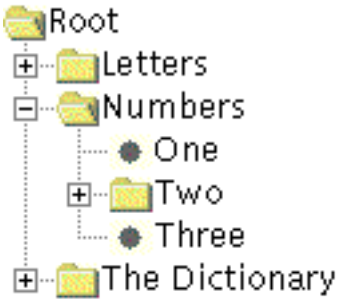
1 Comment utiliser les JTree

La classe `JTree` est employée pour afficher des données organisées hiérarchiquement. Une instance de `JTree` ne contient pas réellement de données, il en offre simplement une vue. Tout comme n'importe quel composant Swing non-trivial, l'arbre récupère les données dans un modèle de données. Voici une image affichée par un arbre :



Comme le montre cette figure, un `JTree` affiche ses données verticallement. Chaque ligne affichée correspond à exactement un élément de données, appelé un **noeud**. Tout arbre possède un noeud particulier duquel sont issus tous les autres et qui lui-même ne possède pas de parent. Ce noeud s'appelle la **racine**. Par défaut, l'arbre affiche la racine, mais on peut en décider autrement. Un noeud peut avoir des **enfants** ou pas. Les noeuds sans enfants sont des **feuilles**. Les autres noeuds sont des **branches**.

Une branches peut avoir un ou plusieurs enfants. À l'aide de la souris, l'utilisateur peut déployer ou au contraire replier les branches, rendant ainsi leur enfants visible ou invisible. Par défaut, toutes les branches exceptées la racine sont fermées au départ. Un programme peut détecter le déploiement ou la fermeture d'une branche en écoutant des événements de type `TreeExpansionEvent` en utilisant un *TreeWillExpandListener* ou un *TreeExpandListener*.

Un noeud peut être identifié dans un arbre soit par une instance de `TreePath`, un objet qui encapsule un noeud et tous ses ancêtres jusqu'à la racine, soit par son numéro de ligne.

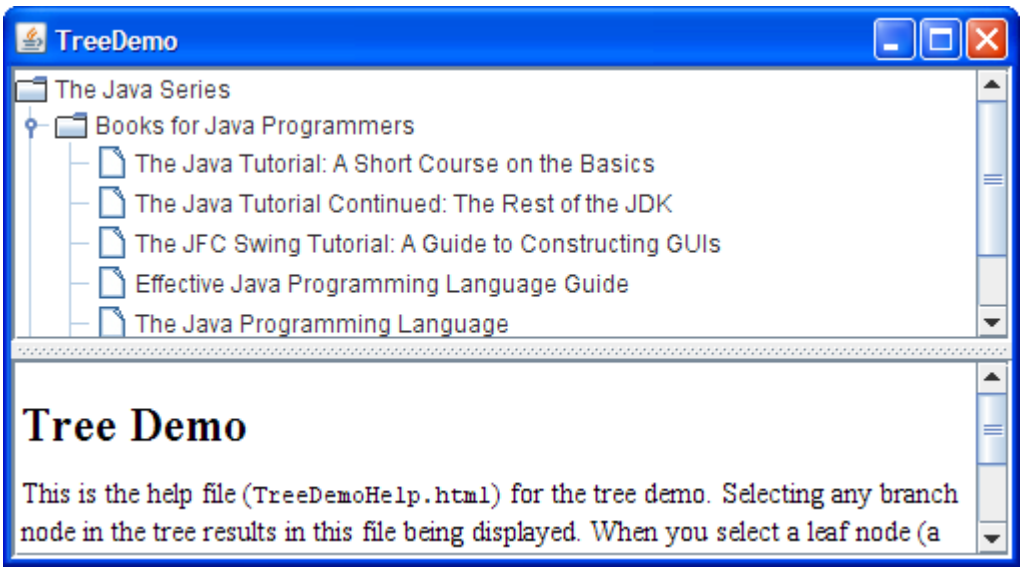
- Un nœud étendu est une branche qui affiche ses enfants quand tous ses ancêtres sont étendus.
- Un nœud fermé les cache.
- Un nœud caché en est un dont un des ancêtres est fermé.

