In this chapter:

- *What Is Drag and Drop?*
- *The Drop API*
- *The Drag Gesture API*
- *The Drag API*
- *Rearranging Trees*
- *Finishing Touches*

1

Drag and Drop

Until the Java 2 platform hit the streets, drag and drop support (specifically support for interacting with the native windowing system underneath the JVM) was lacking. The ability to let users drag a file from their file choosers into your application is almost a requirement of a modern, commercial user interface. The `java.awt.dnd` package gives you and your Java programs access to that support. Now you can create applications that accept information dropped in from an outside source. You can create Java programs that build up draggable information that you export to other applications. And of course, you can add both the drop and the drag capabilities to a single application to make its interface much more rich and intuitive.

"But wait!" you exclaim. "I recognize that package name. That's an AWT package!" You're right. Technically, drag and drop support is provided under the auspices of the AWT, not as part of Swing. However, one driving force behind Swing is to provide your application with a more mature, sophisticated user interface. Since drag and drop directly affects that type of interface, we figured you'd like to hear about it—even if it is not directly part of Swing. And just to try and help hide that fact, we'll be using Swing components in all of the examples. However, we should note for completeness that drag and drop support can be added just as easily to good old-fashioned AWT components—they just don't look as nice.

What Is Drag and Drop?

If you have ever used a graphical file system manager to move a file from one folder to another, you have used drag and drop (often abbreviated DnD). The term "drag and drop" refers to a GUI action whereby the end user "picks up" an object such as a file or piece of text by pressing the mouse button down on that object. Then without letting up on the button, the user "drags" it over some area of the screen and lets up on the mouse button to "drop" the object. This process is meant to extend the desktop metaphor. Just like your

real desktop, you can rearrange things by picking them up, moving them, and dropping them in a filing cabinet, a trash can, an in-box, a shoebox, or on the floor.

Some programmers have added drag and drop functionality to individual components. For example, you might want to have a graphically rearrangeable `JTree`. Even without the DnD package, you can accomplish that with a clever bit of programming and a good deal of time. With the DnD package, however, not only do you not need the clever bit of programming, but also you are not limited to one component. You can drag information from one component to another in the same application, in two different Java applications (using separate JVMs), or even between your Java application and the native windowing system. That's what we're here to tackle.

Programming with DnD

So how do we add DnD functionality to a component? On the surface, it's very easy. Drag and drop functionality uses the same technique as most other GUI features: events. There are drag start events, drag source events, and drop events. To play with these events, you implement the corresponding listener interfaces. This process should sound familiar to anyone who has set up event handlers for other GUI components. For example, to respond to a dropped object, we create an event handler that implements the `DropTargetListener` interface. *What you do* with the dropped object is the fun part.

Just to prove that this stuff really works, let's take a look at a simple example of the drop listener. Figure 1-1 shows a simple application that displays the names of any files you drag from your native windowing system's file manager.

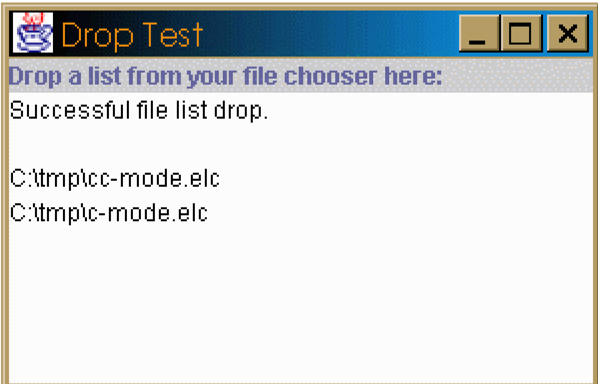

Figure 1-1. Two files dragged from a file manager onto our `DropTest` *application*

Admittedly, a screenshot does not do this test program justice. You really should compile and run `DropTest.java` to get the full effect. But trust us, it does work! We'll look at the source code for this example later in this chapter. First let's look at an overview of what's involved.

You have three main areas to work with—a drop target, a drag recognizer, and a drag source. A drop target accepts an incoming drag. The process of accepting the dragged information generates a series of events that you can respond to. The source of the dragged item might be your application, another Java application, or some native window system application like your file manager. The source doesn't matter to the drop target.

Your application can also be a source for draggable items. This functionality requires the other two parts of DnD, a drag recognizer and a drag source. The drag recognizer is really just a glorified event adapter that listens for events that indicate a drag has begun. In the simplest cases, that's usually a `mouseDragged()` event from the `MouseMotion-Listener` interface. (Later in this chapter you will see how you could make other events trigger a drag if you needed to.) Once a drag has been recognized, the drag source can start a drag and properly encapsulate the dragged information in an object that a drop target would recognize. Like the drop target, the information in a drag source can be used inside your own application, another Java application, or a native application. Of course, all of those examples need to know what to do with the information once it arrives, but there are no *a priori* restrictions on where you drop your data.

While you'll definitely want to get familiar with the `DropTarget` and `DragSource` classes themselves, the driving force behind the DnD package is the event handling. Events exist for each of the three major players: drop target events, recognizer events, and drag source events. You use recognizer events to start the whole process. Your response to target and source events dictates the behavior of your application. Let's look at a typical series of events in drag and drop scenarios (Table 1-1).

Table 1-1. Common DnD Event Methods

Event on System	**Gesture Recognizer Methods**	**Drag Source Methods**	**Drop Target Methods**
Click and drag an item such as a file	`dragGestureRecog-nized()`		
Drag item into a possible drop area		`dragEnter()`—called first `dragOver()`—called continuously until exit	`dragEnter()`—called first `dragOver()`—called continuouslsy until exit
Drag item out of a possible drop area		`dragExit()`	`dragExit()`
Drop the item on a drop target that accepts the drop		`dragExit()` `dragDropEnd()`1	`dragExit()` `drop()`
Drop the item on a drop target that rejects the drop		`dragExit()` `dragDropEnd()`1	`dragExit()` `drop()`
Drop the item on anything other than an active drop target		`dragExit()` `dragDropEnd()`1	

[1]*You can determine the success status of a drop from the event passed to* `dragDropEnd()`.

Throughout the entire process, you can determine whether the user wants to copy or move the information based on keyboard modifiers such as the Control key. You'll find the API for all of these events in the discussions below. Let's get started!

The Drop API

Our simple example, for which you will see the code later, uses several of the classes found in the `java.awt.dnd` package. Figure 1-2 shows how these classes fit together. Remember that we're looking at just the drop side of DnD. We'll explore the drag side in the next section, and autoscrolling at the end of this chapter.

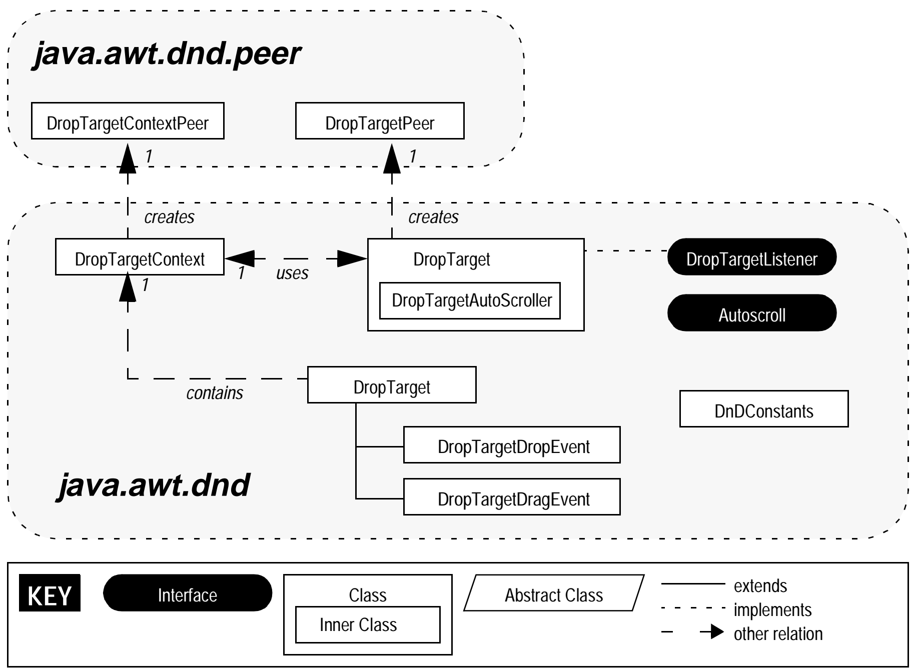

Figure 1-2. DnD class hierarchy for the Drop side

The DropTarget Class

An obvious starting point for playing with this API is the `DropTarget` class. This class encapsulates the functionality you add to regular GUI components so that they can respond to drop events. You can make just about any `Component` be a drop target. Typically, you pick the component that will respond to the drop—like the text area in our example—but you can also pick a proxy component that simply helps produce all the right events while your event handler plays with the dropped data.

Even though the DropTarget class implements the DropTargetListener interface, you can't assume that the target is already handling drop events for you. The DropTarget class implementation of the interface supports the internal mechanisms for responding to events; it's not for use by us as programmers. We still have to create our own listener.

Properties

The DropTarget class contains five properties governing the context of this target and its data transfer capabilities, as seen in Table 1-2.

Table 1-2. DropTarget *Properties*

Property	Data Type	get	is	set	Bound	Default Value
active	boolean		•	•		true
component	Component	•		•		null
defaultActions	int	•		•		ACTION_COPY_OR_MOVE
dropTargetContext	DropTarget-Context	•				null
flavorMap	FlavorMap	•		•		SystemFlavorMap.getDefaultFlavorMap()

The active property handles exactly what you would expect—the active state of this target. If active is false, this target will not accept any dropped information. The component property references the component that this target is using on screen. Notice that you can set this property, too. You could have one DropTarget that uses a variety of components during different stages of your application. This is possible because dropTargetContext keeps a reference to the native peer supporting the drop target, and that native peer only requires something on screen that can activate drop events.

The defaultActions and flavorMap properties determine the types of transfers (move, copy, link) and the types of data (file names, strings, Objects, etc.) you accept, respectively. The complete details of the FlavorMap class are beyond the intent of this chapter, but we'll look at the pertinent aspects later when we create our own data to drag.

Fortunately, it is often simple to figure out what "flavor" a dropped object uses. If you know more about the incoming types of data than Java does, you can supply your own flavor map, but usually the convenience methods supplied by the DataFlavor class are sufficient. You can easily find other Java classes, byte streams, and filename lists. For example, we'll be working with lists of files created by your windowing system's native file manager. You can use the isFlavorJavaFileListType() method to deter-

mine if the user dropped valid filename data on your application. Many similar methods exist for the primary types of information that your Java application can handle.

Events

A `DropTarget` produces `DropTargetEvent` objects as the user enters, moves over, exits, and eventually drops information on the target. Depending on the specific action—dragging actions such as enter and exit versus dropping—a subclass of `DropTargetEvent` will be used. In either case, you implement the `DropTargetListener` interface to respond to these events. The standard event handler methods are present:

public void addDropTargetListener(DropTargetListener l) throws TooManyListenersException
public void removeDropTargetListener(DropTargetListener l)

These methods allow you to add and remove one listener at a time. Attempting to add more than one listener results in a `TooManyListenersException`.

The `DropTarget` class itself implements the `DropTargetListener` interface (dicussed in detail below), and as such defines the following methods:

public void dragEnter(DropTargetDragEvent dtde)
public void dragExit(DropTargetEvent dte)
public void dragOver(DropTargetDragEvent dtde)
public void drop(DropTargetDropEvent dtde)
public void dropActionChanged(DropTargetDragEvent dtde)

These methods are all defined to trap drop events before the registered listener gets them. If the drop target is not active, these methods all return without notifying the listener. Otherwise, the registered listener is indeed notified, and the autoscroll mechanism is updated. (You can see more on autoscrolling in the "Autoscrolling" section at the end of this chapter.)

Note that this trapping behavior prevents the drop target from being its own listener. You'll receive an `IllegalArgumentException` if you try to add the target as its own listener.

