Drag and Drop in Swing

In computer graphical user interfaces, drag-and-drop is the action of (or support for the action of) clicking on a virtual object and dragging it to a different location or onto another virtual object. In general, it can be used to invoke many kinds of actions, or create various types of associations between two abstract objects. (Wikipedia)

Drag and drop functionality is one of the most visible aspects of the graphical user interface. Drag and drop operation enables users to do complex things intuitively.

Usually, we can drag and drop two things. Data or some graphical objects. If we drag an image from one application to another, we drag and drop binary data. If we drag a tab in Firefox and move it to another place, we drag and drop a graphical component.

The sheer amount of various classes involved with drag and drop operations in Java Swing toolkit might be overwhelming. The best way how to cope with this complexity is to create a small example for all situations. And slowly make it to more complex examples.

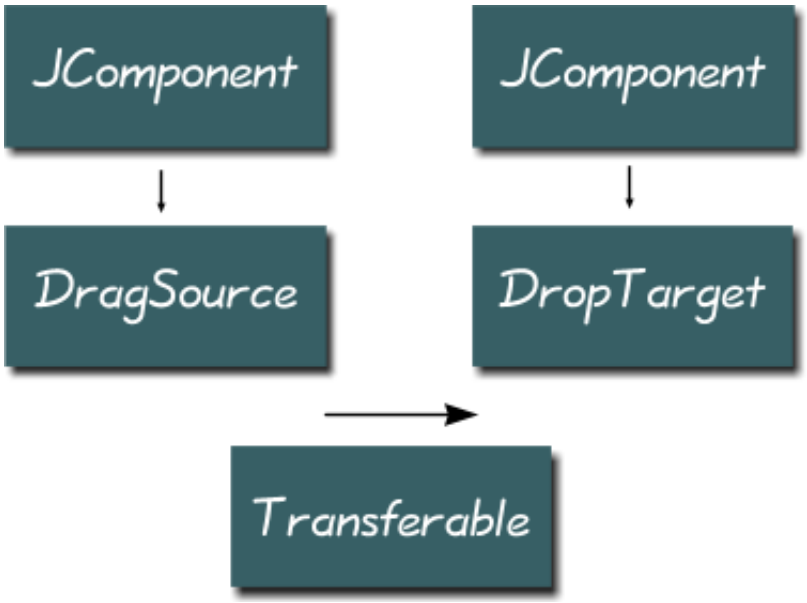

Figure: Drag and drop in Swing

The component, where the drag operation begins must have a `DragSource` object registered. A `DropTarget` is an object responsible for accepting drops in an drag and drop operation. A `Transferable` encapsulates data being transferred. The transferred data can be of various type. A `DataFlavor` object provides information about the data being transferred.

Several Swing components have already a built-in support for drag and drop operations. In such cases, a Swing programmer uses a `TransferHandler` to manage the drag and drop functionality. In situations, where there is no built-in support, the programmer has to create everything from scratch.

A simple drag and drop example

We will demonstrate a simple drag and drop example. We will work with built-in drag and drop support. We utilise a `TransferHandler` class.

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.TransferHandler;


public class SimpleDnD extends JFrame {

    JTextField field;
    JButton button;

    public SimpleDnD() {

        setTitle("Simple Drag & Drop");

        setLayout(null);

        button = new JButton("Button");
        button.setBounds(200, 50, 90, 25);

        field = new JTextField();
        field.setBounds(30, 50, 150, 25);

        add(button);
        add(field);

        field.setDragEnabled(true);
        button.setTransferHandler(new TransferHandler("text"));

        setSize(330, 150);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        setVisible(true);
    }

    public static void main(String[] args) {
        new SimpleDnD();
    }
}
```

In our example we have a text field and a button. We can drag a text from the field and drop it onto the button.

```
field.setDragEnabled(true);
```

The text field has a built in support for dragging. We must enable it.