La suite de cette section parle des sujets suivants :

- Créer un arbre.
- Réagir à la sélection d'un noeud.
- Personnaliser l'affichage d'un arbre.
- Modifier dynamiquement un arbre.
- Créer un modèle de donnée.
- L'API concernant les arbres.
- Quelques exemples exploitant des arbres.

2 Créer un arbre

Voici l'image d'une application, la moitié nord affiche un arbre dans un panneau à ascenseur. La moitié sud affiche éventuellement un texte associé à la feuille sélectionnée (si une feuille est sélectionnée).



Il est possible de déployer ou de replier une branche en cliquant dans le petit cercle à gauche de celle-ci.

Le code suivant crée une instance de `JTree` et le place dans un panneau à ascenseur (scroll pane) :

```
 //Where instance variables are declared:
private JTree tree;
...
public TreeDemo() {
    ...
    // Cette instruction pour créer une racine.
    DefaultMutableTreeNode top = new DefaultMutableTreeNode("The Java Series");
    // La méthode "createNodes" construit l'arbre à partir de sa racine.
    // Cette méthode est décrite plus loin.
    createNodes(top);
    // Création de la vue d'un arbre dont on donne sa racine.
    tree = new JTree(top);
    ...
}
```

Le code précédent crée une instance de `DefaultMutableTreeNode` appelée `top` et utilisée comme racine du futur arbre. Le reste de l'arbre est créé dans la méthode `createNodes`. Une instance de `JTree` est réalisée avec la racine en argument de son constructeur.

Voici maintenat le code qui crée les noeuds issus de la racine :

```
private void createNodes(DefaultMutableTreeNode top) {
    DefaultMutableTreeNode category = null;
    DefaultMutableTreeNode book = null;

    category = new DefaultMutableTreeNode("Books for Java Programmers");
    top.add(category);

    //original Tutorial
    book = new DefaultMutableTreeNode(new BookInfo
        ("The Java Tutorial: A Short Course on the Basics",
        "tutorial.html"));
    category.add(book);

    //Tutorial Continued
```

```
    book = new DefaultMutableTreeNode(new BookInfo
        ("The Java Tutorial Continued: The Rest of the JDK",
        "tutorialcont.html"));
    category.add(book);

    //Swing Tutorial
    book = new DefaultMutableTreeNode(new BookInfo
        ("The Swing Tutorial: A Guide to Constructing GUIs",
        "swingtutorial.html"));
    category.add(book);

    //...add more books for programmers...

    category = new DefaultMutableTreeNode("Books for Java Implementers");
    top.add(category);

    //VM
    book = new DefaultMutableTreeNode(new BookInfo
        ("The Java Virtual Machine Specification",
         "vm.html"));
    category.add(book);

    //Language Spec
    book = new DefaultMutableTreeNode(new BookInfo
        ("The Java Language Specification",
         "jls.html"));
    category.add(book);
}
```

L'argument du constructeur d'une instance de `DefaultMutableTreeNode` est un objet associé à ce noeud. Ce peut être une simple chaîne (`String`) et ce sera cette chaîne qui sera affichée. Mais ce peut être également un objet personnalisé, dans ce cas l'affichage d'un noeud correspond au résultat de la méthode `toString` de son objet associé. Il est dont important que `toString` soit correctement redéfinie et retourne quelque chose de significatif. Parfois il n'est possible de redéfinir `toString`. Le cas échéant il faut redéfinir la méthode `convertValueToText` de `JTree` pour associer un objet et sa chaîne à afficher.

Par exemple, la classe `BookInfo` utilisée dans le précédent morceau de code est une classe personnalisée qui encapsule deux données, le nom d'un livre et une url donnant sa description. `toString` est redéfinie pour retourner le nom du livre. Ainsi, chaque noeud étant associé à un `BookInfo` affiche son nom.