Constructors

Several constructors are available for creating `DropTarget` objects. Depending on how much of the default information you want (or how much information you need to supply at runtime after the `DropTarget` is instantiated), you can pick an appropriate constructor. You can create drop targets right along with your components, or you can attach them at runtime. The latter might be useful if you have a dynamic interface and the visual component that accepts incoming drops can change. The available constructors are:

public DropTarget()

This constructor creates an "empty" drop target. It is currently an active target that supports copy or move operations. You still need to supply a component and a listener to make it useful.

public DropTarget(Component c, DropTargetListener dtl)
public DropTarget(Component c, int ops, DropTargetListener dtl)
public DropTarget(Component c, int ops, DropTargetListener dtl, boolean act)
public DropTarget(Component c, int ops, DropTargetListener dtl, boolean act, FlavorMap fm)

These methods supply various parts of a typical target. The arguments include the component associated with the drop target (`c`), the listener responding to drop events (`dtl`), the drop operations supported (`ops`; copy and move are both supported by default; see Table 1-3 later in this chapter for the constant values to use for this argument), whether or not the target is active (`act`, true by default), and a flavor map to use (`fm`, `null` by default).

Typically, you should find that supplying three main items—the component, a listener, and which operations you want to support—is sufficient to get started. If your drop target is active only in certain situations, starting out with a disabled target may be useful.

Peer Context Methods

To get the native support we need for DnD, we do have native peers. The `DropTargetContext` class handles a majority of the communication between the `DropTarget` we program and the native peer. The following methods are used to set up that context and make sure communication is established:

public void addNotify(java.awt.peer.ComponentPeer peer)
public void removeNotify(java.awt.peer.ComponentPeer peer)

These methods establish (or stop) communication between the drop target and the peer for the component it's currently using.

protected DropTargetContext createDropTargetContext()

This method creates a `DropTargetContext` object for this `DropTarget` to use. This is handled automatically. (The code example in "Drop Example" later in this chapter shows some cases where retrieving the context is useful.)

Autoscroll Methods

In dealing with drag and drop operations for larger applications, you often run into scenarios like this: you're dragging data into your word processing program but the paragraph you want to drop the data into is on the next page of the document. You could let go of the data, manually scroll your document down until the appropriate paragraph is visible, start a new drag with the data, and finally drop it in the correct spot. A better

application would allow you to scroll *while* you're dragging the data. That's precisely what autoscrolling does. Autoscrolling involves the process of dragging your data near the edge of your drop target and having the drop target start scrolling for you. In our word processing example, if you need to scroll down, you move to the bottom edge of the document, and the word processor should automatically move you to the next page. Depending on the application, scrolling up, left, or right might also be supported.

If your drop target component can be scrolled (such as a text area or a large list), you can use a `DropTargetAutoScroller` to manage the scrolling. You'll notice that all of the methods are protected. The autoscroller will be created for you automatically if your GUI component implements the `Autoscroll` interface that we'll discuss at the end of this chapter. That discussion also includes an example of creating a `JTree` that implements the `Autoscroll` interface.

Here are the `Autoscroll` methods:

protected DropTarget.DropTargetAutoScroller createDropTargetAutoScroller(Component c, Point p)
This method creates an autoscroller for the component `c` and initializes it using the point `p`. This method is used only for components that implement the `Autoscroll` interface.

protected void initializeAutoscrolling(Point p)
This method sets the current position of the cursor to point `p`. If no autoscroller exists when this method is called, one is created.

protected void updateAutoscroll(Point dragCursorLocn)
Similar to the `initializeAutoscrolling()` method, this method updates the current position of the cursor to `dragCursorLocn`. However, this method will simply return if no autoscroller exists.

protected void clearAutoscroll()
This method stops any scrolling that might be occurring and removes the current autoscroller. After calling `clearAutoscroll()`, you need to call `initializeAutoScrolling()` to restart any scrolling on your component.

Inner Classes

The `DropTarget` class does include one inner class to aid in the autoscrolling process: `DropTargetAutoscroller`. That class is discussed in more detail in the section on autoscrolling later in this chapter.

The DnDConstants Class

This quick little helper class defines the different types of actions (move, copy, link) that are supported by the DnD framework. Notice that we say "DnD framework" and not just

"DnD." The Java 2 API is ready for all of these actions. But your application or windowing system may not be. For example, while copy and move operations are universally understood, the link operation is not. You still might find that operation useful when dragging and dropping within a single application. Table 1-3 lists the constants defined by this class.

Table 1-3. `DnDConstants` *Constants*

Constant	Type	Description
`ACTION_COPY`	`int`	Supports copying of dragged item(s).
`ACTION_COPY_OR_MOVE`	`int`	Supports copying or moving of dragged item(s).
`ACTION_LINK` `ACTION_REFERENCE`	`int`	Supports referencing dragged item(s). What "reference" means depends on your application and the native platform. No direct mouse or keyboard action starts a link operation in the current implementation.
`ACTION_MOVE`	`int`	Supports moving dragged item(s).
`ACTION_NONE`	`int`	Does not support anything (i.e., ignore this action).

The DropTargetContext Class

As mentioned above, supporting a `DropTarget` with all the useful information about the drop is a `DropTargetContext`. This class provides you access to the DnD peer. Most of the methods update the native peer to keep the drag and drop state of your windowing system consistent.

For programmers, this class represents the primary location for interesting information when handling a drop or drag event. You can retrieve an instance of the context from any drop event and use it to provide user feedback. You can also use it to retrieve the drop target itself if you need it.

Properties

Reflecting the fact that this class serves primarily as peer support, many of its properties are protected as shown in Table 1-4.

Table 1-4. `DropTargetContext` *Properties*

Property	Data Type	get	is	set	Bound	Default Value
`component`	`Component`	•				from `dropTarget` (via the constructor)
`currentDataFlavors`[1]	`DataFlavor[]`	•				non-null empty array
`currentDataFlavorsAsList`[1]	`List`	•				non-null empty list

Table 1-4. `DropTargetContext` *Properties (Continued)*

Property	Data Type	get	is	set	Bound	Default Value
`dropTarget`	`DropTarget`	•				from constructor
`targetActions`[1]	`int`	•		•		from peer
`transferable`[1]	`Transferable`	•				`new TransferableProxy()`

[1]*The get and set methods for these properties are protected.*

The `component` and `dropTarget` properties are both publicly accessible, and they simply return references to the `DropTarget` object (and its `Component`) associated with this context. The `currentDataFlavors` and `currentDataFlavorsAsList` properties return the data flavors supported currently. This feature can be useful when trying to update a drag cursor. (For example, if the drop target cannot accept any of the current flavors, a "no drop" cursor could be displayed.) Which of these two properties you access depends entirely on your preference for using either an array of objects or a `List`. The `targetActions` property indicates the supported operations for the drop target. These actions can be any of the values from the `DnDConstants` class shown in Table 1-3. The `transferable` property is a bundled reference to the data being transferred via the drag and drop operation.

Constructors

The `DropTargetContext` class has no public or protected constructors. A package private constructor does exist, and this constructor is used when you call the `getDropTargetContext()` method from the `DropTarget` class.

Drag and Drop Qualification Methods

During a drag and drop operation, the events that get generated have an active reference to the drop target context. Through those events, a programmer can accept or reject drags and drops. Those event methods in turn pass those notifications on to the context through the following methods (you rarely call these methods in a `DropTargetContext` object directly):

public void dropComplete(boolean success) throws InvalidDnDOperationException

The only public method in the bunch, this method can be used to indicate that a drop is finished. You use the `success` argument to show a successful (`true`) drop or a failed (`false`) drop. In practice, you probably won't worry about this class since the event classes give you front-line access to most of the real action.

protected void acceptDrag(int dragOperation)
protected void acceptDrop(int dropOperation)

These methods allow a drag or drop for the specified `dropOperation`.

protected void rejectDrag()
protected void rejectDrop()

These methods disallow a drag or drop, regardless of the type of operation.

Transfer Data Methods

If you have subclassed the `DropTargetContext` class, you can access information about the data being transferred with these methods:

protected boolean isDataFlavorSupported(DataFlavor df)

As its name suggests, this method tests the given data flavor `df` and returns true if the drop target supports it and false otherwise.

protected Transferable createTransferableProxy(Transferable t, boolean local)

This method creates a special Transferable object which can help you determine if the dragged data came from the same VM as the drop target. This method returns an instance of the `TransferableProxy` inner class, discussed below.

Peer Methods

Since we do deal with the underlying windowing system during the drag and drop process, native peers are required. Similar to the AWT components, there is a `java.awt.dnd.peer` package devoted to the peer classes for the various DnD contexts. These methods are:

public void addNotify(DropTargetContextPeer dtcp)
public void removeNotify()

As with AWT components, these methods exist to associate (and disassociate) this `DropTargetContext` with a peer object.

Inner Classes

The primary purpose of this class is to wrap normal `Transferable` objects and provide efficiencies where possible when dealing with local objects (objects that were dragged and dropped in the same VM). However, access to the class for programmers is restricted to subclasses of the `DropTargetContext` class. You would only subclass `DropTargetContext` if you needed to remap the behavior of the native windowing system.

protected class DropTargetContext.TransferableProxy

This class defines the type of object returned by the `createTransferableProxy()` method above.

The DropTargetListener Interface

If you want to respond to drop events, you need to implement this interface and then register with your `DropTarget` object. You get only one listener per drop target. Unlike many other event handlers, you do have some hidden obligations in the implementation of the interface methods. Specifically, you need to accept or reject the operation generating the events. This acceptance or rejection keeps the visual clues—typically the appearance of the mouse cursor—in sync with your application logic. We'll look at an example of this shortly.

Methods

The `DropTargetListener` methods break a drop event into five categories. You might find these methods similar to `MouseListener` and `MouseMotionListener` methods, and they are. They describe the key moments in a drop process. Notice however, that these methods use a variety of the `DropTargetEvent` class and its subclasses as arguments. `DropTargetEvent` is described in more detail in the next section.

public void dragEnter(DropTargetDragEvent dtde)

This method is called when the user drags an item over an active drop target. If you implement this method, at some point in the code you should indicate whether or not you would accept a drop using the `acceptDrag()` or `rejectDrag()` method of `dtde`.

public void dragExit(DropTargetEvent dte)

This method is called when the user drags an item out of an active drop target.

public void dragOver(DropTargetDragEvent dtde)

This method is called continuously while the user is over (i.e., inside the boundaries of) an active drop target. If you implement this method, at some point in the code you should indicate whether or not you would accept a drop using the `acceptDrag()` or `rejectDrag()` method of `dtde`.

public void drop(DropTargetDropEvent dtde)

This method is called when the user drops a dragged item on an active drop target. All of the hard work involved in accepting a drop starts here. As with the `dragEnter()` and `dragOver()` methods, you can initially accept or reject the drop, but eventually you should terminate the drop process with a call to the `dropComplete()` method. You'll see several examples of this method throughout the rest of this chapter.

public void dropActionChanged(DropTargetDragEvent dtde)

This method is called when the user changes the "type" of the drop. The most common change is from copy to move (or vice versa).

The DropTargetEvent Classes

As you can see, the event handling methods for a drop listener receive a DropTargetEvent. The DropTargetEvent class serves as the basis for the more specific events that get passed to drop() and various drag methods. It provides access to the DropTargetContext through its only property, shown in Table 1-5.

Table 1-5. DropTargetEvent *Properties*

Property	Data Type	get	is	set	Bound	Default Value
dropTargetContext	DropTarget-Context	•				from constructor

As the property name implies, it gives you access to the DropTargetContext associated with the target that generated the event. You can use the context to get at the GUI component associated with the event. (The getSource() method returns the DropTarget object, not the component.)

Depending on whether you pick up drag events or the final drop event, you get one of two DropTargetEvent subclasses: DropTargetDragEvent and DropTargetDropEvent. Both contain more or less the same information, but drop events give you the capacity to retrieve the transferred data.

The DropTargetDragEvent Class

As the user drags a piece of information over your drop target, several drag events are generated. You can monitor these events and use them to dynamically control the cursor or the drop target. For instance, you have probably seen a cursor change to a big "not here" symbol over invalid drop targets.

Properties

The properties of a DropTargetDragEvent object should give you all the information you need to make such changes. These properties are listed in Table 1-6.

The currentDataFlavors and currentDataFlavorsAsList properties give you access to the properties of the same name in the DropTargetContext class. The location property gives you the coordinates of the mouse pointer for this particular event. The dropAction property indicates which operation (move, copy, link) the user intends to perform, while the sourceActions property indicates which operations the source of the data will support. If you find an incompatibility, you could reject the drag event using one of the drag methods listed below.

Table 1-6. DropTargetDragEvent *Properties*

Property	Data Type	get	is	set	Bound	Default Value
currentDataFlavors	DataFlavor[]	•				from dropTarget-Context (via the constructor)
currentDataFlavorsAsList	List	•				from dropTarget-Context (via the constructor)
dropAction	int	•				ACTION_NONE
location	Point	•				new Point(0,0)
sourceActions	int	•				ACTION_NONE

Constructors

The DropTargetDragEvent class does have a public constructor:

public DropTargetDragEvent(DropTargetContext dtc, Point cursorLocn, int dropAction, int srcActions)

This builds an event and uses the arguments to fill out the dropTargetContext, location, dropAction, and sourceActions properties respectively.