```
button.setTransferHandler(new TransferHandler("text"));
```

The `TransferHandler` is a class responsible for transferring data between components. The constructor takes a property name as a parameter.

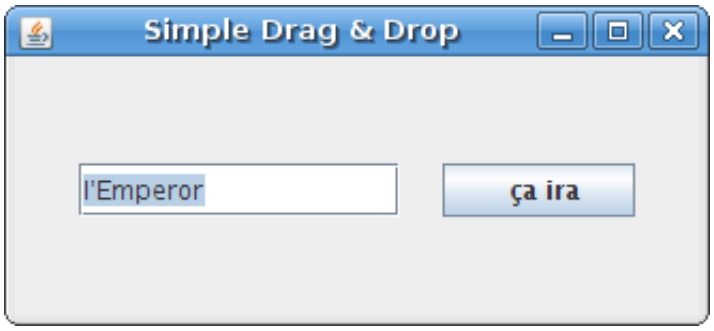

Figure: Simple drag & drop example

Icon drag & drop

Some of the Java Swing components do not have built in drag support. `JLabel` component is such a component. We have to code the drag functionality ourselves.

We will drag and drop icons. In the previous example, we used a text property. This time we will use an icon property.

```
import java.awt.FlowLayout;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.TransferHandler;


public class IconDnD extends JFrame {


    public IconDnD() {
```

```
        setTitle("Icon Drag & Drop");

        JPanel panel = new JPanel(new FlowLayout(FlowLayout.LEFT, 50, 15));

        ImageIcon icon1 = new ImageIcon("sad.png");
        ImageIcon icon2 = new ImageIcon("plain.png");
        ImageIcon icon3 = new ImageIcon("crying.png");

        JButton button = new JButton(icon2);
        button.setFocusable(false);

        JLabel label1  = new JLabel(icon1, JLabel.CENTER);
        JLabel label2  = new JLabel(icon3, JLabel.CENTER);

        MouseListener listener = new DragMouseAdapter();
        label1.addMouseListener(listener);
        label2.addMouseListener(listener);

        label1.setTransferHandler(new TransferHandler("icon"));
        button.setTransferHandler(new TransferHandler("icon"));
        label2.setTransferHandler(new TransferHandler("icon"));

        panel.add(label1);
        panel.add(button);
        panel.add(label2);
        add(panel);

        pack();
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        setVisible(true);
    }

    class DragMouseAdapter extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
            JComponent c = (JComponent) e.getSource();
            TransferHandler handler = c.getTransferHandler();
            handler.exportAsDrag(c, e, TransferHandler.COPY);
        }
    }

    public static void main(String[] args) {
        new IconDnD();
    }
}
```

In the code example, we have two labels and a button. Each component displays an icon. The two labels enable drag gestures, the button accepts a drop gesture.

```
MouseListener listener = new DragMouseAdapter();
label1.addMouseListener(listener);
label2.addMouseListener(listener);
```

The drag support is not enabled by default for the label. We register a custom mouse adapter for both labels.

```
label1.setTransferHandler(new TransferHandler("icon"));
button.setTransferHandler(new TransferHandler("icon"));
label2.setTransferHandler(new TransferHandler("icon"));
```

Each of the three components has a `TransferHandler` class for an icon property. The `TransferHandler` is needed for both drag sources and drag targets as well.

```
JComponent c = (JComponent) e.getSource();
TransferHandler handler = c.getTransferHandler();
handler.exportAsDrag(c, e, TransferHandler.COPY);
```

These code lines initiate the drag support. We get the drag source. In our case it is a label instance. We get its transfer handler object and finally initiate the drag support with the `exportAsDrag()` method call.

Figure: Icon drag & drop example

Custom JList drop example

Some components do not have a default drop support. One of them is a `JList` component. There is a good reason for this. We do not know if the data will be inserted into one row, or two or more rows. So we must implement manually the drop support for the list component.

The comma separated text will be inserted into two or more rows. Text without a comma goes into one row.

