d'un changement de modèle de données
 - lors d'un évènement indiquant un changement de la structure complète de la table
- On peut contrôler ce comportement avec la propriété `autoCreateColumnsFromModel` (**boolean**, RW, liée) des `JTables`

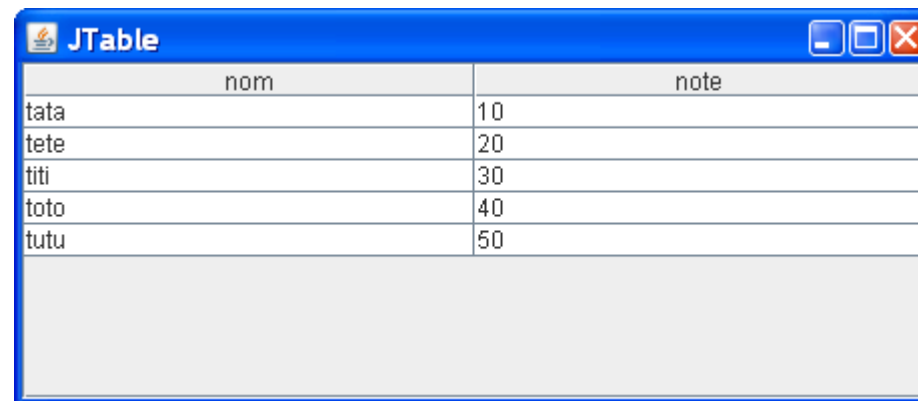
- Exemple : une table à colonnes doublées

```
void duplicateColumns(JTable t) {  
    DefaultTableColumnModel dtcm = new DefaultTableColumnModel();  
    for (int i = 0; i < t.getModel().getColumnCount(); i++) {  
        dtcm.addColumn(new TableColumn(i));  
        dtcm.addColumn(new TableColumn(i));  
    }  
    t.setColumnModel(dtcm);  
    // t.getColumnCount() == 2 * t.getModel().getColumnCount()  
}
```

nom	nom	note	note
tata	tata	10	10
tete	tete	20	20
titi	titi	30	30
toto	toto	40	40
tutu	tutu	50	50

- Exemple : une table privée de certaines colonnes

```
void doNothing(JTable t) {  
    duplicateColumns(t);  
    TableColumnModel dtcm = t.getColumnModel();  
    for (int i = dtcm.getColumnCount() - 1; i >= 0; i--) {  
        if (i % 2 == 1) {  
            dtcm.removeColumn(dtcm.getColumn(i));  
        }  
    }  
}
```



nom	note
tata	10
tete	20
titi	30
toto	40
tutu	50

Changement d'état du modèle de colonnes

- **Interface** `TableColumnModelListener`
 - `columnAdded(TableColumnModelEvent)`
 - `columnMoved(TableColumnModelEvent)`
 - `columnRemoved(TableColumnModelEvent)`
 - `columnMarginChanged(ChangeEvent)`
 - `columnSelectionChanged(ListSelectionEvent)`

Retailage des colonnes

- Deux manières possibles :
 - retailler la `JTable`
 - retailler une (des) colonne(s) individuellement
- Deux stratégies possibles :
 - globale
 - propriété `autoResizeMode` (**int**, RW, liée) de `JTable`
 - locale
 - propriété `resizable` (**boolean**, RW, liée) de `TableColumn`
 - propriétés (**int**, RW, liée) de `TableColumn` :
 - `maxWidth`, `minWidth`, `preferredWidth`, `width`

Défilement et entête

- Toujours associer une `JTable` à un `JScrollPane`
 - `JTable` ne gère pas le défilement
 - `JTable` ne gère pas l'affichage de l'entête
 - l'entête des colonnes ne s'affiche que dans la partie `columnHeader` d'un `JScrollPane`
 - Rq. : si l'on ne veut pas d'entête
 - `table.setTableHeader(null);`
`table.removeNotify();`

Conserver une colonne fixe

```
table = new JTable(  
    new Object[][] {  
        {"tata", "10", "pas terrible"},  
        ...  
    },  
    new Object[] {"nom", "note", "appréciation"}  
);  
  
// créer la table fixe  
JTable tableFixe = new JTable();  
tableFixe.setAutoCreateColumnsFromModel(false);  
tableFixe.setModel(table.getModel());  
TableColumn noms = table.getColumn("nom");  
tableFixe.addColumn(noms);  
table.removeColumn(noms);  
  
// partager le modèle de sélection  
tableFixe.setSelectionModel(table.getSelectionModel());  
  
// fixer la taille du rowHeader  
Dimension size = tableFixe.getPreferredSize();  
JViewport vp = new JViewport();  
vp.setView(tableFixe);  
vp.setPreferredSize(size);  
vp.setMaximumSize(size);  
jsp.setRowHeaderView(vp);  
  
// fixer le header de la table fixe  
jsp.setCorner(  
    ScrollPaneConstants.UPPER_LEFT_CORNER,  
    tableFixe.getTableHeader()  
);
```

colonnes "défilables"
dans le viewportview

nom	note	appréciation
tata	10	pas terrible
tete	20	pas terrible
titi	30	terrible
toto	40	terrible
tutu	50	terrible
toto	40	terrible
tutu	50	terrible
toto	40	terrible
tutu	50	terrible
toto	40	terrible

