

Drag and drop lets you move data from one place in an Adobe Flex application to another. This feature is especially useful in a visual application where you can drag data between two lists, drag controls in a container to reposition them, or drag Flex components between containers.

This topic describes the drag-and-drop operation, and how to add this functionality to your application.

Contents

About drag and drop	1045
Using drag-and-drop with list-based controls	1047
Manually adding drag-and-drop support	1054
Drag and drop examples	1067
Moving and copying data	1077

About drag and drop

Visual development environments typically let you manipulate objects in an application by selecting them with a mouse and moving them around the screen. Drag and drop lets you select an object, such as an item in a List control, or a Flex control such as an Image control, and then drag it over another component to add it to that component.

You can add support for drag and drop to all Flex components. Flex also includes built-in support for the drag-and-drop operation for certain controls such as [List](#), [Tree](#), and [DataGrid](#), that automate much of the processing required to support drag and drop.

About the drag-and-drop operation

The drag-and-drop operation has three main stages: initiation, dragging, and dropping:

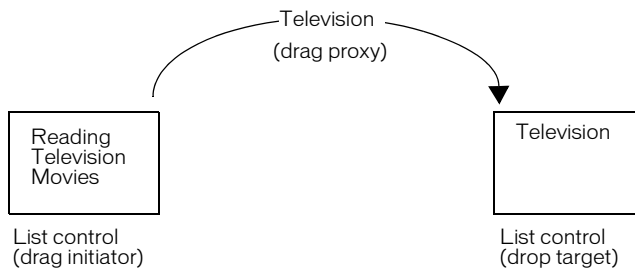
Initiation A user initiates a drag-and-drop operation by using the mouse to select a Flex component, or an item in a Flex component, and then moving the component or item while holding down the mouse button. For example, a user selects an item in a List control with the mouse and, while holding down the mouse button, moves the mouse several pixels. The selected component, the List control in this example, is the *drag initiator*.

Dragging While still holding down the mouse button, the user moves the mouse around the Flex application. Flex displays an image during the drag, called the *drag proxy*. A *drag source* object (an object of type DragSource) contains the data being dragged.

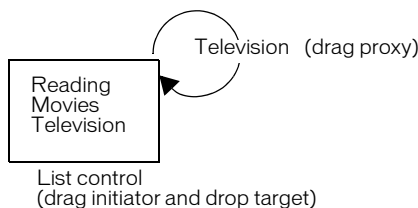
Dropping When the user moves the drag proxy over another Flex component, that component becomes a possible *drop target*. The drop target inspects the drag source object to determine whether the data is in a format that the target accepts and, if so, allows the user drop the data onto it. If the drop target determines that the data is not in an acceptable format, the drop target disallows the drop.

A drag-and-drop operation either copies or moves data from the drag initiator to the drop target. Upon a successful drop, Flex adds the data to the drop target and, optionally, deletes it from the drag initiator in the case of a move.

The following figure shows one List control functioning as the drag initiator and a second List control functioning as the drop target. In this example, you use drag and drop to move the 'Television' list item from the drag initiator to the drop target:



A single Flex component can function as both the drag initiator and the drop target. This lets you move the data within the component. The following example shows a List control functioning as both the drag initiator and the drop target:



By specifying the List control as both the drag initiator and the drop target, you can use drag and drop to rearrange the items in the control. For example, if you use a Canvas container as the drag initiator and the drop target, you can then use drag and drop to move controls in the Canvas container to rearrange them.

Performing a drag and drop

Drag and drop is event driven. To configure a component as a drag initiator or as a drop target, you have to write event handlers for specific events, such as the `dragDrop` and `dragEnter` events. For more information, see [“Manually adding drag-and-drop support” on page 1054](#).

For some components that you often use with drag and drop, Flex provides built-in event handlers to automate much of the drag and drop operation. These controls are all subclasses of the `ListBase` class, and are referred to as list-based controls. For more information, see [“Using drag-and-drop with list-based controls” on page 1047](#).

For a move operation, meaning you move the drag data from the drag initiator to the drop target, the list-based controls can handle all of the events required by a drag-and-drop operation. However, if you want to copy the drag data to the drop target, you have to write an event handler for both list-based and nonlist-based controls. For more information, see [“Moving and copying data” on page 1077](#).

Using drag-and-drop with list-based controls

The following controls include built-in support for the drag-and-drop operation:

- [DataGrid](#)
- [HorizontalList](#)
- [List](#)
- [PrintDataGrid](#)
- [TileList](#)
- [Tree](#)

The built-in support for these controls lets you move items by dragging them from a drag-enabled control to a drop-enabled control. However, to copy items, you must add additional logic. For more information, see [“Moving and copying data” on page 1077](#).

The following drag-and-drop example lets you move items from one [List](#) control to another:

```
<?xml version="1.0"?>
<!-- dragdrop\SimpleListToListMove.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  creationComplete="initApp();">

  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      private function initApp():void {
        srclist.dataProvider =
          new ArrayCollection(['Reading', 'Television', 'Movies']);
        destlist.dataProvider = new ArrayCollection([]);
      }
    ]]>
  </mx:Script>

  <mx:HBox>
    <mx:VBox>
      <mx:Label text="Available Activities"/>
      <mx:List id="srclist"
        allowMultipleSelection="true"
        dragEnabled="true"
        dragMoveEnabled="true"/>
    </mx:VBox>

    <mx:VBox>
      <mx:Label text="Activities I Like"/>
      <mx:List id="destlist"
        dropEnabled="true"/>
    </mx:VBox>
  </mx:HBox>
</mx:Application>
```

By setting the `dragEnabled` property to `true` on the first `List` and the `dropEnabled` property to `true` on the second `List` control, you enabled users to drag items from the first list to the second without worrying about any of the underlying event processing.

For all list classes except the `Tree` control, the default value of the `dragMoveEnabled` property is `false`, so you can only copy elements from one control to the other. By setting the `dragMoveEnabled` to `true` in the first `List` control, you can move and copy data. For the `Tree` control, the default value of the `dragMoveEnabled` property is `true`.

When the `dragMoveEnabled` property is set to `true`, the default drag-and-drop action is to move the drag data. To perform a copy, hold down the `Control` key during the drag-and-drop operation.

The only requirement on the drag and drop operation is that the structure of the data providers must match for the two controls. In this example, the data provider for `sclist` is an Array of Strings, and the data provider for the destination List control is an empty Array. If the data provider for `destlist` is an Array of some other type of data, `destlist` might not display the dragged data correctly.