```
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.datatransfer.DataFlavor;
import java.awt.datatransfer.Transferable;
```

```
import javax.swing.DefaultListModel;
import javax.swing.DropMode;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextField;
import javax.swing.ListSelectionModel;
import javax.swing.TransferHandler;


public class ListDrop extends JFrame {

    JTextField field;
    DefaultListModel model;

    public ListDrop() {

        setTitle("ListDrop");

        JPanel panel = new JPanel(new FlowLayout(FlowLayout.LEFT, 15, 15));

        JScrollPane pane = new JScrollPane();
        pane.setPreferredSize(new Dimension(180, 150));

        model = new DefaultListModel();
        JList list = new JList(model);

        list.setDropMode(DropMode.INSERT);
        list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        list.setTransferHandler(new ListHandler());

        field = new JTextField("");
        field.setPreferredSize(new Dimension(150, 25));
        field.setDragEnabled(true);

        panel.add(field);
        pane.getViewport().add(list);
        panel.add(pane);

        add(panel);

        pack();

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        setLocationRelativeTo(null);
        setVisible(true);
    }


    private class ListHandler extends TransferHandler {
        public boolean canImport(TransferSupport support) {
            if (!support.isDrop()) {
                return false;
            }

            return support.isDataFlavorSupported(DataFlavor.stringFlavor);
         }

         public boolean importData(TransferSupport support) {
            if (!canImport(support)) {
              return false;
            }

            Transferable transferable = support.getTransferable();
            String line;
            try {
              line = (String)
transferable.getTransferData(DataFlavor.stringFlavor);
            } catch (Exception e) {
              return false;
            }

            JList.DropLocation dl = (JList.DropLocation)
support.getDropLocation();
            int index = dl.getIndex();

            String[] data = line.split(",");
            for (String item: data) {
                if (!item.isEmpty())
                   model.add(index++, item.trim());
            }
            return true;
         }
    }

    public static void main(String[] args) {
        new ListDrop();
    }
}
```

In the above example, we have a text field and a list component. The text in the text field can be dragged and dropped into the list. If the text is comma separated, the words will be split into rows. If not, the text will be inserted into one row.

```
list.setDropMode(DropMode.INSERT);
```

Here we specify a drop mode. The `DropMode.INSERT` specifies, that we are going to insert new items into the list component. If we chose `DropMode.INSERT`, we would drop new items onto the existing ones.

```
list.setTransferHandler(new ListHandler());
```

We set a custom transfer handler class.

```
field.setDragEnabled(true);
```

We enable the drag support for the text field component.

```
public boolean canImport(TransferSupport support) {
    if (!support.isDrop()) {
        return false;
    }
    return support.isDataFlavorSupported(DataFlavor.stringFlavor);
}
```

This method tests suitability of a drop operation. Here we filter out the clipboard paste operations and allow only String drop operations. If the method returns false, the drop operation is cancelled.

```
public boolean importData(TransferSupport support) {
...
}
```

The `importData()` method transfers the data from the clipboard or from the drag and drop operation to the drop location.

```
Transferable transferable = support.getTransferable();
```

The `Transferable` is the class, where the data is bundled.

```
line = (String) transferable.getTransferData(DataFlavor.stringFlavor);
```

We retrieve our data.

```
JList.DropLocation dl = (JList.DropLocation) support.getDropLocation();
int index = dl.getIndex();
```

We get a drop location for the list. We retrieve the index, where the data will be inserted.

