TD5 – NF28 : Drag & Drop

Présentation

Objectif

L'objectif de ce TD est d'implémenter une fonctionnalité d'une interface graphique très courante et devenue indispensable : le Drag and Drop (DnD).

Scénario

Partant de l'application du TD4, un item du menu d'édition permet d'ajouter un contact anonyme qui devra par la suite être modifié. Ce contact vient se placer dans la liste des contacts mais en dehors de toute catégorie. Par glissement, l'utilisateur pourra le positionner dans la bonne catégorie. Il pourra également déplacer un contact d'une catégorie à une autre ou encore déplacer une catégorie. Relâcher (un drop) un contact sur un contact ne doit produire aucun effet.

Principe du Geste

La plupart des composants reconnaissent le geste Drag dès que leur méthode `setDragEnabled(true)` est invoquée. Les principaux composants acceptent le geste Drop notamment les composants texte. Pour spécialiser le comportement lors du Drag and/or Drop il est nécessaire d'implémenter un `TransferHandler`. Cette classe fournit un mécanisme simple lors du transfert de données à partir et/ou vers un `JComponent`. La donnée sélectionnée lors d'un mouvement est encapsulée dans un objet dit Transferable.
Pour adapter un Drag and/or Drop il suffit de créer deux classes, l'une héritant de `TransferHandler` et l'autre implémentant l'interface `Transferable` et de surcharger/implémenter les méthodes nécessaires.
La méthode `setTransferHandler(<TransferHandler>)` permet d'attacher un handler sur un composant.

Méthodes de `TransferHandler` pour le Drag

`getSourceActions(JComponent)` : fournit les actions acceptées par le composant source du drag telles que `COPY`, `MOVE`, or `LINK`.

`createTransferable(JComponent)` : crée et retourne l'objet transferable au cœur du mouvement.

Méthodes de `TransferHandler` pour le Drop

`canImport(TransferHandler.TransferSupport)` : retourne `true` si la zone sous le curseur est capable d'accepter le Drop et `false` sinon. Cette méthode doit questionner le TransferSupport en paramètre s'il est en situation de drop (isDrop()) et le transferable qu'on obtient par cet objet s'il est en mesure de relâcher un objet compatible avec ce que demande la zone. Ce « dialogue » se fait grâce à la notion de DataFlavor.

`importData(TransferHandler.TransferSupport)` : si le drop est accepté cette méthode définit le transfert de données à partir du Transferable vers le composant cible. La méthode retourne `true` si le transfert réussit, `false` sinon.

`TransferSupport.getComponent()` retourne le composant sur lequel se fait le drop.

Proposition de phases de réalisation

Phase 1 : Création de l'objet Transferable

Un objet Transferable encapsule les données à transférer, c'est-à-dire l'objet source et détermine l'objet destination défini lors du « drop » qui pourrait être différent de l'objet source. L'objet source est un nœud (accompagné de son UserObject). L'objet destination est ici le même, mais ce pourrait être une chaîne si la cible était un champ texte.
Créer une classe `NodeTransferable` et un constructeur à partir d'un DefaultMutabletreeNode. Cette classe implémente l'interface Transferable. Compléter les méthodes requises. Les objets échangés doivent être accompagnés de méta-données servant notamment à déterminer l'objet à restituer à la cible.

```
protected static final DataFlavor nodeFlavor = new DataFlavor(
                  DataFlavor.javaJVMLocalObjectMimeType,"ContactNode");

public static DataFlavor getNodeFlavor () {
      return nodeFlavor;}

public DataFlavor[] getTransferDataFlavors() {
      DataFlavor[] result = {nodeFlavor};
      return result;}

public boolean isDataFlavorSupported(DataFlavor arg0) {
      return Arrays.asList(getTransferDataFlavors()).contains(arg0);}

public Object getTransferData(DataFlavor arg0)
                  throws UnsupportedFlavorException, IOException {
            if (arg0 == nodeFlavor)
                  return node;
            return null;}
```

Phase 2 : Création de TransferHandler

Créer une classe héritant de TransferHandler et implémentant les quatre méthodes précédentes puisque elle sert pour le geste drag et le geste drop.

`getSourceActions(JComponent)` : retourne le geste `MOVE`.

`createTransferable(JComponent c)` : récupère le nœud sélectionné dans l'arbre et l'encapsule dans un `NodeTransferable`.

`canImport(TransferHandler.TransferSupport)` : vérifie que le `TransferSupport` est en état drop (`isDrop()`) et teste le transferable pour voir s'il accepte le `NodeTransferable.nodeFlavor`.

```
public boolean importData(TransferSupport support) {
 if (canImport(support)) {
  try {
   Transferable t = support.getTransferable();
   DefaultMutableTreeNode dmt = (DefaultMutableTreeNode)
                        t.getTransferData(NodeTransferable.nodeFlavor);
   JTree.DropLocation dl =    (JTree.DropLocation) support.getDropLocation();
   TreePath tp = dl.getPath();
   if (tp == null) {return false;}
   DefaultMutableTreeNode parent = (DefaultMutableTreeNode)
                                                tp.getLastPathComponent();
   if (parent.getUserObject() instanceof Contact) {return false;}
   JTree tree = (JTree)support.getComponent();
   DefaultTreeModel tm = (DefaultTreeModel) tree.getModel();
   parent.add(dmt);
   tm.reload();
   tree.expandPath(tp);
   return true;
  }
  catch(Exception ex) {ex.printStackTrace();}
 }
 return false;
}
```

Phase 3 : Spécialisation du Transferable

Adapter le transferable pour qu'il puisse répondre positivement à une demande de `DataFlavor.stringFlavor` de la part d'un champ texte dans une situation de Drop. On pourra tester avec l'un des champs texte déjà présent dans l'interface.
On pourra également implémenter un `FileTransferHandler` qui permettra de faire un drag d'un fichier à partir du bureau de l'ordinateur et un drop sur la fenêtre. La fenêtre en faisant le parsing comme dans le cas d'une ouverture de fichier classique.
Java possède un dataflavor particulier pour cette situation : `DataFlavor.javaFileListFlavor` et le type de l'objet récupéré à partir du transferable est `List<File>`.