You can modify the dragged data as part of a drag-and-drop operation to make the dragged data compatible with the destination. For an example of dragging data from one control to another when the data formats do not match, see [“Example: Copying data from a List control to a DataGrid control”](#) on page 1080.

Performing a two-way drag and drop

You can allow two-way drag and drop by setting the `dragEnabled`, `dropEnabled`, and `dragMoveEnabled` properties to `true` for both list-based controls, as the following example shows for two `DataGrid` controls. In this example, you can drag and drop rows from either `DataGrid` control to the other:

```
<?xml version="1.0"?>
<!-- dragdrop\SimpleDGTODG.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  creationComplete="initApp();">

  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      private function initApp():void {
        srcgrid.dataProvider = new ArrayCollection([
          {Artist:'Carole King', Album:'Tapestry', Price:11.99},
          {Artist:'Paul Simon', Album:'Graceland', Price:10.99},
          {Artist:'Original Cast', Album:'Camelot', Price:12.99},
          {Artist:'The Beatles', Album:'The White Album', Price:11.99}
        ]);

        destgrid.dataProvider = new ArrayCollection([]);
      }
    ]]>
  </mx:Script>

  <mx:HBox>
    <mx:VBox>
      <mx:Label text="Available Albums"/>
      <mx>DataGrid id="srcgrid"
        allowMultipleSelection="true"
        dragEnabled="true"
        dropEnabled="true"
        dragMoveEnabled="true">
        <mx:columns>
          <mx>DataGridColumn dataField="Artist"/>
          <mx>DataGridColumn dataField="Album"/>
          <mx>DataGridColumn dataField="Price"/>
        </mx:columns>
      </mx>DataGrid>
    </mx:VBox>

    <mx:VBox>
      <mx:Label text="Buy These Albums"/>
      <mx>DataGrid id="destgrid"
        allowMultipleSelection="true"
        dragEnabled="true"
      </mx>DataGrid>
    </mx:VBox>
  </mx:HBox>
</mx:Application>
```

```
        dropEnabled="true"  
        dragMoveEnabled="true">  
        <mx:columns>  
            <mx:DataGridColumn dataField="Artist"/>  
            <mx:DataGridColumn dataField="Album"/>  
            <mx:DataGridColumn dataField="Price"/>  
        </mx:columns>  
    </mx:DataGrid>  
</mx:VBox>  
</mx:HBox>  
</mx:Application>
```

Dragging and dropping in the same control

One use of drag and drop is to let you reorganize the items in a list-based control by dragging the items and then dropping them in the same control. In the next example, you define a Tree control, and let the user reorganize the nodes of the Tree control by dragging and dropping them. In this example, you set the `dragEnabled` and `dropEnabled` to true for the Tree control (the `dragMoveEnabled` property defaults to true for the Tree control):

```
<?xml version="1.0"?>
<!-- dragdrop\SimpleTreeSelf.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            // Initialize the data provider for the Tree.
            private function initApp():void {
                firstList.dataProvider = treeDP;
            }
        ]]>
    </mx:Script>

    <mx:XML id="treeDP">
        <node label="Mail">
            <node label="Inbox"/>
            <node label="Personal Folder">
                <node label="Demo"/>
                <node label="Personal"/>
                <node label="Saved Mail"/>
                <node label="bar"/>
            </node>
            <node label="Calendar"/>
            <node label="Sent"/>
            <node label="Trash"/>
        </node>
    </mx:XML>

    <mx:Tree id="firstList"
        showRoot="false"
        labelField="@label"
        dragEnabled="true"
        dropEnabled="true"
        allowMultipleSelection="true"
        creationComplete="initApp();"/>
</mx:Application>
```

Drag and drop properties for list-based controls

List-based controls provide properties and methods for managing the drag-and-drop operation. The following table lists these properties and methods:

Property/ Method	Description
<code>dropIndicatorSkin</code>	Specifies the name of the skin to use for the drop-insert indicator which shows where the dragged data will be inserted. The default value is <code>ListDropIndicator</code> .
<code>dragEnabled</code>	A Boolean value that specifies whether the control is a drag initiator. The default value is <code>false</code> . When <code>true</code> , users can drag selected items from the control. When a user drags items from the control, Flex creates a <code>DragSource</code> object that contains the following data objects: <ul style="list-style-type: none"> A copy of the selected item or items in the control. For all controls except for <code>Tree</code>, the format string is <code>"items"</code> and the items implement the <code>IDataProvider</code> interface. For <code>Tree</code> controls the format string is <code>"treeItems"</code> and the items implement the <code>ITreeDataProvider</code> API interface. A copy of the initiator, with a format String of <code>"source"</code>.
<code>dropEnabled</code>	A Boolean value that specifies whether the control can be a drop target that uses default values for handling items dropped onto it. The default value is <code>false</code> , which means that you must write event handlers for the drag events. When the value is <code>true</code> , you can drop items onto the control by using the default drop behavior. When you set <code>dropEnabled</code> to <code>true</code> , Flex automatically calls the <code>showDropFeedback()</code> and <code>hideDropFeedback()</code> methods to display the drop indicator.
<code>dragMoveEnabled</code>	If the value is <code>true</code> , and the <code>dragEnabled</code> property is <code>true</code> , specifies that you can move or copy items from the drag initiator to the drop target. When you move an item, the item is deleted from the drag initiator when you add it to the drop target. If the value is <code>false</code> , you can only copy an item to the drop target. For a copy, the item in the drag initiator is not affected by the drop. When the <code>dragMoveEnabled</code> property is <code>true</code> , you must hold down the <code>Control</code> key during the drop operation to perform a copy. The default value is <code>false</code> for all list controls except the <code>Tree</code> control, and <code>true</code> for the <code>Tree</code> control.
<code>calculateDropIndex</code>	Returns the item index in the drop target where the item will be dropped. Used by the <code>dragDrop</code> event handler to add the items in the correct location. Not available in the <code>TileList</code> or <code>HorizontalList</code> controls.

Property/ Method	Description
hideDropFeedback()	Hides drop target feedback and removes the focus rectangle from the target. You typically call this method from within the handler for the <code>dragExit</code> and <code>dragDrop</code> events.
showDropFeedback()	Specifies to display the focus rectangle around the target control and positions the drop indicator where the drop operation should occur. If the control has active scroll bars, hovering the mouse pointer over the control's top or bottom scrolls the contents. You typically call this method from within the handler for the <code>dragOver</code> event.

Manually adding drag-and-drop support

The list-based controls have built-in support for drag and drop but you can use drag and drop with any Flex component. To support drag-and-drop operations with non-list-based components, or to explicitly control drag and drop with list-based controls, you must handle the drag and drop events.

