



Interface Graphique en Java 1.6

Presse-papier et Drag and Drop

Sébastien Paumier



Le presse-papier

- le **Clipboard** sert à stocker des données pour faire du copier/couper/coller
- presse-papier système:

```
Toolkit toolkit=Toolkit.getDefaultToolkit();  
Clipboard clipboard=toolkit.getSystemClipboard();
```

- presse-papier de sélection (ne marche pas sur tous les systèmes):

```
Toolkit toolkit=Toolkit.getDefaultToolkit();  
Clipboard clipboard=toolkit.getSystemSelection();
```

- on peut aussi en créer: `Clipboard c=new Clipboard("biniou");`



Le presse-papier

- les données peuvent être présentées sous plusieurs formes, caractérisées par des **DataFlavor**
- exemple: du texte enrichi peut être vu soit comme du texte enrichi, soit comme du texte brut, et ce, sous différents encodages



Le presse-papier

- une **DataFlavor** est définie par:
 - un type MIME
 - une classe représentant les données:
 - soit les données elles-mêmes (**String**, **Image**, ...)
 - soit un moyen d'y accéder (exemple: **Reader** pour du texte)
- 3 **DataFlavor** prédéfinies:
 - DataFlavor.stringFlavor (String)**
 - DataFlavor.imageFlavor (Image)**
 - DataFlavor.javaFileListFlavor (List<File>)**



Le presse-papier

- les méthodes de **Clipboard** sont:
 - `DataFlavor[] getAvailableDataFlavors()`
 - connaître les **DataFlavor** disponibles pour le contenu courant du presse-papier
 - `boolean isDataFlavorAvailable(DataFlavor flavor)`
 - tester une **DataFlavor** en particulier
 - `void setContents(Transferable contents, ClipboardOwner owner)`
 - définir le contenu du presse-papier



Le presse-papier

- 2 façons de lire le contenu:
 - `Object getData(DataFlavor flavor)`
 - obtenir le contenu sous la forme indiquée par `flavor`
 - `Transferable getContents(Object requestor)`
 - obtenir le contenu sous **TOUTES** les formes possibles
 - risque de grosse consommation mémoire !
- et pour écouter les changements:
 - `void addFlavorListener(FlavorListener l)`
 - `void removeFlavorListener(FlavorListener l)`



Le presse-papier

- récupération du contenu:

```
public static void main(String[] args) {
    final JFrame f = new JFrame("Clipboard paste demonstration");
    final JDesktopPane desktop=new JDesktopPane();
    JButton paste=new JButton("Paste system clipboard content");
    Toolkit toolkit=Toolkit.getDefaultToolkit();
    final Clipboard clipboard=toolkit.getSystemClipboard();
    paste.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            if (clipboard.isDataFlavorAvailable(DataFlavor.imageFlavor)) {
                desktop.add(createInternalFrame("Image", getPastedImage(clipboard)));
            }
            else if (clipboard.isDataFlavorAvailable(DataFlavor.stringFlavor)) {
                desktop.add(createInternalFrame("Text", getPastedText(clipboard)));
            }
            else if (clipboard.isDataFlavorAvailable(DataFlavor.javaFileListFlavor)) {
                desktop.add(createInternalFrame("Files", getPastedFiles(clipboard)));
            }
        }
    });
    final DefaultListModel model=new DefaultListModel();
    updateModel(clipboard,model);
    f.getContentPane().add(new JScrollPane(new JList(model)), BorderLayout.WEST);
    ...
}
```



Le presse-papier

- on écoute les changements pour mettre à jour la liste des **DataFlavor**:

```
public static void main(String[] args) {
    ...
    final DefaultListModel model=new DefaultListModel();
    updateModel(clipboard,model);
    f.getContentPane().add(new JScrollPane(new JList(model)),BorderLayout.WEST);
    clipboard.addFlavorListener(new FlavorListener() {
        @Override
        public void flavorsChanged(FlavorEvent e) {
            updateModel(clipboard,model);
        }
    });
    ...
}

protected static void updateModel(Clipboard clipboard, DefaultListModel model) {
    model.clear();
    DataFlavor[] flavors=clipboard.getAvailableDataFlavors();
    for (int i=0;i<flavors.length;i++) {
        model.addElement(flavors[i].getMimeType());
    }
}
```



Le presse-papier

- récupération d'une image:

```
protected static JComponent getPastedImage(Clipboard clipboard) {  
    try {  
        Image img = (Image) clipboard.getData(DataFlavor.imageFlavor);  
        return new JLabel(new ImageIcon(img));  
    }  
    catch (IllegalStateException e1) {  
        /* Clipboard access can fail */  
        return null;  
    }  
    catch (UnsupportedFlavorException e) {  
        return null; /* Should not happen */  
    }  
    catch (IOException e) {  
        return null; /* Should not happen */  
    }  
}
```

les accès peuvent échouer sur certains systèmes



Le presse-papier

- on obtient du texte ou une liste de fichiers de la même façon:

```
protected static JComponent getPastedText(Clipboard clipboard) {  
    try {  
        String s = (String) clipboard.getData(DataFlavor.stringFlavor);  
        return new JTextArea(s);  
    }  
    ...  
}
```

```
protected static JComponent getPastedFiles(Clipboard clipboard) {  
    try {  
        List<File> files = (List<File>)clipboard.getData(DataFlavor.javaFileListFlavor);  
        return new JList(files.toArray());  
    }  
    ...  
}
```



Le presse-papier

- résultat:

Clipboard paste demonstration

Paste system clipboard content

image/x-java-image; class=java.awt.Image

Files

- X:\BOULOT\Enseignement\IG\Cours 8-dessin.pdf
- X:\BOULOT\Enseignement\IG\Cours 9-LnF et créer un composant.pdf
- X:\BOULOT\Enseignement\IG\Cours 10-tables.pdf
- X:\BOULOT\Enseignement\IG\GuiExampleViewer.jar
- X:\BOULOT\Enseignement\IG\syntax.jar
- X:\BOULOT\Enseignement\IG\GUISeb.zip
- X:\BOULOT\Enseignement\IG\LaF MacOS.tiff
- X:\BOULOT\Enseignement\IG\laf_macos_pressed.tiff
- X:\BOULOT\Enseignement\IG\laf_macos_unpressed.tiff
- X:\BOULOT\Enseignement\IG\Cours 1-introduction à Swing.odp
- X:\BOULOT\Enseignement\IG\Cours 2-gestionnaires de géomé

Text

```
protected static JComponent getPastedText(Clipboard clipboard) {  
    try {  
        String s = (String) clipboard.getData(DataFlavor.stringFlavor);  
        return new JTextArea(s);  
    }  
}
```



Mettre du contenu personnalisé

- on peut mettre ce que l'on veut dans le presse-papier, pour peu que l'on définisse une classe de représentation qui implémente **Transferable**:
 - **Object** `getTransferData(DataFlavor flavor)`
 - obtenir les données sous une forme précise
 - **DataFlavor[]** `getTransferDataFlavors()`
 - obtenir la liste des DataFlavor supportées par les données
 - **boolean** `isDataFlavorSupported(DataFlavor f)`



Mettre du contenu personnalisé

- exemple: transférer des **JButton**

```
public class ButtonSelection implements Transferable {

    public final static DataFlavor buttonFlavor=new DataFlavor(JButton.class,"buttonFlavor");
    private final static DataFlavor[] supportedFlavors=new DataFlavor[] {buttonFlavor};
    private JButton button;

    public ButtonSelection(JButton b) {
        this.button=b;
    }

    @Override public Object getTransferData(DataFlavor flavor)
        throws UnsupportedOperationException, IOException {
        if (!buttonFlavor.equals(flavor)) {
            throw new UnsupportedOperationException(flavor);
        }
        return button;
    }

    @Override public DataFlavor[] getTransferDataFlavors() {
        return supportedFlavors;
    }

    @Override public boolean isDataFlavorSupported(DataFlavor flavor) {
        return buttonFlavor.equals(flavor);
    }
}
```



Mettre du contenu personnalisé

- application: utiliser des **JPopupMenu** pour couper/coller des **JButton**
- création de boutons avec des listeners sérialisables:

```
static abstract class SerializableActionListener implements ActionListener, Serializable {
    /* We just need to indicate that we want a serializable ActionListener */
}

final static Random random=new Random();
private static Component createButton(String title,final String message) {
    int n=random.nextInt(1000);
    JButton b=new JButton(title+" "+n);
    /* We don't want to inheritate the JPanel popup menu */
    b.setInheritsPopupMenu(false);
    b.setComponentPopupMenu(cutPopup);
    final String id="Message ID: "+n;
    b.addActionListener(new SerializableActionListener () {
        @Override public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(null,message,id,JOptionPane.INFORMATION_MESSAGE);
        }
    });
    return b;
}
```



Mettre du contenu personnalisé

- création de l'action "Cut": on enlève le bouton de son panel et le met dans le presse-papier

```
final Action cut=new AbstractAction("Cut button") {
    @Override
    public void actionPerformed(ActionEvent e) {
        JButton b=(JButton) cutPopup.getInvoker();
        /* We don't want the popup menu to be serialized, since we
         * have a static final reference on it. Better to restore it
         * at paste time.
         */
        b.setComponentPopupMenu(null);
        JPanel p=(JPanel)b.getParent();
        p.remove(b);
        p.revalidate();
        p.repaint();
        try {
            clipboard.setContents(new ButtonSelection(b), null);
        } catch (IllegalStateException e1) {
            /* Do nothing */
        }
    }
};
cutPopup.add(new JMenuItem(cut));
```



Mettre du contenu personnalisé

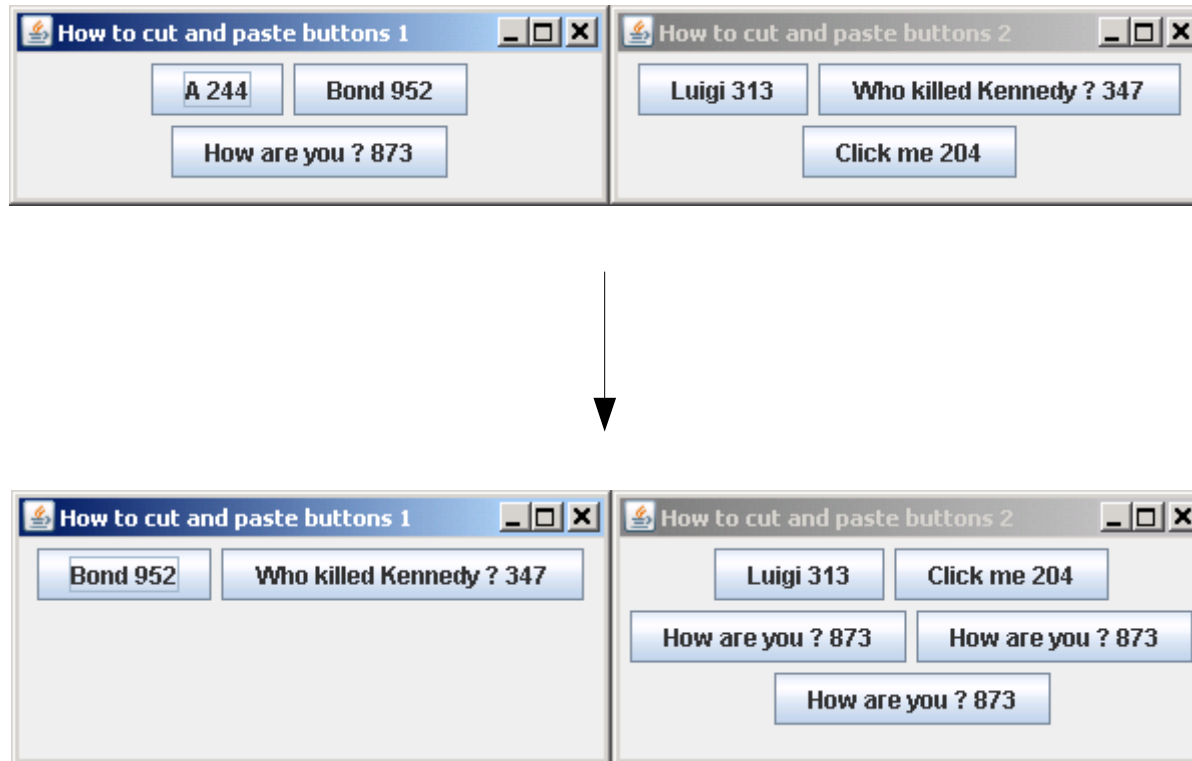
- création de l'action "Paste": on récupère un bouton du presse-papier et on le met dans le panel

```
final Action paste=new AbstractAction("Paste button") {
    @Override
    public void actionPerformed(ActionEvent e) {
        JPanel p=(JPanel)pastePopup.getInvoker();
        try {
            JButton b=(JButton) clipboard.getData(ButtonSelection.buttonFlavor);
            /* We have to restore the popup menu */
            b.setComponentPopupMenu(cutPopup);
            p.add(b);
            p.revalidate();
            p.repaint();
        } catch(IllegalStateException e1) {
            /* Do nothing */
        } catch (UnsupportedFlavorException e1) {
            /* Should not happen */
        } catch (IOException e1) {
            /* Should not happen */
        }
    }
};
```