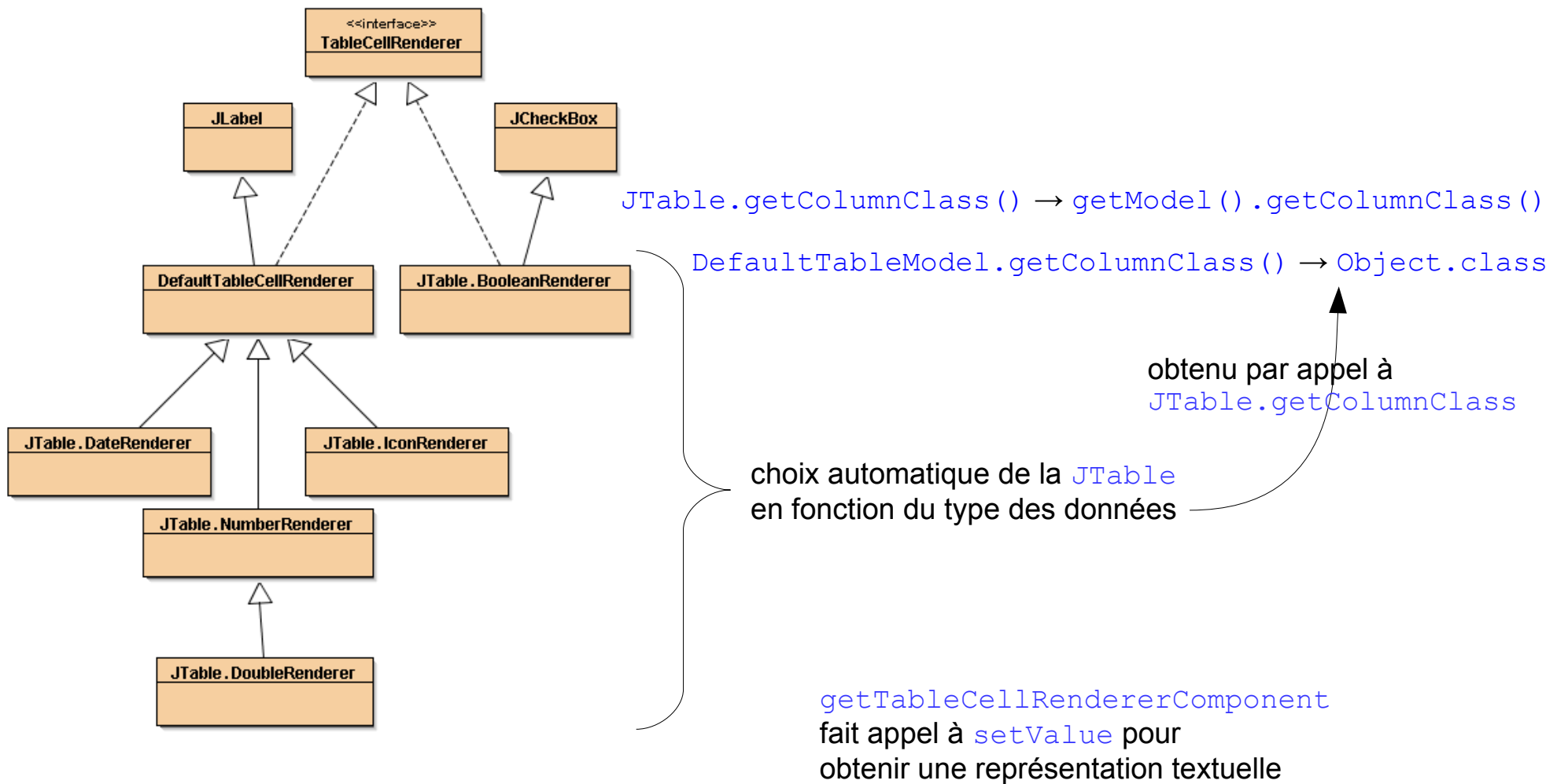
colonne fixe dans
le rowheaderview

Rendu visuel des cellules

- Géré par un composant obtenu d'un `TableCellRenderer` :

```
- Component getTableCellRendererComponent(  
    JTable table,           // la table  
    Object value,           // la valeur à afficher  
    boolean isSelected,     // état de sélection de la ¢  
    boolean hasFocus,       // état de focus de la ¢  
    int row,                // numéro de ligne de la ¢  
    int column              // numéro de colonne de la ¢  
)
```

- Implantation standard
 - `DefaultTableCellRenderer`

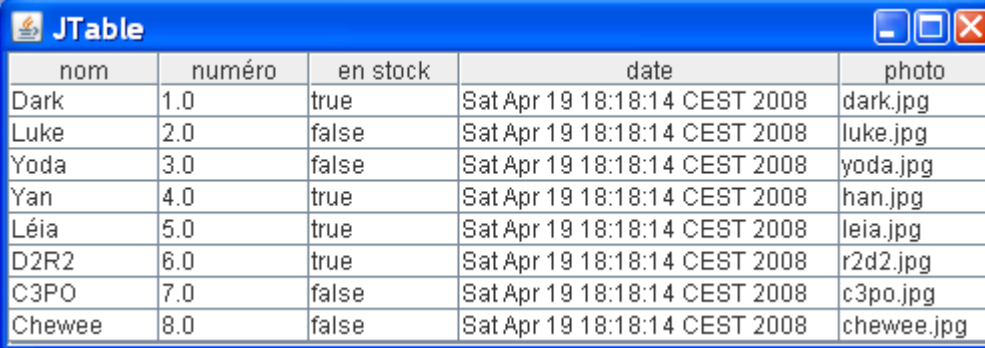


<code>getColumnClass</code>	Type de renderer	Effet de <code>setValue</code>
<code>Object.class</code>	<code>DefaultTableCellRenderer</code>	<code>setText(v.toString())</code>
<code>Date.class</code>	<code>JTable.DateRenderer</code>	<code>setText(DateFormat.format(v))</code>
<code>Number.class</code>	<code>JTable.NumberRenderer</code>	<code>setText(v.toString())</code>
<code>Float.class</code>	<code>JTable.DoubleRenderer</code>	<code>setText(NumberFormat.format(v))</code>
<code>Double.class</code>	<code>JTable.DoubleRenderer</code>	<code>setText(NumberFormat.format(v))</code>
<code>Icon.class</code>	<code>JTable.IconRenderer</code>	<code>setIcon(v)</code>
<code>ImageIcon.class</code>	<code>JTable.IconRenderer</code>	<code>setIcon(v)</code>
<code>Boolean.class</code>	<code>JTable.BooleanRenderer</code>	<code>setSelected(v)</code>

```

table = new JTable(
    new Object[][] {
        {"Dark",      1.0, true,  new Date(), new ImageIcon("dark.gif")},
        {"Luke",      2.0, false, new Date(), new ImageIcon("luke.jpg")},
        {"Yoda",      3.0, false, new Date(), new ImageIcon("yoda.gif")},
        {"Yan",       4.0, true,  new Date(), new ImageIcon("han.jpg")},
        {"Léia",      5.0, true,  new Date(), new ImageIcon("leia.jpg")},
        {"D2R2",      6.0, true,  new Date(), new ImageIcon("r2d2.gif")},
        {"C3PO",      7.0, false, new Date(), new ImageIcon("c3po.jpg")},
        {"Chewee",    8.0, false, new Date(), new ImageIcon("chewee.jpg")},
    },
    new Object[] {"nom", "numéro", "en stock", "date", "photo"}
);

```






nom	numéro	en stock	date	photo
Dark	1.0	true	Sat Apr 19 18:18:14 CEST 2008	dark.jpg
Luke	2.0	false	Sat Apr 19 18:18:14 CEST 2008	luke.jpg
Yoda	3.0	false	Sat Apr 19 18:18:14 CEST 2008	yoda.jpg
Yan	4.0	true	Sat Apr 19 18:18:14 CEST 2008	han.jpg
Léia	5.0	true	Sat Apr 19 18:18:14 CEST 2008	leia.jpg
D2R2	6.0	true	Sat Apr 19 18:18:14 CEST 2008	r2d2.jpg
C3PO	7.0	false	Sat Apr 19 18:18:14 CEST 2008	c3po.jpg
Chewee	8.0	false	Sat Apr 19 18:18:14 CEST 2008	chewee.jpg

```

class MyModel extends AbstractTableModel {
    public Class<?> getColumnClass(int i) {
        switch (i) {
            case 0:
                return String.class;
            case 1:
                return Double.class;
            case 2:
                return Boolean.class;
            case 3:
                return Date.class;
            case 4:
                return ImageIcon.class;
            default:
                return super.getColumnClass(i);
        }
        ...
    };
}

-----
table = new JTable(new MyModel()) ;

```

JTable				
nom	numéro	en stock	date	photo
Dark	1	<input checked="" type="checkbox"/>	19 avr. 2008	
Luke	2	<input type="checkbox"/>	19 avr. 2008	
Yoda	3	<input type="checkbox"/>	19 avr. 2008	
Yan	4	<input checked="" type="checkbox"/>	19 avr. 2008	
Léia	5	<input checked="" type="checkbox"/>	19 avr. 2008	
D2R2	6	<input checked="" type="checkbox"/>	19 avr. 2008	
C3PO	7	<input type="checkbox"/>	19 avr. 2008	
Chewee	8	<input type="checkbox"/>	19 avr. 2008	

Définir ses propres renderers

1) Planter `TableCellRenderer`

2) Associer un renderer aux colonnes concernées :

- une ou plusieurs colonnes spécifiques :

- ```
TableColumnModel tcm = table.getColumnModel();
TableColumn tc = tcm.getColumn(colIndex);
// ou aussi tc = table.getColumn(colName);
tc.setCellRenderer(r);
```

- toutes les colonnes de type `X` :

- Redéfinir `getColumnClass` dans le modèle pour qu'elle retourne `X.class` sur les indices des colonnes de type `X`
- ```
table.setDefaultRenderer(X.class, r);
```

Algorithme de recherche du renderer adéquat

- Obtenir le renderer d'une `TableColumn tc` :
 - Si un renderer a été fixé par (1) pour `tc` Alors
 - Utiliser ce renderer
 - Sinon [un renderer a été fixé par (2)] :
 - Obtenir le type `X` des éléments de `tc` par `getColumnClass`
 - TantQue aucun renderer n'a été fixé pour `X` Faire
 - `X <- super type de X`
 - Utiliser le renderer fixé pour le type `X`
- Notes :
 - (1) par `tc.setCellRenderer(TCR)`
 - (2) par `table.setDefaultRenderer(Class, TCR)`
 - On est assuré de sortir du TantQue, au pire sur le type `Object` (par défaut, un `DTCR` est fixé pour `Object`)

Éditeurs de cellules

- Points communs avec les renderers :
 - on peut associer un éditeur à une ou plusieurs colonnes spécifiques
 - `TableColumn.setCellEditor(TCE)`
 - on peut associer un éditeur à un type de cellules
 - `JTable.setDefaultEditor(Class, TCE)`
 - on peut utiliser des composants Swing pour éditer les cellules
 - `JComboBox`, `JCheckBox`, `JTextField`, ...