Classes used in drag-and-drop operations

You use the following classes to implement the drag-and-drop operation:

Class	Function
DragManager	Manages the drag-and-drop operations; for example, its <code>doDrag()</code> method starts the drag operation.
DragSource	Contains the data being dragged. It also provides additional drag management features, such as the ability to add a handler that is called when data is requested.
DragEvent	Represents the event object for all drag-and-drop events.

Drag-and-drop events for a drag initiator

A component that acts as a drag initiator handles the following events to manage the drag-and-drop operation:

Drag initiator event	Description	Handler required	Implemented by list controls
<code>mouseDown</code> and <code>mouseMove</code>	The <code>mouseDown</code> event is dispatched when the user selects a control with the mouse and holds down the mouse button. The <code>mouseMove</code> event is dispatched when the mouse moves. For most controls, you initiate the drag-and-drop operation in response to one of these events. For an example, see “Example: Handling the <code>dragOver</code> and <code>dragExit</code> events for the drop target” on page 1072.	Yes, for nonlist controls	No
<code>dragStart</code>	Dispatched by a list-based component when a drag operation starts. This event is used internally by the list-based controls; you do not handle it when implementing drag and drop. If you want to control the start of a drag-and-drop operation, use the <code>mouseDown</code> or <code>mouseMove</code> event.	Yes, for list controls	Yes
<code>dragComplete</code>	Dispatched when a drag operation completes, either when the drag data drops onto a drop target, or when the drag-and-drop operation ends without performing a drop operation. You can use this event to perform any final cleanup of the drag-and-drop operation. For example, if a user moves data from one component to another, you can use this event to delete the item from the drag initiator. For an example, see “Example: Moving and a copying data for a nonlist-based control” on page 1082.	No	Yes

When adding drag-and-drop support to a component, you must implement an event handler for either the `mouseDown` or `mouseMove` event, and optionally for the `dragComplete` event.

When you set the `dragEnabled` property to `true` for a list-based control, Flex automatically adds event handlers for the `dragStart` and `dragComplete` events.

NOTE

Do not add an event handler for the `dragStart` event. That is an internal event handled by Flex.

Drag-and-drop events for a drop target

To use a component as a drop target, you handle the following events:

Drop target event	Description	Handler required	Implemented by list controls
<code>dragEnter</code>	<p>Dispatched when a drag proxy moves over the drop target from outside the drop target. A component <i>must</i> define an event handler for this event to be a drop target. The event handler determines whether the data being dragged is in an accepted format. To accept the drop, the event handler calls the <code>DragManager.acceptDragDrop()</code> method. You must call the <code>DragManager.acceptDragDrop()</code> method for the drop target to receive the <code>dragOver</code>, <code>dragExit</code>, and <code>dragDrop</code> events. In the handler, you can change the appearance of the drop target to provide visual feedback to the user that the component can accept the drag operation. For example, you can draw a border around the drop target, or give focus to the drop target. For an example, see “Example: Simple drag-and-drop operation for a nonlist-based control” on page 1059.</p>	Yes	Yes
<code>dragOver</code>	<p>Dispatched while the user moves the mouse over the target, after the <code>dragEnter</code> event. You can handle this event to perform additional logic before allowing the drop operation, such as dropping data to various locations within the drop target, reading keyboard input to determine if the drag-and-drop operation is a move or copy of the drag data, or providing different types of visual feedback based on the type of drag-and-drop operation. For an example, see “Example: Handling the dragOver and dragExit events for the drop target” on page 1072.</p>	No	Yes

Drop target event	Description	Handler required	Implemented by list controls
<code>dragDrop</code>	Dispatched when the user releases the mouse over the drop target. Use this event handler to add the drag data to the drop target. For an example, see “Example: Simple drag-and-drop operation for a nonlist-based control” on page 1059.	Yes	Yes
<code>dragExit</code>	Dispatched when the user moves the drag proxy off of the drop target, but does not drop the data onto the target. You can use this event to restore the drop target to its normal appearance if you modified its appearance in response to a <code>dragEnter</code> event or other event. For an example, see “Example: Handling the dragOver and dragExit events for the drop target” on page 1072.	No	Yes

When adding drag-and-drop support to a nonlist-based component, you must implement an event handler for the `dragEnter` and `dragDrop` events, and optionally for the other events. When you set the `dropEnabled` property to `true` for a list-based control, Flex automatically adds event handlers for all events.

The drag-and-drop operation

The following steps define the drag-and-drop operation:

1. A component becomes a drag-and-drop initiator in either of the following ways:
 - List-based components with `dragEnabled=true`** Flex automatically makes the component an initiator when the user clicks and moves the mouse on the component.
 - Nonlist-based components, or list-based components with `dragEnabled=false`** The component must detect the user’s attempt to start a drag operation and explicitly become an initiator. Typically, you use the `mouseMove` or `mouseDown` event to start the drag-and-drop operation.
 - a. The component creates an instance of the [mx.core.DragSource](#) class that contains the data to be dragged, and specifies the format for the data.
 - b. The component calls the `mx.managers.DragManager.doDrag()` method, to initiate the drag-and-drop operation.

- While the mouse button is still pressed, the user moves the mouse around the application. Flex displays the drag proxy image in your application. The `DragManager.defaultDragImageSkin` property defines the default drag proxy image. You can define your own drag proxy image. For more information, see [“Example: Specifying the drag proxy” on page 1070](#).

NOTE

Releasing the mouse button when the drag proxy is not over a target ends the drag-and-drop operation. Flex generates a `DragComplete` event on the drag initiator, and the `DragManager.getFeedback()` method returns `DragManager.NONE`.

- If the user moves the drag proxy over a Flex component, Flex dispatches a `dragEnter` event for the component.

List-based components with `dropEnabled=true` Flex checks to see if the component can be a drop target.

Nonlist-based components, or list-based components with `dropEnabled=false` The component must define an event handler for the `dragEnter` event to be a drop target.

The `dragEnter` event handler can examine the `DragSource` object to determine whether the data being dragged is in an accepted format. To accept the drop, the event handler calls the `DragManager.acceptDragDrop()` method. You must call the `DragManager.acceptDragDrop()` method for the drop target to receive the `dragOver`, `dragExit`, and `dragDrop` events.