Drag Methods

Once you have had a chance to look at the event, you can decide whether or not to accept it. The following methods facilitate announcing that decision. As you accept() or reject() the drag, the cursor should follow suit to give the user proper visual feedback.

public void acceptDrag(int dragOperation)

This method indicates that you will accept the drag. The dragOperation argument dictates which operation is acceptable. If you want to accept both copy and move operations, you need to figure out which operation is currently underway and accept that operation. While you can syntactically accept an ACTION_COPY_OR_MOVE operation, this does not succeed in reality. The example in the section "Finishing Touches" later in this chapter shows how to accept both kinds of operations.

public void rejectDrag()

This method indicates that you will not accept the drag.

Transfer Data Methods

The `DropTargetDragEvent` class also carries some information about the data wrapped up in the event. The method for accessing this informatin is:

public boolean isDataFlavorSupported(DataFlavor df)

Similar to the `currentDataFlavors` property, this method gives you access to the method of the same name in the `DropTargetContext`. It returns `true` if `df` is a flavor that the drop target will accept.

The DropTargetDropEvent Class

This class is very similar to the `DropTargetDragEvent` class, but now that we have a real drop, we can gain access to the information that the user was dragging, not just the meta-information. We can also use the `DropTargetDropEvent` class to signal the original drag source of a successful (or failed) drop.

Properties

Similar to the drag version, the properties for `DropTargetDropEvent` shown in Table 1-7 give you access to just about everything that's useful.

Table 1-7. `DropTargetDropEvent` *Properties*

Property	Data Type	get	is	set	Bound	Default Value
`currentDataFlavors`	`DataFlavor[]`	•				non-null empty array
`currentDataFlavorsAsList`	`List`	•				non-null empty list
`dropAction`	`int`	•				`ACTION_NONE`
`localTransfer`	`boolean`		•			`false`
`location`	`Point`	•				`new Point(0,0)`
`sourceActions`	`int`	•				`ACTION_NONE`
`transferable`	`Transferable`	•				from `dropTarget-Context` (via the constructor)

The `currentDataFlavors`, `currentDataFlavorsAsList`, `dropAction`, `location`, and `sourceActions` properties all mimic their respective counterparts in the `DropTargetDragEvent` class. There are also two new properties, `local-Transfer` and `transferable`. The `localTransfer` property tells you whether or not the data being transferred came from the same virtual machine. You could use this information to process the transfer more efficiently. The `transferable` property is a `Transferable` object that represents the actual transferred data. Again, `Transfer-`

able comes from the java.awt.datatransfer package, but we'll see some examples of extracting the data from a Transferable object in the code shortly.

Constructors

We have two public constructors for DropTargetDropEvents:

public DropTargetDropEvent(DropTargetContext dtc, Point cursorLocn, int dropAction, int srcActions)

public DropTargetDropEvent(DropTargetContext dtc, Point cursorLocn, int dropAction, int srcActions, boolean isLocal)

Both of these constructors both fill out the dropTargetContext, location, dropAction, and sourceActions properties. The second constructor also lets you determine whether the drop is local (from the same VM) or not. The first constructor leaves the localProperty transfer at the default of false.

Drop Methods

Similar to the drag events, you can accept or reject drops. The methods are:

public void acceptDrop(int dropAction)

This method will accept a drop of type dropAction. If you decide to accept the drop, call this method, process the drop, and then call the dropComplete() method below.

public void rejectDrop()

This method rejects the drop.

public void dropComplete(boolean success)

This method tells the source of the drag that the drop was completed. The success argument should be true if the drop was successful, false otherwise.

Transfer Data Methods

Also like the drag events, you can check for specific data flavor support using this method:

public boolean isDataFlavorSupported(DataFlavor df)

This method gives you access to the method of the same name in the DropTargetContext. It returns true if df is a flavor that the drop target will accept.

Drop Example

Well, after all that theory, here's the source code from the file list drop application shown earlier in Figure 1-1. This example simply generates status messages for most of the

methods, but it does correctly process the drop event in the `drop()` method. Later examples will put more of the listener methods to use.

```
/*
 * DropTest.java
 * A simple drop tester application.
 */

import java.awt.*;
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;

public class DropTest extends JFrame implements DropTargetListener {

  DropTarget dt;
  JTextArea ta;

  public DropTest() {
    super("Drop Test");
    setSize(300,300);
    addWindowListener(new BasicWindowMonitor());

    // Make a quick label for instructions and create the
    // text area component.
    getContentPane().add(
        new JLabel("Drop a list from your file chooser here:"),
        BorderLayout.NORTH);
    ta = new JTextArea();
    ta.setBackground(Color.white);
    getContentPane().add(ta, BorderLayout.CENTER);

    // Set up our text area to receive drops...
    // This class will handle drop events.
    dt = new DropTarget(ta, this);
    setVisible(true);
  }

  // For now, we'll just report the "ancilliary" events to the console.
  // These include dragEnter, dragExit, dragOver, and dropActionChanged.
  public void dragEnter(DropTargetDragEvent dtde) {
    System.out.println("Drag Enter");
  }

  public void dragExit(DropTargetEvent dte) {
    System.out.println("Drag Exit");
  }

  public void dragOver(DropTargetDragEvent dtde) {
    System.out.println("Drag Over");
  }
```

```
  public void dropActionChanged(DropTargetDragEvent dtde) {
    System.out.println("Drop Action Changed");
  }

  public void drop(DropTargetDropEvent dtde) {
    try {
      // Ok, get the dropped object and try to figure out what it is.
      Transferable tr = dtde.getTransferable();
      DataFlavor[] flavors = tr.getTransferDataFlavors();
      for (int i = 0; i < flavors.length; i++) {
        System.out.println("Possible flavor: " + flavors[i].getMimeType());
        // Check for file lists specifically.
        if (flavors[i].isFlavorJavaFileListType()) {
          // Great!  Accept copy drops...
          dtde.acceptDrop(DnDConstants.ACTION_COPY);
          ta.setText("Successful file list drop.\n\n");

          // And add the list of file names to our text area
          java.util.List list =
              (java.util.List)tr.getTransferData(flavors[i]);
          for (int j = 0; j < list.size(); j++) {
            ta.append(list.get(j) + "\n");
          }

          // If we made it this far, everything worked.
          dtde.dropComplete(true);
          return;
        }
      }
      // Hmm, the user must not have dropped a file list.
      System.out.println("Drop failed: " + dtde);
      dtde.rejectDrop();
    } catch (Exception e) {
      e.printStackTrace();
      dtde.rejectDrop();
    }
  }

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

The interesting code for this application falls into two methods: the constructor and the `drop()` method. In the constructor, we build a simple application with a label and a text area. To create a drop target and associate it with the text area takes only one line:

```
dt = new DropTarget(ta, this);
```

Now we have a drop target; its associated component is the text area `ta`. The `this` argument is the target's `DropTargetListener`. That simply means that our `DropTest` application will be handling the events that `dt` generates.

The `drop()` method deciphers the dropped information in a series of steps:

1. It checks the MIME type of the transfer to make sure it's acceptable.
2. If the type is acceptable, the target accepts the drop.
3. The target retrieves the data from the `Transferable` object.
4. Finally, the target marks the drop complete.

In our example, the first step is handled using a convenience method from the `DataFlavor` class (more on data flavors later). We're looking for a list of files specifically; we can test for a list of files by calling `isFlavorJavaFileTypeList()`, a method of the `DataFlavor` class. If we find that we have a file list, we proceed to the second step and call `acceptDrop()`. Pulling the data out of the `Transferable` object is straightforward; we call `getTransferableData()` on the `Transferable` object, with the data flavor as an argument. We still must cast the result to the appropriate type (in this case, `java.util.List`) before we can do anything with it. We'll see examples of other types of transferable data in the next section. Finally, if nothing went wrong, we mark the drop a success by calling `dropComplete(true)`. If we didn't find a file list or if any error occurred trying to retrieve the data, we call `rejectDrop()`.

Transferable Contents

The DnD API makes use of the foundations laid by the system clipboard support in the `java.awt.datatransfer` package. We won't go into the details of data transfer here, but we will take a short tangent to discuss some of the valid data flavors. The data flavor is the key to successfully retrieving the transfered data.

A data flavor describes a possible means for interpreting a piece of information. We use these flavors to "tag" data as a particular type of information. Other applications read the tag to see if they can understand and use the data. Normally, this tag is associated with a clipboard, but the same idea applies with information the user drags from one application to another. DnD is really just a quick, mouse version of copy and paste.

While you can certainly create your own flavors—we will do this later—the `DataFlavor` class predefines a few useful flavors: `plainTextFlavor`, `stringFlavor`, and `javaFileListFlavor`. The plain text flavor is quite universal. Almost every application out there can read plain text. The problem, of course, is that some objects are quite difficult to represent in plain text. Still, good old-fashioned text makes up a large percentage of the data that users transfer, so this flavor will definitely come in handy. If you are transferring text from one Java application to another Java application (whether or not they use the same VM), you will probably use `stringFlavor`. The string flavor has a mime type of `text/plain`, but its representation class is `String`, so you don't have to worry about converting the plain text to a `String` with the correct character set.

The other pre-defined flavor, `javaFileListFlavor`, comes in handy for another extremely common task: accepting a list of files generated in a file chooser application. Most windowing systems these days have a graphical file manager. From that manager, you can select several files and drag them to another application. This flavor represents a *list of the names* of those selected files. What your application does with the names (or with the files themselves) is up to you.

Let's expand our example to handle more than just a list of files. Here's an improved version of the `drop()` method that also handles serialized objects and input streams:

```
public void drop(DropTargetDropEvent dtde) {
    try {
      // Ok, get the dropped object and try to figure out what it is.
      Transferable tr = dtde.getTransferable();
      DataFlavor[] flavors = tr.getTransferDataFlavors();
      for (int i = 0; i < flavors.length; i++) {
        System.out.println("Possible flavor: " + flavors[i].getMimeType());
        // Check for file lists specifically.
        if (flavors[i].isFlavorJavaFileListType()) {
          // Great!  Accept copy drops...
          dtde.acceptDrop(DnDConstants.ACTION_COPY);
          ta.setText("Successful file list drop.\n\n");

          // And add the list of file names to our text area.
          java.util.List list =
              (java.util.List)tr.getTransferData(flavors[i]);
          for (int j = 0; j < list.size(); j++) {
            ta.append(list.get(j) + "\n");
          }

          // If we made it this far, everything worked.
          dtde.dropComplete(true);
          return;
        }
        // Ok, is it another Java object?
        else if (flavors[i].isFlavorSerializedObjectType()) {
          dtde.acceptDrop(DnDConstants.ACTION_COPY_OR_MOVE);
          ta.setText("Successful Object drop.\n\n");
          Object o = tr.getTransferData(flavors[i]);
          // Normally we would try to cast o as something useful.
          // For now, we just want to print out a success message.
          ta.append("Object: " + o);
          dtde.dropComplete(true);
          return;
        }
        // How about an input stream?
        else if (flavors[i].isRepresentationClassInputStream()) {
          dtde.acceptDrop(DnDConstants.ACTION_COPY_OR_MOVE);
          ta.setText("Successful stream drop.\n\n");
          ta.read(new InputStreamReader(
                  (InputStream)tr.getTransferData(flavors[i])),
                  "from a drop");
          dtde.dropComplete(true);
```

