Interface Graphique en Java 1.6

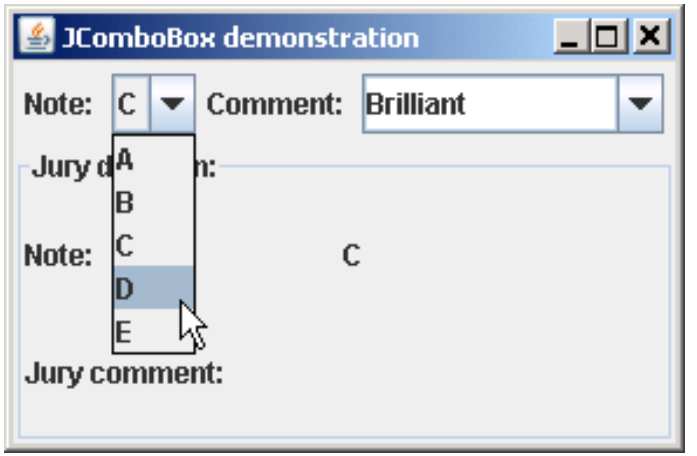
Combos, sliders, spinners et arbres

Sébastien Paumier

La JComboBox

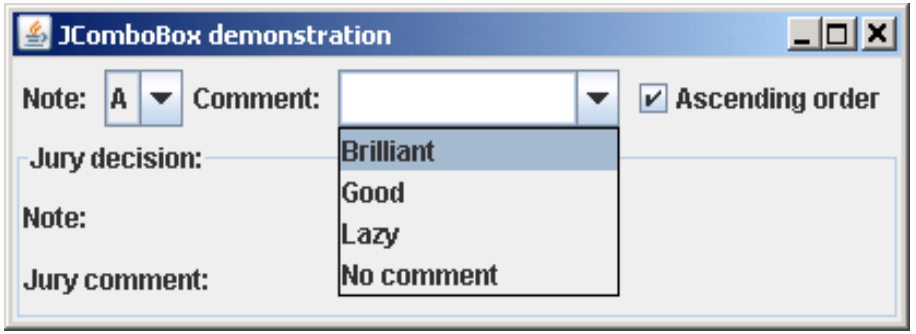
- boîte déroulante qui combine une zone de texte et une liste
- utilise un **ComboBoxModel**
- on peut sélectionner un élément
- on est prévenu de la sélection par un **ActionListener**:

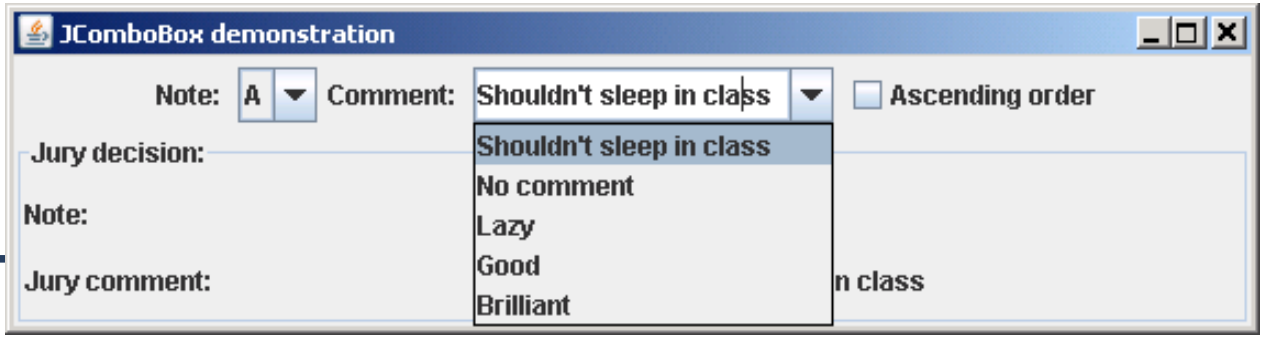
```
combo1.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        String s=(String)combo1.getSelectedItem();
        if (s!=null) {
            noteLabel.setText(s);
        }
    }
});
```



La JComboBox

- peut devenir éditable avec `setEditable(true)`
- le `ActionListener` est aussi invoqué quand on valide le texte dans l'éditeur, mais on doit gérer à la main l'insertion





Un ComboBoxModel

- pour ajouter un élément, sans doublon, et à la bonne place, on va écrire notre propre modèle:

```
/* We extend AbstractListModel in order to have the fire... methods */
public class MyComboModel extends AbstractListModel implements MutableComboBoxModel {

    private String selectedItem;
    private ArrayList<String> elements;

    boolean ascending=true;

    public MyComboModel(String... items) {
        elements=new ArrayList<String>();
        for (String s:items) {
            addElement(s);
        }
    }
...
}
```

on veut ajouter des éléments

Un ComboBoxModel

- pour l'ajout, on cherche la position d'insertion:

```
@Override public void addElement(Object obj) {
    int index=Arrays.binarySearch(elements.toArray(),obj,myComparator);
    if (index>=0) {
        /* We don't want duplicates */
        return;
    }
    index=-1-index;
    insertElementAt(obj,index);
}

Comparator<Object> myComparator=new Comparator<Object>() {
    @Override public int compare(Object o1, Object o2) {
        String s1=(String)o1;
        String s2=(String)o2;
        return s1.compareTo(s2)*(ascending?1:-1);
    }
};
```

Un ComboBoxModel

- quand on change l'ordre de tri, on doit modifier les données:

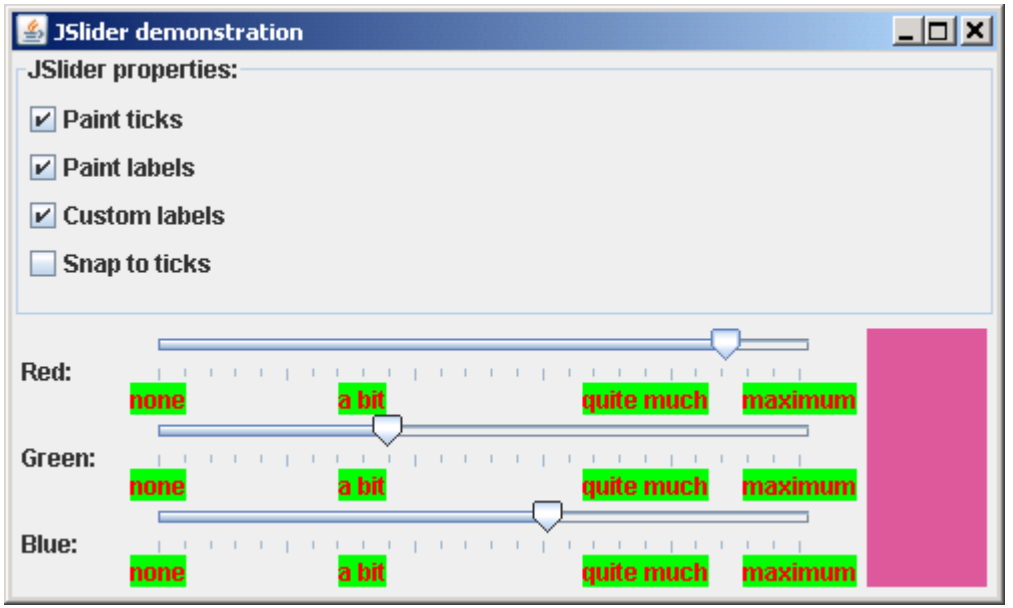
```
public void setAscendingSort(boolean ascending) {
    if (this.ascending==ascending) {
        return;
    }
    this.ascending=ascending;
    Collections.sort(elements,myComparator);
    fireContentsChanged(this,0,getSize());
}
```