En résumé, un arbre peut être construit en invoquant le constructeur d'un `JTree` avec un objet de type `TreeNode` en argument. Il est probable que cet arbre doive se trouver dans un panneau à ascenseur à cause de la place qu'il peut prendre. Il n'y rien de particulier à faire pour que l'utilisateur puisse avec la souris déployer ou replier les branches d'un arbre. Cependant, il faut ajouter du code si l'on veut que l'arbre produise un comportement particulier lorsqu'une branche est sélectionnée ou dé-sélectionnée en cliquant sur son noeud.

Par exemple :

3 Réagir à la sélection d'un noeud

Il est assez simple de réagir à la sélection d'un noeud. Il suffit d'implémenter un « listener » de sélection d'arbre. Le code suivant montre comment le programme `TreeDemo` réagit à la sélection d'un noeud.

```
 //Where the tree is initialized:
    tree.getSelectionModel().setSelectionMode
            (TreeSelectionModel.SINGLE_TREE_SELECTION);

    //Listen for when the selection changes.
    tree.addTreeSelectionListener(this);
...
public void valueChanged(TreeSelectionEvent e) {
//Returns the last path element of the selection.
//This method is useful only when the selection model allows a single selection.
    DefaultMutableTreeNode node = (DefaultMutableTreeNode)
                       tree.getLastSelectedPathComponent();

    if (node == null)
    //Nothing is selected.
    return;

    Object nodeInfo = node.getUserObject();
    if (node.isLeaf()) {
        BookInfo book = (BookInfo)nodeInfo;
        displayURL(book.bookURL);
    } else {
        displayURL(helpURL);
    }
}
```

Le code précédent conduit les tâches suivantes :

- Il récupère le `TreeSelectionModel` par défaut de l'arbre et définit le mode de sélection des noeuds en précisant qu'un seul noeud peut être sélection à la fois.
- Il ajoute à cet arbre un `TreeSelectionListener` qui est en fait lui-même (`this`).
- Il faut donc que la classe développée implémente `TreeSelectionListener` et donc redéfinisse la méthode `valueChanged`. C'est le cas.
- La méthode `valueChanged` fait les choses suivantes :
  - Récupérer le noeud sélectionné.
  - Il peut ne pas y en avoir (dans ce cas, rien n'est fait)
  - Récupérer l'objet associé à ce noeud qui est un `BookInfo`
  - Afficher une aide ou le résumé du livre selon que le noeud sélectionné soit une feuille ou non.

4 Personnaliser l'affichage d'un arbre

Voici ci-dessous trois styles d'arbre, selon Java, selon Windows et selon Mac OS.

Style Java Style Windows Style Mac

Comme le montrent les figures précédentes, un arbre affiche traditionnellement un icône et du texte pour chaque noeud. On peut les personnaliser, comme on va le voir bientôt.

Un arbre se charge représenter les relations entre les noeuds. La façon dont cela est fait peut aussi être un peu personnalisée. Tout d'abord, il est possible de rendre la racine visible ou invisible en utilisant la méthode `setRootVisible` d'un `JTree`. Il est possible aussi de changer la visibilité de la poignée de la racine, celle qui permet de la déployer et la replier, grâce à `setShowsRootHandles` de`JTree`.

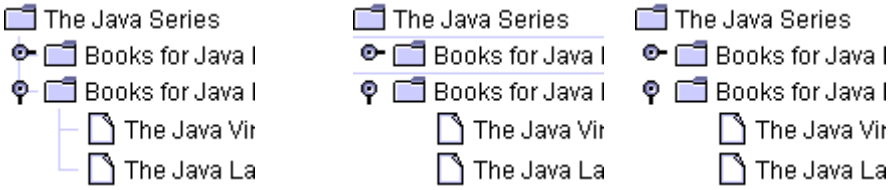
Avec le style Java, il est possible de personnaliser la façon dont les lignes sont dessinés les liens entre les noeuds. Par défaut, le style Java dessine des lignes angulaires entre les noeuds. En modifiant la propriété `JTree.lineStyle` d'un arbre, une autre convention peut être choisie. Par exemple, pour que le style Java utilise des lignes horizontales pour regrouper les noeuds, l'instruction suivante peut être utilisée

```
// Où tree est une instance de JTree
 tree.putClientProperty("JTree.lineStyle", "Horizontal");
```

Pour ne pas dessiner de ligne, utiliser ce code :

```
// Où tree est une instance de JTree
tree.putClientProperty("JTree.lineStyle", "None");
```

Les images suivantes montrent les résultats pour les différents styles de ligne du « look » Java.