```
          return;
        }
      }
      System.out.println("Drop failed: " + dtde);
      dtde.rejectDrop();
    } catch (Exception e) {
      e.printStackTrace();
      dtde.rejectDrop();
    }
}
```

In this expanded `drop()` method, we check for two other common data transfer types: a Java `Object` and an `InputStream`. If it is a Java `Object`, we can simply grab the transfer data directly using `getTransferData()`. Presumably, your application knows what types of objects the user might be dragging in and can use `instanceof` tests to determine what it received.

If it's not an `Object`, we check to see if it can be represented by a stream. If so, we open an `InputStreamReader` and grab the data that way. Text built outside of another Java application could be transferred this way. Again, the receiving application needs to know what to do with the data in that stream.

We will go through an example of creating your own `Transferable` type when we explore the rearrangable `JTree` later in this chapter.

The Drag Gesture API

Now that the drop side is working, how do we make the drag side go? That's a good question, and we're about to answer it. First, we need a bit of background information. To successfully accomplish a drag in Java, we first must recognize what constitutes a "drag gesture." A drag gesture is an action the user takes that indicates he or she is starting a drag. Typically that would be a mouse drag event, but it is not hard to imagine other gestures that could be used. For example, a voice-activated system might listen for the words "pick up" or something similar.

The API for drag gestures is fairly simple. Three classes and one interface make up the core; Figure 1-3 shows the relationships between these classes.

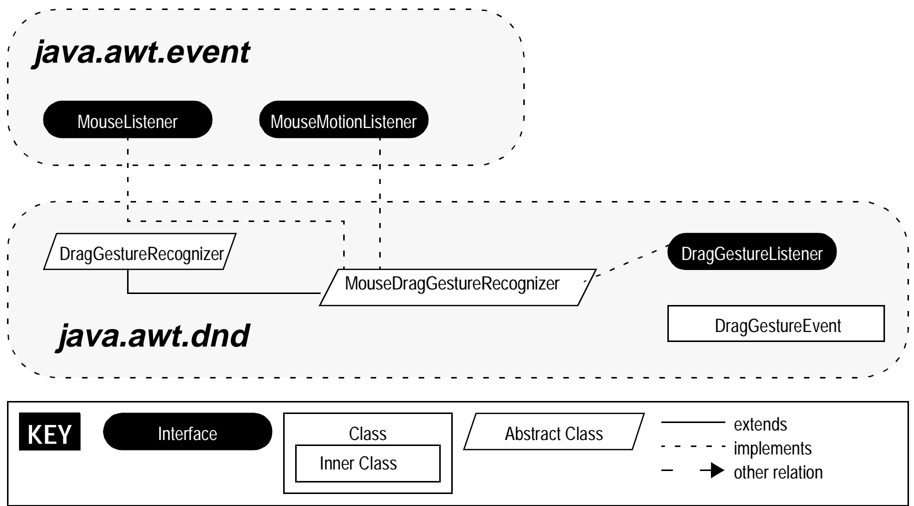

Figure 1-3. Drag Recognition class hierarchy

The DragGestureRecognizer Class

At the heart of this API is the `DragGestureRecognizer` class. The recognizer is responsible, quite simply, for recognizing a sequence of events that means "start dragging." This could be quite different for different platforms, so the `DragGestureRecognizer` class is abstract and must be subclassed to create an appropriate implementation. (The `MouseDragGestureRecognizer` in the next section takes a few more steps towards completing the class and makes assumptions about using a mouse as the trigger for drag events. You can use the toolkit on your system to retrieve a concrete version.) This process sounds more complex than it is. The idea is that you attach a drag gesture recognizer to a component with code similar to this:

```
JLabel jl = new JLabel("Drag Me!");
DragSource ds = new DragSource();
DragGestureRecognizer dgr = ds.createDefaultDragGestureRecognizer(
                            jl, DnDConstants.ACTION_COPY, this);
```

Now when you click down on the label `jl` and start dragging away, a drag gesture is recognized. The system then reports that recognition to your application through a registered `DragGestureListener`. You will see a completed example later in the chapter.

Properties

The DragGestureRecognizer keeps several pieces of information about a drag operation around as properties shown in Table 1-8.

Table 1-8. DragGestureRecognizer *Properties*

Property	Data Type	get	is	set	Bound	Default Value
component	Component	•		•		null
dragSource	DragSource	•				passed through constructor
sourceActions	int	•		•		ACTION_NONE
triggerEvent	InputEvent	•				null

The component property refers to the component registered as the source for possible drag initiations. The dragSource property represents a DragSource object that wraps the component generating drags much the same way a DropTarget object wraps components receiving drops. The sourceActions property is the set of acceptable drag operations (coming from DnDConstants) for the drag source. The trigger-Event property returns the first event from the list of events that led to this recognizer declaring a drag was indeed started. If no drag is currently recognized, this property will be null.

Events

As mentioned before, the DragGestureRecognizer reports its events to a Drag-GestureListener and sends a DragGestureEvent. The usual add and remove methods for the listener are available and include:

public void addDragGestureListener(DragGestureListener dgl) throws TooManyListenersException
public void removeDragGestureListener(DragGestureListener dgl)

These methods supply the usual support for registering and unregistering listeners with the recognizer. Note that this is a unicast event; only one listener is allowed.

protected abstract void registerListeners()
protected abstract void unregisterListeners()

These protected methods register (and unregister) the appropriate trigger event listeners with the component associated with this recognizer. For example, if a drag starts with the user dragging the mouse, these methods would register the recognizer with the component as a MouseMotionListener. Which listeners a subclass registers to receive depends entirely on which events can trigger a drag.

protected void fireDragGestureRecognized(int dragAction, Point p)

This method creates a `DragGestureEvent` and notifies the registered listener (if any) that a drag gesture has been recognized. The `dragAction` argument will be one of the `DnDConstants` actions. The location of the mouse when the trigger event occurred is contained in `p`.

Fields

`DragGestureRecognizer` defines the following fields:

protected DragSource dragSource
protected Component component
protected int sourceActions

These fields support the properties of the same names.

protected DragGestureListener dragGestureListener

This field stores the active listener registered with the recognizer.

protected ArrayList events

This field stores the list of events that triggered the recognition of a drag gesture. The `triggerEvent` property corresponds to the first element in this list. This field can be manipulated through both the `appendEvent()` and `clearRecognizer()` methods below.

Constructors

All of the constructors for `DragGestureRecognizer` are protected. Since the class is abstract, a subclass would be needed to create an instance of `DragGestureRecognizer` anyway. In real programs, you would use the `DragSource.createDefaultDragGestureRecognizer()` method or the `Toolkit.createDragGestureRecognizer()` method to build a recognizer. The constructors are:

protected DragGestureRecognizer(DragSource ds)
protected DragGestureRecognizer(DragSource ds, Component c)
protected DragGestureRecognizer(DragSource ds, Component c, int sa)
protected DragGestureRecognizer(DragSource ds, Component c, int sa, DragGestureListener dgl)

These methods create `DragGestureRecognizer` objects and use the supplied arguments to override the default property values for `dragSource`, `component`, and `sourceActions`, respectively. If supplied, `dgl` will be used as the listener for any recognition events.

Trigger Event List Methods

Since it is possible that a sequence of events (such as pressing the Tab key and then dragging the mouse) will be responsible for causing a drag, the following methods exist to help set up the list of trigger events correctly:

protected void appendEvent(InputEvent awtie)
This method allows individual events to be stored in the recognizer. The list can be cleared using the `resetRecognizer()` method below.

public void resetRecognizer()
This method clears any events in the trigger event list. If a drag is currently in process, the drag is effectively cancelled.

The MouseDragGestureRecognizer Class

For many applications, we use the mouse drag as the drag gesture. The `MouseDragGestureRecognizer` class implements the appropriate recognizer functions for us. Note that this class is still abstract. Individual platforms must subclass this class and fill out the specifics of what constitutes a valid drag initiation. However, this class does make it much easier to create your own recognizer, rather than starting from the `DragGestureRecognizer` class directly.

As you start out playing with DnD functionality, you will most likely rely on the prebuilt recognizers you can retrieve from the `DragSource` class. We'll tackle `DragSource` in the next section.

Constructors

The `MouseDragGestureRecognizer` class contains the same set of constructors found in its parent class:

protected MouseDragGestureRecognizer(DragSource ds)
protected MouseDragGestureRecognizer(DragSource ds, Component c)
protected MouseDragGestureRecognizer(DragSource ds, Component c, int act)
protected MouseDragGestureRecognizer(DragSource ds, Component c, int act, DragGestureListener dgl)
All four constructors perform the same initialization functions as those described for the `DragGestureRecognizer` class.

Mouse Methods

As this class recognizes gestures started by mouse events, it does implement both the `MouseListener` and `MouseMotionListener` interfaces. The following methods come from these interfaces:

public void mouseClicked(MouseEvent e)
public void mouseDragged(MouseEvent e)
public void mouseEntered(MouseEvent e)
public void mouseExited(MouseEvent e)
public void mouseMoved(MouseEvent e)
public void mousePressed(MouseEvent e)
public void mouseReleased(MouseEvent e)

In the `MouseDragGestureRecognizer` class, these methods are all empty. Subclasses must override the methods to provide appropriate functionality.

Listener Methods

This class implements the abstract methods inherited from `DragGestureListener`:

protected void registerListeners()
protected void unregisterListeners()

These methods register (and unregister) this recognizer as a `MouseListener` and a `MouseMotionListener` with its component.

The Drag Gesture Events and Listeners

The `DragGestureListener` interface provides one method to indicate a drag gesture has been recognized:

public void dragGestureRecognized(DragGestureEvent dge)

The `DragGestureEvent` provides you with everything you need to know about the drag the user has initiated. If you like everything you see, you can start the drag process: get the information to be dragged (entirely up to you), package it as a `Transferable`, and pass it to the event's `startDrag()` method. Or, if something is wrong (the wrong action, the wrong object to drag, the application is not in a draggable mode, etc.) you can ignore the gesture by returning without doing anything.

Now that we know how to write a `DragGestureListener`, let's talk about the events that we send to those listeners. Not surprisingly, they're called `DragGestureEvents`.

Properties

Most of the properties stored in a `DragGestureEvent` provide convenient access to similar properties in the `DragGestureRecognizer` that generated the event. These are listed in Table 1-9.

The `dragOrigin` property corresponds to the location of the mouse when the gesture was recognized. The `sourceAsDragGestureRecognizer` property is merely a convenience property allowing you to access the source of the event as a `Drag-`

Table 1-9. `DragGestureEvent` *Properties*

Property	Data Type	get	is	set	Bound	Default Value
component	Component	•				null
dragAction	int	•				ACTION_NONE
dragOrigin	Point	•				null
dragSource	DragSource	•				null
sourceAsDragGesture-Recognizer	DragGesture-Recognizer	•				null
triggerEvent	InputEvent	•				from dragSource (during a valid recognition)

GestureRecognizer rather than as an Object, which is what you get from the source property inherited from the EventObject class.

Event List Methods

In discussing the DragGestureRecognizer fields, we noted that several events could be sequenced together to create a drag. The entire list of those events can be retrieved by using the methods in this group:

public Iterator iterator()

This method returns an iterator for the ArrayList storing the events. If you've worked with the Collections APIs, this method may be more to your liking.

public Object[] toArray()
public Object[] toArray(Object[] array)

These methods return the list of events in an Object array. The first method creates a new array while the second method fills the supplied array.

Drag Initiation Methods

Probably the whole reason we even worry about DragGestureEvent objects is that they have the mechanism for starting a drag operation. The methods below are called to get a drag underway; you usually call them as part of your dragGestureRecognized() method. In other words, the DragGestureRecognizer is intercepting various events and watching them to see if anything looks like the beginning of a drag. When it sees a drag, it sends you a DragGestureEvent, packaging up any information it has. You can then look at the event, and decide whether or not it is the beginning of a drag. If it is, you package up the data to be dragged, pick a cursor to indicate that a successful drag start has occurred, and call one of the startDrag() methods. (You can pick one of the cursors predefined for you in the DragSource class. See Table 1-11 for

a complete list.) If you decide that this is not a valid drag initiation, just don't call `startDrag()`; if this method isn't called, the drag is ignored. This may sound complicated, but the programming examples show how little work on your part is really required. The methods are:

public void startDrag(Cursor dragCursor, Transferable transferable, DragSourceListener dsl) throws InvalidDnDOperationException

This method starts a drag operation with the initial cursor defined by `dragCursor`. The data to be transferred in the operation needs to be wrapped up in a `Transferable` object. The `dsl` argument should be the `DragSourceListener` associated with the drag source. Often you can bundle the `DragSourceListener` and the `DragGestureListener` together in one class.

public void startDrag(Cursor dragCursor, Image dragImage, Point imageOffset, Transferable transferable, DragSourceListener dsl) throws InvalidDnDOperationException

This method is similar to the first `startDrag()` with the exception of the `dragImage` and `imageOffset` arguments. If draggable images are supported on your platform (they may not be!), then you can supply an image and an offset to use during the drag operation. The offset represents the location of the mouse pointer relative to the upper-left corner of the image. You can find out if your platform supports drag images using the `isDragImageSupported()` method of the `DragSource` class discussed in the next section.

A Simple Gesture

While it won't do us any good on its own, we need to recognize these drag gestures before we can get to anything fun in the drag and drop process. Here's a simple example of `JList` object containing draggable list items. At least, it will contain them eventually. Right now we'll just make sure that we can recognize a drag gesture when it happens in our list object.