```
String[] data = line.split(",");
for (String item: data) {
    if (!item.isEmpty())
        model.add(index++, item.trim());
}
```

Here we split the text into parts and insert it into one or more rows.

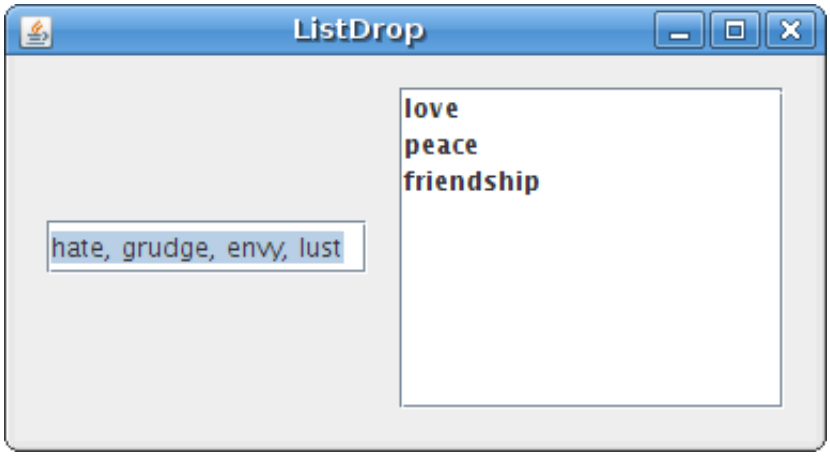

Figure: JList drop example

The previous examples used components with built-in drag and drop support. Next we are going to create a drag and drop functionality from scratch.

Drag Gesture

In the following example we will inspect a simple drag gesture. We will work with several classes needed to create a drag gesture. A `DragSource`, `DragGestureEvent`, `DragGestureListener`, `Transferable`.

```
import java.awt.Color;
import java.awt.Cursor;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.datatransfer.DataFlavor;
import java.awt.datatransfer.Transferable;
import java.awt.dnd.DnDConstants;
import java.awt.dnd.DragGestureEvent;
import java.awt.dnd.DragGestureListener;
import java.awt.dnd.DragSource;
```

```
import javax.swing.JFrame;
import javax.swing.JPanel;


public class DragGesture extends JFrame implements
    DragGestureListener, Transferable {

    public DragGesture() {

        setTitle("Drag Gesture");

        JPanel panel = new JPanel(new FlowLayout(FlowLayout.LEFT, 50, 15));

        JPanel left = new JPanel();
        left.setBackground(Color.red);
        left.setPreferredSize(new Dimension(120, 120));

        DragSource ds = new DragSource();
        ds.createDefaultDragGestureRecognizer(left,
            DnDConstants.ACTION_COPY, this);

        panel.add(left);
        add(panel);

        pack();
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        setVisible(true);
    }

    public void dragGestureRecognized(DragGestureEvent event) {
        System.out.println("grag gesture");
        Cursor cursor = null;
        if (event.getDragAction() == DnDConstants.ACTION_COPY) {
            cursor = DragSource.DefaultCopyDrop;
        }
        event.startDrag(cursor, this);
    }

    public static void main(String[] args) {
        new DragGesture();
    }

    public Object getTransferData(DataFlavor flavor) {
        return null;
```

```
    }

    public DataFlavor[] getTransferDataFlavors() {
        return new DataFlavor[0];
    }

    public boolean isDataFlavorSupported(DataFlavor flavor) {
        return false;
    }
}
```

This simple example demonstrates a drag gesture. The drag gesture is created, when we click on a component and move a mouse pointer, while the button is pressed. The example will show, how we can create a DragSource for a component.

```
public class DragGesture extends JFrame implements
   DragGestureListener, Transferable {
```

The DragGesture implements two interfaces. The `DragGestureListener` will listen for drag gestures. The `Transferable` handles data for a transfer operation. In the example, we will not transfer any data. We will only demonstrate a drag gesture. So the three necessary methods of the `Transferable` interface are left unimplemented.

```
DragSource ds = new DragSource();
ds.createDefaultDragGestureRecognizer(left,
    DnDConstants.ACTION_COPY, this);
```

Here we create a `DragSource` object and register it for the left panel. The DragSource is the entity responsible for the initiation of the Drag and Drop operation. The `createDefaultDragGestureRecognizer()` associates a drag source and `DragGestureListener` with a particular component.

```
public void dragGestureRecognized(DragGestureEvent event) {

}
```

The `dragGestureRecognized()` method responds to a drag gesture.

```
Cursor cursor = null;
if (event.getDragAction() == DnDConstants.ACTION_COPY) {
    cursor = DragSource.DefaultCopyDrop;
}
event.startDrag(cursor, this);
```

The `startDrag()` method of the `DragGestureEvent` finally starts the drag operation. We will specify two parameters. The cursor type and the `Transferable` object.