Angulaire (par défaut) Horizontal Sans

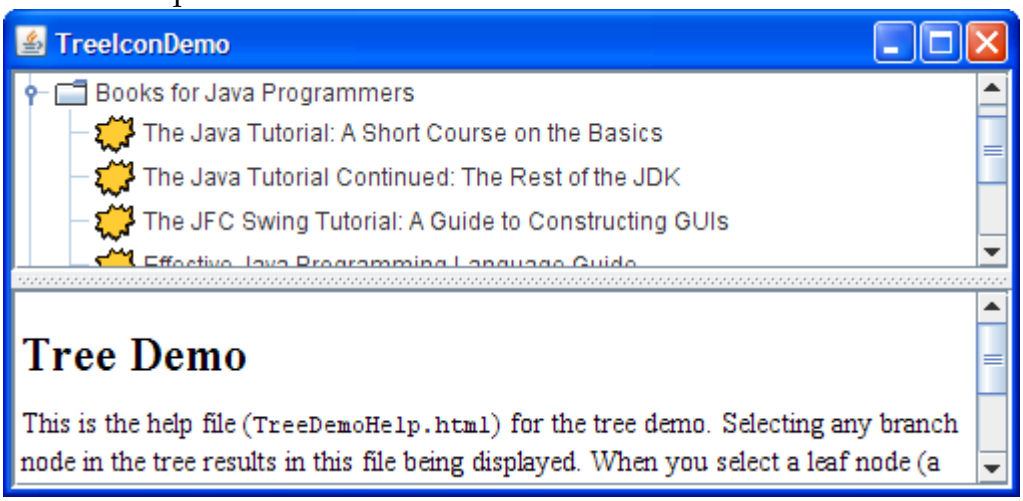
Quel que soit le style choisi, l'icône par défaut affiché pour un noeud change s'il s'agit d'une feuille ou non. Celui d'une branche change aussi selon qu'il soit déployer ou replier. Par exemple, si le style Windows est choisi, l'icône par défaut pour un noeud est un point alors qu'il s'agit d'une représentation d'une page de papier pour les deux autres styles. Dans tous les styles, l'icône dédié aux branches ressemble à un dossier. Le style MacOS les représente différemment selon que la branche soit déployée ou repliée.

On peut facilement changer l'icône par défaut pour les feuilles, les branches déployées ou repliées. Pour ce faire, il faut commencer par créer une instance de `DefaultTreeCellRenderer`. Ensuite, on peut spécifier les icônes pour les feuilles, les branches déployée ou repliée, que l'on veut à l'aide des méthodes `setLeafIcon`, `setOpenIcon` et `setClosedIcon`. Pour préciser l'absence d'icône pour un type de noeud il suffit de mettre `null` en argument de la méthode appropriée (parmi `setLeafIcon`, `setOpenIcon` et `setClosedIcon`). Il faut ensuite affecter l'arbre dont on veut personnaliser l'affichage en employant la méthode `setCellRenderer` avec votre `DefaultTreeCellRenderer` en argument. Il est possible aussi de créer sa propre implémentation de `TreeCellRenderer`.

Voici un exemple tiré de `TreeIconDemo.java`

```
ImageIcon leafIcon = createImageIcon("images/middle.gif");
if (leafIcon != null) {
    DefaultTreeCellRenderer renderer =
        new DefaultTreeCellRenderer();
    renderer.setLeafIcon(leafIcon);
    tree.setCellRenderer(renderer);
}
```

Voici une copie d'écran de `TreeIconDemo` :