Le JSlider

- curseur à déplacer sur une barre, défini par un minimum, un maximum et une valeur courante, tous entiers
- utilise un `BoundedRangeModel`
- utilise un `ChangeListener` pour prévenir qu'une de ces valeurs a changé
- peut afficher ou non des encoches, grandes et petites, ainsi que des labels, personnalisés ou non

Le JSlider

- pour personnaliser les labels, il faut créer un `Dictionary<Integer,JComponent>` et le passer à `setLabelTable`



Le JSlider

- exemple de labels personnalisés:

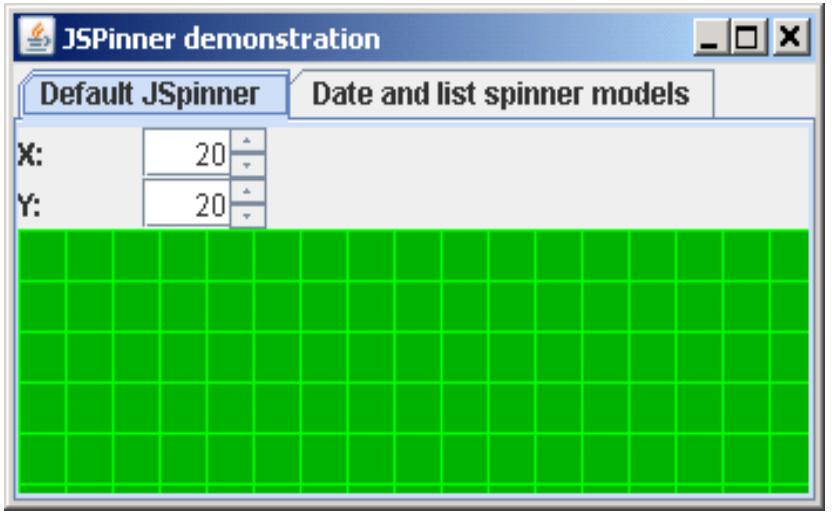
```
final Hashtable<Integer,JComponent> customLabels=new Hashtable<Integer,JComponent>();
customLabels.put(0,makeLabel("none"));
customLabels.put(80,makeLabel("a bit"));
customLabels.put(190,makeLabel("quite much"));
customLabels.put(255,makeLabel("maximum"));
```

on décide des valeurs qui auront un label

```
private static JLabel makeLabel(String string) {
    JLabel l=new JLabel(string);
    l.setOpaque(true);
    l.setForeground(Color.RED);
    l.setBackground(Color.GREEN);
    return l;
}
```

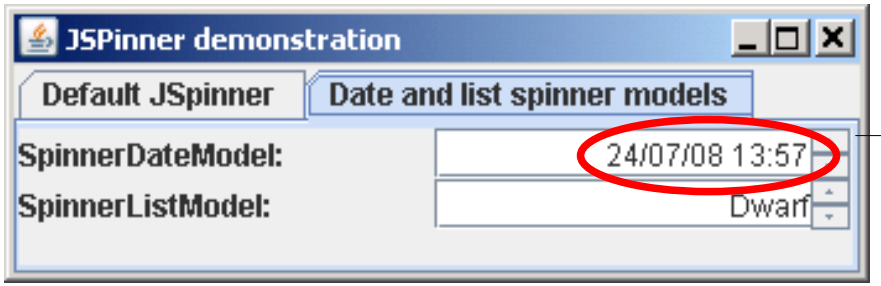
Le JSpinner

- composant destiné à éditer une valeur, soit au clavier, soit avec 2 petites flèches
- comportement similaire au `JSlider` (minimum, maximum, etc)



Le JSpinner

- le plus souvent, on utilise un **`SpinnerNumberModel`**
- possibilité d'utiliser d'autres modèles prédéfinis:
  - **`SpinnerDateModel`**: valeurs=dates
  - **`SpinnerListModel`**: valeurs à choisir dans une liste prédéfinie



le spinner peut n'agir que sur une partie de l'entrée

Saisies filtrées

- les `SpinnerModel` refusent les valeurs qui ne correspondent pas à ce qu'ils attendent, comme par exemple un mot au lieu d'un nombre
- ce type de filtrage peut également être mis en œuvre grâce au `JFormattedTextField`
- exemple: filtrage d'adresses IP

Saisies filtrées

- on définit un formateur capable de passer d'une chaîne à un objet et vice-versa
- lève une **`ParseException`** en cas d'erreur

```
public class IPFormatter extends AbstractFormatter {

    @Override
    public String valueToString(Object value) throws ParseException {
        if (value==null) {
            return null;
        }
        try {
            return ((Inet4Address)value).getHostAddress();
        } catch (ClassCastException e) {
            throw new ParseException("Invalid IPv4 address",0);
        }
    }

...

}
```

Saisies filtrées

```
public class IPFormatter extends AbstractFormatter {

    private final static Pattern pattern=Pattern.compile(
            "([0-9]{1,3})\\.([0-9]{1,3})\\.([0-9]{1,3})\\.([0-9]{1,3})");

    @Override
    public Inet4Address stringToValue(String text) throws ParseException {
        Matcher matcher=pattern.matcher(text);
        if (!matcher.matches()) {
            throw new ParseException("Invalid IPv4 address", 0);
        }
        byte[] b = new byte[4];
        for (int i=0;i<4;i++) {
            int n = Integer.parseInt(matcher.group(1+i));
            if (n < 0 || n > 255)
                throw new ParseException("Invalid IPv4 address", 0);
            b[i] = (byte) n;
        }
        try {
            return (Inet4Address) InetAddress.getByAddress(b);
        } catch (UnknownHostException e) {
            throw new ParseException("Invalid IPv4 address", 0);
        }
    }

...

}
```

Saisies filtrées

- on passe ensuite ce formateur au constructeur de **`JFormattedTextField`**
- à chaque édition, on va tester si l'entrée est valide:

```
text.addCaretListener(new CaretListener() {
    @Override public void caretUpdate(CaretEvent e) {
        try {
            text.commitEdit();
            text.setForeground(Color.GREEN);
            InetAddress a=(InetAddress) text.getValue();
            if (a==null) return;
            multicast.setText("Multicast: "+a.isMulticastAddress());
            loopback.setText("Loopback address: "+a.isLoopbackAddress());
        } catch (ParseException e1) {
            text.setForeground(Color.RED);
            multicast.setText("Multicast: ");
            loopback.setText("Loopback address: ");
        }
    }
});
```

lève une **`ParseException`** en cas de problème

Saisies filtrées

- on obtient ainsi une belle application:

Les arbres

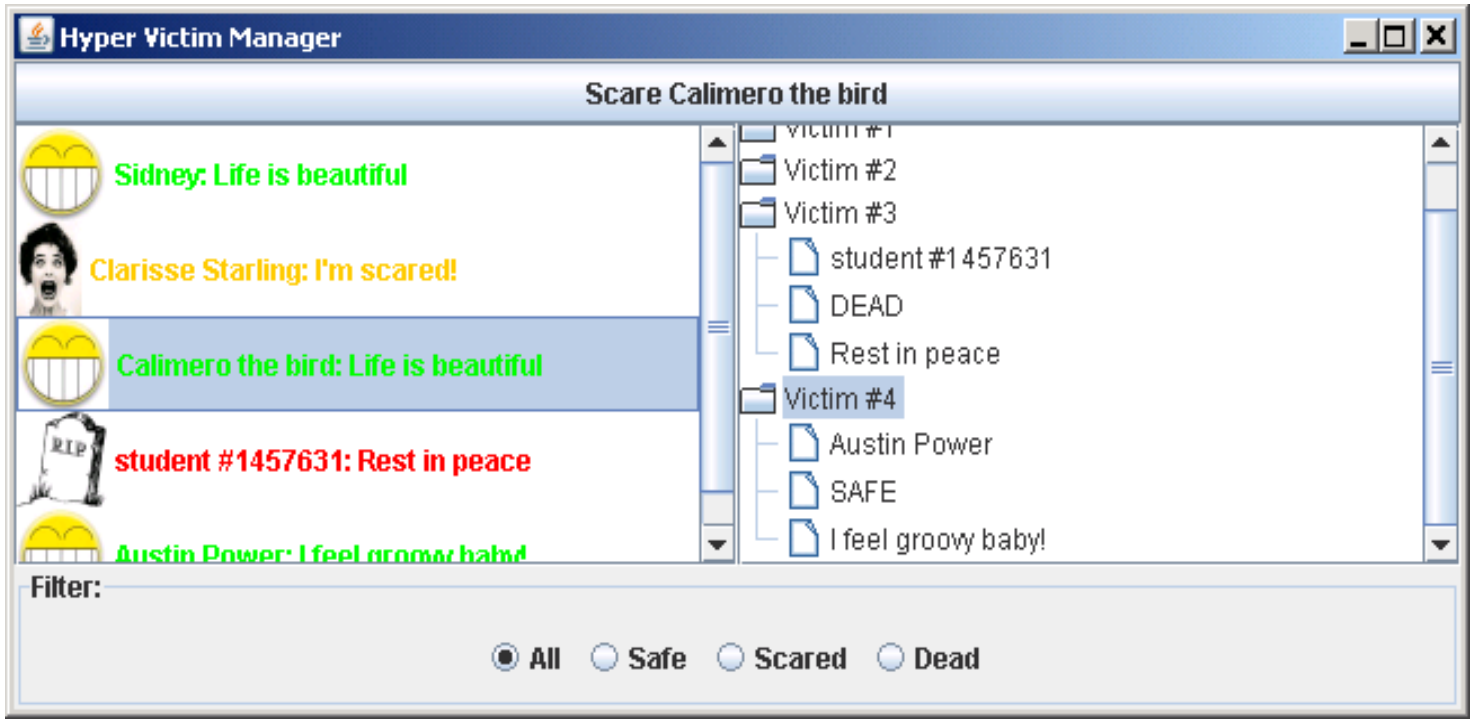
- les acteurs:
  - `JTree`: la vue graphique
  - `TreeModel`: le modèle de données
  - `TreeNode`: les noeuds
- 2 façons de gérer les arbres:
  - le modèle qui sait tout et fait tout
  - les noeuds qui se connaissent entre eux

Le modèle qui sait tout

- 1ère façon de gérer des arbres: un modèle qui fait tout, comme pour les listes
- utile quand on connaît de façon globale les informations à représenter sous forme d'arbre
- exemple: adapteur d'une liste sous forme d'arbre

VictimManager, le retour

- voici ce que l'on veut obtenir:



Le VictimTreeModel

- on doit adapter le **VictimListModel** pour en faire un **TreeModel**:

```
public class VictimTreeModel implements TreeModel {

    private VictimListModel model;
    final DefaultMutableTreeNode root=new DefaultMutableTreeNode();

    public VictimTreeModel(VictimListModel model) {
        this.model=model;
        model.addListDataListener(new ListDataListener() {
            @Override
            public void contentsChanged(ListDataEvent e) {
                someNodesHaveChanged(getChildIndices(e.getIndex0(),e.getIndex1()));
            }

            ...

    }
...
}
```

quand la liste change, on met l'arbre à jour

Le VictimTreeModel

- quand une victime change, on ne modifie que le noeud correspondant à son état:

```
protected void someNodesHaveChanged(int[] childIndices) {
    for (int i=0;i<childIndices.length;i++) {
        Victim v=model.getElementAt(childIndices[i]);
        fireTreeNodeChanged(this,new Object[] {root,v},new int[]{1},
                            new Object[] {v.getState()});
    }
}

protected void fireTreeNodeChanged(Object source, Object[] path,
                                   int[] childIndices, Object[] children) {
    firing=true;
    try {
        TreeModelEvent event=new TreeModelEvent(source,path,childIndices, children);
        for (TreeModelListener l:listeners) {
            l.treeNodesChanged(event);
        }
    } finally {firing=false;}
}
```

Le VictimTreeModel

- une fois réglée la question de la mise à jour, il faut remplir le contrat d'un **TreeModel**

1) savoir trouver un enfant:

```
@Override public Object getChild(Object parent,int index) {
    if (parent==root) {
        return model.getElementAt(index);
    }
    if (parent instanceof Victim) {
        Victim v=(Victim)parent;
        switch(index) {
        case 0: return v.getName();
        case 1: return v.getState();
        case 2: return v.getComment();
        }
    }
    throw new AssertionError("getChild is not supposed to be invoked on a leaf");
}
```

Le VictimTreeModel

2) savoir combien il y a d'enfants par noeud:

```
@Override public int getChildCount(Object parent) {
    if (parent==root) { return model.getSize(); }
    if (parent instanceof Victim) { return 3; }
    throw new AssertionError("getChildCount is not supposed to be invoked on a leaf");
}
```

3) savoir qui est la racine:

```
final DefaultMutableTreeNode root=new DefaultMutableTreeNode();

@Override public Object getRoot() {
    return root;
}
```

La racine

- quand on fabrique son propre **`TreeModel`**, on doit gérer soi-même la racine
- **attention** aux erreurs de cast dans les méthodes du modèle:

```
/* Fake root: we just want a node list */
final ArrayList<TreeNode> root=new ArrayList<TreeNode>();

@Override public Object getRoot() {
    return root;
}

@Override public boolean isLeaf(Object node) {
    return ((TreeNode)node).isLeaf();
}
```

```
Exception in thread "main" java.lang.ClassCastException:
java.util.ArrayList cannot be cast to javax.swing.tree.TreeNode
```