Mettre du contenu personnalisé

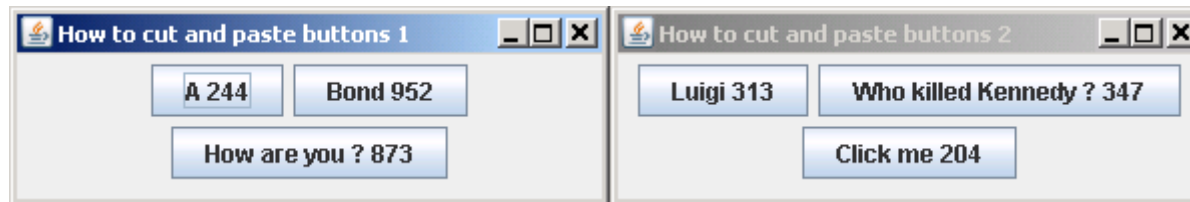
- on peut ainsi couper-coller des boutons, éventuellement plusieurs fois:





Mettre du contenu personnalisé

- comme on utilise le presse-papier système, on peut coller un bouton qui a été coupé dans une session précédente:





Le Drag and Drop

- principe: déplacer à la souris des données d'une source vers une destination, en stockant les données dans un objet **Transferable**
- avantage: la source et la destination ne sont pas forcément dans la même application



Le Drag and Drop

- exemple: implémenter un jeu d'échecs
- échiquier=grille de 8×8 contenant des **JLabel**
- les labels afficheront les images des pièces
- chaque label sera à la fois une zone de drag et une zone de drop



Le Drag and Drop

- pour commencer, il faut détecter un début de DnD du côté de la source
- pour cela, il faut gérer 2 conditions:
 - A) est-ce que le mouvement de souris correspond à un début de DnD ?
 - B) si c'est le cas, est-ce que l'objet sur lequel on veut démarrer un DnD est d'accord ?



Le Drag and Drop

- pour gérer le début de DnD, on utilise un objet **DragSource**
- si l'on veut personnaliser la condition A (par exemple pour imposer une combinaison de touches), on crée son propre **DragSource**
- sinon, on utilise celui du système:

`DragSource.getDefaultDragSource()`



Le Drag and Drop

- pour gérer la condition B, on utilise un **DragGestureRecognizer**
- le plus simple est de l'obtenir avec la méthode suivante de **DragSource**:

```
createDefaultDragGestureRecognizer (  
    Component c, int actions,  
    DragGestureListener dgl)
```

- qui crée un **DragGestureRecognizer** et le met en activité sur le composant **c**



Le Drag and Drop

- le paramètre **actions** définit le type d'opération associé au DnD, grâce à des constantes de **DnDConstants**:
 - **ACTION_COPY**
 - **ACTION_MOVE**
 - **ACTION_MOVE_OR_COPY**
 - **ACTION_LINK**
 - **ACTION_NONE**



Le Drag and Drop

- le `DragGestureListener` définit si oui ou non on doit démarrer un DnD
- le test se fait dans la méthode:

```
public void  
    dragGestureRecognized(DragGestureEvent dge)
```

- pour démarrer le DnD, on invoque sur `dge` la méthode:

```
dge.startDrag(Cursor c, Transferable t)
```
- il y a des curseurs prédéfinis pour le DnD dans `DragSource`



Le Drag and Drop

- `dge.startDrag(...)` est un raccourci vers la méthode `startDrag` de `DragSource`
- **piège:** certaines versions de `startDrag` proposent des paramètres `dragImage` et `imageOffset` censés être utilisés pour afficher un truc qui se déplace pendant le DnD
- ces paramètres ne sont pas utilisables!!!