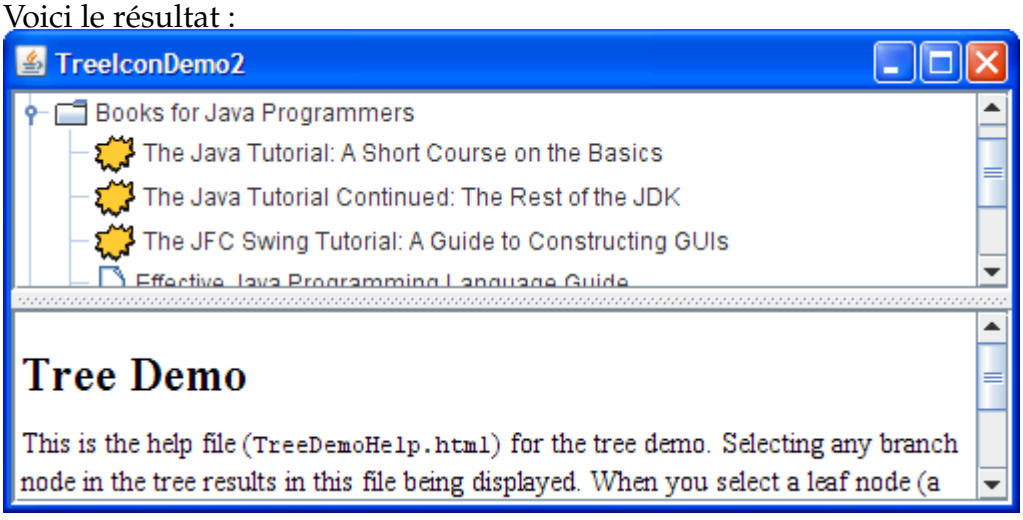
Si vous voulez un contrôle amélioré des icônes associé aux noeuds ou voulez les doter de notice-conseil (tool tips), cela peut se faire en créant une sous-classe de `DefaultTreeCellRenderer` et de redéfinir la méthode `getTreeCellRendererComponent`. Puisque `DefaultTreeCellRenderer` est une sous-classe de `JLabel`, vous pouvez utiliser n'importe quelle méthode de `JLabel` – tel que `setIcon` – pour votre personnalisation.

Le code suivant montre comment créer un « rendu de cellule » qui fait varier l'icône d'une feuille selon que le mot « Tutorial » apparaisse dans le texte du noeud. Des notices (tool-tip) sont également programmées.

```
 //...where the tree is initialized:
    //Enable tool tips.
    ToolTipManager.sharedInstance().registerComponent(tree);

    ImageIcon tutorialIcon = createImageIcon("images/middle.gif");
    if (tutorialIcon != null) {
        tree.setCellRenderer(new MyRenderer(tutorialIcon));
    }
...
class MyRenderer extends DefaultTreeCellRenderer {
    Icon tutorialIcon;

    public MyRenderer(Icon icon) {
        tutorialIcon = icon;
    }

    public Component getTreeCellRendererComponent(
                        JTree tree,
                        Object value,
                        boolean sel,
                        boolean expanded,
```

```
                        boolean leaf,
                        int row,
                        boolean hasFocus) {

        super.getTreeCellRendererComponent(
                        tree, value, sel,
                        expanded, leaf, row,
                        hasFocus);
        if (leaf && isTutorialBook(value)) {
            setIcon(tutorialIcon);
            setToolTipText("This book is in the Tutorial series.");
        } else {
            setToolTipText(null); //no tool tip
        }

        return this;
    }

    protected boolean isTutorialBook(Object value) {
        DefaultMutableTreeNode node =
                (DefaultMutableTreeNode)value;
        BookInfo nodeInfo =
                (BookInfo)(node.getUserObject());
        String title = nodeInfo.bookName;
        if (title.indexOf("Tutorial") >= 0) {
            return true;
        }

        return false;
    }
}
```

Voici le résultat :

