

KAPITEL 4

von Anja Austermann



Drag and Drop

»Drag and Drop« ist die Bezeichnung für den Datentransfer zwischen unterschiedlichen Anwendungen mit grafischer Benutzeroberfläche. Zur Zeit arbeitet Drag and Drop in Java nur unter Windows wirklich zufriedenstellend. Mit anderen Plattformen kann es zu Schwierigkeiten kommen, die aber mit den nächsten Releases beseitigt werden dürfen.

Das Prinzip von Drag and Drop beruht darauf, daß Dateien, beispielsweise vom Desktop oder aus dem Windows-Explorer einfach in eine aktive Anwendung gezogen werden können, die diese automatisch öffnet, wenn sie in der Lage ist, mit Daten des entsprechenden Typs umzugehen.

Die Anwendung, von der die Daten verschickt werden, wird als »Drag Source« bezeichnet, während die Anwendung, die die Daten empfängt, »Drop Target« genannt wird.

Im Rahmen dieses Kapitels werden wir ausschließlich Drag and Drop von Dateien aus dem Explorer oder vom Desktop in eine Java-Anwendung behandeln, der umgekehrte Weg oder der Datentransfer zwischen zwei Java-Anwendungen sowie die direkte Übertragung von Text ist aber auch möglich.

Genug der Vorrede – widmen wir uns jetzt der konkreten Realisierung von Drag and Drop in der Programmiersprache Java.

Alle Klassen, die für Drag and Drop notwendig sind, befinden sich im Package *java.awt.dnd*, das – wie das 2D-API oder Swing – zu den Java Foundation Classes (JFC) zählt.

Damit ein Java-Programm per Drag and Drop Daten lesen kann, muß das Interface *DropTargetListener* implementiert werden. Hierdurch wird es notwendig, diese Methoden zu implementieren, die in *DropTargetListener* deklariert sind:

- ◆ *public void dragEnter(DropTargetDragEvent evt)* – wird ausgelöst, wenn der Mauszeiger eine Datei über eine Komponente zieht.
- ◆ *public void dragOver(DropTargetDragEvent evt)* – wird ausgelöst, so lange der Mauszeiger über der Komponente verbleibt.
- ◆ *public void dropActionChanged(dropTargetDragEvent evt)* – wird ausgelöst, wenn der Benutzer ein anderes Objekt zum »dragen« auswählt.
- ◆ *public void dragExit(DropTargetDragEvent evt)* – wird ausgelöst, wenn der Mauszeiger den Bereich der Komponente wieder verläßt.

- ◆ *public void drop(DropTargetDragEvent evt)* – wird ausgelöst, wenn eine Datei in die Komponente verschoben und losgelassen wird. Normalerweise erfolgt hier die eigentliche Dateiverarbeitung via Drag and Drop.

Damit eine Komponente Dateien mittels Drag and Drop öffnen kann, ist es notwendig, zuerst ein Objekt vom Typ *DropTarget* zu erzeugen und dieses mit *setComponent(Component c)* der Komponente zuzuweisen, die später die Drop-Aktion bearbeiten soll. Also beispielsweise mit:

```
DropTarget dt = new DropTarget();
dt.setComponent(EineBeliebigeKomponente);
```

Außerdem muß dem *DropTarget*-Objekt ein *Listener* zugewiesen werden, der permanent überwacht, ob eine Datei in den Bereich der Komponente gezogen wurde. Hierbei ist es möglich, daß eine *TooManyListenersException* auftritt, und zwar dann, wenn mehr als ein *Listener*-Objekt gleichzeitig bei einer Komponente registriert werden sollen. Auch wenn dieser Fehler im größten Teil der Drag-and-Drop-Programme vom Programmierer umgangen werden kann, macht das Konzept des Exception Handlings von Java es notwendig, ihn mit Hilfe eines *try/catch*-Blocks folgendermaßen abzufangen:

```
try {
    dt.addDropTargetListener(this);
}
catch (TooManyListenersException e) {
    System.out.println("Es kann nur EIN Listener bei der " +
        "Komponente registriert werden!");
}
```

Außer diesen Schritten, die im Konstruktor des Drag-and-Drop-fähigen Objekts – also bei seiner Initialisierung – ausgeführt werden, muß die Funktion der im Interface *DropTargetListener* deklarierten Methoden spezifiziert werden. Um einfach nur eine Datei in einem Programm zu öffnen, so wie wir es tun möchten, brauchen Sie für die Funktionen *dragEnter*, *dragOver*, *acceptDrag*, *dropActionChanged* und *dragExit* keinen zusätzlichen Code zu schreiben – wichtig ist einzig und allein die *drop*-Methode. Hier erledigt Ihr Programm die Hauptarbeit.