```
/*
 * GestureTest.java
 * A simple (?) test of the DragSource classes to see if we
 * can create a draggable object in a Java application.
 */

import java.awt.*;
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;

public class GestureTest extends JFrame implements DragGestureListener {
```

```
  DragSource ds;
  JList jl;
  String[] items = {"Java", "C", "C++", "Lisp", "Perl", "Python"};

  public GestureTest() {
    super("Gesture Test");
    setSize(200,150);
    addWindowListener(new BasicWindowMonitor());

    // Create a JList object just like you normally would and add it
    // to our application.
    jl = new JList(items);
    jl.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    getContentPane().add(new JScrollPane(jl), BorderLayout.CENTER);

    // Now create a DragRecognizer...we're using the DragSource class
    // to do this; you could use a Toolkit if you'd rather. Notice that
    // we specify our JList object when creating the recognizer, *not*
    // when creating the drag source.
    ds = new DragSource();
    DragGestureRecognizer dgr = ds.createDefaultDragGestureRecognizer(
                                jl, DnDConstants.ACTION_COPY, this);
    setVisible(true);
  }

  // Ok, report the recognized drag event to the console.
  public void dragGestureRecognized(DragGestureEvent dge) {
    System.out.println("Drag Gesture Recognized!");
    // In real life, we would check the drag gesture for validity,
    // and call startDrag() to get things going. In this toy program, we
    // merely announce that the event occurred.
  }

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

There's really not much to it. The constructor creates a `DragSource`, and uses that to create a standard recognizer by calling `createDefaultDragGestureRecognizer()`. The recognizer ties the drag's source and the current component (`this`) together. When a drag is recognized, our `dragGestureRecognized()` method is called. For the moment, we just print an acknowledgement—we really haven't gotten the drag source working yet. In real life, we would make sure that the drag is valid and, if so, call `startDrag()`.

The Drag API

Once you recognize a drag gesture, you can start an actual drag. The drag source receives many of the same types of events as the drop target. Like a drop target, a drag source has

a native peer in the form of a DragSourceContext and receives events as the user drags an object over the source and finally drops it on a drop target. The classes of the drag API even fall into a similar hierarchy, shown in Figure 1-4.

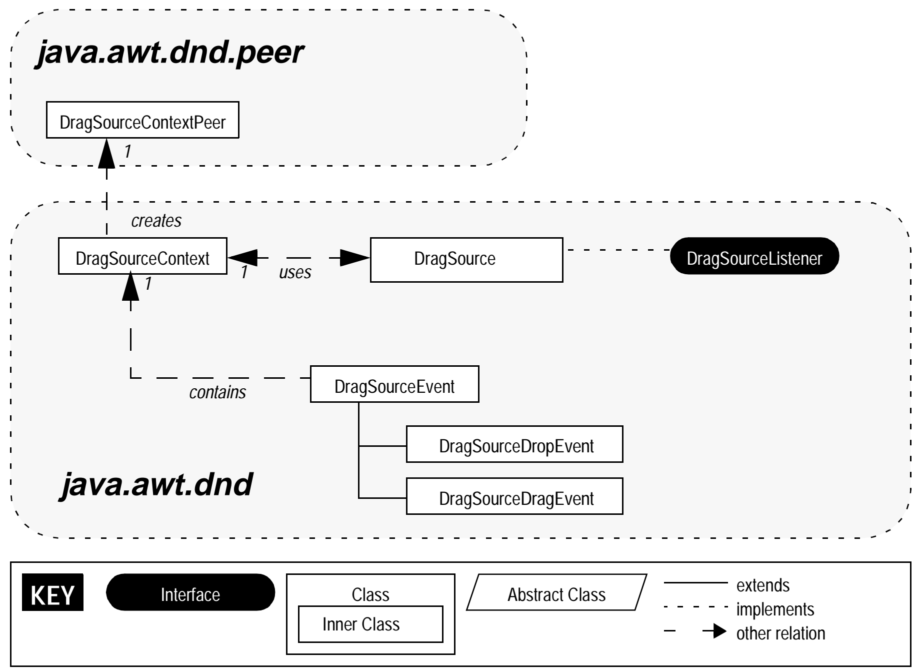

Figure 1-4. Drag side of the DnD class hierarchy

The DragSource Class

The flip side of DropTarget, the DragSource class provides support for creating draggable information. Recall from the beginning of this chapter that unlike the DropTarget class, a regular GUI component is *not* used as a base. Rather, the base component for starting a drag is encapsulated in the DragGestureRecognizer. The recognizer can in turn start a drag for the DragSource. The DragSource class encompasses the context of the drag and provides a focal point for maintaining the visual state of the drag (i.e., selecting which cursor to display for the drag operation). Several pre-defined cursors are built into this class. You should remember, though, that most times you will update the cursor when handling events through the DropTargetListener methods. We'll see some good examples of this process later in the chapter.

Properties

The DragSource class has only three properties, two of which are static and apply to the overall DnD system. These properties are shown in Table 1-10.

Table 1-10. `DragSource` *Properties*

Property	Data Type	get	is	set	Bound	Default Value
defaultDragSource[1]	DragSource	•				new DragSource()
flavorMap	FlavorMap	•				SystemFlavorMap.getDefaultFlavorMap()
dragImageSupported[1]	boolean		•			false

[1]*These methods are* `static` *and can thus be accessed at any time.*

The `defaultDragSource` property provides a "platform" drag source that can be retrieved from any other class. As you do not really associate a component with a drag source, you can rely entirely on this default source during your drags. (This is not required; you can build your own.) The `flavorMap` property stores the flavor map for the source to help translate between native names and MIME type names for the data to be transferred. The `dragImageSupported` property indicates whether or not your system can drag an `Image` along with the cursor during a drag operation.

Constants

Several predefined cursors are at your disposal as constants in `DragSource`. These cursor constants can be used when starting a drag operation and are listed in Table 1-11. The sample images are merely representative of the types of cursors that appear. They may vary from system to system, and with Java 2, you are certainly free to build and use your own.

Table 1-11. `DragSource` *Constants*

Constant	Type	Description
DefaultCopyDrop	Cursor	
DefaultCopyNoDrop	Cursor	
DefaultLinkDrop	Cursor	
DefaultLinkNoDrop	Cursor	

Table 1-11. `DragSource` *Constants (Continued)*

Constant	Type	Description
DefaultMoveDrop	Cursor	
DefaultMoveNoDrop	Cursor	

Constructors

One public constructor for `DragSource` exists:

public DragSource()
This constructor creates a new `DragSource`.

Helper Creation Methods

The `DragSource` class has a few helper methods to create the other parts required for initiating a drag:

public DragGestureRecognizer createDefaultDragGestureRecognizer(Component c, int actions, DragGestureListener dgl)
public DragGestureRecognizer createDragGestureRecognizer(Class recognizerClass, Component c, int actions, DragGestureListener dgl)
These methods can be used to create `DragGestureRecognizer` objects to use with your drag-capable component (`c`). The `actions` argument again comes from `DnDConstants` and dictates the types of actions (move, copy, link) that are acceptable. You can give a null listener for the `dgl` argument and attach one to the recognizer later. The second version of this method uses `recognizerClass` as a prototype for the recognizer. The `recognizerClass` argument should be a subclass of `DragGestureRecognizer`.

protected DragSourceContext createDragSourceContext(DragSourceContextPeer dscp, DragGestureEvent dgl, Cursor dragCursor, Image dragImage, Point imageOffset, Transferable t, DragSourceListener dsl)
This protected method creates a new context object for the `DragSource`. As with other peer operations, this is not something you normally need to use. However, subclasses can override this to create context objects suited for a particular task, such as always supplying a certain image to go with the drag operation.

Start Methods

As you saw before in the DragGestureEvent class, once a drag gesture has been recognized, you need to start the drag manually. Either you can use the drag event versions, or if you need a bit more control, you can use any of the following methods as well:

public void startDrag(DragGestureEvent trigger, Cursor dragCursor, Transferable transferable, DragSourceListener dsl) throws InvalidDnDOperationException

The simplest of the start methods, this one requires only a trigger event, (trigger), an initial cursor to display (dragCursor), the data to be transferred, (transferable), and a source listener (which can be null). If the system is not in a state where a drag initiation is allowed (typically because a peer DragSourceContext cannot be created) it will throw the InvalidDnDOperationException.

public void startDrag(DragGestureEvent trigger, Cursor dragCursor, Transferable transferable, DragSourceListener dsl, FlavorMap flavorMap) throws InvalidDnDOperationException

Similar to the method above, this version allows you to specify your own flavor map.

public void startDrag(DragGestureEvent trigger, Cursor dragCursor, Image dragImage, Point dragOffset, Transferable transferable, DragSourceListener dsl) throws InvalidDnDOperationException

If your system supports drag images, you can specify both the image and the offset in that image where you want the cursor to sit. Remember that not all systems support drag images.

public void startDrag(DragGestureEvent trigger, Cursor dragCursor, Image dragImage, Point imageOffset, Transferable transferable, DragSourceListener dsl, FlavorMap flavorMap) throws InvalidDnDOperationException

Similar to the other startDrag() methods, but of course, you can specify both a drag image and a flavor map if you have them.

The DragSourceContext Class

As you may have gathered from reading through the previous section, DragSource objects use DragSourceContext to handle the necessary native windowing code much the same way DropTarget objects do. You don't have to create an instance of this class on your own, but you can access the context from the DragSource class if you need any of the information provided by the context.

Properties

DragSourceContext has several properties, which are listed in Table 1-12; these properties are all read-only, meaning that DragSourceContext supplies information about a drag operation, but doesn't directly allow you to modify the operation.

Table 1-12. DragSourceContext *Properties*

Property	Data Type	get	is	set	Bound	Default Value
component	Component	•				from trigger (via the constructor)
cursor	Cursor	•				from constructor
dragSource	DragSource	•				from trigger (via the constructor)
sourceActions	int	•				from trigger (via the constructor)
transferable	Transferable	•				from constructor
trigger	InputEvent	•				from constructor

These properties give you access to the "way things look" during a drag operation. With the exception of the cursor property, which tells you which cursor is currently displayed, these properties are similar to the properties of the DragGestureEvent class.

Events

The DragSourceContext class generates DragSource events, which tell their listeners (typically the DragSource) what is happening with the drag. The listener uses the events to react appropriately—for example, deleting an object that has been "moved" after it has been successfully dropped. If you think about this example, you'll see why it's necessary for the source of a drag operation to track the drag's progress: if the user is moving an object, the source will eventually want to remove that object. But it can't remove the object until it knows that the drag succeeded.

DragSourceContext contains the necessary methods for adding and removing listeners. While you can use these methods to attach (or detach) a listener, you typically pass in your listener when you call a startDrag() method. It's rare to call the add and remove methods explicitly. These event methods are:

public void addDragSourceListener(DragSourceListener dsl) throws TooManyListenersException
public void removeDragSourceListener(DragSourceListener dsl)

These methods add and remove a drag source listener for this context. Notice that this is a unicast event.

Protected Constants

Table 1-13 defines several protected constants found in the `DragSourceContext` class. These constants are used to identify cursor actions in calls to the `updateCurrentCursor()` method.

Table 1-13. `DragSourceContext` *Constants*

Constant	Type	Description
`CHANGED`	`int`	The user action has changed (usually between copy and move).
`DEFAULT`	`int`	Should switch to the default cursor.
`ENTER`	`int`	The cursor entered a drop target.
`OVER`	`int`	The cursor is over a drop target.

Constructors

You can create a `DragSourceContext` with all the frills using its only constructor. However, to call the constructor, you need a valid `DragSourceContextPeer` for your platform. Normally, you call one of the `startDrag()` methods of the `DragGestureEvent`, which creates a context for you behind the scenes. The constructor is:

public DragSourceContext(DragSourceContextPeer dscp, DragGestureEvent trigger, Cursor dragCursor, Image dragImage, Point offset, Transferable t, DragSourceListener dsl)

This constructor allows you to specify all of the properties listed in Table 1-12 for your context object.

Event Methods

Like the `DropTargetContext` class, `DragSourceContext` implements the `DragSourceListener` interface itself. The event handling methods are:

public void dragEnter(DragSourceDragEvent dsde)
public void dragOver(DragSourceDragEvent dsde)
public void dragExit(DragSourceEvent dse)
public void dropActionChanged(DragSourceDragEvent dsde)
public void dragDropEnd(DragSourceDropEvent dsde)

These methods come from the `DragSourceListener` interface. They trap incoming events, and if a valid listener exists, the events are passed on. The cursor is then updated for you automatically.