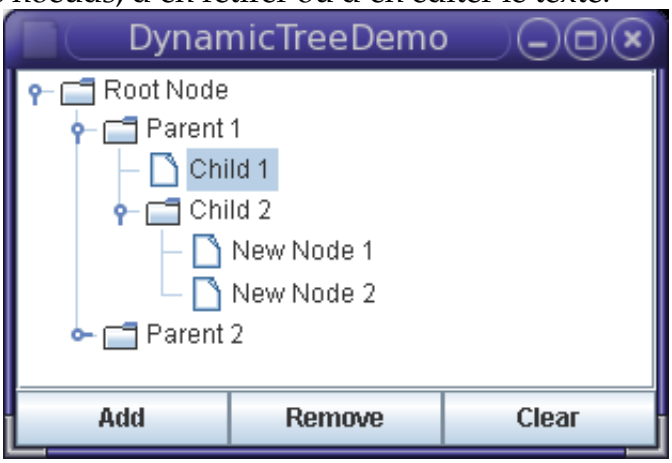


Vous pourriez vous demander comment le « rendu de cellule » fonctionne. Quand un arbre peint des noeuds, ni le `JTree` ni son implémentation de style spécifique ne contiennent effectivement le code pour les peindre. En fait, l'arbre utilise le code du « rendu de cellule » pour peindre le noeud. Par exemple pour peindre une feuille dont le texte est `"The Java Programming Language"`, l'arbre demande son « rendu de cellule » pour retourner un composant capable de peindre une feuille avec cette chaîne. Si le « rendu de cellule » est un `DefaultTreeCellRenderer`, alors il retourne un label pour peindre la feuille.

Un « rendu de cellule » ne fait que peindre mais il ne gère pas les événements. Pour qu'un arbre réagisse à des événements, il faut utiliser des « écouteurs ».

5 Changer un arbre

La figure suivante montre une application appelée `DynamicTreeDemo` qui permet d'ajouter des noeuds, d'en retirer ou d'en éditer le texte.



Voici le code qui initialise l'arbre :

```
rootNode = new DefaultMutableTreeNode("Root Node");
treeModel = new DefaultTreeModel(rootNode);
treeModel.addTreeModelListener(new MyTreeModelListener());

tree = new JTree(treeModel);
tree.setEditable(true);
tree.getSelectionModel().setSelectionMode(TreeSelectionModel.SINGLE_TREE_SELECTION);
tree.setShowsRootHandles(true);
```

En créant explicitement le modèle d'arbre, le code garantit que le modèle d'arbre est une instance de `DefaultTreeModel`. Ainsi, nous savons toutes les méthodes que le modèle supporte. Par exemple, nous savons que quand nous invoquons la méthode `insertNodeInto` du modèle que supporte l'arbre, quand bien même cette méthode n'est pas requise par l'interface `TreeModel`.

Pour rendre éditables les noeuds, nous invoquons la méthode `setEditable(true)` de l'arbre. Quand l'utilisateur termine l'édition d'un noeud, le modèle génère un événement qui signale à tous les écouteurs — y compris ceux de l'arbre — qu'un noeud a été modifié. Notez que bien que `DefaultMutableTreeNode` a des méthodes pour changer le contenu d'un noeud, les changements se font à traver le `DefaultTreeModel`. Otherwise, the tree model events would not be generated, and listeners such as the tree would not know about the updates.

Pour être notifié d'un changement, un `TreeModelListener` peut être implémenté. Voici un exemple d'un listener de modèle d'arbre qui détecte quand l'utilisateur a tapé un nouveau nom pour un noeud :

```
class MyTreeModelListener implements TreeModelListener {
    public void treeNodesChanged(TreeModelEvent e) {
        DefaultMutableTreeNode node;
        node = (DefaultMutableTreeNode)(e.getTreePath().getLastPathComponent());

        /*
```

```
         * If the event lists children, then the changed
         * node is the child of the node we have already
         * gotten. Otherwise, the changed node and the
         * specified node are the same.
         */
        try {
            int index = e.getChildIndices()[0];
            node = (DefaultMutableTreeNode)(node.getChildAt(index));
        } catch (NullPointerException exc) {}

        System.out.println("The user has finished editing the node.");
        System.out.println("New value: " + node.getUserObject());
    }
    public void treeNodesInserted(TreeModelEvent e) {
    }
    public void treeNodesRemoved(TreeModelEvent e) {
    }
    public void treeStructureChanged(TreeModelEvent e) {
    }
}
```