- If the drop target does not accept the drop, the drop target component's parent chain is examined to determine if any component in the chain accepts the drop data.
 - If the drop target or a parent component accepts the drop, Flex dispatches the `dragOver` event as the user moves the proxy over the target.
- (Optional) The drop target can handle the `dragOver` event. For example, the drop target can use this event handler to set the focus on itself.
 - (Optional) If the user decides not to drop the data onto the drop target and moves the drag proxy outside of the drop target without releasing the mouse button, Flex dispatches a `dragExit` event for the drop target. The drop target can optionally handle this event; for example, to undo any actions made in the `dragOver` event handler.
 - If the user releases the mouse while over the drop target, Flex dispatches a `dragDrop` event on the drop target.

List-based components with `dropEnabled=true` Flex automatically adds the drag data to the drop target. If this is a copy operation, you have to implement the event handler for the `dragDrop` event for a list-based control. For more information, see [“Example: Copying data from one List control to another List control” on page 1078](#).

Nonlist-based components, or list-based components with `dropEnabled=false` The drop target must define an event listener for the `dragDrop` event handler to add the drag data to the drop target.

7. (Optional) When the drop operation completes, Flex dispatches a `dragComplete` event. The drag initiator can handle this event; for example, to delete the drag data from the drag initiator in the case of a move.

List-based components with `dragEnabled=true` If this is a move operation, Flex automatically removes the drag data from the drag initiator.

Nonlist-based components, or list-based components with `dragEnabled=false` The drag initiator completes any final processing required. If this was a move operation, the event handler must remove the drag data from the drag initiator. For an example of writing the event handler for the `dragComplete` event, see [“Example: Moving and a copying data for a nonlist-based control” on page 1082](#).

Example: Simple drag-and-drop operation for a nonlist-based control

The following example lets you set the background color of a Canvas container by dropping either of two colors onto it. You are not copying or moving any data; instead, you are using the two drag initiators as a color palette. You then drag the color from one palette onto the drop target to set its background color.

The drag initiators, two Canvas containers, implement an event handler for the `mouseDown` event to initiate the drag and drop operation. This is the only event required to be handled by the drag initiator. The drop target is required to implement event handlers for the `dragEnter` and `dragDrop` events.

```
<?xml version="1.0"?>
<!-- dragdrop\DandDCanvas.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    backgroundColor="white">

    <mx:Script>
        <![CDATA[

            import mx.core.DragSource;
            import mx.managers.DragManager;
            import mx.events.*;
            import mx.containers.Canvas;

            // Initializes the drag and drop operation.
            private function mouseMoveHandler(event:MouseEvent):void {

                // Get the drag initiator component from the event object.
                var dragInitiator:Canvas=Canvas(event.currentTarget);

                // Get the color of the drag initiator component.
                var dragColor:int = dragInitiator.getStyle('backgroundColor');

                // Create a DragSource object.
                var ds:DragSource = new DragSource();

                // Add the data to the object.
                ds.addData(dragColor, 'color');

                // Call the DragManager doDrag() method to start the drag.
                DragManager.doDrag(dragInitiator, ds, event);
            }

            // Called when the user moves the drag proxy onto the drop target.
            private function dragEnterHandler(event:DragEvent):void {

                // Accept the drag only if the user is dragging data
                // identified by the 'color' format value.
                if (event.dragSource.hasFormat('color')) {

                    // Get the drop target component from the event object.
                    var dropTarget:Canvas=Canvas(event.currentTarget);
                    // Accept the drop.
                    DragManager.acceptDragDrop(dropTarget);
                }
            }
        ]]>
    </mx:Script>
</mx:Application>
```

```

// Called if the target accepts the dragged object and the user
// releases the mouse button while over the Canvas container.
private function dragDropHandler(event:DragEvent):void {

    // Get the data identified by the color format
    // from the drag source.
    var data:Object = event.dragSource.dataForFormat('color');
    // Set the canvas color.
    myCanvas.setStyle("backgroundColor", data);
}
]]>
</mx:Script>

<!-- A horizontal box with red and green canvases the user can drag -->
<mx:HBox>
    <mx:Canvas
        width="30" height="30"
        backgroundColor="red"
        borderStyle="solid"
        mouseMove="mouseMoveHandler(event);"/>
    <mx:Canvas
        width="30" height="30"
        backgroundColor="green"
        borderStyle="solid"
        mouseMove="mouseMoveHandler(event);"/>
</mx:HBox>

<mx:Label text="Drag a color onto the Canvas container."/>

<!-- Handles dragEnter and dragDrop events to allow dropping -->
<mx:Canvas id="myCanvas"
    width="100" height="100"
    backgroundColor="#FFFFFF"
    borderStyle="solid"
    dragEnter="dragEnterHandler(event);"
    dragDrop="dragDropHandler(event);"/>
</mx:Application>

```

The following sections describe the event handlers for the `mouseDown`, `dragEnter`, and `dragDrop` events.

Writing the `mouseDown` event handler

The event handler that initiates a drag-and-drop operation must do two things:

1. Create a [DragSource](#) object and initialize it with the drag data and the data format.

The `DragSource` object contains the drag data and a description of the drag data, called the data format. The event object for the `dragEnter` and `dragDrop` events contains a reference to this object in their `dragSource` property, which allows the event handlers to access the drag data.

You use the `DragSource.addData()` method to add the drag data and format to the `DragSource` object, where the `addData()` method has the following signature:

```
addData(data:Object, format:String):void
```

The `format` argument is a text string such as "color", "list data", or "employee record". In the event handler for the `dragEnter` event, the drop target examines this string to determine whether the data format matches the type of data that the drop target accepts. If the format matches, the drop target lets users drop the data on the target; if the format does not match, the target does not enable the drop operation.

One example of using the format string is when you have multiple components in your application that function as drop targets. Each drop target examines the `DragSource` object during its `dragEnter` event to determine if the drop target supports that format. For more information, see [“Handling the dragEnter event” on page 1063](#).

NOTE

The list controls have predefined values for `format` argument. For all list controls other than the `Tree` control, the format String is "items". For the `Tree` control, the format String is "treeItems". For more information, see [“Using drag-and-drop with list-based controls” on page 1047](#).

If you drag large or complex data items, consider creating a handler to copy the data, and specify it by calling the `DragSource.addListener()` method instead of using the `DragSource.addData()` method. If you do this, the data does not get copied until the user drops it, which avoids the processing overhead of copying the data if a user starts dragging data but never drops it. The implementation of the list-based classes use this technique.

2. Call the `DragManager.doDrag()` method to start the drag-and-drop operation.

The `doDrag()` method has the following signature:

```
doDrag(
    dragInitiator:IUIComponent,
    dragSource:DragSource,
    mouseEvent:MouseEvent,
    dragImage:IFlexDisplayObject = null,
    xOffset:Number = 0, yOffset:Number = 0,
    imageAlpha:Number = 0.5,
    allowMove:Boolean = true):void
```

The `doDrag()` method requires three arguments: a reference to the component that initiates the drag operation (identified by the `event.currentTarget` object); the `DragSource` object that you created in step 1, and the event object passed to the event handler.