Miscellaneous Methods

Two other methods exist to round out the functionality required of the `DragSource-Context` class:

public void transferablesFlavorsChanged()

This method can also be used to indicate that something has changed in the current drag process. If the data flavor for the `Transferable` object changes, this method can be called to notify the native peer of that change.

protected void updateCurrentCursor(int dropOp, int targetAct, int status)

Each of the event methods in the previous section use this method to update the cursor as the user moves over potential drop targets and other parts of the application. The `dropOp` argument is the current action the user has undertaken. The `targetAction` argument lists the acceptable drop actions for the current target, and `status` indicates what type of update this is for the current cursor. This can be any one of the values from Table 1-13.

The DragSourceListener Interface

If you care about events being sent to the drag source, you should implement the `Drag-SourceListener` interface. All drag events generated over the source object and the drop event (either an accepted drop over a valid drop target or a rejected drop) get sent here.

Event Methods

The types of events presented here mimic the types we saw with the `DropTargetLis-tener` interface:

public void dragEnter(DragSourceDragEvent dsde)

This method handles events generated as the user enters a viable drop target.

public void dragOver(DragSourceDragEvent dsde)

This method handles events that are generated continuously as the user moves around inside the bounds of a viable drop target.

public void dropActionChanged(DragSourceDragEvent dsde)

This method is called with an appropriate event if the user changes the drag operation from copy to move or vice versa.

public void dragExit(DragSourceEvent dse)

This method handles the events generated when the user leaves the bounds of a viable target.

public void dragDropEnd(DragSourceDropEvent dsde)

Not found in the `DropTargetListener` interface, this method gets notified when a drop has completed on a drop target. Notification does not imply success; you need to check the `dsde` event for that. The events are discussed in the next section.

Drag Source Events

Not surprisingly, we have drag source events similar to the drop target events. The `DragSourceEvent` class serves as the base of the event hierarchy and gives you access to the source context through its sole property listed in Table 1-14.

Table 1-14. `DragSourceEvent` *Properties*

Property	Data Type	get	is	set	Bound	Default Value
`dragSourceContext`	`DragSourceContext`	•				from constructor

Constructors

This class has one constructor:

public DragSourceEvent(DragSourceContext dsc)

If you have a valid `DragSourceContext`, you can build your own `DragSourceEvent` with this constructor.

The DragSourceDragEvent Class

The subclasses of `DragSourceEvent` distinguish between drag events and drop events. `DragSourceDragEvent` is used in a majority of the events reported, with the exception of exit and drop complete events.

Properties

The `DragSourceDragEvent` class properties help you decide how the source of the drag should respond to various things the user does. Most of the properties deal with the appropriate types of action the user can take and, of course, what the user is really doing. These properties are shown in Table 1-15.

Table 1-15. DragSourceDragEvent *Properties*

Property	Data Type	get	is	set	Bound	Default Value
dropAction	int	•				from constructor
gestureModifiers	int	•				from constructor
targetActions	int	•				same as dropAction
userAction	int	•				from constructor

The dropAction represents the effective drop action found by combining the user's currently selected drop action (userAction) and the actions acceptable to the target (targetActions). Differences in dropAction and userAction can be used to provide helpful feedback to the user during the drag operation. The gestureModifiers property describes the state of any input modifiers, such as holding down the Control key.

Constructors

This class has one constructor:

public DragSourceDragEvent(DragSourceContext dsc, int dropAction, int actions, int modifiers)

This constructor creates a new DragSourceDragEvent and fills in the appropriate properties. (The actions argument represents the targetActions property.) The userAction property is filled in at runtime and can change as the user alters the modifiers.

The DragSourceDropEvent Class

While it's not required, your source might care about the success (or failure) of a drop. For example, it would be the source's responsibility to delete an object after it has been moved by a drag. To know when to delete the object, the source needs to know whether the drop succeeded. This class encapsulates the drop event and its status.

Properties

Table 1-16 shows the two properties available from a DragSourceDropEvent.

The dropAction property contains the final drop action taken by the user. If the drop target accepted the drop, dropSuccess will be true; false otherwise. You may recall the dropComplete() method from the DropTargetDropEvent class. The dropSuccess property reflects the value passed to that method.

Table 1-16. `DragSourceDropEvent` *Properties*

Property	Data Type	get	is	set	Bound	Default Value
dropAction	int	•				ACTION_NONE
dropSuccess	boolean	•				false

Constructors

public DragSourceDropEvent(DragSourceContext dsc)
public DragSourceDropEvent(DragSourceContext dsc, int action, boolean success)

The first constructor creates an event that does not result in an actual drop. If you notice that the default value for the drop action is `ACTION_NONE`, no drop target should really accept a drop event built like this. Instead, you would use the second constructor.

Completing the Gesture

We can go back to our `GestureTest` program now and finish the job. We'll complete the `DragSource` object and add in some listener methods so that we can see the different events as they are reported to the `DragSourceListener`. You'll also want to pay attention to what we do in the `dragGestureRecognized()` method. We manually start the drag operation there. Figure 1-5 shows the resulting pair of applications. `DragTest` is the newest version of our gesture recognizer; it presents a list of items that can be dragged. We used `DropTest2` with the expanded `drop()` method for the receiving end.

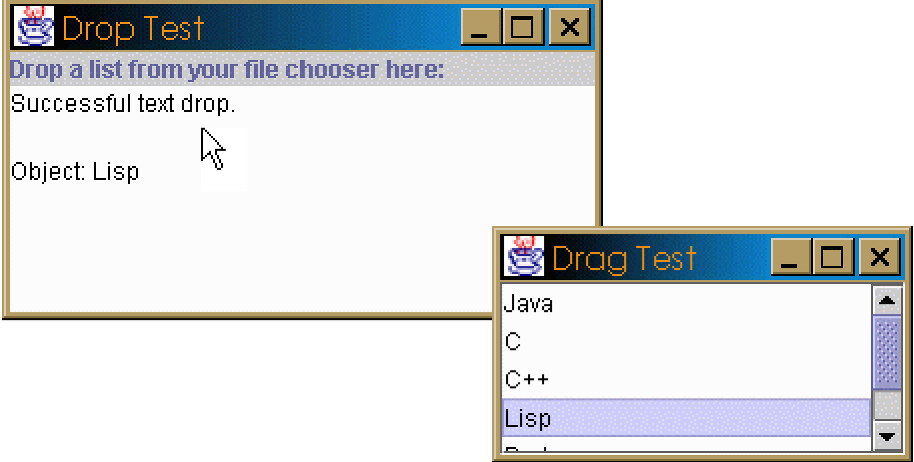

Figure 1-5. A drag operation started in a `JList` *and dropped in a* `JTextArea`

And here's the source for the complete `DragTest` application:

```
/*
 * DragTest.java
 * A simple (?) test of the DragSource classes to see if we
 * can create a draggable object in a Java application.
 */

import java.awt.*;
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;

public class DragTest extends JFrame implements DragSourceListener,
DragGestureListener {

  DragSource ds;
  JList jl;
  StringSelection transferable;
  String[] items = {"Java", "C", "C++", "Lisp", "Perl", "Python"};

  public DragTest() {
    super("Drag Test");
    setSize(200,150);
    addWindowListener(new BasicWindowMonitor());

    // Set up our JList and DragRecognizer as before. The difference
    // this time is that we will use the recognized drag to start a
    // real drag process.
    jl = new JList(items);
    jl.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    getContentPane().add(new JScrollPane(jl), BorderLayout.CENTER);

    ds = new DragSource();
    DragGestureRecognizer dgr = ds.createDefaultDragGestureRecognizer(
        jl, DnDConstants.ACTION_COPY, this);
    setVisible(true);
  }

  // In addition to reporting a successful gesture, create a piece of
  // transferable data using the StringSelection class.
  public void dragGestureRecognized(DragGestureEvent dge) {
    System.out.println("Drag Gesture Recognized!");
    transferable = new StringSelection(jl.getSelectedValue().toString());
    ds.startDrag(dge, DragSource.DefaultCopyDrop, transferable, this);
    // You could also use the dge to start the drag:
    // dge.startDrag(DragSource.DefaultCopyDrop, transferable, this);
  }

  public void dragEnter(DragSourceDragEvent dsde) {
    System.out.println("Drag Enter");
```

```
  }

  public void dragExit(DragSourceEvent dse) {
    System.out.println("Drag Exit");
  }

  public void dragOver(DragSourceDragEvent dsde) {
    System.out.println("Drag Over");
  }

  public void dragDropEnd(DragSourceDropEvent dsde) {
    System.out.print("Drag Drop End: ");
    if (dsde.getDropSuccess()) {
      System.out.println("Succeeded");
    }
    else {
      System.out.println("Failed");
    }
  }

  public void dropActionChanged(DragSourceDragEvent dsde) {
    System.out.println("Drop Action Changed");
  }

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

With this version of the `dragGestureRecognized()` method, we create a real piece of draggable information. In our case, we create a `String` object with the `StringSelection` class. This object could be picked up by other Java applications as a `String`, or other applications as a serialized Java object—admittedly not the most ideal transfer medium if you are serious about dropping this text in some native application. Fortunately, the `StringSelection` class also supports the "plain text" flavor. That flavor will be more useful to native window applications. Once we've created the `StringSelection`, which implements `Transferable`, we're ready to start the drag by calling the `startDrag()` method of our drag source. (We could have also called the `startDrag()` method that's packaged with the event itself.) In this call, we specify the event that triggered the gesture, the drag cursor we want to use (`DragSource.DefaultCopyDrop`), the data we're dragging, and the listener for `DragGestureEvents` (`this`).

Rearranging Trees

We can apply all this drag and drop functionality to `JTree` objects to overcome a deficiency in their user interface. On its own, `JTree` has no facility for visually rearranging its leaves and branches. We can create appropriate drag and drop event handlers and our

own transferable type to implement this functionality for any `JTree` we build. Figure 1-6 shows the rearrangeable tree application.

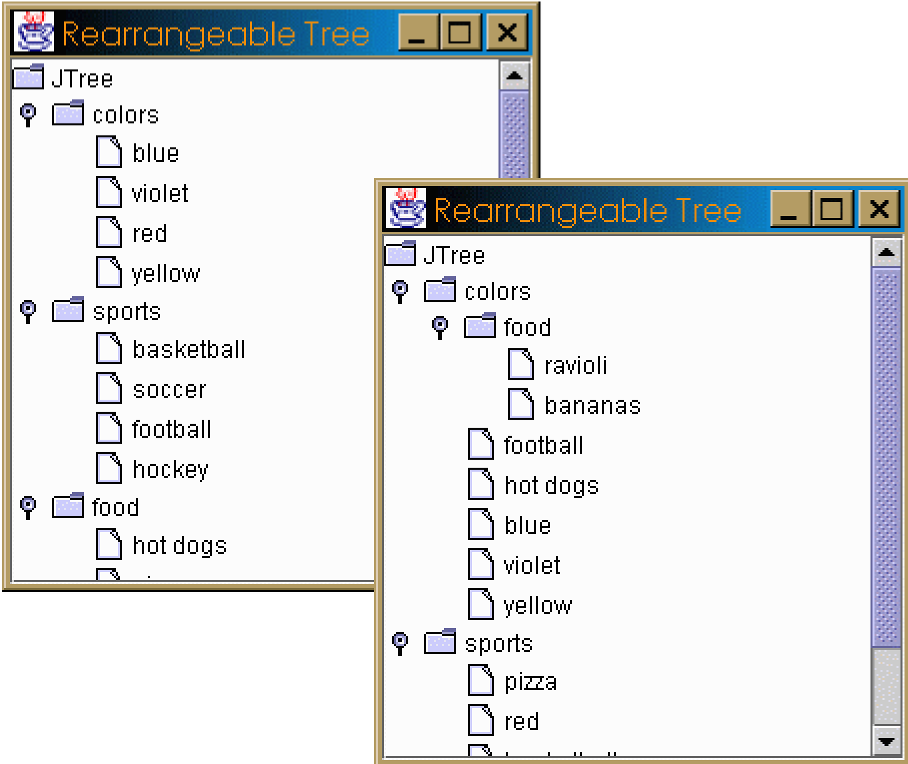

Figure 1-6. An editable tree before (left) and after (right) items have been dragged and dropped

Ok, so "before and after" screen shots still don't do this application justice. But notice that you can move entire folders as well as individual items. Any node can be dragged and dropped into any folder. You are, however, limited to dropping items into a folder. As with the other examples in this chapter, you'll have to compile the code and play with the application to get the full effect. In the meantime, here's the source code for the drag half of our example. Don't worry too much about the `TransferableTreePath` class; we'll discuss that in greater detail in the next section. For now, just think of it as a DnD wrapper for a tree node.