La source du DnD

- on peut suivre le déplacement, soit du point de vue de la source, soit de celui de la destination (soit les deux)
- pour la source, on utilise un **DragSourceListener**:

```
void dragDropEnd(DragSourceDropEvent dsde)
```

- prévient la source que le DnD est fini, avec succès ou non (consulter `getDropSuccess()`)

```
void dragEnter(DragSourceDragEvent dsde)
```

- la souris vient d'entrer dans une zone susceptible d'accepter le drop



La source du DnD

`void dragExit(DragSourceEvent dse)`

- la souris n'est plus sur une zone de drop possible

`void dragOver(DragSourceDragEvent dsde)`

- la souris survole une zone susceptible d'accepter le drop

`void dropActionChanged(DragSourceDragEvent dsde)`

- l'utilisateur a modifié les conditions requises de début de DnD, par exemple en relâchant une touche du clavier
- le **DragSourceListener** est passé en paramètre à la méthode **startDrag**



La destination du DnD

- méthodes de **DropTargetListener**:

void dragEnter(DropTargetDragEvent dtde)

- la souris entre dans une zone susceptible d'accepter le drop

void dragExit(DropTargetEvent dte)

- la souris est sortie de la zone de drop

void dragOver(DropTargetDragEvent dtde)

- la souris survole une zone de drop possible
- attention: cette méthode est invoquée à intervalles réguliers, **même si la souris ne bouge pas !**



La destination du DnD

`void dropActionChanged(DropTargetDragEvent dtde)`

- l'utilisateur a modifié les conditions de drop (exemple: la zone de drop n'est plus active)

`void drop(DropTargetDropEvent dte)`

- invoquée quand l'utilisateur relâche la souris
- la méthode doit, soit refuser le drop avec `rejectDrop()`, soit l'accepter avec `acceptDrop(int dropAction)`; `dropAction` permet de dire ce que l'on accepte (copier, coller, ...)



La destination du DnD

- suite de `drop()` :
 - si l'on accepte le drop, **et pas avant**, on peut récupérer les données du `Transferable`
 - une fois les données récupérées, **et pas avant**, on signale si le drop s'est bien passé avec `dropComplete(boolean success)`

exemple: on a accepté un fichier, mais après examen, on voit qu'il n'a pas le bon contenu



La destination du DnD

- pour installer un **DropTargetListener** sur un composant, il faut utiliser un **DropTarget**:

```
DropTarget dropTarget=new DropTarget(component,dropTargetListener);  
dropTarget.setActive(true);  
component.setDropTarget(dropTarget);
```

- la propriété **active** indique si la zone accepte ou non les drops
- note: c'est le **DropTarget** qui gère l'autoscrolling pendant le DnD (au besoin)



Pendant le DnD

- une fois le DnD démarré, si l'on souhaite voir bouger une image en même temps que la souris, il faut la gérer soi-même
- solution:
 - mettre un **GlassPane** sur la fenêtre
 - avoir une référence sur l'image, par exemple dans le **Transferable** contenant les données
 - mettre à jour le **GlassPane** quand la souris bouge



Pendant le DnD

- exemple de **GlassPane**:

```
public class ChessGlassPane extends JComponent {

    private Image image;
    private int x,y;

    public void setImage(Image image,int x,int y) {
        this.image=image;
        this.x=x;
        this.y=y;
        paintImmediately(0,0,getWidth(),getHeight());
    }

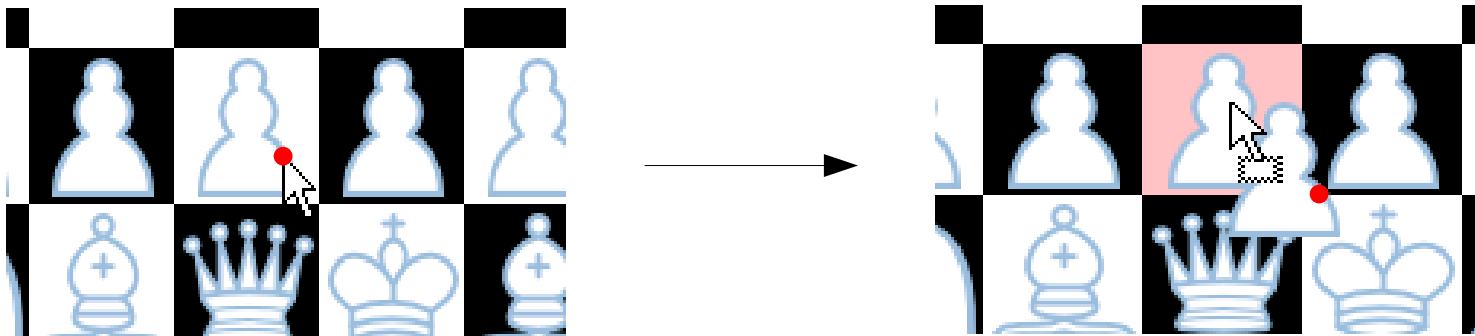
    @Override
    protected void paintComponent(Graphics g) {
        if (image==null) return;
        g.drawImage(image,x,y,null);
    }

    @Override
    public boolean contains(int x, int y) {
        /* If we don't do that, we can click through the glass pane, but
         * it's the glass pane's mouse cursor that is taken into account */
        return false;
    }
}
```



Pendant le DnD

- attention: si l'image correspond à ce que l'on voit sur la source, il faut faire attention à la position de l'image par rapport à la souris, sinon:



la position du pointeur par rapport à l'image a changé, ce qui choque l'œil de l'utilisateur