```
public Object getTransferData(DataFlavor flavor) {
    return null;
}

public DataFlavor[] getTransferDataFlavors() {
return new DataFlavor[0];
}

public boolean isDataFlavorSupported(DataFlavor flavor) {
    return false;
}
```

The object that implements the `Transferable` interface must implement these three methods. As I have already mentioned, we left these methods unimplemented for now.

A complex drag and drop example

In the following example, we create a complex drag and drop example. We create a drag source a drop target and a transferable object.

```
import java.awt.Color;
import java.awt.Cursor;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.datatransfer.DataFlavor;
import java.awt.datatransfer.Transferable;
import java.awt.datatransfer.UnsupportedFlavorException;
import java.awt.dnd.DnDConstants;
import java.awt.dnd.DragGestureEvent;
import java.awt.dnd.DragGestureListener;
import java.awt.dnd.DragSource;
import java.awt.dnd.DropTarget;
import java.awt.dnd.DropTargetAdapter;
import java.awt.dnd.DropTargetDropEvent;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JColorChooser;
import javax.swing.JFrame;
import javax.swing.JPanel;
```

```
public class ComplexExample extends JFrame
    implements DragGestureListener {

    JPanel panel;
    JPanel left;

    public ComplexExample() {

        setTitle("Complex Example");

        panel  = new JPanel(new FlowLayout(FlowLayout.LEFT, 50, 15));

        JButton openb = new JButton("Choose Color");
        openb.setFocusable(false);

        left = new JPanel();
        left.setBackground(Color.red);
        left.setPreferredSize(new Dimension(100, 100));

        openb.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                JColorChooser clr = new JColorChooser();
                Color color = clr.showDialog(panel, "Choose Color",
Color.white);
                left.setBackground(color);
            }
        });

        JPanel right = new JPanel();
        right.setBackground(Color.white);
        right.setPreferredSize(new Dimension(100, 100));

        new MyDropTargetListener(right);

        DragSource ds = new DragSource();
        ds.createDefaultDragGestureRecognizer(left,
            DnDConstants.ACTION_COPY, this);

        panel.add(openb);
        panel.add(left);
        panel.add(right);
        add(panel);

        pack();
```

```
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        setVisible(true);
    }

    public void dragGestureRecognized(DragGestureEvent event) {
        Cursor cursor = null;
        JPanel panel = (JPanel) event.getComponent();

        Color color = panel.getBackground();

        if (event.getDragAction() == DnDConstants.ACTION_COPY) {
            cursor = DragSource.DefaultCopyDrop;
        }

        event.startDrag(cursor, new TransferableColor(color));
    }

    class MyDropTargetListener extends DropTargetAdapter {

        private DropTarget dropTarget;
        private JPanel panel;

     public MyDropTargetListener(JPanel panel) {
        this.panel = panel;

        dropTarget = new DropTarget(panel, DnDConstants.ACTION_COPY,
            this, true, null);
      }


      public void drop(DropTargetDropEvent event) {
        try {

          Transferable tr = event.getTransferable();
          Color color = (Color)
tr.getTransferData(TransferableColor.colorFlavor);

            if (event.isDataFlavorSupported(TransferableColor.colorFlavor))
{

              event.acceptDrop(DnDConstants.ACTION_COPY);
              this.panel.setBackground(color);
              event.dropComplete(true);
              return;
```

```
            }
          event.rejectDrop();
        } catch (Exception e) {
          e.printStackTrace();
          event.rejectDrop();
        }
      }
    }

    public static void main(String[] args) {
        new ComplexExample();
    }
}


class TransferableColor implements Transferable {

    protected static DataFlavor colorFlavor =
        new DataFlavor(Color.class, "A Color Object");

    protected static DataFlavor[] supportedFlavors = {
        colorFlavor,
        DataFlavor.stringFlavor,
    };