Optional arguments specify the drag proxy image and the characteristics of the image. For an example that specifies a drag proxy, see [“Example: Specifying the drag proxy” on page 1070](#).

Handling the `dragEnter` event

Flex generates a `dragEnter` event when the user moves the drag proxy over any control. A control *must* define a handler for a `dragEnter` event to be a drop target. The event handler typically performs the following actions:

- Use the format information in the `DragSource` object to determine whether the drag data is in a format accepted by the drop target.
- If the drag data is in a compatible format, the handler *must* call the `DragManager.acceptDragDrop()` method to enable the user to drop the data on the drop target. If the event handler does not call this method, the user cannot drop the data and the drop target will not receive the `dragOver`, `dragExit`, and `dragDrop` events.
- Optionally, perform any other actions necessary when the user first drags a drag proxy over a drop target.

The value of the `action` property of the event object for the `dragEnter` event is `DragManager.MOVE`, even if you are doing a copy. This is because the `dragEnter` event occurs before the drop target recognizes that the `Control` key is pressed to signal a copy.

The Flex default event handler for the `dragOver` event for a list-based control automatically sets the `action` property. For nonlist-based controls, or if you explicitly handle the `dragOver` event for a list-based control, use the `DragManager.showFeedback()` method to set the `action` property to a value that signifies the type of drag operation: `DragManager.COPY`, `DragManager.LINK`, `DragManager.MOVE`, or `DragManager.NONE`. For more information on the `dragOver` event, see [“Example: Handling the `dragOver` and `dragExit` events for the drop target” on page 1072](#).

Handling the `dragDrop` event

The `dragDrop` event occurs when the user releases the mouse to drop data on a target, and the `dragEnter` event handler has called the `DragManager.acceptDragDrop()` method to accept the drop. You must define a handler for the event to add the drag data to the drop target.

The event handler uses the `DragSource.dataForFormat()` method to retrieve the drag data. In the previous example, the drag data contains the new background color of the drop target. The event handler then calls `setStyle()` to set the background color of the drop target.

Example: handling drag and drop events in a list-based control

Flex automatically defines default event handlers for the drag-and-drop events when you set `dragEnabled` or `dropEnabled` property to `true` for a list-based control. You can either use these default event handlers, which requires you to do no additional work in your application, or define your own event handlers.

There are three common scenarios for using event handlers with the list-based controls:

Use the default event handlers When you set `dragEnabled` to `true` for a drag initiator, or when you set `dropEnabled` to `true` for a drop target, Flex handles all drag-and-drop events for you for a move. You only have to define your own `dragDrop` event handler when you want to copy data as part of the drag-and-drop operation. For more information, see [“Moving and copying data” on page 1077](#).

Define your own event handlers If you want to control the drag-and-drop operation for a list-based control, you can explicitly handle the drag-and-drop events, just as you can for any component. In this scenario, set the `dragEnabled` property to `false` for a drag initiator, or set the `dropEnabled` property to `false` for a drop target. For more information on handling these events, see [“Example: Simple drag-and-drop operation for a nonlist-based control” on page 1059](#).

Define your own event handlers and use the default event handlers You might want to add your own event handler for a drag-and-drop event, and also use the build in drag-and-drop handlers. In this case, your event handler executes first, then the default event handler provided by Flex executes. If, for any reason, you want to explicitly prohibit the execution of the default event handler, call the `Event.preventDefault()` method from within your event handler.

NOTE

If you call `Event.preventDefault()` in the event handler for the `dragComplete` or `dragDrop` event for a `Tree` control when dragging data from one `Tree` control to another, it prevents the drop.

Because of the way data to a Tree control is structured, the Tree control handles drag and drop differently from the other list-based controls. For the Tree control, the event handler for the `dragDrop` event only performs an action when you drag and drop data within the same Tree control. If you drag data from one Tree control and drop it onto another Tree control, the event handler for the `dragComplete` event actually performs the work to add the data to the destination Tree control, rather than the event handler for the `dragDrop` event, and also removes the data from the source Tree control for a move operation.

Therefore, if you call `Event.preventDefault()` in the event handler for the `dragDrop` or `dragComplete` events, you implement the drop behavior yourself. For more information, see [“Example: Moving and a copying data for a nonlist-based control” on page 1082](#).

The following example modifies the example shown in the section [“Performing a two-way drag and drop” on page 1050](#) to define an event handler for the `dragDrop` event that accesses the data dragged from one `DataGrid` control to another. This event handler is executed before the default event handler for the `dragDrop` event to display in an `Alert` control the `Artist` field of each `DataGrid` row dragged from the drag initiator to the drop target:

```
<?xml version="1.0"?>
<!-- dragdrop\simpleDGToDGAAlert.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  creationComplete="initApp();">

  <mx:Script>
    <![CDATA[
      import mx.events.DragEvent;
      import mx.controls.Alert;
      import mx.collections.ArrayCollection;

      private function initApp():void {
        srcgrid.dataProvider = new ArrayCollection([
          {Artist:'Carole King', Album:'Tapestry', Price:11.99},
          {Artist:'Paul Simon', Album:'Graceland', Price:10.99},
          {Artist:'Original Cast', Album:'Camelot', Price:12.99},
          {Artist:'The Beatles', Album:'The White Album', Price:11.99}
        ]);

        destgrid.dataProvider = new ArrayCollection([]);
      }

      // Define the event listener.
      public function dragDropHandler(event:DragEvent):void {
        // dataForFormat() always returns an Array
        // for the list-based controls
        // in case multiple items were selected.
        var dragObj:Array=
          event.dragSource.dataForFormat("items") as Array;

        // Get the Artist for all dragged albums.
        var artistList:String='';
        for (var i:Number = 0; i < dragObj.length; i++) {
          artistList+='Artist: ' + dragObj[i].Artist + '\n';
        }

        Alert.show(artistList);
      }
    ]]>
  </mx:Script>

  <mx:HBox>
    <mx:VBox>
```

```

    <mx:Label text="Available Albums"/>
    <mx:DataGrid id="srcgrid"
      allowMultipleSelection="true"
      dragEnabled="true"
      dropEnabled="true"
      dragMoveEnabled="true">
      <mx:columns>
        <mx:DataGridColumn dataField="Artist" />
        <mx:DataGridColumn dataField="Album" />
        <mx:DataGridColumn dataField="Price" />
      </mx:columns>
    </mx:DataGrid>
  </mx:VBox>

  <mx:VBox>
    <mx:Label text="Buy These Albums"/>
    <mx:DataGrid id="destgrid"
      allowMultipleSelection="true"
      dragEnabled="true"
      dropEnabled="true"
      dragMoveEnabled="true"
      dragDrop="dragDropHandler(event);">
      <mx:columns>
        <mx:DataGridColumn dataField="Artist" />
        <mx:DataGridColumn dataField="Album" />
        <mx:DataGridColumn dataField="Price" />
      </mx:columns>
    </mx:DataGrid>
  </mx:VBox>
</mx:HBox>
</mx:Application>

```

Notice that the `dataFormat()` method specifies an argument value of "items". This is because the list-based controls have predefined values for the data format of drag data. For all list controls other than the Tree control, the format String is "items". For the Tree control, the format String is "treeItems".