Pendant le DnD

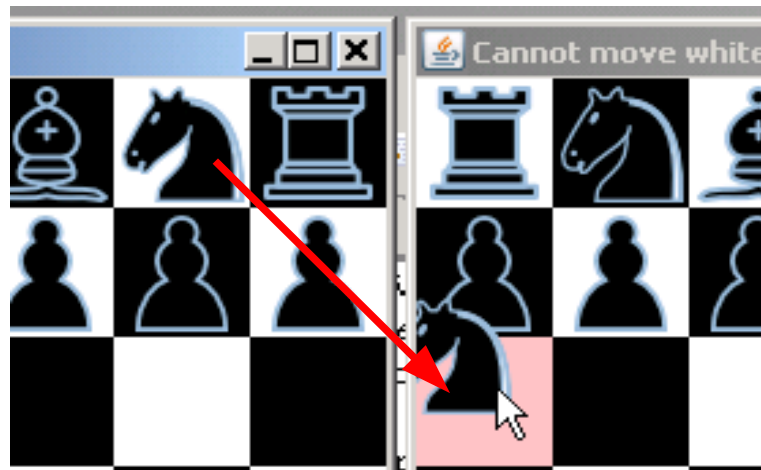
- solution: garder ce décalage lors du rafraîchissement du **GlassPane**
- pour cela, on peut stocker dans le **Transferable** ce décalage qui est donné par **getDragOrigin()**:

```
DragGestureListener dragGestureListener=new DragGestureListener() {
    @Override
    public void dragGestureRecognized(DragGestureEvent dge) {
        ChessPiece piece=model.getPiece(row,column);
        if (piece!=null && piece.isWhite()==model.whiteMustPlay()) {
            dge.startDrag(DragSource.DefaultMoveDrop,
                new PieceSelection(piece,row,column,dge.getDragOrigin()));
        }
    }
};
```



Pendant le DnD

- pour dessiner l'image associée au DnD, il vaut mieux suivre les mouvements depuis la destination, car ça permet de dessiner correctement, même si l'on change de fenêtre:





Pendant le DnD

- on va donc utiliser un **DropTargetListener**
- pour dessiner l'image sur le **GlassPane**, il faut connaître la position de la souris par rapport à celui-ci
- rappels:
 - le **GlassPane** couvre toute la fenêtre, moins ses marges (bords et barre de titre)
 - on peut obtenir les coordonnées des composants en relatif ou en absolu



Pendant le DnD

- on calcule la position que l'image doit avoir sur le **GlassPane**:

```
int lastX=-1,lastY=-1;

@Override
public void dragOver(DropTargetDragEvent dtde) {
    /* p's coordinates are relative to the drop JLabel */
    Point p=dtde.getLocation();
    int a=p.x;
    int b=p.y;
    if (lastX==a && lastY==b) {
        /* Remember that dragOver is called even if the mouse
         * doesn't move. We don't want unnecessary updates */
        return;
    }
    lastX=a;
    lastY=b;
    Point framePosition=f.getLocationOnScreen();
    Point labelPosition=l.getLocationOnScreen();
    int frameRelativeX=a+labelPosition.x-framePosition.x-f.getInsets().left;
    int frameRelativeY=b+labelPosition.y-framePosition.y-f.getInsets().top;
    ...
}
```



Pendant le DnD

- et on la place, sans oublier le décalage dû à la position de la souris au moment du démarrage du DnD:

```
@Override
public void dragOver(DropTargetDragEvent dtde) {

    ...

    PieceSelection.Data data;
    try {
        data = (PieceSelection.Data) dtde.getTransferable().getTransferData(
            PieceSelection.chessPieceFlavor);
        glassPane.setImage(data.piece.getIcon().getImage(),
            frameRelativeX-data.offset.x,
            frameRelativeY-data.offset.y);
    } catch (UnsupportedFlavorException e) {
        /* Should not happen */
        e.printStackTrace();
    } catch (IOException e) {
        /* Should not happen */
        e.printStackTrace();
    }
}
```



Pendant le DnD

- pour faciliter la vie des joueurs, on va indiquer si une case peut recevoir ou non la pièce déplacée grâce à **dragEnter**:

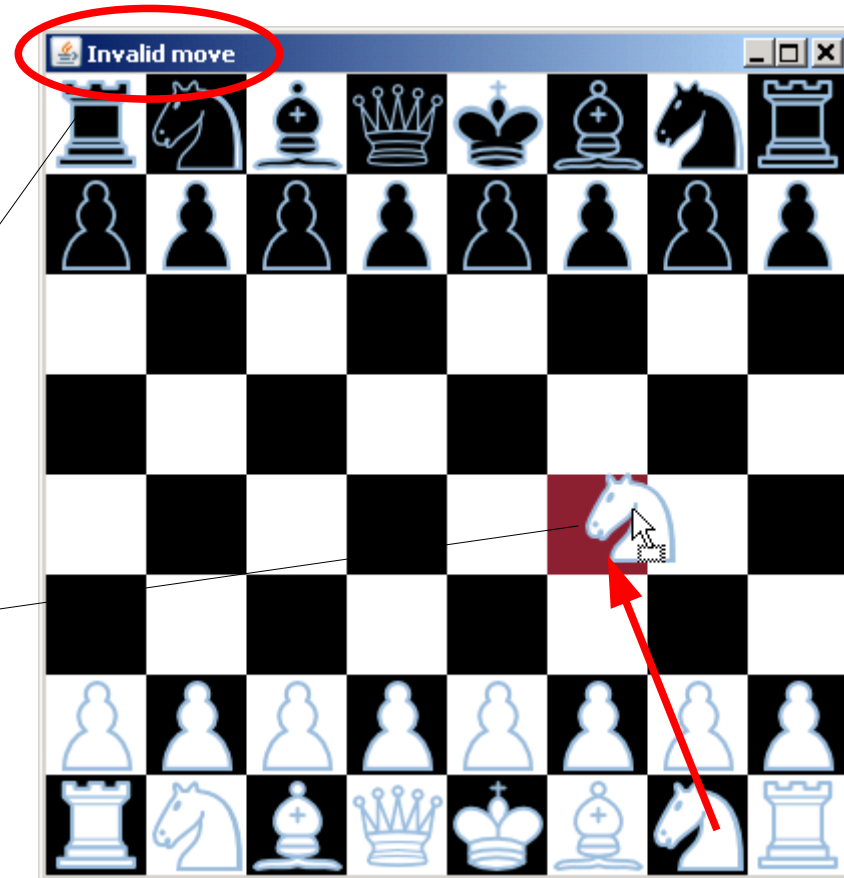
```
@Override public void dragEnter(DropTargetDragEvent dtde) {
    PieceSelection.Data data;
    try {
        data = (PieceSelection.Data) dtde.getTransferable().getTransferData(
            PieceSelection.chessPieceFlavor);
        ChessError ret=model.isRegularMove(data.raw,data.column,row,column,
            model.whiteMustPlay(),true);

        if (ret!=ChessError.OK) {
            l.setBackground(forbidden);
        } else {
            l.setBackground(allowed);
        }
        f.setTitle(ret.getMessage(model.whiteMustPlay()));
    } catch (UnsupportedFlavorException e) {
        /* Should not happen */
        e.printStackTrace();
    } catch (IOException e) {
        /* Should not happen */
        e.printStackTrace();
    }
}
```