Le VictimTreeModel

4) savoir trouver l'indice d'un enfant:

```
/* WARNING: bugs if a Victim uses the same String as name and as comment */
@Override public int getIndexOfChild(Object parent, Object child) {
    if (parent==root) {
        int n=model.getSize();
        for (int i=0;i<n;i++) {
            Victim v=model.getElementAt(i);
            if (v.equals(child)) {
                return i;
            }
        }
    }
    if (parent instanceof Victim) {
        Victim v=(Victim)parent;
        if (v.getName().equals(child))
            return 0;
        if (v.getState().equals(child))
            return 1;
        if (v.getComment().equals(child))
            return 2;
    }
    throw new AssertionError(
                "getIndexOfChild is not supposed to be invoked on a leaf");
}
```

Le VictimTreeModel

5) savoir si un noeud est une feuille:

```
@Override public boolean isLeaf(Object node) {
    if (node==root || node instanceof Victim) return false;
    return true;
}
```

6) savoir quoi faire si l'utilisateur édite un noeud:

```
@Override public void valueForPathChanged(TreePath path, Object newValue) {
    throw new UnsupportedOperationException();
}
```

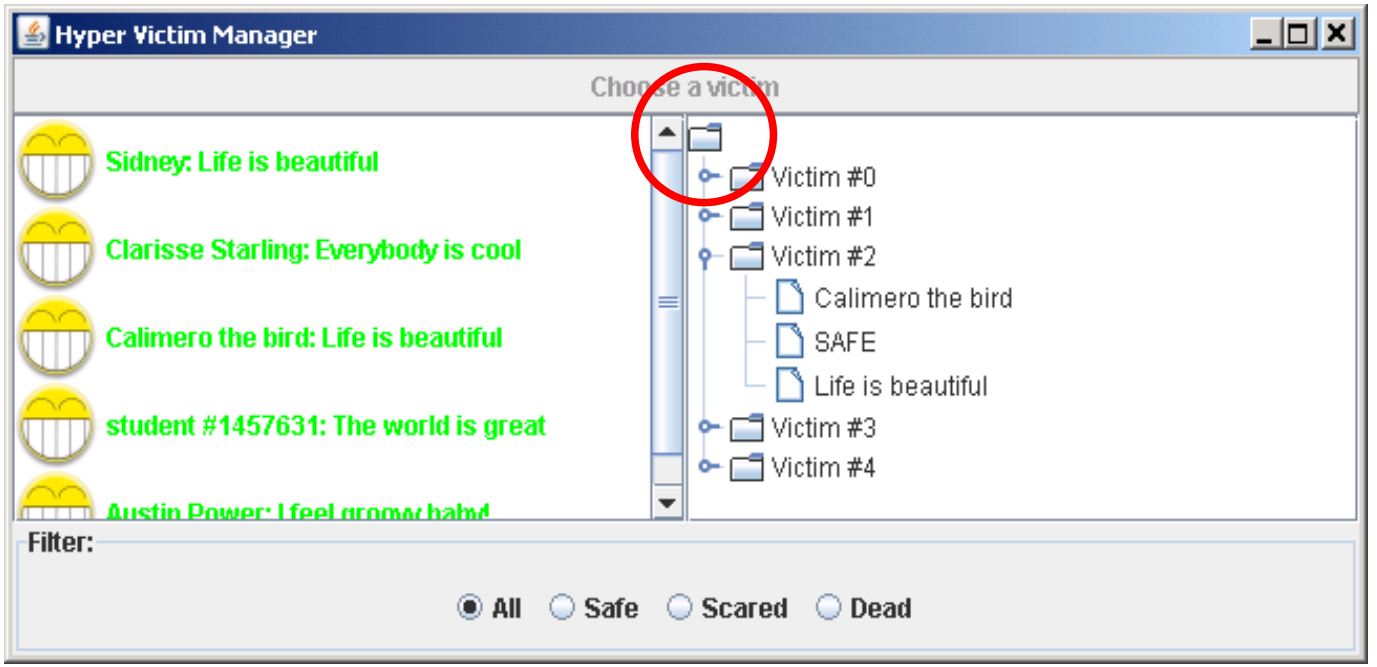
Le VictimTreeModel

- il n'y a plus qu'à créer le modèle et l'arbre, et à définir un renderer:

```
JTree tree=new JTree(new VictimTreeModel(model));
tree.setCellRenderer(new DefaultTreeCellRenderer() {
    @Override
    public Component getTreeCellRendererComponent(JTree tree, Object value, boolean sel,
                                    boolean expanded, boolean leaf, int row, boolean hasFocus) {
        if (value instanceof Victim) {
            /* Victim #row would be wrong */
            value="Victim #"+tree.getModel().getIndexOfChild(tree.getModel().getRoot(),value);
        }
        return super.getTreeCellRendererComponent(tree,value,sel,expanded,leaf,row,hasFocus);
    }
});
```

Le VictimTreeModel

- on obtient ceci:

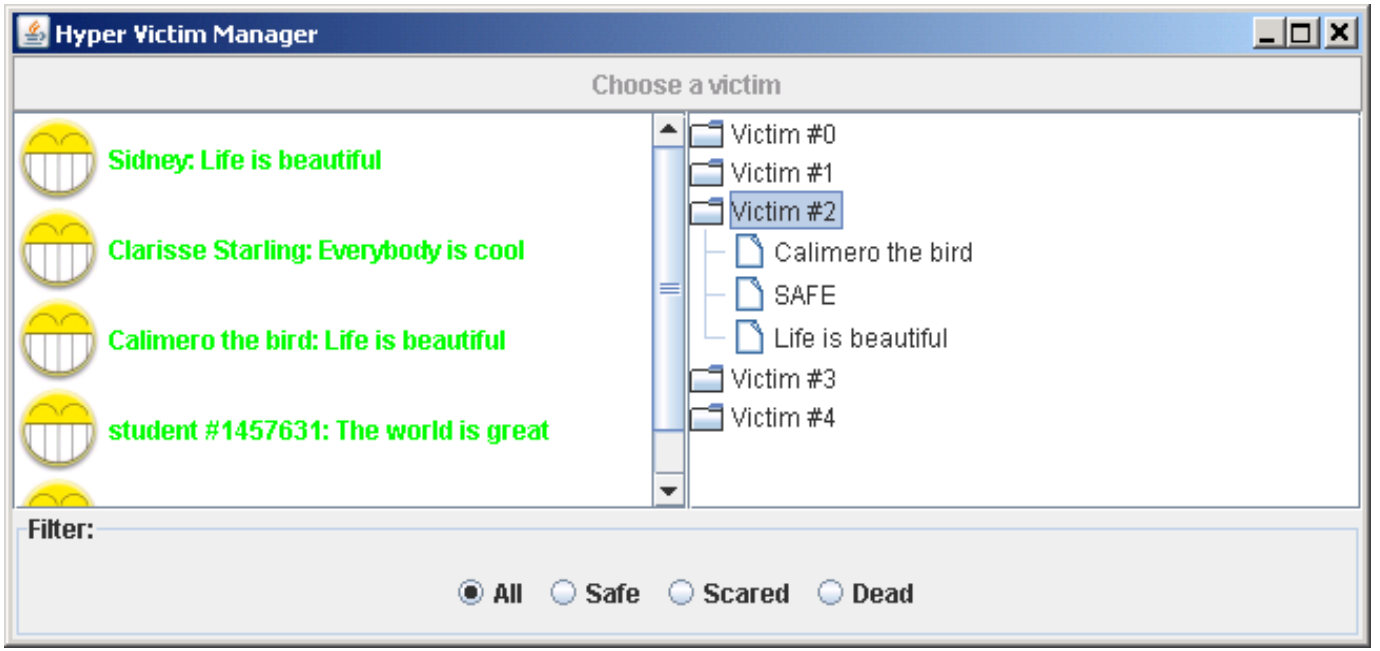


- problème: on n'a pas une "vraie" racine
- pour ne pas l'afficher:

```
tree.setRootVisible(false);
```