Détection d'édition

- Clic sur une cellule → la table détermine si la cellule est éditable ou non
(`TableModel.isCellEditable`)
- Si réponse positive → la table retrouve l'éditeur pour la cellule et demande si la cellule doit être éditée (`CellEditor.isCellEditable`)
- L'éditeur répond en fonction du nombre de clics et de la politique d'édition
- Il y a édition seulement si les deux réponses sont positives

Fonctionnement de l'édition

- Lorsque l'utilisateur clique sur une cellule pour l'éditer, si la table accepte l'édition
 - appel de `getTableCellEditorComponent` qui prépare et configure l'éditeur pour refléter correctement le contenu de la cellule
 - puis la table retaille l'éditeur avec les dimensions de la cellule et le superpose au dessus de la cellule pour en capturer l'édition
 - enfin, l'éditeur se met en œuvre

TableCellEditor

- Lorsque la table détecte une situation d'édition, elle obtient l'éditeur configuré par appel de

- `Component getTableCellEditorComponent (`

```
    JTable table,           // la table
    Object value,           // valeur à afficher
    boolean isSelected,     // état de sélection de la ¢
    int row,                 // ligne de la ¢
    int column              // colonne de la ¢
```

```
)
```

sur un `TableCellEditor`

- Cette interface étend l'interface `CellEditor`

CellEditor

- **void** addCellEditorListener(CellEditorListener)
 - Enregistre un écouteur auprès de l'éditeur, des notifications auront lieu quand l'éditeur arrêtera ou annulera l'édition
- **void** cancelCellEditing()
 - Annule l'édition sans prendre en compte la valeur en cours d'édition ; doit notifier les écouteurs enregistrés
- **Object** getCellEditorValue()
 - La valeur courante stockée par l'éditeur
- **boolean** isCellEditable(EventObject)
 - Deuxième méthode appelée durant le mécanisme de détection d'édition, l'évènement est celui qui est à l'origine de l'édition
- **void** removeCellEditorListener(CellEditorListener)
 - Supprime l'écouteur de cet éditeur
- **boolean** shouldSelectCell(EventObject)
 - Indique si la cellule en cours d'édition doit être sélectionnée ou non, l'évènement est celui qui est à l'origine de l'édition
- **boolean** stopCellEditing()
 - Arrête l'édition en prenant en compte la valeur en cours d'édition comme valeur de l'éditeur ; doit notifier les écouteurs enregistrés

CellEditorListener

- **void** `editingCanceled(ChangeEvent)`
 - Comportement à adopter quand l'éditeur a annulé l'édition
 - Activée par `CellEditor.cancelCellEditing`
- **void** `editingStopped(ChangeEvent)`
 - Comportement à adopter quand l'éditeur a terminé l'édition
 - Activée par `CellEditor.stopCellEditing`
- Normalement, seule la table est un écouteur de son éditeur

Exemple : un éditeur de dates

```
public class DateEditor extends AbstractCellEditor implements TableCellEditor {
    private static final Calendar CAL = Calendar.getInstance();
    private JTextField delegate;
    public DateEditor() {
        delegate = new JTextField();
        delegate.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                DateEditor.this.stopCellEditing();
            }
        });
    }
    public Object getCellEditorValue() {
        CAL.clear();
        try {
            CAL.set(Long.parseLong(delegate.getText()));
        } catch (NumberFormatException e) { /* rien */ }
        return CAL.getTime();
    }
    public boolean isCellEditable(EventObject e) {
        if (e instanceof MouseEvent) {
            return ((MouseEvent) e).getClickCount() >= 2;
        }
        return false;
    }
    public Component getTableCellEditorComponent( ... ) {
        if (value instanceof Date) {
            delegate.setText(String.valueOf(((Date) value).getTime()));
        } else {
            delegate.setText("0");
        }
        return delegate;
    }
}
```

```
table.setDefaultEditor(Date.class, new DateEditor());
```

DefaultCellEditor

- Éditeur par défaut pour les cellules de `JTable`
- 3 constructeurs :
 - `DefaultCellEditor(JCheckBox)`
 - pour des cellules contenant des booléens
 - édition sur un seul clic
 - `DefaultCellEditor(JComboBox)`
 - pour des cellules contenant des éléments d'une liste prédéfinie
 - édition sur un seul clic
 - `DefaultCellEditor(JTextField)`
 - pour des cellules dont la représentation textuelle sera éditée
 - édition sur double clic

DefaultCellEditor.EditorDelegate

- Classe interne qui gère toutes les commandes d'édition au sein de DCE
 - `Object getCellEditorValue()`
 - **void** `setValue(Object value)`
 - **boolean** `startCellEditing(EventObject)`
 - **void** `cancelCellEditing()`
 - **boolean** `stopCellEditing()`
 - **boolean** `isCellEditable(EventObject)`
 - **void** `itemStateChanged(ItemEvent)`
 - **boolean** `shouldSelectCell(EventObject)`

Retour sur l'éditeur de dates

```
public class DateEditor extends DefaultCellEditor {  
    public DateEditor() {  
        super(new JTextField());  
        final JTextField tf = (JTextField) editorComponent;  
        tf.setHorizontalAlignment(JTextField.RIGHT);  
        delegate = new EditorDelegate() {  
            public Object getCellEditorValue() {  
                CAL.clear();  
                try {  
                    CAL.set(Long.parseLong(tf.getText()));  
                } catch (NumberFormatException e) {  
                    // rien  
                }  
                return CAL.getTime();  
            }  
            public void setValue(Object value) {  
                if (value instanceof Date) {  
                    tf.setText(String.valueOf(((Date) value).getTime()));  
                } else {  
                    tf.setText("0");  
                }  
            }  
        };  
    }  
}  
  
table.setDefaultEditor(Date.class, new DateEditor());
```

composant Swing effectuant l'édition

classe interne à DCE

objet qui gère toutes les commandes d'édition

Sélection

- Propriétés de `JTable` (**boolean**, RW, liées)
 - sélection des lignes (défaut : **true**)
 - `rowSelectionAllowed`
 - sélection des colonnes (défaut : **false**)
 - `columnSelectionAllowed`
 - sélection des cellules (défaut : **false**)
 - `cellSelectionEnabled`
- 2 modèles de sélection
 - lignes : `selectionModel` de `JTable` (`ListSelectionModel`, RW, liée)
 - cols : `selectionModel` de `TableColumnModel` (`ListSelectionModel`, RW, non liée)