```
/*
 * TreeDragSource.java
 * A more involved test of the drag and drop system to see if
 * we can create a self-contained, rearrangable JTree. This is
 * the drop half.
 */

import java.awt.*;
```

```
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import javax.swing.tree.*;

public class TreeDragSource
implements DragSourceListener, DragGestureListener {

  DragSource source;
  DragGestureRecognizer recognizer;
  TransferableTreePath transferable;
  DefaultMutableTreeNode oldNode;
  JTree sourceTree;

  public TreeDragSource(JTree tree, int actions) {
    sourceTree = tree;
    source = new DragSource();
    recognizer = source.createDefaultDragGestureRecognizer(
         sourceTree, actions, this);
  }

  /*
   * Drag Gesture Handler
   */
  public void dragGestureRecognized(DragGestureEvent dge) {
    TreePath path = sourceTree.getSelectionPath();
    if ((path == null) || (path.getPathCount() <= 1)) {
      // We can't really move the root node (or an empty selection).
      return;
    }
    // Remember which node was dragged off, so that we can delete
    // it to complete a move operation.
    oldNode = (DefaultMutableTreeNode)path.getLastPathComponent();

    // Make a version of the node that we can use in the DnD system...
    transferable = new TransferableTreePath(path);
    // ...and start the drag process. We start with a no-drop cursor
    // assuming that the user won't want to drop the item right where they
    // picked it up. As soon as they move the mouse, it should change to
    // an "OK" cursor...
    dge.startDrag(DragSource.DefaultMoveNoDrop, transferable, this);

    // If you support dropping the node anywhere, you should probably
    // start with a valid move cursor:
    // dge.startDrag(DragSource.DefaultMoveDrop, transferable, this);
  }

  /*
   * Drag Event Handlers
   */
  public void dragEnter(DragSourceDragEvent dsde) { }
```

```
  public void dragExit(DragSourceEvent dse) { }
  public void dragOver(DragSourceDragEvent dsde) { }
  public void dropActionChanged(DragSourceDragEvent dsde) { }

  public void dragDropEnd(DragSourceDropEvent dsde) {
    if (dsde.getDropSuccess()) {
      // Only remove the node if the drop was successful.
      ((DefaultTreeModel)sourceTree.getModel())
            .removeNodeFromParent(oldNode);
    }
  }
}
```

The setup in the constructor for `TreeDragSource` should be fairly familiar by now: we create a `DragSource`, then create the default `DragGestureRecognizer` for our tree. Note that we are assuming the JTree in question has been set up with single selection mode. You could certainly handle multiple selections by transferring an array of selected nodes. With Java 2, however, recognizing a drag with multiple selections is tricky—the click that starts the drag often de-selects the item under the mouse. Presumably support for DnD in future releases will remove this obstacle.

The most interesting part of the class is `dragGestureRecognized()`, which figures out whether or not we really have a valid drag in progress. The gesture recognizer knows that drag gestures involving a tree starts with a selection, so we can start by asking for the current selection path. We discard the drag if the selection is empty, or if the user selected the root node, which can't be dragged. If the drag is valid, we extract the last component of the path by calling `getLastPathComponent()`, and packaging that node up as `Transferable`, using our `TransferableTreePath` class. Then we call `startDrag()` to start the actual drag operation. Here's the source code for our transfer wrapper:

```
/*
 * TransferableTreePath.java
 * A Transferable TreePath to be used with drag and drop applications.
 */

import java.io.*;
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import javax.swing.tree.*;

public class TransferableTreePath implements Transferable {

  public static DataFlavor TREE_PATH_FLAVOR =
       new DataFlavor(TreePath.class, "Tree Path");
  DataFlavor flavors[] = { TREE_PATH_FLAVOR };
  TreePath path;

  public TransferableTreePath(TreePath tp) {
    path = tp;
  }
```

```
  public synchronized DataFlavor[] getTransferDataFlavors() {
    return flavors;
  }

  public boolean isDataFlavorSupported(DataFlavor flavor) {
    return (flavor.getRepresentationClass() == TreePath.class);
  }

  public synchronized Object getTransferData(DataFlavor flavor)
    throws UnsupportedFlavorException, IOException {
    if (isDataFlavorSupported(flavor)) {
      return (Object)path;
    } else {
      throw new UnsupportedFlavorException(flavor);
    }
  }
}
```

As you can see, it is not very complicated. The simplicity of this `Transferable` implementation stems from the fact that we really only support transferring `TreeNode` objects from one Java application to another. (We do not restrict the transfer to a single VM, though!) The `getTransferData()` method should return the data in a format compatible with the flavor the user passes in. This allows you to support multiple data formats (including non-Java formats). You can take a look at the source code of the `java.awt.dnd.StringSelection` class if you want to see another example of implementing `Transferable`.

In addition to listening to drag gesture events, we also need to listen to drag source events to find out what happens while the drag is in progress. Most of these events aren't interesting, so most of the methods in the `DragSourceListener` interface have null implementations (`dragEnter()` and the rest—later we'll see how to use these methods to keep the cursor updated and do other useful things). The one exception is the `drop()` method, which is called when the drag-and-drop operation has completed. If the drop was successful, we want to delete the dragged node from the source tree. The code for this is relatively simple: we call `getDropSuccess()` to find the result, and we call `removeModeFromParent()` on the tree's model to delete the dragged node if the operation succeeded.

The `TreeDragSource` that we developed allows us to recognize a drag and build a valid `Transferable` object to accompany the drag. The drop side needs to take the drop, figure out where it landed, and decide what to do with it. If the drop occurred over a folder and contained one of our special tree node objects, we accept the drop. If it was over a file or was not a tree node, we reject the drag. (And we finally get to put some of those obtuse `JTree` methods to use!) Here's the code for the drop target:

```
/*
 * TreeDropTarget.java
```

```
 * A more involved test of the drag and drop system to see if
 * we can create a self-contained, rearrangable JTree. This is
 * the drop half.
 */

import java.awt.*;
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import javax.swing.tree.*;

public class TreeDropTarget implements DropTargetListener {

  DropTarget target;
  JTree targetTree;

  public TreeDropTarget(JTree tree) {
    targetTree = tree;
    target = new DropTarget(targetTree, this);
  }

  /*
   * Drop Event Handlers
   */
  public void dragEnter(DropTargetDragEvent dtde) { }
  public void dragOver(DropTargetDragEvent dtde) { }
  public void dragExit(DropTargetEvent dte) { }
  public void dropActionChanged(DropTargetDragEvent dtde) { }

  public void drop(DropTargetDropEvent dtde) {
    // Figure out where the drop occurred (in relation to the target tree).
    Point pt = dtde.getLocation();
    DropTargetContext dtc = dtde.getDropTargetContext();
    JTree tree = (JTree)dtc.getComponent();
    TreePath parentpath = tree.getClosestPathForLocation(pt.x, pt.y);
    DefaultMutableTreeNode parent =
      (DefaultMutableTreeNode)parentpath.getLastPathComponent();

    // Now check to see if it was dropped on a folder. If not, reject it.
    if (parent.isLeaf()) {
      dtde.rejectDrop();
      return;
    }

    try {
      // Grab the data...
      Transferable tr = dtde.getTransferable();
      DataFlavor[] flavors = tr.getTransferDataFlavors();
      for (int i = 0; i < flavors.length; i++) {
        if (tr.isDataFlavorSupported(flavors[i])) {
          // Ok, it's a usable node, so pull it out of the transferable
```

```
          // object and add it to our tree.
          dtde.acceptDrop(DnDConstants.ACTION_MOVE);
          TreePath p = (TreePath)tr.getTransferData(flavors[i]);
          DefaultMutableTreeNode node =
            (DefaultMutableTreeNode)p.getLastPathComponent();
          DefaultTreeModel model = (DefaultTreeModel)tree.getModel();
          model.insertNodeInto(node, parent, 0);

          // And last but not least, mark the drop a success.
          dtde.dropComplete(true);
          return;
        }
      }
      dtde.rejectDrop();
    } catch (Exception e) {
      e.printStackTrace(); // just for debugging, really.
      dtde.rejectDrop();
    }
  }
}
```

It's obvious where all the action is. Our `drop()` method, one of the methods required by `DropTargetListener`, does all the work of figuring out exactly where the user dropped the tree node, getting the data back from the drop event, and putting the new node in the tree. First we figure out where exactly the drop occurred by calling the `getLocation()` method of the drop event `dtde`. We then get the `DropTargetContext`, which lets us retrieve the actual component (obviously, a `JTree`) on which the drop occurred. Now that we have both the component and the location of the drop in hand, we can call `getClosestPathForLocation()` to figure out exactly where, in terms of the target tree, the drop occurred. We extract the last tree node from this path. If this node is a leaf node, we reject the drop, because it only makes sense to drop a tree node into a folder. If we have a folder, we proceed to extract the data and add the node.

To extract the data, we call `getTransferable()`, and then check whether or not the object that we received has a data flavor that we understand. If it does, we accept the drop, specifying that we want to move the node from the old tree into the new one. (The source object is eventually informed that we wish to move the object rather than copy it, and can decide whether or not to delete it. The current version of `TreeDragSource` ignores this information.) We then call `getTransferData()` to construct a tree path, extract the last node from that path, and insert that new node into the tree's data model under the parent node that we extracted earlier, using the method `insertNodeInto()`.

Now that we have a `TreeDragSource` and a `TreeDropTarget`, putting the pieces together in a working application does not take much effort. If we want to create a single rearrangeable tree, we simply make it both a drag source and a drop target. With the separation of our drag and drop handlers, we could also enforce a "source-only" tree by

attaching only the drag source half. Without the drop half, drops on the tree would simply be ignored.

```
/*
 * TreeDragTest.java
 * A more involved test of the drag and drop system to see if
 * we can create a self-contained, rearrangable JTree. This is
 * the test framework that builds a tree and attaches both the
 * drag and the drop support.
 *.
 */

import java.awt.*;
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import javax.swing.tree.*;

public class TreeDragTest extends JFrame {

  TreeDragSource ds;
  TreeDropTarget dt;
  JTree tree;

  public TreeDragTest() {
    super("Rearrangeable Tree");
    setSize(300,200);
    addWindowListener(new BasicWindowMonitor());

    // Create a quick JTree to use for demonstration purposes.
    // The default constructor for JTree creates just such a
    // tree with a few categories like food and sports to play
    // around with.
    tree = new JTree();
    getContentPane().add(new JScrollPane(tree), BorderLayout.CENTER);

    ds = new TreeDragSource(tree, DnDConstants.ACTION_MOVE);
    dt = new TreeDropTarget(tree);
    setVisible(true);
  }

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

This class is extremely simple. All we do is get a `JTree`, put it in a `JFrame`, and then instantiate our `TreeDragSource` and `TreeDropTarget` around the same tree.

Finishing Touches

With the DnD API, we can add a few more features that will help us achieve the goal of user interface maturity. Since we can't drop our objects on other leaves, we really should keep the drag cursor consistent. If we're over a folder, we can accept the drag and show the "ok to drop here" cursor. Anywhere else, we'll reject the drag and show the "not here" version. The current implementation of our handlers allows only nodes to be moved. We could add in support for "copy or move" drags. (We can also keep our cursor in sync with the new support so users know if they're copying or moving.) Once all that is working, we should also set up autoscrolling on our tree to give the user access to any off-screen parts of the tree.

Dynamic Cursors

The DnD package does ship with some standard cursors for indicating "ok to drop here" and "not here." Those cursors are displayed automatically when you `accept()` or `reject()` an item as it is dragged over potential drop targets. The code for deciding whether or not it's okay to drop belongs in the `dragEnter()` and `dragOver()` methods of the `DropEventListener` interface, which is implemented by our `TreeDropTarget`. (One potential source for confusion: `DragSourceListener` has a very similar group of methods, and it's easy to imagine implementing this logic in the wrong interface. Just remember that it's the drop target's job to determine whether or not the drop is in a legitimate place.)