Notice also that the return value of the `dataFormat()` method is an Array. The `dataFormat()` method always returns an Array for a list-based control, even if you are only dragging a single item, because list-based controls let you select multiple items.

Drag and drop examples

This section contains several drag-and-drop examples.

Example: Using a container as a drop target

To use a container as a drop target, you must use the `backgroundColor` property of the container to set a color. Otherwise, the background color of the container is transparent, and the Drag and Drop Manager is unable to detect that the mouse pointer is on a possible drop target.

In the following example, you use the `<mx:Image>` tag to load a draggable image into a [Canvas](#) container. you then add event handlers to let the user drag the Image control within the Canvas container to reposition it:

```
<?xml version="1.0"?>
<!-- dragdrop\DandDImage.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            //Import classes so you don't have to use full names.
            import mx.managers.DragManager;
            import mx.core.DragSource;
            import mx.events.DragEvent;
            import flash.events.MouseEvent;

            // Embed icon image.
            [Embed(source='assets/globe.jpg')]
            public var globeImage:Class;

            // The mouseMove event handler for the Image control
            // initiates the drag-and-drop operation.
            private function mouseMoveHandler(event:MouseEvent):void
            {
                var dragInitiator:Image=Image(event.currentTarget);
                var ds:DragSource = new DragSource();
                ds.addData(dragInitiator, "img");

                DragManager.doDrag(dragInitiator, ds, event);
            }

            // The dragEnter event handler for the Canvas container
            // enables dropping.
            private function dragEnterHandler(event:DragEvent):void {
                if (event.dragSource.hasFormat("img"))
                {
                    DragManager.acceptDragDrop(Canvas(event.currentTarget));
                }
            }

            // The dragDrop event handler for the Canvas container
            // sets the Image control's position by
            // "dropping" it in its new location.
            private function dragDropHandler(event:DragEvent):void {
                Image(event.dragInitiator).x =
                    Canvas(event.currentTarget).mouseX;
                Image(event.dragInitiator).y =
                    Canvas(event.currentTarget).mouseY;
            }
        ]]>
    </mx:Script>
</mx:Application>
```

```

</mx:Script>

<!-- The Canvas is the drag target -->
<mx:Canvas id="v1"
  width="500" height="500"
  borderStyle="solid"
  backgroundColor="#DDDDDD"
  dragEnter="dragEnterHandler(event);"
  dragDrop="dragDropHandler(event);">

  <!-- The image is the drag initiator. -->
  <mx:Image id="myimg"
    source="@Embed(source='assets/globe.jpg')"
    mouseMove="mouseMoveHandler(event);"/>
</mx:Canvas>
</mx:Application>

```

Example: Specifying the drag proxy

In the event handler for the `mouseDown` or `mouseUp` event, you can optionally specify a drag proxy in the `doDrag()` method of the [DragManager](#) class. If you do not specify a drag proxy, Flex uses a default proxy image. The `doDrag()` method takes the following optional arguments to specify the drag proxy and its properties.

Argument	Description
<i>dragImage</i>	The image that specifies the drag proxy. To specify a symbol, such as a JPEG image of a product that a user wants to order, use a string that specifies the symbol's name, such as <code>myImage.jpg</code> . To specify a component, such as a Flex container or control, create an instance of the control or container, configure and size it, and then pass it as an argument to the <code>doDrag()</code> method.
<i>xOffset</i>	Number that specifies the x offset, in pixels, for the <code>dragImage</code> . This argument is optional. If omitted, the drag proxy is shown at the upper-left corner of the drag initiator. The offset is expressed in pixels from the left edge of the drag proxy to the left edge of the drag initiator, and is usually a negative number.
<i>yOffset</i>	Number that specifies the y offset, in pixels, for the <code>dragImage</code> . This argument is optional. If omitted, the drag proxy is shown at the upper-left corner of the drag initiator. The offset is expressed in pixels from the top edge of the drag proxy to the top edge of the drag initiator, and is usually a negative number.
<i>imageAlpha</i>	A Number that specifies the alpha value used for the drag proxy image. If omitted, Flex uses an alpha value of 0.5. A value of 0 corresponds to transparent and a value of 1.0 corresponds to fully opaque.

You must specify a size for the drag proxy image, otherwise it does not appear. The following example modifies the example in [“Example: Using a container as a drop target” on page 1068](#) to use a 15 pixel by 15 pixel [Image](#) control as the drag proxy:

```
<?xml version="1.0"?>
<!-- dragdrop\DandDImage.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            // Import classes so you don't have to use full names.
            import mx.managers.DragManager;
            import mx.core.DragSource;
            import mx.events.DragEvent;
            import flash.events.MouseEvent;

            // Embed icon image.
            [Embed(source='assets/globe.jpg')]
            public var globeImage:Class;

            // The mouseMove event handler for the Image control
            // initiates the drag-and-drop operation.
            private function mouseOverHandler(event:MouseEvent):void
            {
                var dragInitiator:Image=Image(event.currentTarget);
                var ds:DragSource = new DragSource();
                ds.addData(dragInitiator, "img");

                // The drag manager uses the Image control
                // as the drag proxy and sets the alpha to 1.0 (opaque),
                // so it appears to be dragged across the Canvas.
                var imageProxy:Image = new Image();
                imageProxy.source = globeImage;
                imageProxy.height=15;
                imageProxy.width=15;
                DragManager.doDrag(dragInitiator, ds, event,
                    imageProxy, -15, -15, 1.00);
            }

            // The dragEnter event handler for the Canvas container
            // enables dropping.
            private function dragEnterHandler(event:DragEvent):void {
                if (event.dragSource.hasFormat("img"))
                {
                    DragManager.acceptDragDrop(Canvas(event.currentTarget));
                }
            }

            // The dragDrop event handler for the Canvas container
            // sets the Image control's position by
```

```

        // "dropping" it in its new location.
        private function dragDropHandler(event:DragEvent):void {
            Image(event.dragInitiator).x =
                Canvas(event.currentTarget).mouseX;
            Image(event.dragInitiator).y =
                Canvas(event.currentTarget).mouseY;
        }
    ]]>
</mx:Script>

<!-- The Canvas is the drag target -->
<mx:Canvas id="v1"
    width="500" height="500"
    borderStyle="solid"
    backgroundColor="#DDDDDD"
    dragEnter="dragEnterHandler(event);"
    dragDrop="dragDropHandler(event);">

    <!-- The image is the drag initiator. -->
    <mx:Image id="myimg"
        source="@Embed(source='assets/globe.jpg')"
        mouseMove="mouseOverHandler(event);"/>
</mx:Canvas>
</mx:Application>