Modes de sélection

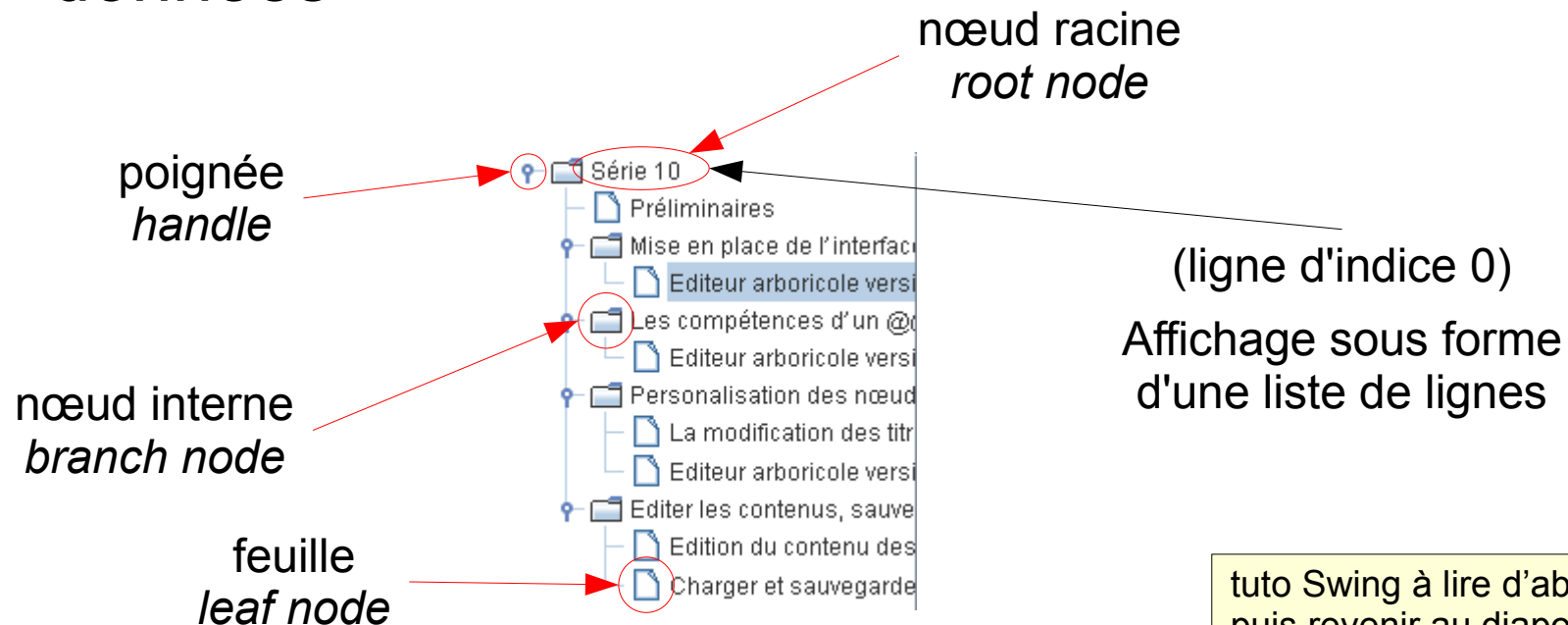
- `selectionMode` de `JTable` (**int**, W, non liée) contrôle le mode de sélection des lignes **et** des colonnes
 - `MULTIPLE_INTERVAL_SELECTION` (défaut)
 - `SINGLE_INTERVAL_SELECTION`
 - `SINGLE_SELECTION`
- Différencier le comportement ligne/colonne
 - ```
tbl.setSelectionMode(
 ListSelectionModel.SINGLE_SELECTION
);
```
  - ```
tbl.getColumnModel().getSelectionModel().setSelectionMode(  
    ListSelectionModel.MULTIPLE_INTERVAL_SELECTION  
);
```

Ajouter une ligne à une table

- Ajouter une ligne vierge à une table, après que la table aura été affichée :
 - modèle mutable (comme `DefaultTableModel`) doté d'une méthode `addRow` (par ex.)
- S'assurer que la ligne deviendra visible juste après l'ajout
 - rajouter un écouteur de modification du modèle de données qui fait défiler les lignes jusqu'au bon endroit
 - utiliser `scrollRectToVisible` de la classe `JComponent` (voir JavaDoc et TP 7)

Présentation de JTree

- `javax.swing.JTree`
 - affichage et édition d'un modèle arborescent de données



tuto Swing à lire d'abord,
puis revenir au diaporama
et lire les diapos 60-100

Vocabulaire

- Accès à un nœud
 - par son chemin (*path*) à partir de la racine
 - instance de `TreePath`
 - par un numéro de ligne d'affichage (*row index*)
 - le numéro de ligne change en fonction du pliage
- Deux états possibles
 - **nœud déplié** (*expanded*)
 - nœud interne, les enfants directs sont exposés à la vue
 - mais pas nécessairement tous visibles
- État des nœuds
 - **affichable** (*viewable*) / **masqué** (*hidden*)
 - tous les ancêtres dépliés / un ancêtre au moins est plié
 - **visible** (*displayed*) :
 - affichable et dans la zone visible de l'arbre

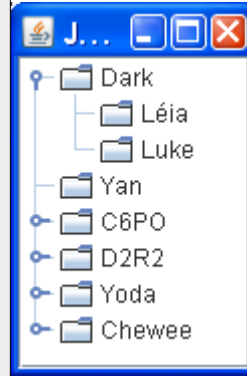
nœud plié (*collapsed*) :
nœud interne non déplié

Utilitaires de pliage/dépliage

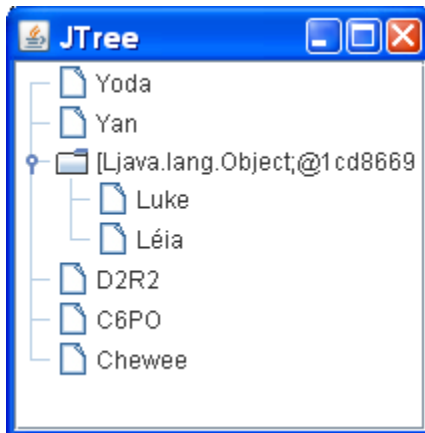
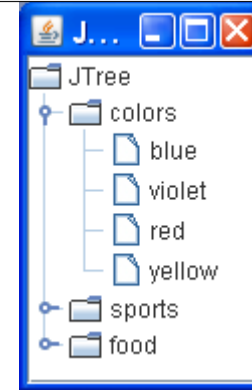
- Quelques méthodes utilitaires de `JTree` :
 - `boolean isCollapsed(TreePath)`
 - `boolean isExpanded(TreePath)`
 - `void collapsePath(TreePath)`
 - `void expandPath(TreePath)`
 - `boolean isVisible(TreePath)`
 - indique si le nœud de chemin donné est affichable
 - `void makeVisible(TreePath)`
 - rend affichable le nœud de chemin donné

Mises en place simples

```
Hashtable m = new Hashtable();
Hashtable m2 = new Hashtable();
m.put("Yoda", new Hashtable());
m.put("Yan", new Hashtable());
m.put("Dark", m2); {
    m2.put("Luke", new Hashtable());
    m2.put("Léia", new Hashtable());
}
m.put("D2R2", new Hashtable());
m.put("C6PO", new Hashtable());
m.put("Chewee", new Hashtable());
JTree tree = new JTree(m);
```



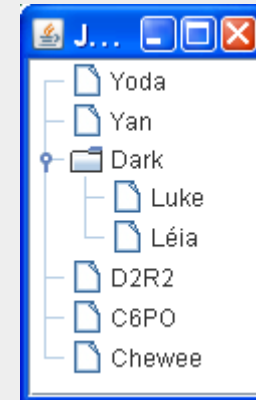
```
JTree tree = new JTree();
```



```
Object[] m = new Object[] {
    "Yoda", "Yan",
    new Object[] {"Luke", "Léia" },
    "D2R2", "C6PO", "Chewee"
};
JTree tree = new JTree(m);
```

```
Vector m = new Vector();
Vector m2 = new Vector() {
    public String toString() {
        return "Dark";
    }
};
```

```
m.add("Yoda");
m.add("Yan");
m.add(m4); {
    m2.add("Luke");
    m2.add("Léia");
}
m.add("D2R2");
m.add("C6PO");
m.add("Chewee");
JTree tree = new JTree(m);
```



Gestion du défilement

- `JTree` **implements** `Scrollable`
 - gestion du défilement à l'aide d'un `JScrollPane`
- **`void`** `scrollPathToVisible(TreePath)`
 - Assure qu'un nœud de chemin donné est visible
- **`void`** `scrollRowToVisible(int)`
 - Assure qu'une ligne donnée (à partir de 0) est visible
 - /!\ une ligne n'affiche pas toujours le même nœud, tout dépend du pliage des nœuds intermédiaires

- **boolean** `isRootVisible()`
 - indique si la racine est visible
- **void** `setRootVisible(boolean)`
 - rend la racine visible ou non
- **int** `getVisibleRowCount()`
 - indique le nombre de lignes visibles
- **void** `setVisibleRowCount(int)`
 - fixe le nombre de lignes visibles