Pendant le DnD

- exemple de mouvement interdit:

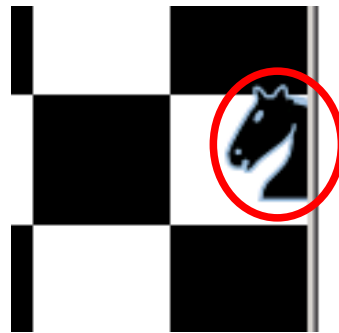


on met à jour le
titre de la fenêtre
et la couleur de la
case visée



Pendant le DnD

- on doit penser à rafraîchir quand on sort d'une zone de drop, sinon:



```
try {  
    date  
    Ches  
    if  
    } el
```

la souris n'est plus sur une zone de drop

- il suffit d'utiliser **dragExit**:

```
@Override  
public void dragExit(DropTargetEvent dte) {  
    l.setCursor(Cursor.getDefaultCursor());  
    l.setBackground(background);  
    glassPane.setImage(null, 0, 0);  
}
```



Le drop

- notre méthode **drop** doit gérer le déplacement d'une pièce:

```
@Override public void drop(DropTargetDropEvent dtde) {
    dtde.acceptDrop(DnDConstants.ACTION_MOVE);
    PieceSelection.Data data;
    try {
        data = (PieceSelection.Data) dtde.getTransferable().getTransferData(
            PieceSelection.chessPieceFlavor);
        ChessError ret=model.isRegularMove(data.raw,data.column,row,column,
            model.whiteMustPlay(),true);

        l.setBackground(background);
        glassPane.setImage(null,0,0);
        if (ret!=ChessError.OK) {
            dtde.dropComplete(false);
            return;
        }
        GameState state=model.movePiece(data.raw,data.column,row,column);
        switch(state) {
            case OK: f.setTitle((model.whiteMustPlay()?"White":"Black")+ " turn");
                updateDndCursors(board,model);
                break;
            case CHECK: f.setTitle("Check"); break;
            case CHECKMATE: f.setTitle("Checkmate"); /* End the game */ break;
        }
        dtde.dropComplete(true);
    } ...
}
```

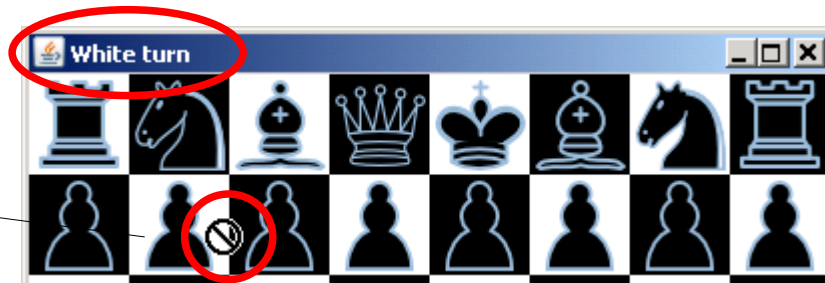


Le drop

- on met les curseurs à jour, pour montrer quelles sont les pièces à jouer:

```
static void updateDndCursors(JPanel board, ChessModel model) {  
    int y=0;  
    for (int r=0;r<8;r++) {  
        for (int c=0;c<8;c++) {  
            ChessPiece p=model.getPiece(r,c);  
            if (p==null || p.isWhite()==model.whiteMustPlay()) {  
                board.getComponent(y).setCursor(Cursor.getDefaultCursor());  
            } else {  
                board.getComponent(y).setCursor(DragSource.DefaultMoveNoDrop);  
            }  
            y++;  
        }  
    }  
}
```