Le VictimTreeModel

- et voilà le travail:



- c'est bien joli, mais comment fait-on quand on a une structure d'arbre où chaque noeud connaît son père et ses fils ?

Les noeuds qui se parlent

- 2ème façon de gérer des arbres, avec des noeuds personnalisés et un modèle standard
- exemple d'utilisation: utiliser un `JTree` pour représenter un arbre binaire de recherche d'entiers (c'est de l'algo, mais ça fait pas mal)

Les noeuds

- nous devons implémenter un **TreeNode** correspondant à notre besoin
- **children**: énumération des enfants

```
public class BinarySearchTreeNode implements TreeNode
...
    @Override public Enumeration<BinarySearchTreeNode> children() {
        ArrayList<BinarySearchTreeNode> tmp=new ArrayList<BinarySearchTreeNode>();
        if (left != null) {
            tmp.add(left);
        }
        if (right != null) {
            tmp.add(right);
        }
        return Collections.enumeration(tmp);
    }
...
}
```

Les noeuds

- **getChildAt**: enfant #n
- **getChildCount**: nombre d'enfants

```
@Override public TreeNode getChildAt(int childIndex) {
    if (childIndex == 1)
        return right;
    if (childIndex == 0) {
        if (left != null) {
            return left;
        } else {
            return right;
        }
    }
    throw new AssertionError("Invalid child index");
}

@Override public int getChildCount() {
    int n = 0;
    if (left != null) n++;
    if (right != null) n++;
    return n;
}
```

Les noeuds

- **`getIndex`**: indice d'un enfant donné
- **`getParent`**: le parent, ou **`null`** si le noeud est la racine

```
@Override public int getIndex(TreeNode node) {
    if (left != null && left.equals(node)) return 0;
    if (right != null && right.equals(node)) {
        return (left != null) ? 1 : 0;
    }
    throw new AssertionError("Cannot ask for the index of a non child object");
}

@Override public BinarySearchTreeNode getParent() {
    return parent;
}
```

Les noeuds

- **`isLeaf`**: le noeud est-il une feuille ?
- **`getAllowsChildren`**: le noeud accepte-il qu'on lui ajoute des enfants ?

```
@Override public boolean isLeaf() {
    return getChildCount() == 0;
}

@Override public boolean getAllowsChildren() {
    return true;
}
```

- une fois les méthodes de **`TreeNode`** implémentées, il nous reste à coder nos fonctions d'ajout/suppression de noeuds

Les noeuds

- accès à la valeur et aux enfants:

```
public void addLeftChild(int n) {
    if (left != null)
        throw new IllegalStateException("Left child already exists !");
    left = new BinarySearchTreeNode(this, n);
}

public void addRightChild(int n) {
    if (right != null)
        throw new IllegalStateException("Right child already exists !");
    right = new BinarySearchTreeNode(this, n);
}

public int getValue() {
    return value;
}

public BinarySearchTreeNode getLeft() {
    return left;
}

public BinarySearchTreeNode getRight() {
    return right;
}
```

Les noeuds

- ajout d'une valeur:

```
public void addValue(int value,BinarySearchTreeModel model) {
    int currentValue=getValue();
    if (currentValue==value) {
        /* Nothing to do, the value is already in the tree */
        return;
    }
    if (value<currentValue) {
        if (getLeft()==null) {
            addLeftChild(value);
            model.nodesWereInserted(this,new int[] {0});
        } else { getLeft().addValue(value,model); }
    } else {
        if (getRight()==null) {
            addRightChild(value);
            model.nodesWereInserted(this,new int[] {getChildCount()-1});
        } else { getRight().addValue(value,model); }
    }
}
```

on a besoin du modèle pour le prévenir qu'on a changé

Les noeuds

- idem pour la suppression d'une valeur:

```
public void removeValueFromNode(int value,BinarySearchTreeModel model) {
    if (value<getValue() && getLeft()!=null) {
        getLeft().removeValueFromNode(value,model);
        return;
    }
    if (value>getValue() && getRight()!=null) {
        getRight().removeValueFromNode(value,model);
        return;
    }
    if (getLeft()==null && getRight()==null) {
        /* No child ? We can rawly remove the node */
        BinarySearchTreeNode parent=getParent();
        if (this==model.getRoot()) {
            model.setRoot(null);
            /* Nothing to do: setRoot will fire as needed */
            return;
        }
        int index=parent.getIndex(this);
        parent.removeChild(this);
        model.nodesWereRemoved(parent,new int[]{index},new Object[]{this});
        return;
    }
...
}
```

Le modèle d'arbre

- le modèle d'arbre ne sert qu'à:
  - proposer les méthodes d'ajout et suppression de valeurs
  - renseigner la vue sur les changements
- il suffit d'étendre le **`DefaultTreeModel`**:

```
@SuppressWarnings("serial")
public class BinarySearchTreeModel extends DefaultTreeModel {

    public BinarySearchTreeModel(TreeNode root) {
        super(root);
    }

...

}
```

Le modèle d'arbre

- pour l'ajout et la suppression, on laisse les noeuds travailler:

```
public void addInteger(int n) {
    if (getRoot()==null) {
        setRoot(new BinarySearchTreeNode(null,n));
        /* No need to fire..., setRoot does it */
        return;
    }
    BinarySearchTreeNode root=(BinarySearchTreeNode) getRoot();
    root.addValue(n,this);
}

public void removeValue(int value) {
    BinarySearchTreeNode root=(BinarySearchTreeNode)getRoot();
    root.removeValueFromNode(value,this);
}
```

on leur donne le modèle pour qu'ils puissent prévenir des changements (pas très MVC tout ça, mais bon)

Prévenir le modèle

- dans cette architecture, ce sont les noeuds qui préviennent des changements, en utilisant des méthodes de `DefaultTreeModel`:
  - `nodeChanged(TreeNode node)`:
    le noeud a changé
  - `nodesChanged(TreeNode node, int[] childIndices)`:
    certains enfants du noeud donné ont changé