```

To use a control with specific contents, such as a [VBox](#) control with a picture and label, you must create a custom component that contains the control or controls, and use an instance of the component as the *dragProxy* argument.

Example: Handling the dragOver and dragExit events for the drop target

The `dragOver` event occurs when the user moves the mouse over a drag-and-drop target whose `dragEnter` event handler has called the `DragManager.acceptDragDrop()` method. This event is dispatched continuously as the user drags the mouse over the target. The `dragOver` event handler is optional; you do not have to define it to perform a drag-and-drop operation.

The `dragOver` event is useful for specifying the visual feedback that the user gets when the mouse is over a drop target. For example, you can use the `DragManager.showFeedback()` method to specify the drag-feedback indicator that appears within the drag proxy image. This method uses four constant values for the argument, as the following table shows:

Argument value	Icon
<code>DragManager.COPY</code>	A green circle with a white plus sign indicating that you can perform the drop.
<code>DragManager.LINK</code>	A grey circle with a white arrow sign indicating that you can perform the drop.
<code>DragManager.MOVE</code>	A plain arrow indicating that you can perform the drop.
<code>DragManager.NONE</code>	A red circle with a white x appears indicating that a drop is prohibited. This is the same image that appears when the user drags over an object that is not a drag target.

You typically show the feedback indicator based on the keys pressed by the user during the drag-and-drop operation. The `DragEvent` object for the `dragOver` event contains Boolean properties that indicate whether the Control or Shift keys are pressed at the time of the event: `ctrlKey` and `shiftKey`, respectively. No key pressed indicates a move, the Control key indicates a copy, and the Shift key indicates a link. You then call the `showFeedback()` method as appropriate for the key pressed.

Another use of the `showFeedback()` method is that it determines the value of the `action` property of the `DragEvent` object for the `dragDrop`, `dragExit`, and `dragComplete` events. If you do not call the `showFeedback()` method in the `dragOver` event handler, the `action` property of the `DragEvent` is always set to `DragManager.MOVE`.

The `dragExit` event is dispatched when the user drags the drag proxy off the drop target, but does not drop the data onto the target. You can use this event to restore any visual changes that you made to the drop target in the `dragOver` event handler.

In the following example, you set the `dropEnabled` property of a List control to `true` to configure it as a drop target and to use the default event handlers. However, you want to provide your own visual feedback, so you also define event handlers for the `dragEnter`, `dragExit`, and `dragDrop` events. The `dragOver` event handler completely overrides the default event handler, so you call the `Event.preventDefault()` method to prohibit the default event handler from execution.

The `dragOver` event handler determines whether the user is pressing a key while dragging the proxy over the target, and sets the feedback appearance based on the key that is pressed. The `dragOver` event handler also sets the border color of the drop target to green to indicate that it is a viable drop target, and uses the `dragExit` event handler to restore the original border color.

For the `dragExit` and `dragDrop` handlers, you only want to remove any visual changes that you made in the `dragOver` event handlers, but otherwise you want to rely on the default Flex event handlers. Therefore, these event handlers do not call the `Event.preventDefault()` method:

```
<?xml version="1.0"?>
<!-- dragdrop\DandDListToListShowFeedback.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  creationComplete="initApp();">

  <mx:Script>
    <![CDATA[
      import mx.managers.DragManager;
      import mx.events.DragEvent;
      import mx.collections.ArrayCollection;

      private function initApp():void {
        firstList.dataProvider = new ArrayCollection([
          {label:"First", data:"1"},
          {label:"Second", data:"2"},
          {label:"Third", data:"3"},
          {label:"Fourth", data:"4"}
        ]);
        secondList.dataProvider = new ArrayCollection([]);
      }

      // Variable to store original border color.
      private var tempBorderColor:uint;

      // Flag to indicate that tempBorderColor has been set.
      private var borderColorSet:Boolean = false;

      private function dragOverHandler(event:DragEvent):void {

        // Explicitly handle the dragOver event.
        event.preventDefault();

        // Since you are explicitly handling the dragOver event,
        // call showDropFeedback(event) to have the drop target
        // display the drop indicator.
        // The drop indicator is removed
        // automatically for the list controls by the built-in
        // event handler for the dragDrop event.
        event.currentTarget.showDropFeedback(event);

        if (event.dragSource.hasFormat("items"))
        {
          // Set the border to green to indicate that
          // this is a drop target.
          // Since the dragOver event is dispatched continuously
```

```

        // as you move over the drop target, only set it once.
        if (borderColorSet == false) {
            tempBorderColor =
                event.currentTarget.getStyle('borderColor');
            borderColorSet = true;
        }

        // Set the drag-feedback indicator based on the
        // type of drag-and-drop operation.
        event.currentTarget.setStyle('borderColor', 'green');
        if (event.ctrlKey) {
            DragManager.showFeedback(DragManager.COPY);
            return;
        }
        else if (event.shiftKey) {
            DragManager.showFeedback(DragManager.LINK);
            return;
        }
        else {
            DragManager.showFeedback(DragManager.MOVE);
            return;
        }
    }

    // Drag not allowed.
    DragManager.showFeedback(DragManager.NONE);
}

private function dragDropHandler(event:DragEvent):void {
    dragExitHandler(event);
}

// Restore the border color.
private function dragExitHandler(event:DragEvent):void {
    event.currentTarget.setStyle('borderColor', tempBorderColor);
    borderColorSet = true;
}
]]>
</mx:Script>

<mx:HBox id="myHB">
    <mx:List id="firstList"
        dragEnabled="true"/>

    <mx:List id="secondList"
        borderThickness="2"
        dropEnabled="true"
        dragMoveEnabled="true"
        dragOver="dragOverHandler(event);"
        dragDrop="dragExitHandler(event);"

```

```
        dragExit="dragExitHandler(event);"/>  
    </mx:HBox>  
</mx:Application>
```

Moving and copying data

This section describes how to implement a move and a copy as part of a drag-and-drop operation.