Modèle de données

- `TreeModel` : modèle de données des `JTree`
 - `Object getChild(Object parent, int index)`
 - fils à la position `index` du nœud `parent`
 - `index` = position du fils par rapport à `parent` (pour l'affichage, à partir de 0)
 - `int getChildCount(Object parent)`
 - nombre de fils du nœud `parent`
 - `int getIndexOfChild(Object parent, Object child)`
 - position du nœud `child` dans le nœud `parent` (pour l'affichage)
 - -1 si pas de filiation

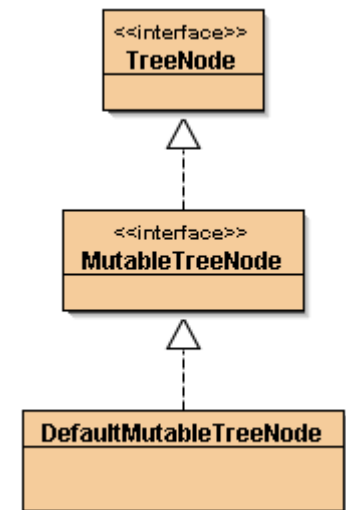
- `Object getRoot()`
 - nœud racine de l'arbre
- `boolean isLeaf(Object node)`
 - `node` est une feuille ?
- `void valueForPathChanged(TreePath p, Object v)`
 - signale que l'utilisateur a modifié la valeur du nœud identifié par `p`
 - modèle doit notifier (`fireTreeNodesChanged`) en cas de réelle modification
- `void addTreeModelListener(TreeModelListener)`
`void removeTreeModelListener(TreeModelListener)`
 - Ajout/suppression d'un écouteur de modification du modèle

Mise en place avec un modèle

- `JTree (TreeModel)`
 - constructeur qui utilise le modèle fourni
 - affiche la racine, contrairement aux mises en places simples

Gestion des nœuds

- Une classe pour les nœuds mutables
- Deux interfaces
 - nœuds mutables
 - nœuds non mutables
- interface `TreeNode` :
 - `Enumeration children()`
 - `Enumeration` des fils de ce nœud
 - `boolean getAllowsChildren()`
 - le nœud peut-il porter des fils ?
 - `TreeNode getParent()`
 - nœud père de ce nœud



- `TreeNode getChildAt(int childIndex)`
 - nœud fils d'index `childIndex`
 - utilisée par `TreeModel.getChild`
- `int getChildCount()`
 - nombre de fils de ce nœud
 - utilisée par `TreeModel.getChildCount`
- `int getIndex(TreeNode node)`
 - index de `node` pour ce nœud
 - utilisée par `TreeModel.getIndexOfChild`
- `boolean isLeaf()`
 - ce nœud est-il une feuille ?
 - utilisée par `TreeModel.isLeaf`

Mise en place avec une racine

- `JTree(TreeNode, boolean)`
 - crée son propre modèle (DTM) :
 - racine = nœud passé en argument
 - paramètre booléen = manière de déterminer les feuilles :
 - **false** : `getChildCount() == 0` \Rightarrow feuille
 - **true** : `!getAllowsChildren()` \Rightarrow feuille
 - affiche la racine, contrairement aux mises en places simples
- `JTree(TreeNode) \equiv JTree(TreeNode, false)`

Nœuds mutables

- `TreeNode` → ensemble des services que propose un nœud au modèle
 - non mutables
 - ne peuvent pas changer de position dans l'arbre
 - ne peuvent pas changer de valeur
- `MutableTreeNode` **extends** `TreeNode` pour gérer la mutabilité (position et valeur)

- **void** insert(MutableTreeNode child, **int** index)
 - insère `child` en `index` comme fils de ce nœud
 - penser à détacher `child` de son père
 - penser à attacher **this** comme nouveau père
- **void** remove(**int** index)
 - supprime le fils en `index` pour ce nœud
 - penser à détacher le fils en `index` de ce nœud
- **void** remove(MutableTreeNode node)
 - supprime `node` des fils de ce nœud
 - penser à détacher `node` de ce nœud
- **void** removeFromParent()
 - détache ce nœud de son père
- **void** setParent(MutableTreeNode newParent)
 - attache ce nœud à `newParent`
- **void** setUserObject(Object object)
 - fixe la nouvelle étiquette de ce nœud à `object`

TreePath

- Notion de chemin à partir de la racine
- Seule méthode sûre pour identifier un nœud :
 - Selon l'état de pliage des nœuds, l'index d'un nœud dans l'arbre peut être variable ou indéterminé
- Implémentation sous forme de tableau de nœuds, ordonné de la racine jusqu'au dernier nœud du chemin
- Notion de chemin **descendant** :
 - $p2$ descend de $p1 \Leftrightarrow p1$ est préfixe de $p2$

- `Object getLastPathComponent()`
 - dernier nœud de ce chemin
- `TreePath getParentPath()`
 - nouveau chemin constitué des nœuds de ce chemin sauf le dernier
- `Object[] getPath()`
 - ce chemin sous forme de tableau ordonné
- `Object getPathComponent(int index)`
 - nœud de ce chemin situé en `index`
- `int getPathCount()`
 - nombre de nœuds de ce chemin
- `boolean isDescendant(TreePath aTreePath)`
 - `aTreePath` est-il un descendant de ce chemin ?
- `TreePath pathByAddingChild(Object child)`
 - nouveau chemin constitué des nœuds de ce chemin plus `child` à la fin

Gestion des évènements de modification du modèle de données

- `TreeModelListener` pour écouter
 - modifications, insertions et suppressions de nœuds
- Attention :
 - Modification de la structure par le modèle ⇒ notification automatique des écouteurs
 - Modification de la structure par les nœuds ⇒ notification à la charge du programmeur
 - Avantage : permet d'effectuer plusieurs modifications et une seule notification (efficacité)
 - Inconvénient : ne pas oublier la notification !

TreeModelListener

- **void** `treeNodesChanged(TreeModelEvent evt)`
 - modification des fils de `evt.getTreePath()`
 - fils modifiés dans `evt.getChildren()`
 - structure de l'arbre inchangée
- **void** `treeStructureChanged(TreeModelEvent evt)`
 - modification importante de la structure de l'arbre sous le nœud `evt.getPath()`
- **void** `treeNodesInserted(TreeModelEvent evt)`
 - insertion des nœuds `evt.getChildren()` sous le nœud `evt.getPath()`
- **void** `treeNodesRemoved(TreeModelEvent evt)`
 - suppression des nœuds `evt.getChildren()` sous le nœud `evt.getPath()`

TreeModelEvent

- `Object[] getPath()`
 - chemin d'accès au nœud sous lequel a eu lieu la modification
- `TreePath getTreePath()`
 - chemin d'accès au nœud sous lequel a eu lieu la modification
 - dans `treeStructureChanged`, retourne l'ancêtre commun à tous les nœuds modifiés
 - dans `treeNodesChanged`, retourne la racine

- `int[] getChildIndices()`
 - positions des enfants modifiés
 - suppression → indices des enfants enlevés par rapport au tableau initial
 - insertion → indices des enfants ajoutés par rapport au tableau initial
 - dans `treeStructureChanged` → indices des modifications (tableau à plusieurs dimensions ?)
 - dans `treeNodesChanged` → **`null`**
- `Object[] getChildren()`
 - enfants du nœud `getPath()` aux positions indiquées par `getChildIndices()`

Exemple

- Différence entre modification du modèle et modification des nœuds :

```
DefaultTreeModel model = (DefaultTreeModel) tree.getModel();  
for (int i = 0; i < nodesToInsert.length; i++) {  
    model.insertNodeInto(nodesToInsert[i], parent, i);  
}
```

```
-----  
  
DefaultTreeModel model = (DefaultTreeModel) tree.getModel();  
int[] indices = new int[nodesToInsert.length];  
for (int i = 0; i < nodesToInsert.length; i++) {  
    parent.insert(nodesToInsert[i], i);  
    indices[i] = i;  
}  
model.nodesWhereInserted(parent, indices);
```