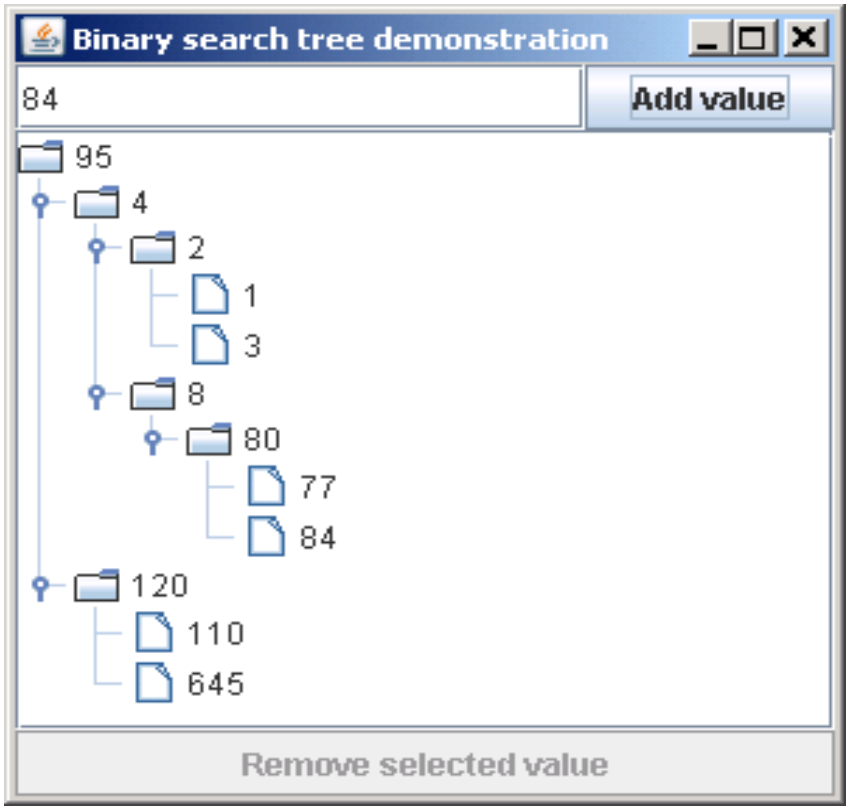
Prévenir le modèle

- `nodeStructureChanged(TreeNode node)`:
  tout le sous-arbre a changé
- `nodesWereInserted(TreeNode node, int[] childIndices)`:
  on a ajouté des enfants au noeud donné
- `nodesWereRemoved(TreeNode node, int[] childIndices)`:
  on a enlevé des enfants au noeud donné

- plus pratique que les `fire...` car on ne doit pas se préoccuper des chemins jusqu'aux noeuds concernés

Résultat

- on obtient bien un arbre binaire de recherche:



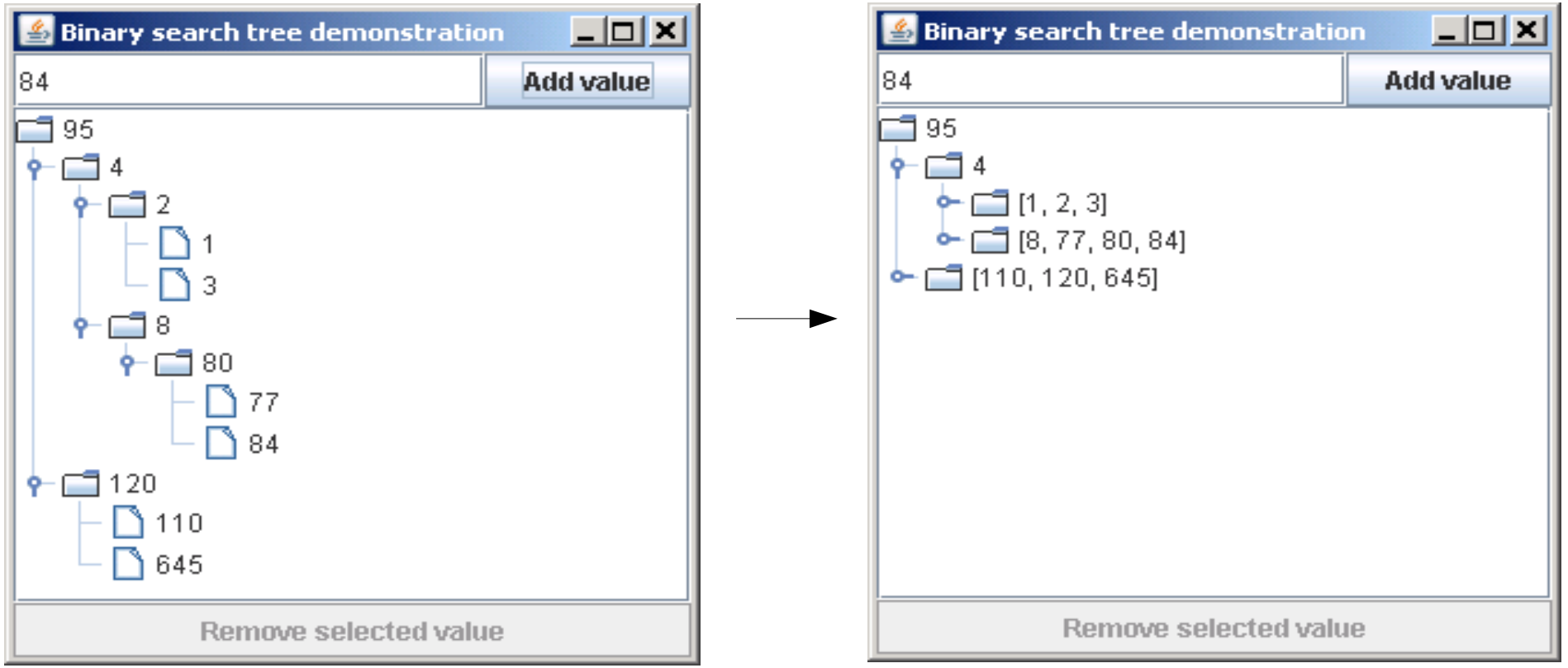
Rendu de l'arbre

- on aimerait pouvoir voir les valeurs d'une branche quand un noeud est replié
- on travaille donc sur le renderer:

```
tree.setCellRenderer(new DefaultTreeCellRenderer() {
    @Override
    public Component getTreeCellRendererComponent(JTree tree, Object value,
                    boolean selected, boolean expanded, boolean leaf, int row,
                    boolean hasFocus) {
        BinarySearchTreeNode node = (BinarySearchTreeNode) value;
        String s=node.getValue()+"";
        if (!expanded && !leaf) {
            s=node.getValueList();
        }
        super.getTreeCellRendererComponent(tree,s,selected,expanded,
                                           leaf,row,hasFocus);
        return this;
    }
});
```

Rendu de l'arbre

- on récupère la liste des valeurs et on l'affiche:

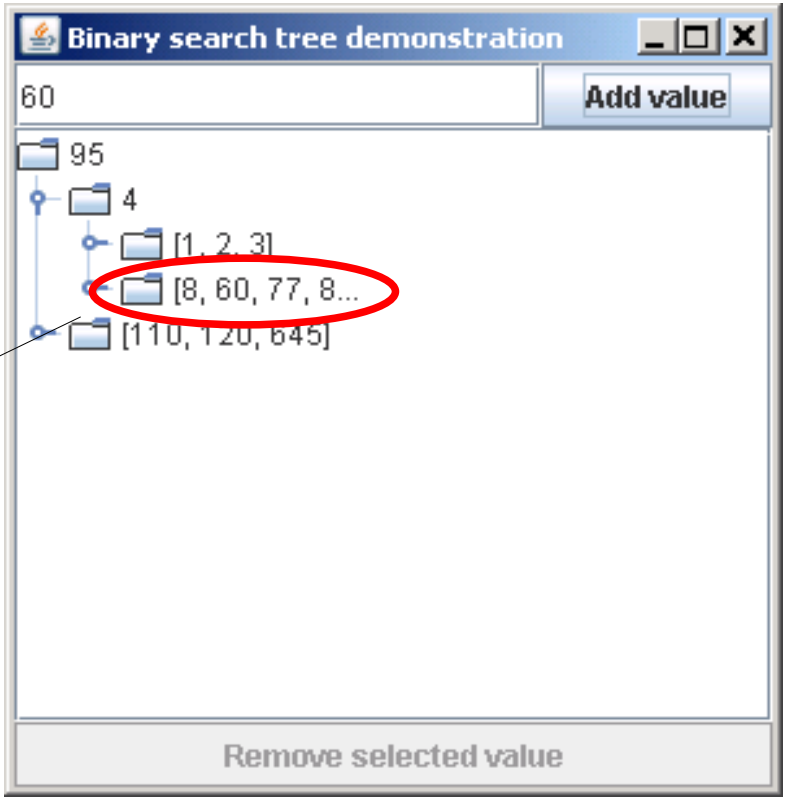


Bug de rendu

- si on ajoute un noeud à une branche repliée:

mauvais rafraîchissement

=problème de design de Swing **:(**



Bug de rendu

- solution: prévenir soi-même tous les parents du noeud ajouté
  - seul le premier parent replié suffirait, mais il faudrait connaître la vue

```
private void updateParents(BinarySearchTreeModel model) {
    TreeNode tmp=this;
    while (tmp!=null) {
        model.nodeChanged(tmp);
        tmp=tmp.getParent();
    }
}
```

Bug de rendu

- on invoque **updateParents** à chaque ajout:

```
public void addValue(int value,BinarySearchTreeModel model) {
    int currentValue=getValue();
    if (currentValue==value) {
        /* Nothing to do, the value is already in the tree */
        return;
    }
    if (value<currentValue) {
        if (getLeft()==null) {
            addLeftChild(value);
            model.nodesWereInserted(this,new int[] {0});
            updateParents(model);
        } else { getLeft().addValue(value,model); }
    } else {
        if (getRight()==null) {
            addRightChild(value);
            model.nodesWereInserted(this,new int[] {getChildCount()-1});
            updateParents(model);
        } else { getRight().addValue(value,model); }
    }
}
```

Supprimer des noeuds

- 2 choses à faire:
  - gérer la sélection d'un noeud (et d'un seul)
  - coder la suppression d'un noeud (c'est déjà fait dans notre **BinarySearchTreeNode**)

- définissons le mode de sélection:

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

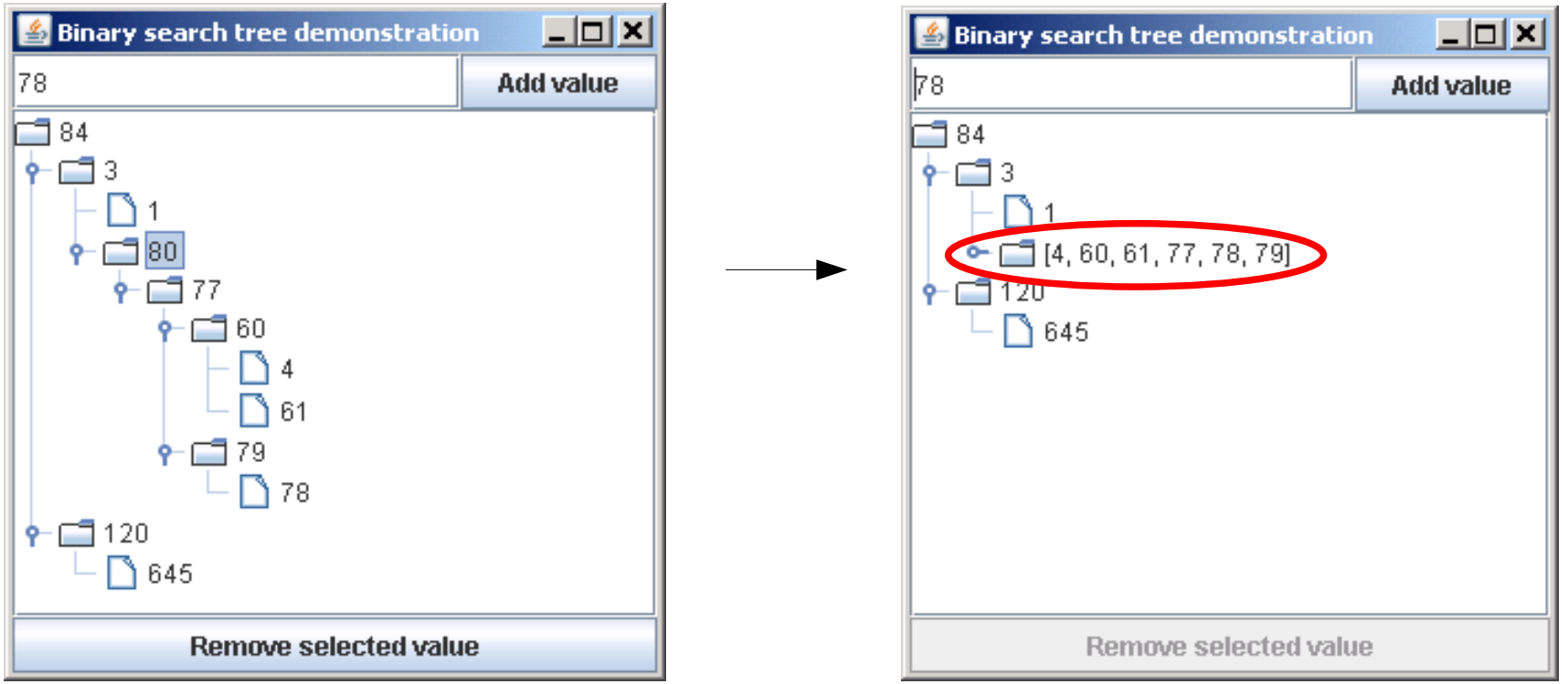
Supprimer des noeuds

- création du bouton "Remove", qui supprime la valeur du noeud courant, et qui est désactivé si aucun noeud n'est sélectionné:

```
final JButton remove=new JButton("Remove selected value");
remove.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        BinarySearchTreeNode node=
          (BinarySearchTreeNode) tree.getSelectionPath().getLastPathComponent();
        model.removeValue(node.getValue());
    }
});
tree.getSelectionModel().addTreeSelectionListener(new TreeSelectionListener() {
    @Override
    public void valueChanged(TreeSelectionEvent e) {
        remove.setEnabled(tree.getSelectionCount()!=0);
    }
});
remove.setEnabled(false);
```