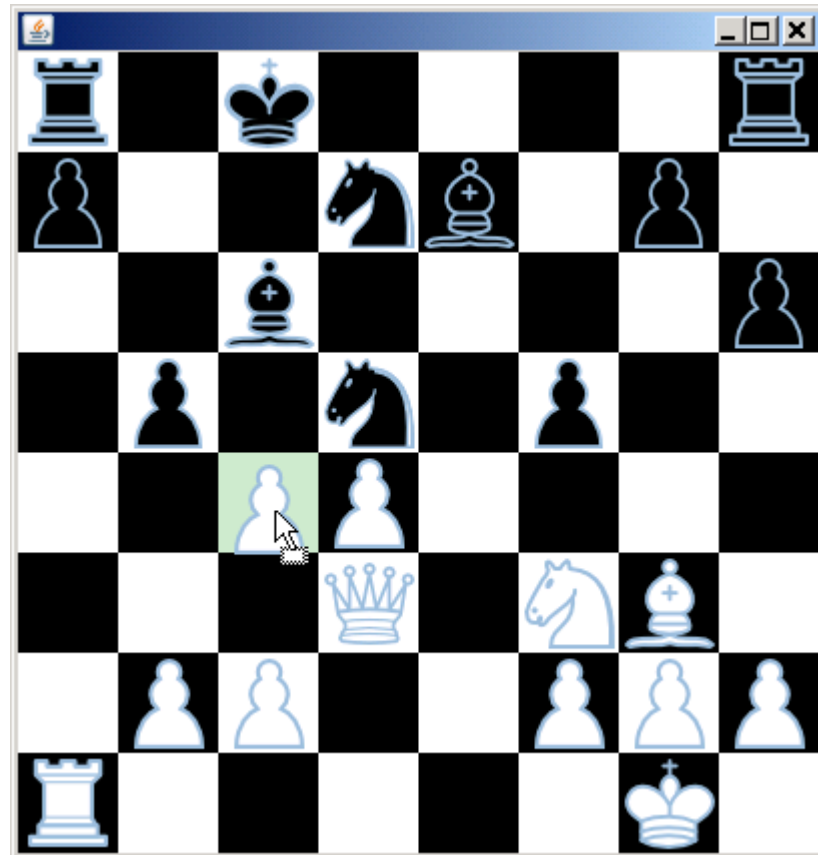
ce n'est pas aux
noirs de jouer





Le drop

- on peut maintenant presque jouer (il manque le roque, la promotion des pions, le pat, une IA, ...)





Le DnD et Swing

- certains composants Swing savent déjà faire du DnD par défaut:
 - **JList, JTree, JTable:**
 - copy vers presse-papier
 - drag copy
 - **JTextField, JTextArea, JTextPane, JEditorPane:**
 - cut/copy/paste vers presse-papier
 - drag copy/move et drop
- il faut invoquer **setDragEnabled(true)**



Le DnD et Swing

- tous les composants Swing peuvent faire facilement du DnD en utilisant `setTransferHandler(TransferHandler th)`
- `TransferHandler` sert à unifier l'import/export de données par presse-papier et DnD
- aurait dû être une interface
- au lieu de ça, c'est une implémentation par défaut qui transfère des propriétés de composants



Le DnD et Swing

- il suffit de définir un **TransferHandler** sur un **JComponent** pour qu'il autorise le le drop
- pour le drag, il faut explicitement invoquer **exportAsDrag** sur le handler:

```
public void exportAsDrag(JComponent comp, InputEvent e, int action)
```

- on peut également exporter vers le presse-papier avec:

```
public void exportToClipboard(JComponent comp, Clipboard clip, int action)
```

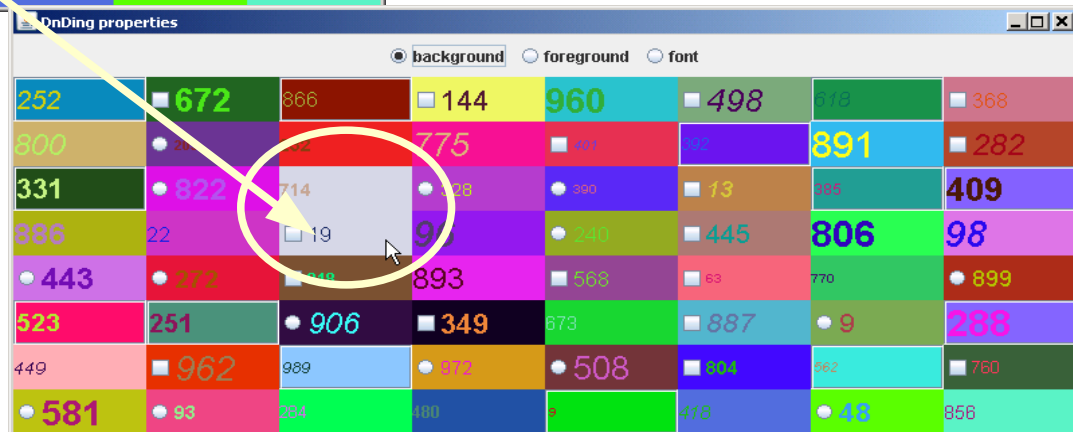


Le DnD et Swing

- exemple: exportation de propriétés graphiques



modification de la couleur de fond





Son propre TransferHandler

- si on veut exporter autre chose qu'une propriété, on doit se préoccuper des méthodes suivantes:
 - **createTransferable**: fabrique l'objet utilisé pour transférer les données, aussi bien par DnD que via le presse-papier
 - **getSourceActions**: indique si on peut faire copy, move ou les deux
 - **exportAsDrag**: a priori, pas besoin d'y toucher
 - même chose pour **exportToClipboard**



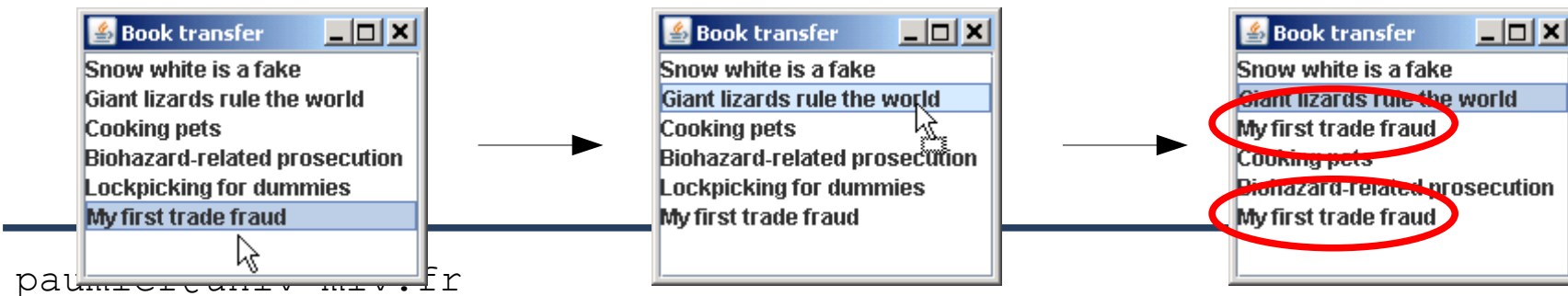
Son propre TransferHandler

- **exportDone**: invoquée quand les données ont été transférées; c'est ici qu'on retirera les données de la source si l'action était *move*
- et pour gérer l'importation de données:
 - **canImport**: indique si le drop ou le paste sont possibles
 - **importData**: récupération effective des données