Rendu visuel des nœuds

- L'interface `TreeCellRenderer` ne contient que la méthode :
 - ```
Component getTreeCellRendererComponent(
 JTree tree, // arbre
 Object value, // le nœud à afficher
 boolean selected, // nœud sélectionné ?
 boolean expanded, // nœud déplié ?
 boolean leaf, // feuille ?
 int row, // ligne d'affichage
 boolean hasFocus // a le focus ?
)
```
- `DefaultTreeCellRenderer` dérive `JLabel` et implémente cette interface

# Calcul du rendu d'un nœud

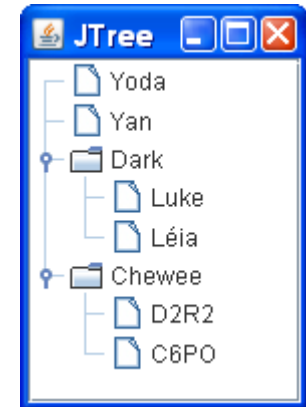
- Un nœud doit savoir afficher son étiquette avec `toString`
- Calcul du texte à afficher :
  - Le `TreeUI` appelle sur le `renderer` `getTreeCellRendererComponent(...)`
    - qui appelle `convertValueToText(value)` sur le `JTree`
      - qui appelle `toString()` sur `value`
    - puis modifie le texte du `renderer` (`JLabel`) avec la valeur retournée
- Pour adapter ce calcul : dériver `JTree` et redéfinir la méthode `convertValueToText`

# Changer l'affichage des nœuds sans changer de renderer

- En utilisant les propriétés de DTCR :

|                             |       |    |
|-----------------------------|-------|----|
| backgroundNonSelectionColor | Color | RW |
| backgroundSelectionColor    | Color | RW |
| borderSelectionColor        | Color | RW |
| closedIcon                  | Icon  | RW |
| defaultClosedIcon           | Icon  | R  |
| defaultLeafIcon             | Icon  | R  |
| defaultOpenIcon             | Icon  | R  |
| font                        | Font  | RW |
| leafIcon                    | Icon  | RW |
| openIcon                    | Icon  | RW |
| textNonSelectionColor       | Color | RW |
| textSelectionColor          | Color | RW |

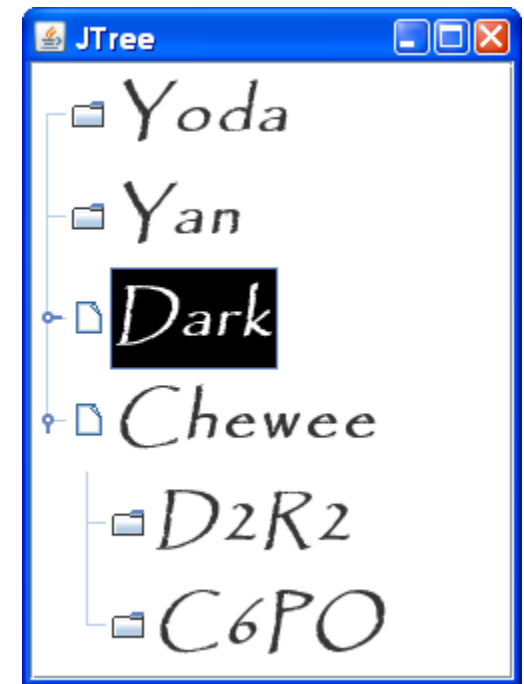
# Example



```
JTree tree = new JTree(new Object[] {...});
DefaultTreeCellRenderer r = tree.getCellRenderer();

r.setBackgroundSelectionColor(Color.BLACK);
r.setTextSelectionColor(Color.WHITE);
r.setLeafIcon(r.getDefaultOpenIcon());
r.setClosedIcon(r.getDefaultLeafIcon());
r.setOpenIcon(r.getDefaultLeafIcon());
r.setFont(
 new Font("Papyrus", Font.BOLD | Font.ITALIC, 32)
);

tree.setRowHeight(Math.min(0, tree.getRowHeight() - 1));
```

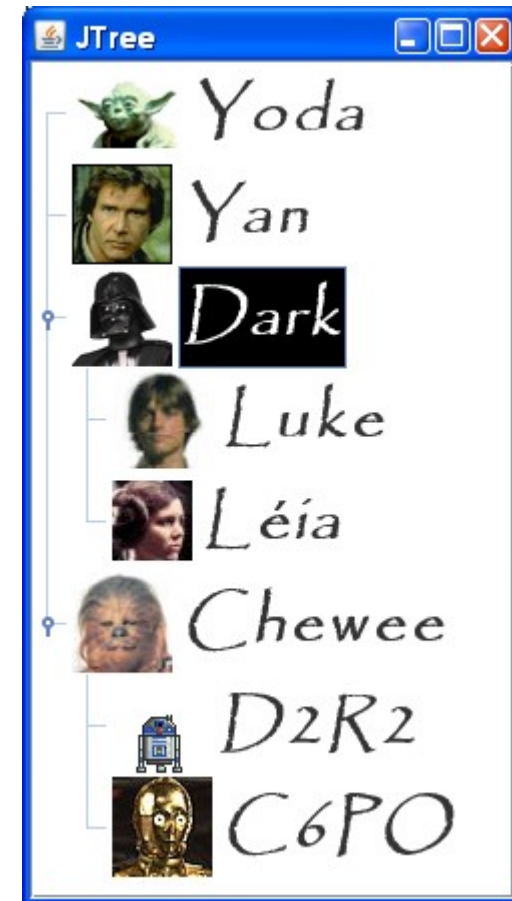


# Définir un *renderer*

```
class MyTreeCellRenderer extends DefaultTreeCellRenderer {
 private Map<String, ImageIcon> icons;
 public MyTreeCellRenderer(Map<String, ImageIcon> icons) {
 this.icons = icons;
 setBackgroundSelectionColor(Color.BLACK);
 setTextSelectionColor(Color.WHITE);
 setFont(new Font("Papyrus", ...));
 }
 public Component getTreeCellRendererComponent(
 JTree tree, Object value, boolean sel,
 boolean expanded, boolean leaf, int row,
 boolean hasFocus) {
 ImageIcon i = (ImageIcon) icons.get(value.toString());
 if (leaf) {
 setLeafIcon(i);
 } else if (expanded)
 setOpenIcon(i);
 } else {
 setClosedIcon(i);
 }
 return super.getTreeCellRendererComponent(
 tree, value, sel, expanded, leaf, row, hasFocus
);
 }
}
```