    Color color;

    public TransferableColor(Color color) { this.color = color; }

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

    public boolean isDataFlavorSupported(DataFlavor flavor) {
    if (flavor.equals(colorFlavor) ||
        flavor.equals(DataFlavor.stringFlavor)) return true;
    return false;
  }


   public Object getTransferData(DataFlavor flavor)
        throws UnsupportedFlavorException
   {
     if (flavor.equals(colorFlavor))
         return color;
     else if (flavor.equals(DataFlavor.stringFlavor))
```

```
            return color.toString();
        else
            throw new UnsupportedFlavorException(flavor);
    }
}
```

The code example shows a button and two panels. The button displays a colour chooser dialog and sets a colour for the first panel. The colour can be dragged into the second panel.

This example will enhance the previous one. We will add a drop target and a custom transferable object.

```
new MyDropTargetListener(right);
```

We register a drop target listener with the right panel.

```
event.startDrag(cursor, new TransferableColor(color));
```

The `startDrag()` method has two parameters. A cursor and a `Transferable` object.

```
public MyDropTargetListener(JPanel panel) {
    this.panel = panel;

    dropTarget = new DropTarget(panel, DnDConstants.ACTION_COPY,
        this, true, null);
}
```

In the MyDropTargetListener we create a drop target object.

```
Transferable tr = event.getTransferable();
Color color = (Color) tr.getTransferData(TransferableColor.colorFlavor);

if (event.isDataFlavorSupported(TransferableColor.colorFlavor)) {

    event.acceptDrop(DnDConstants.ACTION_COPY);
    this.panel.setBackground(color);
    event.dropComplete(true);
    return;
}
```

We get the data being transferred. In our case it is a colour object. Here we set the colour of the right panel.

```
event.rejectDrop();
```

If the conditions for a drag and drop operation are not fulfilled, we reject it.

```
protected static DataFlavor colorFlavor =
    new DataFlavor(Color.class, "A Color Object");
```

In the `TransferableColor`, we create a new `DataFlavor` object.

```
protected static DataFlavor[] supportedFlavors = {
    colorFlavor,
    DataFlavor.stringFlavor,
};
```

Here we specify, what data flavors we support. In our case it is a custom defined colour flavor and a predefined `DataFlavor.stringFlavor`.

```
public Object getTransferData(DataFlavor flavor)
     throws UnsupportedFlavorException
{
  if (flavor.equals(colorFlavor))
      return color;
  else if (flavor.equals(DataFlavor.stringFlavor))
      return color.toString();
  else
      throw new UnsupportedFlavorException(flavor);
}
```

Return an object for a specific data flavour.

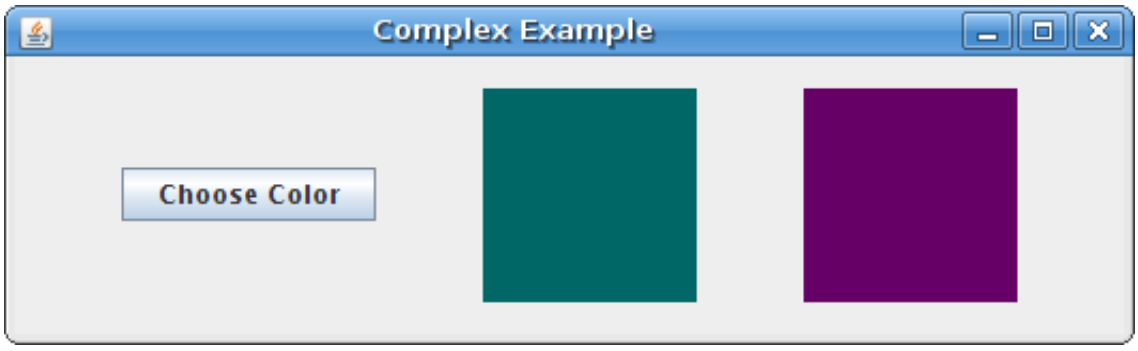

Figure: A complex example

This part of the Java Swing tutorial was dedicated to Swing drap and drop operations.