Notre ABR avec suppression

- quand la structure change, tout le sous-arbre concerné se replie:



Des noeuds éditables

- pour pouvoir éditer les noeuds d'un arbre, il suffit de le rendre éditable:

```
tree.setEditable(true);
```

- de laisser faire le `DefaultTreeCellEditor` installé par défaut
- et de réagir aux demandes de modifications d'une valeur dans le modèle

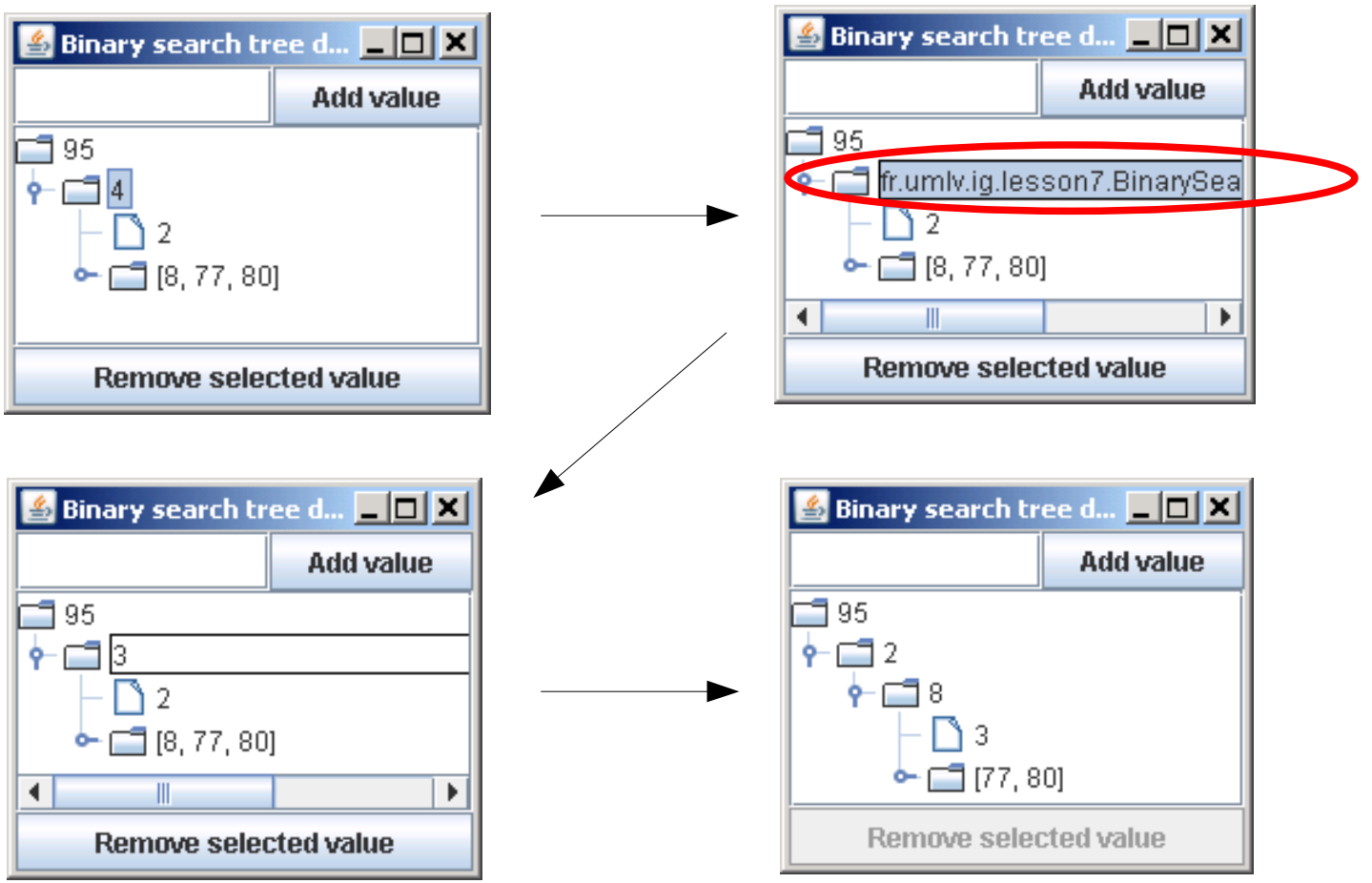
Des noeuds éditables

- **valueForPathChanged** est appelée quand l'utilisateur a changé la valeur d'un noeud avec le **TreeCellEditor**; le modèle doit alors décider quoi faire

```
@Override public void valueForPathChanged(TreePath path, Object newValue) {
    /* Does the String returned by the DefaultTreeCellEditor represent an integer ? */
    try {
        int n=Integer.parseInt((String)newValue);
        BinarySearchTreeNode node=(BinarySearchTreeNode)path.getLastPathComponent();
        removeValue(node.getValue());
        addInteger(n);
    } catch (NumberFormatException e) {
        JOptionPane.showMessageDialog(null,
                "Incorrect integer value: "+newValue,
                "Node edition error",JOptionPane.ERROR_MESSAGE);
    }
}
```

Des noeuds éditables

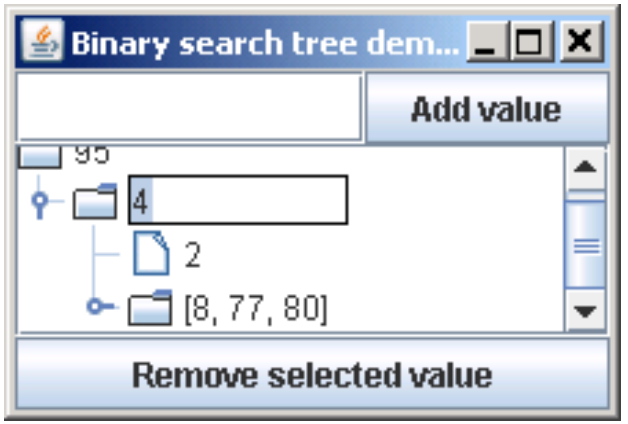
- ça marche, mais on a un léger souci d'affichage:



Des noeuds éditables

- explication: le **`DefaultTreeCellEditor`** utilise **`node.toString()`** comme valeur initiale
- solution: redéfinir **`toString()`** dans **`BinarySearchTreeNode`**:

```
@Override
public String toString() {
    return ""+value;
}
```



MutableTreeNode

- c'est une interface créée pour gérer des noeuds éditables depuis le modèle:
  - c'est le modèle qui fait node.insert...
  - c'est le modèle qui fait les fire...
- c'est un design bâtard qui traîne dans l'API Swing pour des raisons historiques

- à ne pas utiliser!