Here are new versions of the `dragEnter()` and `dragOver()` event handlers in `TreeDragTest` that check to see if the mouse is over a folder in the tree. If it's not, we reject the drag. That rejection causes the cursor to update its appearance. (Note that using the new cursor features in Java 2, you could define your own cursors if you don't like the cursors supplied by your windowing system.)

```
private TreeNode getNodeForEvent(DropTargetDragEvent dtde) {
  // Use the same logic from the drop() method to see if the cursor is
  // over a folder in the target tree.
  Point p = dtde.getLocation();
  DropTargetContext dtc = dtde.getDropTargetContext();
  JTree tree = (JTree)dtc.getComponent();
  TreePath path = tree.getClosestPathForLocation(p.x, p.y);
  return (TreeNode)path.getLastPathComponent();
}

public void dragEnter(DropTargetDragEvent dtde) {
  TreeNode node = getNodeForEvent(dtde);
  if (node.isLeaf()) {
    // This doesn't halt the drag operation, it merely updates the cursor
    // to reflect a "no drop" region.
    dtde.rejectDrag();
  }
```

```
    else {
      // Start by supporting move operations.
      dtde.acceptDrag(DnDConstants.ACTION_MOVE);
    }
  }

  public void dragOver(DropTargetDragEvent dtde) {
    TreeNode node = getNodeForEvent(dtde);
    if (node.isLeaf()) {
      dtde.rejectDrag();
    }
    else {
      // Start by supporting move operations.
      dtde.acceptDrag(DnDConstants.ACTION_MOVE);
    }
  }
```

`dragEnter()` and `dragExit()` are very simple: they both ask a private helper method, `getNodeForEvent()`, to extract the node that the cursor has entered from the `DropTargetDragEvent`. Once we have the node, we can ask the node whether or not it is a leaf, and reject the drag if it is. `getNodeForEvent()` is a simple helper method that retrieves the cursor location from a `DropTargetDragEvent` and uses it to figure out which node the cursor is over.

Changing the Drop Action

So far, we have always assumed that the user wanted to move a node; merely copying a node from one place to another wasn't possible. To support both move and copy operations, we only need to make two modifications. Contrary to what you might infer from the names of the event handling methods, we do not need to override the `dropActionChanged()` method. (You would override that method if you wanted to react to the change *during* the drag operation. We just want to react to the change when the data is finally dropped.)

All we really need to do is make sure that our `acceptDrag()` methods handle both copy and move actions. Regrettably, you cannot accept the `ACTION_MOVE_OR_COPY` action to encompass both types of drops. You need to decide which type of drop occurred and accept that drop specifically. However, if you know you'll accept any drop the user can make, you can try this trick: in `TreeDropTarget.java`, everywhere we had:

```
dtde.acceptDrag(DnDConstants.ACTION_MOVE);
```

Now we have:

```
dtde.acceptDrag(dtde.getDropAction());
```

We just accept whatever the user's desired action is.

To complete the transaction properly, we need to make sure that during a copy operation we don't remove the original node. In `TreeDragSource.java`, we need a slightly smarter `dragDropEnd()` method:

```
public void dragDropEnd(DragSourceDropEvent dsde) {
  /*
   * To support move or copy, we have to check which occurred,
   * and remove the node only if it was a move operation.
   */
  if (dsde.getDropSuccess() &&
      (dsde.getDropAction() == DnDConstants.ACTION_MOVE))
  {
    ((DefaultTreeModel)sourceTree.getModel())
          .removeNodeFromParent(oldNode);
  }
}
```

So now, if the user performed a copy rather than a move, we leave the source tree in tact. Otherwise, we remove the node from the source tree just like we did earlier.

Autoscrolling

A drop target is often contained in a scrollable window. Word processors with much more text than fits on the screen and file managers with a large hierarchy of files and folders are common examples. If you want, you can create drop targets that set up a "autoscroll boundary." This boundary allows a user in the middle of a drag operation to hold the mouse still near an edge of your component and the component will scroll itself to show the user the previously hidden area. Figure 1-7 shows our tree with the auto-scroll boundaries drawn in.

Unfortunately, automatic scrolling isn't implemented by default on components like `JTree` and `JTextArea`. However, you can create autoscrolling versions of these classes by implementing the `Autoscroll` interface.

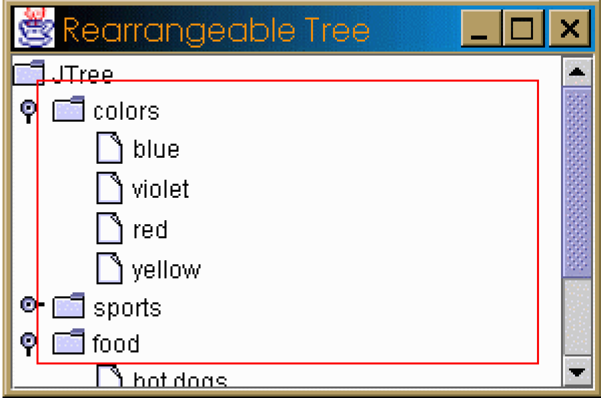

Figure 1-7. The editable tree with autoscroll boundaries

The Autoscroll Interface

You can extend any component that can appear in a `JViewport` or `JScrollPane` to implement the `Autoscroll` interface. This interface allows `DropTarget` objects to pay attention to where your cursor is and initiate an autoscroll if necessary. The work of autoscrolling is handled by an autoscroller, discussed in the next section.

Methods

The `Autoscroll` interface has only two methods:

public Insets getAutoscrollInsets()

This method returns an `Insets` object which, describes the autoscroll activation areas of a target. Figure 1-7 shows a `JTree` with an outline of the insets drawn over the tree.

public void autoscroll(Point cursorLocn)

This method is called by the autoscroller to force the component to reposition itself. Exactly where the component moves to (how fast, in which direction, etc.) is up to you. The `cursorLocn` argument gives you the location of the mouse cursor that prompted the scrolling request. You get full control over the scrolling. You can jump to a new location or use a loop to incrementally inch your way to the target.

We can add an inner class that extends `JTree` and implements this interface to achieve our autoscrolling, rearrangeable tree. Pay special attention to the `Autoscroll-ingJTree` class:

```
/*
 * TreeDragTest.java
 * A simple (?) test of the DragSource classes to see if we
 * can create a draggable object in a Java application.
 */

import java.awt.*;
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import javax.swing.tree.*;

public class TreeDragTest extends JFrame {

  TreeDragSource ds;
  TreeDropTarget dt;
  JTree tree;

  public TreeDragTest() {
    super("Rearrangeable Tree");
    setSize(300,200);
    addWindowListener(new BasicWindowMonitor());

    // If you want autoscrolling, use this line:
    tree = new AutoScrollingJTree();
    // Otherwise, use this line:
    //tree = new JTree();
    getContentPane().add(new JScrollPane(tree), BorderLayout.CENTER);

    ds = new TreeDragSource(tree, DnDConstants.ACTION_COPY_OR_MOVE);
    dt = new TreeDropTarget(tree);
    setVisible(true);
  }

  public class AutoScrollingJTree extends JTree implements Autoscroll {
    private int margin = 12;

    public AutoScrollingJTree() { super(); }

    // Ok, we've been told to scroll because the mouse cursor is in our
    // scroll zone.
    public void autoscroll(Point p) {
      // Figure out which row we're on.
      int realrow = getRowForLocation(p.x, p.y);
      Rectangle outer = getBounds();

      // Now decide if the row is at the top of the screen or at the
      // bottom. We do this to make the previous row (or the next
      // row) visible as appropriate. If we're at the absolute top or
      // bottom, just return the first or last row respectively.
      realrow = (p.y + outer.y <= margin ?
```

```
                realrow < 1 ? 0 : realrow - 1 :
                realrow < getRowCount() - 1 ? realrow + 1 : realrow);
      scrollRowToVisible(realrow);
    }

    // Calculate the insets for the *JTREE*, not the viewport
    // the tree is in. This makes it a bit messy.
    public Insets getAutoscrollInsets() {
      Rectangle outer = getBounds();
      Rectangle inner = getParent().getBounds();
      return new Insets(
        inner.y - outer.y + margin, inner.x - outer.x + margin,
        outer.height - inner.height - inner.y + outer.y + margin,
        outer.width - inner.width - inner.x + outer.x + margin);
    }

    // Use this method if you want to see the boundaries of the
    // autoscroll active region. Toss it out, otherwise.
    public void paintComponent(Graphics g) {
      super.paintComponent(g);
      Rectangle outer = getBounds();
      Rectangle inner = getParent().getBounds();
      g.setColor(Color.red);
      g.drawRect(-outer.x + 12, -outer.y + 12,
                inner.width - 24, inner.height - 24);
    }
  }

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

In the `autoscroll()` method, we check to see if the cursor is near the top edge or the bottom edge. We use `JTree`'s `retRowForLocation()` method to find the row closest to the cursor and then make the next (or previous) row visible. If you were implementing this functionality for a component that did not have something like the `scrollRowToVisible()` method, you could grab the component's container and scroll that. (Most likely, the parent container will be a `JViewport`, so you can code for that. Remember that even in a `JScrollPane`, the component is housed in a `JViewport` that is tied to the scrollbars.)

The `getAutoscrollInsets()` method looks a bit ugly. This is due to the fact that we need to supply insets that match the tree, not the viewport that contains the tree. As

we expand the tree, it gets taller. As it gets taller, we need to make the bottom inset larger. Figure 1-8 illustrates the problem we're facing.

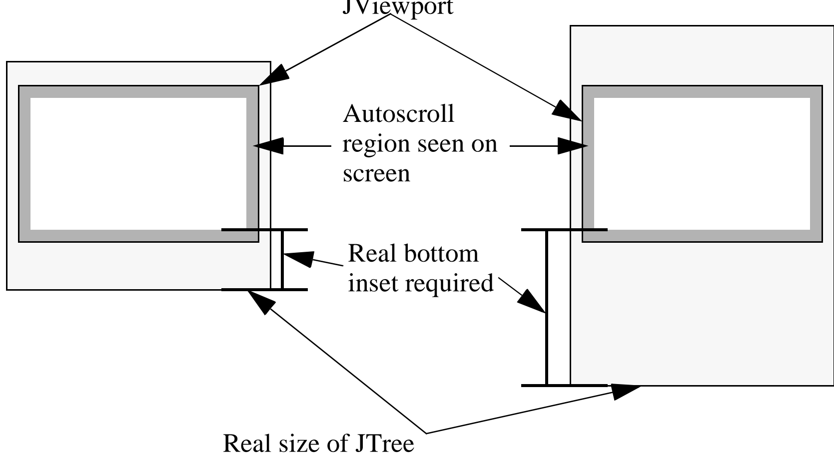

Figure 1-8. Real insets versus visible insets for a component in a `JViewport`

So, all the extra work in that method takes into account the varying size of the tree. Fortunately, that also handles the case where the user resizes the application and changes the space allotted to the viewport. The performance of these calculations is surprising quick and should suffice for many applications. However, for the performance fanatics, you could certainly modify this class to listen to component events on both the tree and the viewport. If size of either of those components changes, you could reset the values of the insets that then do not change during the `getAutoscrollInsets()` method.

The DropTarget.DropTargetAutoScroller Class

As we mentioned above, there is a separate helper class involved in listening to the mouse cursor's position and causing a target to autoscroll when the cursor waits long enough in the active autoscroll region. The `DropTarget` class has an inner class called `DropTargetAutoScroller` devoted to this task. When you register a component with a drop target, the target checks to see if the component is an instance of `Autoscroll`. If so, it creates an autoscroller for you. You could, of course, subclass if you wanted different behavior from the autoscroller.

Constructors

The only constructor for the autoscroller is protected:

protected DropTarget.DropTargetAutoScroller(Component c, Point p)

This creates an autoscroller for the component `c` and stores an initial value for the mouse location of `p`. (This value is used to determine if the mouse has moved since the last time an autoscroll was performed.)

Methods

Three methods are present to accomplish the business of starting a scroll, stopping a scroll, and updating the mouse's location:

public void actionPerformed(ActionEvent e)

The autoscroller works by timing the mouse. If it sits in an active autoscroll region long enough, an autoscroll is started. To do this, it uses a `Timer` object that reports its `ActionEvent` to this class. This method then calls the `autoscroll()` method of your component.

protected void stop()

This method stops the autoscroller's timer.

protected void updateLocation(Point newLocn)

This method keeps the autoscroller in sync with the mouse as it moves around and restarts the timer if necessary.

As with every great feature in Java, you will want to play around with the DnD package in small chunks. You should make tiny apps that accept or produce your `Transferable` information for quick testing. Once that's working, you can integrate them back into your real application and apply the finishing touches discussed earlier in this chapter.