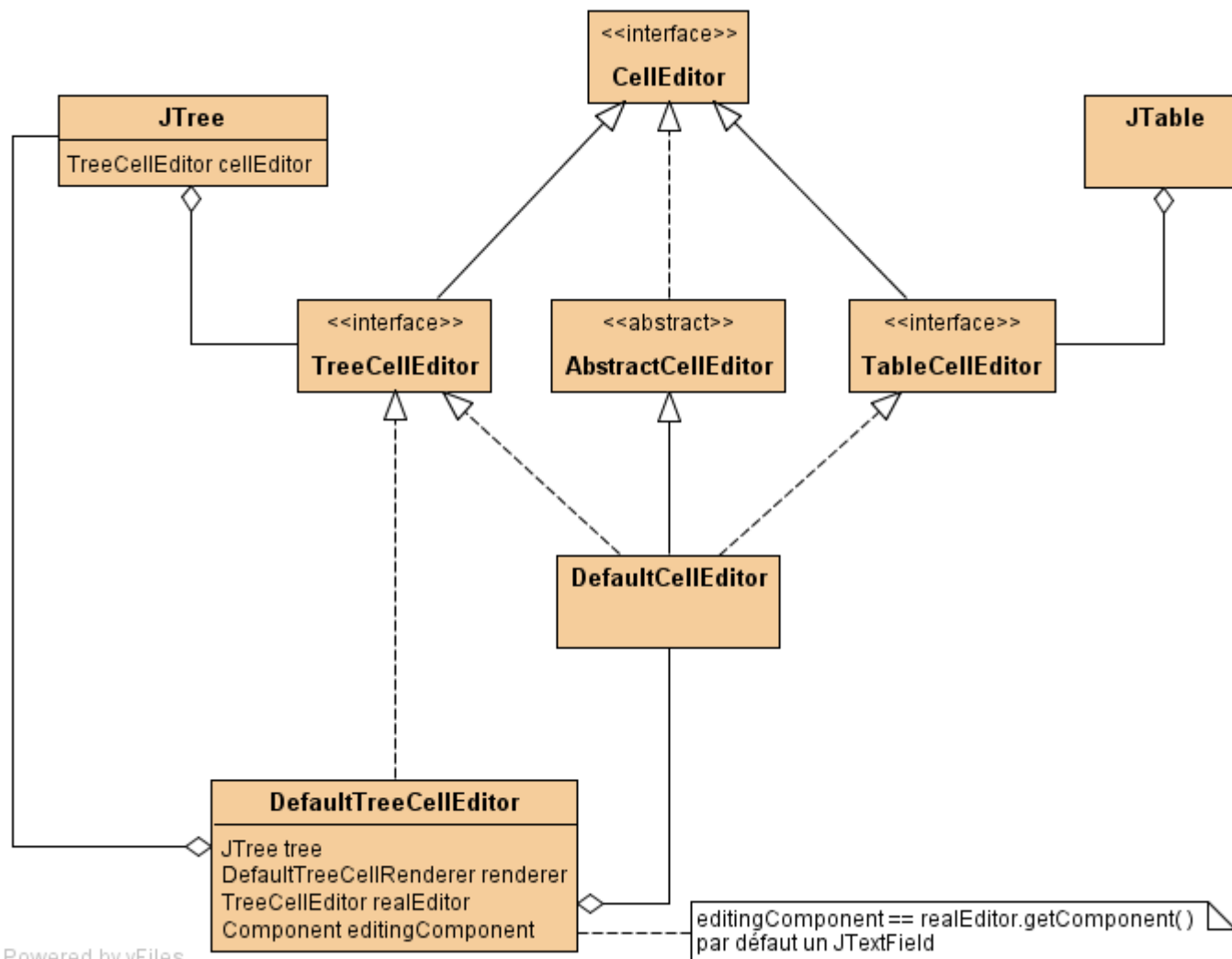
---

```
Map<String, ImageIcon> icons = ...;
JTree tree = new JTree(new Object {...});
tree.setCellRenderer(new MyTreeCellRenderer(icons));
```



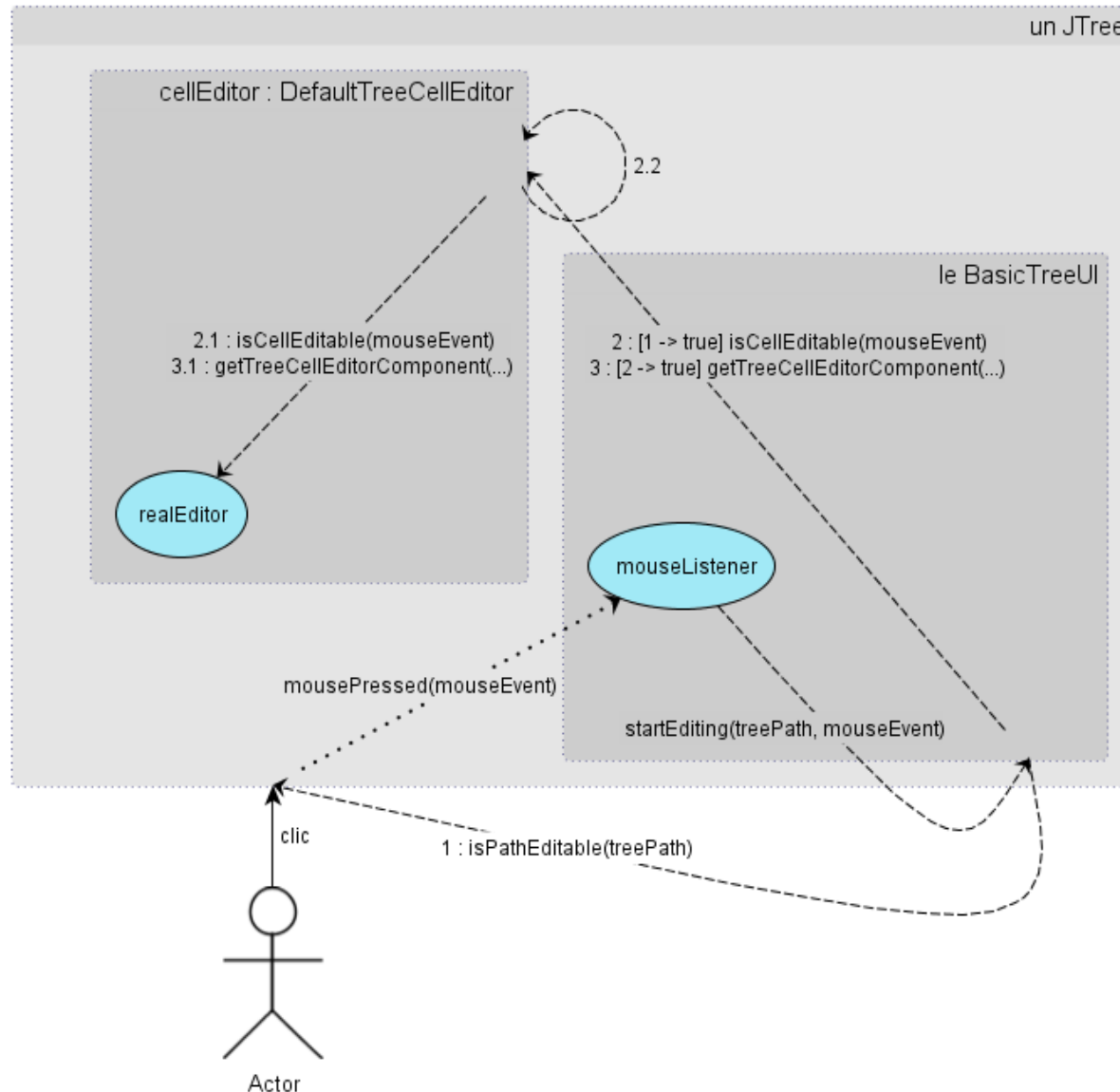
# Edition des nœuds

- Éditeur de nœuds = instance de `TreeCellEditor`
- `DefaultTreeCellEditor` = implémentation par défaut
  - utilise un `TreeCellRenderer` pour afficher l'icône du nœud
  - utilise un délégué de type `DefaultCellEditor` pour éditer le texte du nœud
- `DefaultTreeCellEditor` se construit avec :
  - un `JTree`
  - un `DefaultTreeCellRenderer`
  - [un `TreeCellEditor`]



# Calcul de l'éditabilité d'un nœud

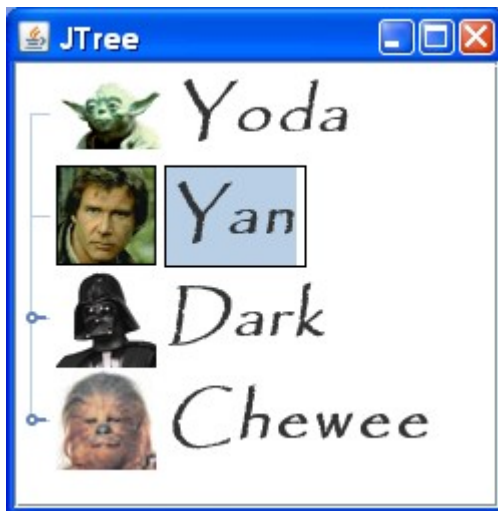
2.2.a : [2.1 -> false] retourne false  
2.2.b : [2.1 -> true && canEditImmediately(mouseEvent)] retourne true  
2.2.c : [2.1 -> true && ! canEditImmediately(mouseEvent) && shouldStartEditingTimer(mouseEvent)] startEditingTimer() ; retourne false  
2.2.d : [2.1 -> true && ! canEditImmediately(mouseEvent) && ! shouldStartEditingTimer(mouseEvent)] retourne false