Voici le code que le bouton `Add` exploite pour ajouter un nouveau noeud à un arbre.

```
treePanel.addObject("New Node " + newNodeSuffix++);
...
public DefaultMutableTreeNode addObject(Object child) {
    DefaultMutableTreeNode parentNode = null;
    TreePath parentPath = tree.getSelectionPath();

    if (parentPath == null) {
        //There is no selection. Default to the root node.
        parentNode = rootNode;
    } else {
        parentNode = (DefaultMutableTreeNode)(parentPath.getLastPathComponent());
    }

    return addObject(parentNode, child, true);
}
...
public DefaultMutableTreeNode addObject(DefaultMutableTreeNode parent, Object child,
    boolean shouldBeVisible) {
    DefaultMutableTreeNode childNode = new DefaultMutableTreeNode(child);
    ...
    treeModel.insertNodeInto(childNode, parent, parent.getChildCount());

    //Make sure the user can see the lovely new node.
    if (shouldBeVisible) {
        tree.scrollPathToVisible(new TreePath(childNode.getPath()));
    }
    return childNode;
}
```

Le code crée un noeud, l'insère dans le modèle d'arbre, ensuite, si nécessaire, déploie tous les noeuds sous lui et déroule l'ascenseur pour que le nouveau noeud soit visible.

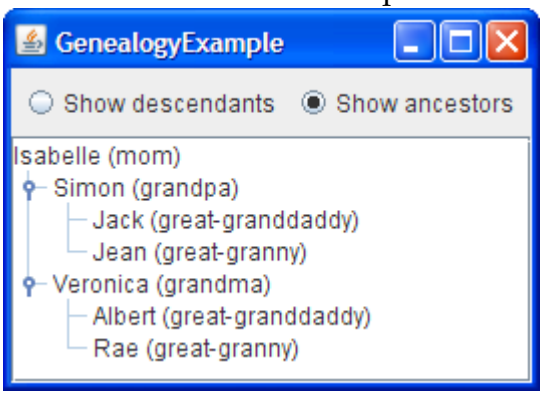
Pour insérer le noeud dans le modèle, le code utilise la méthode `insertNodeInto` fournie par `DefaultTreeModel`.

6 Créer un modèle de données

Si le `DefaultTreeModel` ne convient pas à vos besoins, il est possible d'écrire un modèle de données personnalisé. Votre modèle de données doit implémenter l'interface `TreeModel`. `TreeModel` déclare des méthodes pour particulariser les noeuds d'un arbre, imposer le nombre d'enfants d'un noeud particulier, déterminer si un noeud est une feuille, envoyer des notifications en cas de changement dans l'arbre, ajouter et enlever des « listeners » de modèle d'arbre.

L'interface `TreeModel` prend n'importe objet pour un noeud, ce qui est commode. Il n'est pas nécessaire que les noeuds soient un `DefaultMutableTreeNode`, ni même un `TreeNode`. Ainsi, si l'interface `TreeNode` ne convient pas à votre arbre, libre à vous de concevoir votre propre représentation d'un noeud. Par exemple, si vous avez un structure de données hiérarchique existante, il n'est pas nécessaire de la duppliquer ni même de la transformer. Il suffit seulement d'implémenter votre modèle d'arbre de telle façon qu'il utilise la structure de données existante.

Le code suivant montre une application appelée GenealogyExample qui affiche les descendants et les ancêtres d'une personne.



Puisque le modèle est une sous-classe de `Object` plutôt que, disons, une sous-classe de `DefaultTreeModel`, il doit implémenter l'interface `TreeModel`.