Son propre TransferHandler

- exemple: création d'un handler pour transférer des du texte dans des **JList**
- on veut pouvoir importer du texte de l'extérieur de la liste
- on veut pouvoir déplacer un élément par DnD à l'intérieur d'une même liste
 - dans ce cas, on doit faire attention aux indices de suppression et d'ajout, sinon:





Son propre TransferHandler

- solution:
 - savoir si l'élément à importer provient de la même liste
 - si oui, ajuster l'indice de suppression si nécessaire
- pour caractériser une **JList** précise d'une application précise, on calcule un identifiant unique avec

```
System.currentTimeMillis();
```



Son propre TransferHandler

- voici le handler:

```
public class BookTransferHandler extends TransferHandler {

    private BookListModel model;
    private int removeIndex=-1;
    private final long ID;

    public BookTransferHandler(BookListModel model, long JListID) {
        this.model=model;
        this.ID=JListID;
    }

    @Override
    public int getSourceActions(JComponent c) {
        return COPY_OR_MOVE;
    }

    @Override
    protected Transferable createTransferable(JComponent c) {
        JList list=(JList)c;
        int index=list.getSelectedIndex();
        if (index==-1) return null;
        String book=(String) model.getElementAt(index);
        removeIndex=index;
        return new ListItemSelection(book, ID);
    }
    ...
}
```



Son propre TransferHandler

- on propose deux formats:
 - texte simple: pour faire du copier-coller normal
 - item de liste: pour gérer le DnD correctement d'une liste vers elle-même

```
public class BookTransferHandler extends TransferHandler {  
  
    ...  
    @Override  
    public boolean canImport(JComponent comp, DataFlavor[] transferFlavors) {  
        List<DataFlavor> list=Arrays.asList(transferFlavors);  
        return list.contains(ListItemSelection.listItemFlavor) ||  
            list.contains(DataFlavor.stringFlavor);  
    }  
    ...  
}
```



Son propre TransferHandler

- pour un item de liste, on doit ajuster, si nécessaire, l'indice de suppression:

```
@Override public boolean importData(JComponent comp, Transferable t) {
    JList list = (JList) comp;
    int index = list.getSelectedIndex();
    if (index==-1) {
        index=model.getSize()-1;
    }
    if (t.isDataFlavorSupported(ListItemSelection.listItemFlavor)) {
        ListItem item;
        try {
            item = (ListItem) t.getTransferData(ListItemSelection.listItemFlavor);
        } catch (UnsupportedFlavorException e) {
            throw new AssertionError(e);
        } catch (IOException e) {
            e.printStackTrace();
            return false;
        }
        if (item.ID == this.ID && index <= removeIndex) {
            /* If both drag and drop occur on the same list,
             * we must take care while removing the item */
            removeIndex++;
        }
        model.add(index + 1, item.value);
        return true;
    }
    ...
}
```



Son propre TransferHandler

- pour importer du texte simple, il suffit d'ajouter l'élément à la fin de la liste:

```
@Override
public boolean importData(JComponent comp, Transferable t) {
    ...
    if (t.isDataFlavorSupported(DataFlavor.stringFlavor)) {
        if (removeIndex!=-1) {
            throw new IllegalStateException("Cannot import a String while a move operation");
        }
        try {
            model.add(index+1, (String) t.getTransferData(DataFlavor.stringFlavor));
        } catch (UnsupportedFlavorException e) {
            throw new AssertionError(e);
        } catch (IOException e) {
            e.printStackTrace();
            return false;
        }
    }
    return false;
}
```



Son propre TransferHandler

- en cas de move, on doit retirer l'élément une fois que les données ont été récupérées par la liste de destination:

```
public class BookTransferHandler extends TransferHandler {  
    ...  
    @Override  
    protected void exportDone(JComponent source, Transferable data, int action) {  
        if (action==MOVE) {  
            model.remove(removeIndex);  
            removeIndex=-1;  
        }  
    }  
    ...  
}
```



Son propre TransferHandler

- l'objet contenant les données doit être sérialisable:

```
public class ListItem implements Serializable {  
  
    public String value;  
    public long ID;  
  
    public ListItem(String book, long ID) {  
        this.value=book;  
        this.ID=ID;  
    }  
}
```



Son propre TransferHandler

- le **Transferable** utilisé doit renvoyer soit du texte simple, soit un **ListItem**:

```
public class ListItemSelection implements Transferable {

    public static DataFlavor listItemFlavor=new DataFlavor(ListItem.class,"listItemFlavor");
    private final static DataFlavor[] supportedFlavors=new DataFlavor[] {
                                                listItemFlavor,DataFlavor.stringFlavor};

    private ListItem item;

    public ListItemSelection(String book,long ID) {
        this.item=new ListItem(book,ID);
    }

    @Override
    public Object getTransferData(DataFlavor flavor) throws UnsupportedFlavorException, IOException {
        if (listItemFlavor.equals(flavor)) {
            return item;
        }
        if (DataFlavor.stringFlavor.equals(flavor)) {
            return item.value;
        }
        throw new UnsupportedFlavorException(flavor);
    }
    ...
}
```



Son propre TransferHandler

- enfin, il n'y a plus qu'à utiliser notre handler sur une liste:

```
BookListModel model=new BookListModel();  
final JList list=new JList(model);  
final TransferHandler handler=new BookTransferHandler(model, System.currentTimeMillis());  
list.setTransferHandler(handler);  
list.setDragEnabled(true);
```

- et c'est fini, car la **JList** gère déjà le couper/copier/coller: notre handler va être automatiquement utilisé