Im ersten Arbeitsschritt wird überprüft, welche Art von *Drag*-Operation überhaupt beim *Source*-Objekt ausgeführt wurde. Die wichtigsten Möglichkeiten sind in der Klasse *DnDConstants* definiert und hier aufgelistet:

- ◆ *ACTION_COPY* – Kopieren eines Objekts in das DnD-Programm
- ◆ *ACTION_MOVE* – Verschieben eines Objekts
- ◆ *ACTION_REFERENCE* – Eine Verknüpfung mit dem Objekt erstellen
- ◆ *ACTION_COPY_OR_MOVE* – Verschieben oder Kopieren eines Objekts

Auf die unterschiedlichen Operationen kann Ihr Programm unterschiedlich reagieren, oder im Zweifelsfall eine unerwünschte Operation zurückweisen.

Um in Erfahrung zu bringen, welche Methode tatsächlich angewandt wird, muß das Ergebnis, das die Methode *getSourceActions()* zurückliefert, mit den oben beschriebenen Konstanten verglichen werden.

Die Methode *getSourceActions* befindet sich in der Klasse *DropTargetDropEvent* und liefert einen *Integer*-Wert zurück, wenn man sie auf den Parameter *evt* anwendet, der der Methode *drop* bei ihrem Aufruf übergeben wird. Diesen Wert können Sie wie folgt mit dem als Konstante festgelegten Operationstyp vergleichen:

```
if ((eventParameter.getSourceActions() &
    DnDConstants.ACTION_KONSTANTE) == 0) {
    // hier werden die entsprechenden Maßnahmen durchgeführt,
    // wenn die Operation nicht den gewünschten Typ hat -
    // Insbesondere erfolgt der Aufruf
    eventParameter.rejectDrop();
    // sonst geht es bei else weiter....
} else {
    Transferable trans = eventParameter.getTransferable();
    //...
}
```

Im *else*-Block ist bereits der nächste Schritt angedeutet. Um überhaupt mit dem Objekt arbeiten zu können, müssen sie ein *Transferable*-Objekt erzeugen, das als Referenz auf die entsprechende Datei dient. Mit *getTransferData()* können Sie später ein Objekt erzeugen, das genau die in der Datei gespeicherten Daten enthält.

Vorher müssen Sie allerdings prüfen, ob das Dateiformat der ausgewählten Dateien unterstützt wird. Dateiformate werden von Java als »Data Flavor« bezeichnet. Neben einigen Dateiformaten, die nur für den Datenaustausch zwischen zwei oder mehreren Java-Programmen zu gebrauchen sind, definiert Java in der Klasse *DataFlavor* die Formate *javaFileListFlavor* und *plainTextFlavor* als Konstanten. Letzteres

wird zum Kopieren von Text aus einer Applikation in eine andere ohne den Umweg über die Speicherung als Datei verwendet. Mit der ersten Möglichkeit – also *javaFileListFlavor* – werden wir uns im folgenden näher beschäftigen.

Um zu testen, welche der Dateien diesem Typ entsprechen, muß das Programm alle Dateien durchlaufen und auf Kompatibilität mit *javaFileListFlavor* testen. Dazu muß zuerst ein Array erstellt werden, in dem die Dateitypen aller Dateien als Instanzen von *DataFlavor* gespeichert sind. Die Methode *getCurrentDataFlavors()* aus der Klasse *DropTargetDropEvent* liefert ein solches Array zurück, sobald sie auf das *Event*-Objekt angewendet wird, das der *drop*-Methode als Parameter übergeben wurde. Um die Kompatibilität aller Dateien zu überprüfen, wird dieses Array durchlaufen und überprüft, ob die Bedingung

```
javaFileListFlavor.equals(ArrayElemente[i])
```

erfüllt ist. Ist das der Fall, kann jetzt zur tatsächlichen Verarbeitung der Datei übergegangen werden. Dazu ist es erforderlich, einen *Iterator* zu erzeugen, um alle Dateien der Reihe nach abzuarbeiten. Hier kommt wieder das *Transferable*-Objekt ins Spiel, das bereits vorher erzeugt wurde. Die darin enthaltenen Daten werden als Liste von Dateien interpretiert, aus der dann – wie im Kapitel über Collections beschrieben – ganz einfach mit Hilfe der *iterator()*-Methode ein *Iterator*-Objekt erzeugt werden kann.