Voici à quoi ressemble l'interface `TreeModel` de l'API Java :

```
public interface TreeModel {
    public Object getRoot();
    public Object getChild(Object o, int i);
    public int getChildCount(Object o);
    public boolean isLeaf(Object o);
    public void valueForPathChanged(TreePath tp, Object o);
    public int getIndexOfChild(Object o, Object o1);
    public void addTreeModelListener(TreeModelListener tl);
    public void removeTreeModelListener(TreeModelListener tl);
}
```

Il faut donc implémenter toutes ces méthodes pour avoir des informations à propos des noeuds, comme celui qui est à la racine et que sont ceux d'un noeud particulier. Dans le cas de `GenealogyModel`, chaque noeud représente un objet de type `Person`, une classe qui n'implémente pas `TreeNode`.

Un modèle d'arbre doit aussi implémenter des méthodes pour ajouter et enlever des « listeners » et doit leur lancer des événements de type `TreeModelEvents` quand la structure

de l'arbre ou que les données changent. Par exemple, quand l'utilisateur demande à basculer entre « montrer les ancêtres » et « montrer les descendants », le modèle d'arbre fait les changements et lance un événement pour informer ses listeners.

```
public class GenealogyModel implements TreeModel {
   private boolean showAncestors;
   private Vector<TreeModelListener> treeModelListeners = new Vector<
       TreeModelListener>();
   private Person rootPerson;

   public GenealogyModel(Person root) {
       showAncestors = false;
       rootPerson = root;
   }

   // Used to toggle between show ancestors/show descendant and to change the root
       of the tree.
   public void showAncestor(boolean b, Object newRoot) {
       showAncestors = b;
       Person oldRoot = rootPerson;
       if (newRoot != null) {
          rootPerson = (Person)newRoot;
       }
       fireTreeStructureChanged(oldRoot);
   }

/////////////// Fire events //////////////////////////////////////////
   // The only event raised by this model is TreeStructureChanged with the root as
       path, i.e. the whole tree has changed.
   protected void fireTreeStructureChanged(Person oldRoot) {
       int len = treeModelListeners.size();
       TreeModelEvent e = new TreeModelEvent(this, new Object[] {oldRoot});
       for (TreeModelListener tml : treeModelListeners) {
           tml.treeStructureChanged(e);
       }
   }

/////////////// TreeModel interface implementation ////////////////////
   // Adds a listener for the TreeModelEvent posted after the tree changes.
   public void addTreeModelListener(TreeModelListener l) {
       treeModelListeners.addElement(l);
   }

   // Returns the child of parent at index index in the parent's child array.
   public Object getChild(Object parent, int index) {
       Person p = (Person)parent;
       if (showAncestors) {
           if ((index > 0) && (p.getFather() != null)) {
               return p.getMother();
           }
           return p.getFather();
       }
       return p.getChildAt(index);
   }
```

```
    // Returns the number of children of parent.
    public int getChildCount(Object parent) {
        Person p = (Person)parent;
        if (showAncestors) {
            int count = 0;
            if (p.getFather() != null) {
                count++;
            }
            if (p.getMother() != null) {
                count++;
            }
            return count;
        }
        return p.getChildCount();
    }
    // Returns the index of child in parent.
    public int getIndexOfChild(Object parent, Object child) {
        Person p = (Person)parent;
        if (showAncestors) {
            int count = 0;
            Person father = p.getFather();
            if (father != null) {
                count++;
                if (father == child) {
                    return 0;
                }
            }
            if (p.getMother() != child) {
                return count;
            }
            return -1;
        }

        return p.getIndexOfChild((Person)child);
    }
    // Returns the root of the tree.
    public Object getRoot() {
        return rootPerson;
    }
    // Returns true if node is a leaf.
    public boolean isLeaf(Object node) {
        Person p = (Person)node;
        if (showAncestors) {
            return ((p.getFather() == null)
                 && (p.getMother() == null));
        }
        return p.getChildCount() == 0;
    }

    // Removes a listener previously added with addTreeModelListener().
    public void removeTreeModelListener(TreeModelListener l) {
        treeModelListeners.removeElement(l);
```

```
    }

    // Messaged when the user has altered the value for the item identified by path
        to newValue.
    // Not used by this model.
    public void valueForPathChanged(TreePath path, Object newValue) {
        System.out.println("*** valueForPathChanged : " + path + " --> " + newValue);
    }
}
```