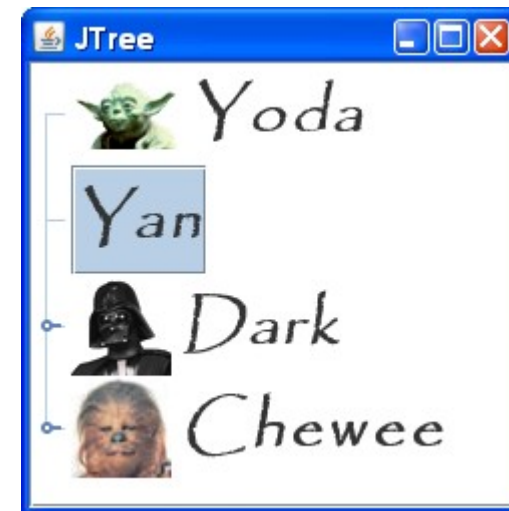


# Comparaison des deux types d'éditeurs

- `DefaultCellEditor` utilise un `JTextField` placé sur toute la surface d'un nœud
- `DefaultTreeCellEditor` utilise l'icône du renderer et le `JTextField` du `DefaultCellEditor`



Avec un `DefaultTreeCellEditor`



Avec un `DefaultCellEditor`

# Contrôle de l'édition des nœuds

- Se fait au niveau du `JTree` :
  - Contrôle global
    - propriété `"editable"` (**boolean**, RW, liée)
    - Gère globalement l'éditabilité de tous les nœuds
  - Contrôle au cas par cas
    - **boolean** `isPathEditable(TreePath)`
    - Méthode appelée au début du calcul de l'éditabilité d'un nœud
    - Retourne `isEditable()` par défaut
    - Adaptable par redéfinition en installant un algorithme filtrant les nœuds éditables de ceux qui ne le sont pas

# Modification des poignées

- Les icônes des poignées ne sont pas gérées par le `render` (comme pour les nœuds) mais par le *UI-delegate* (instance de `BasicTreeUI`)
- Suppression des poignées :
  - Déterminer la classe exacte du *UI-delegate* (`MetalTreeUI`, `MotifTreeUI`, `WindowsTreeUI`)
  - Dériver cette classe en y redéfinissant la méthode `boolean shouldPaintExpandControlMethod()` pour qu'elle retourne **false**

```
MetalTreeUI ui = new MetalTreeUI() {
 protected boolean shouldPaintExpandControl(TreePath p, int r,
 boolean isExpanded, boolean wasExpanded, boolean leaf) {
 return false;
 }
};
tree.setUI(ui);
```

- Modification pour un `JTree` particulier :

- `((BasicTreeUI) tree.getUI()).setExpandedIcon(Icon)`
- `((BasicTreeUI) tree.getUI()).setCollapsedIcon(Icon)`
- `((BasicTreeUI) tree.getUI()).setLeftChildIndent(int)`
- `((BasicTreeUI) tree.getUI()).setRightChildIndent(int)`

- Modification pour tous les `JTree` :

- Utiliser les propriétés de L&F des `BasicTreeUI` gérées par le `UIManager`

- `UIManager.put("Tree.expandedIcon", Icon)`
- `UIManager.put("Tree.collapsedIcon", Icon)`
- `UIManager.put("Tree.leftChildIndent", Integer)`
- `UIManager.put("Tree.rightChildIndent", Integer)`

# Sélection des nœuds

- Avec un `DefaultTreeSelectionModel`
  - Propriété associée du `JTree` :
    - `selectionModel` (`TreeSelectionModel`, RW, liée)
- Suppression de la possibilité de sélectionner les nœuds d'un arbre avec `setSelectionModel(null)`
- Modes de sélection :
  - Propriété associée du modèle de sélection :
    - `selectionMode` (**`int`**, RW, non liée)

# Modes de sélection des nœuds

- Valeurs possibles (`TreeSelectionModel`) :
  - `CONTIGUOUS_TREE_SELECTION`
    - Une seule zone contiguë de nœuds sélectionnés
  - `DISCONTIGUOUS_TREE_SELECTION`
    - Plusieurs zones de nœuds sélectionnés
  - `SINGLE_TREE_SELECTION`
    - Un seul nœud sélectionné
- Remarques :
  - Sélectionner un nœud interne ne sélectionne pas ses descendants.
  - La sélection d'une zone contiguë sur plusieurs niveaux ne sélectionne que les nœuds visibles.

# Notification de sélection

- Écoute les évènements de sélection avec `TreeSelectionListener` :
  - **`void`** `valueChanged(TreeSelectionEvent)`
- Ces écouteurs peuvent être attachés
  - au `JTree` directement
    - source de l'évènement est un `JTree`
  - à son modèle de sélection
    - source de l'évènement est un `TreeSelectionModel`

# TreeSelectionEvent

- `Object cloneWithSource(Object newSource)`
  - copie de l'évènement dont la source est `newSource`
- `TreePath getNewLeadSelectionPath()`
  - chemin le plus bas dans la sélection après le changement de sélection
- `TreePath getOldLeadSelectionPath()`
  - chemin le plus bas dans la sélection avant le changement de sélection



- `TreePath getPath()`
  - chemin ajouté à ou supprimé de la sélection
- `TreePath[] getPaths()`
  - chemins ajoutés à ou supprimés de la sélection
- **boolean** `isAddedPath()`
  - le chemin a-t-il été ajouté à (**true**) ou supprimé de (**false**) la sélection ?
- **boolean** `isAddedPath(int index)`
  - le chemin en `index` dans la sélection a-t-il été ajouté (**true**) ou supprimé (**false**) ?
- **boolean** `isAddedPath(TreePath path)`
  - `path` a-t-il été ajouté à (**true**) ou supprimé de (**false**) la sélection ?

# Sélection par programmation

- Sélection des nœuds directement à l'aide de l'arbre possible par modification des propriétés
  - `lastSelectedPathComponent` (`Object`, R, non liée)
  - `anchorSelectionPath` (`TreePath`, RW, liée)
  - `leadSelectionPath` (`TreePath`, RW, liée)
  - `selectionCount` (**`int`**, R, non liée)
  - `selectionEmpty` (**`boolean`**, R, non liée)
  - `selectionPath` (`TreePath`, RW, non liée)
  - `selectionPaths` (`TreePath[]`, RW, non liée)
  - `leadSelectionRow` (**`int`**, R, non liée)
  - `maxSelectionRow` (**`int`**, R, non liée)
  - `minSelectionRow` (**`int`**, R, non liée)
  - `selectionRow` (**`int`**, W, non liée)
  - `selectionRows` (**`int[]`**, RW, non liée)

# Gestion du pliage

- Pour programmer le (dé)pliage des nœuds internes, on utilise les méthodes de `JTree` :
  - `void [collapse|expand][Path|Row](...)`
- Pour détecter les (dé)ploiages :
  - Juste avant : `TreeWillExpandListener`
    - `void treeWillCollapse(TreeExpansionEvent) throws ExpandVetoException`
    - `void treeWillExpand(TreeExpansionEvent) throws ExpandVetoException`
  - Juste après : `TreeExpansionListener`
    - `void treeCollapsed(TreeExpansionEvent)`
    - `void treeExpanded(TreeExpansionEvent)`

# Intérêt des TreeWillExpandListener

- Permet d'accepter/refuser le (dé)pliage
  - les exceptions servent uniquement à arrêter l'action
  - pour tester si l'action a aboutie, utiliser les méthode de `JTree` :
    - `boolean isExpanded([int|TreePath])`
    - `boolean isCollapsed([int|TreePath])`
- Ex. d'utilisation : peuplement d'un nœud interne juste avant le moment où le client va le déplier