In der Praxis sieht das so aus. Als Dateityp kann hier der Dateityp verwendet werden, der vorher durch Vergleich ermittelt wurde.:

```
Iterator i = ((List)
TransferableObject.getTransferData(Dateityp)).iterator();
```

Da der *Iterator* eine ganze Liste von Dateien enthält, kann er jetzt der Reihe nach abgearbeitet werden. Mit Hilfe der Methoden *hasNext()* und *next()*, die Sie noch aus dem Kapitel über *Collections* kennen, können Sie ein Element nach dem anderen in ein Objekt vom Typ *File* – der Repräsentation von Dateien in Java – umwandeln und auslesen, indem Sie die bekannten *Stream*-Klassen verwenden.

Das folgende Beispielprogramm wendet Drag and Drop an, um Textdateien, die beispielsweise aus dem Explorer oder vom Desktop in das Programmfenster gezogen werden, darzustellen. Das Programm basiert auf der Swing-Bibliothek. Nähere Informationen zur Verwendung der Swing-Klassen – das sind die Klassen, die mit einem »J« beginnen, wie *JScrollPane* oder *JTextArea* –, finden Sie im entsprechenden Buchkapitel über Swing.

Abb. 4.1:
Drag and Drop
in der Praxis:



```

import java.util.*;
import java.io.*;
import java.awt.*;
import javax.swing.*;
import java.awt.datatransfer.*;
import java.awt.dnd.*;
public class DnDApp extends JFrame implements ActionListener {
    public DnDApp() {
        target = new JTextArea(20, 60);
        getContentPane().add(target);
        DropTarget dt = new DropTarget();
        dt.setComponent(target);
        try {
            dt.addDropTargetListener(this);
        }
        catch(TooManyListenersException e) {
            System.out.println("Es kann nur EIN Listener bei " +
                "der Komponente registriert werden");
        }
    }
    public void dragEnter(DropTargetDragEvent dEvent) {}
    public void dragOver(DropTargetDragEvent dEvent) {}
    public void dragExit(DropTargetEvent dEvent) {}
    public void dropActionChaged(DropTargetDragEvent dEvent) {}

    public void drop(DropTargetDropEvent dEvent) {
        boolean ready = false;
        if ((dEvent.getSourceActions() &
            DnDConstants.ACTION_COPY) == 0)
    
```

```

    dEvent.rejectDrop();
else {
    dEvent.acceptDrop(DnDConstants.ACTION_COPY);
    Transferable trans = dEvent.getTransferable();
    DataFlavor[] currentFlavors = dEvent.getCurrentDataFlavors();
    DataFlavor selectedFlavor = null;
    for (int i = 0; i < currentFlavors.length; i++) {
        if (DataFlavor.javaFileListFlavor.equals(
            currentFlavors[i])) {
            selectedFlavor = currentFlavors[i];
            break;
        }
    }
    if (selectedFlavor != null) {
        try {
            Iterator dateien = ((java.util.List)
                trans.getTransferData(selectedFlavor)).iterator();
            while (dateien.hasNext()) {
                File datei = (File) dateien.next();
                target.append(datei.getPath());
                target.append("\n*****\n\n");
                if (datei.canRead() && datei.isFile()) {
                    BufferedReader in = new BufferedReader(
                        new InputStreamReader(
                            new FileInputStream (datei)));
                    String zeile= null;
                    while ((zeile = in.readLine()) != null) {
                        target.append(zeile);
                        target.append("\n");
                    }
                } else {
                    target.append("Datei nicht lesbar" +
                        " - bitte eine neue Datei angeben");
                }
                target.append("\n");
            }
            ready = true;
        }
        catch (Exception e) {
            System.out.println("Es ist ein Fehler aufgetreten: " + e);
        }
    }
}
}

private JTextArea target = null;
}

```

So sieht eine Beispielapplikation aus, in der ein Drag-and-Drop-fähiges Textfenster eingebunden ist:

```
import java. awt. *;  
import java. awt. event. *;  
import javax. swing. *;  
  
public class DndApp {  
    public static void main (String[] args) {  
        JFrame f = new JFrame("Drag and Drop Beispiel");  
        f. getContentPane(). setLayout(new BorderLayout());  
        f. setSize(600, 400);  
        f. addWindowListener (   
            new WindowAdapter() {  
                public void windowClosing(WindowEvent e) {  
                    System. exit(0);  
                }  
            }  
        );  
        f. setCursor(Cursor. getPredefinedCursor(Cursor. DEFAULT_CURSOR ));  
        f. getContentPane(). add(new DndText(), BorderLayout. CENTER);  
        f. setVisible(true);  
    }  
}
```