About moving data

When you move data, you add it to the drop target and delete it from the drag initiator. You use the `dragDrop` event for the drop target to add the data, and the `dragComplete` event for the drag initiator to remove the data.

How much work you have to do to implement the move depends on whether the drag initiator and drop target are list-based controls or nonlist-based controls:

list-based control You do not have to do any additional work; list-based controls handle all of the processing required to move data from one list-based control to another list-based control. For an example, see [“Performing a drag and drop” on page 1047](#).

nonlist-based control If the drag initiator is a nonlist-based control, you have to implement the event handler for the `dragComplete` event to delete the drag data from the drag initiator. If the drop target is a nonlist-based control, you have to implement the event handler for the `dragDrop` event to add the data to the drop target. For an example, see [“Example: Moving and a copying data for a nonlist-based control” on page 1082](#).

About copying data

The list-based controls can automate all of the drag-and-drop operation except for when you copy the drag data to the drop target, rather than move it. If you want to copy data when using a list-based control as the drop target, you must explicitly handle the `dragDrop` event for the drop target.

When using a nonlist-based control as the drop target, you always have to write an event handler for the `dragDrop` event, regardless of whether you are performing a move or copy.

Copying data in an object-oriented environment is not a trivial task. An object may contain pointers to other objects that themselves contain pointers to even more objects. Rather than try to define a universal object copy as part of the drag-and-drop operation, Flex leaves it to you to implement object copying because you will have first-hand knowledge of your data format and requirements.

In some circumstances, you may have objects that implement a clone method that makes it easy to create a byte copy of the object. In other cases, you will have to perform the copy yourself by copying individual fields of the source object to the destination object.

Example: Copying data from one List control to another List control

The following example lets you copy items from one [List](#) control to another: In this example, even though you set the `dropEnabled` property to `true` in the drop target, you still write an event handler for the `dragDrop` event. The event handler calls `Event.preventDefault()` to explicitly prohibit the default event handler from executing. The reason to set the `dropEnabled` property to `true` in the drop target is so that Flex automatically handles all of the other events (`dragEnter`, `dragOver`, and `dragExit`) on the drop target.

NOTE

You call `Event.preventDefault()` because the drop target is a List control, one of the list-based controls that defines a default `dragDrop` handler. In the case of a copy, you want to explicitly handle the `dragDrop` event, but use all the other default event handlers. For a nonlist-based component, you do not have to call `Event.preventDefault()` because only list-based controls define default drag-and-drop event handlers.

The default value of the `dragMoveEnabled` property is `false`, so that you can only copy elements from one List control to the other. If you modify the example to set `dragMoveEnabled` to `true` in the drag initiator, you can move and copy elements. To copy a list element, hold down the Control key during the drag-and-drop operation:

```
<?xml version="1.0"?>
<!-- dragdrop\DandDListToListCopy.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  creationComplete="initApp();">

  <mx:Script>
    <![CDATA[
      import mx.events.DragEvent;
      import mx.managers.DragManager;
      import mx.core.DragSource;
      import mx.collections.IList;
      import mx.collections.ArrayCollection;

      private function initApp():void {
        firstList.dataProvider = new ArrayCollection([
          {label:"First", data:"1"},
          {label:"Second", data:"2"},
          {label:"Third", data:"3"},
          {label:"Fourth", data:"4"},
        ]);
        secondList.dataProvider = new ArrayCollection([]);
      }

      private function dragDropHandler(event:DragEvent):void {
        if (event.dragSource.hasFormat("items"))
        {
          // Explicitly handle the dragDrop event.
          event.preventDefault();

          // Since you are explicitly handling the dragDrop event,
          // call hideDropFeedback(event) to have the drop target
          // hide the drop indicator.
          // The drop indicator is created
          // automatically for the list controls by the built-in
          // event handler for the dragOver event.
          event.currentTarget.hideDropFeedback(event);

          // Get drop target.
          var dropTarget:List=List(event.currentTarget);

          // Get the dragged item from the drag initiator.
          // The List control always writes an Array
          // to the dragSource object,
          // even if there is only one item being dragged.
          var itemsArray:Array =
            event.dragSource.dataForFormat("items") as Array;

          // Copy the dragged data into a new Object.
          var tempItem:Object =
```

```

        {label: itemsArray[0].label, data: itemsArray[0].data};

        // Get the drop location in the destination.
        var dropLoc:int = dropTarget.calculateDropIndex(event);

        // Add the new object to the drop target.
        IList(dropTarget.dataProvider).addItemAt(tempItem, dropLoc);
    }
}
]]>
</mx:Script>

<mx:HBox>
    <mx:List id="firstList"
        dragEnabled="true"/>

    <mx:List id="secondList"
        dropEnabled="true"
        dragDrop="dragDropHandler(event);"/>
</mx:HBox>
</mx:Application>

```

To perform the copy, the event handler creates a new Object, then copies the individual data fields of the drag data into the new Object. Then, the event handler adds the new Object to the drop target's data provider.

Unlike the example in the section [“Example: Moving and a copying data for a nonlist-based control” on page 1082](#), you do not have to write the `dragOver` or `dragComplete` event handlers. While Flex cannot perform the copy, Flex does recognize when you perform a copy or a move, and automatically removes the data from the drag initiator for you on a move.

Example: Copying data from a List control to a DataGrid control

You can use drag and drop to copy data between two different types of controls, or between controls that use different data formats. To handle this situation, you write an event handler for the `dragDrop` event that converts the data from the format of the drag initiator to the format required by the drop target.

In the following example, you can move or copy data from a List control to a DataGrid control. The event handler for the `dragDrop` event adds a new field to the dragged data that contains the date:

```
<?xml version="1.0"?>
<!-- dragdrop\DandDListToDG.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="initApp();">

    <mx:Script>
        <![CDATA[
            import mx.events.DragEvent;
            import mx.managers.DragManager;
            import mx.core.DragSource;
            import mx.collections.IList;
            import mx.collections.ArrayCollection;

            private function initApp():void {
                srcList.dataProvider = new ArrayCollection([
                    {label:"First", data:"1"},
                    {label:"Second", data:"2"},
                    {label:"Third", data:"3"},
                    {label:"Fourth", data:"4"},
                ]);

                destDG.dataProvider = new ArrayCollection([]);
            }

            private function dragDropHandler(event:DragEvent):void {
                if (event.dragSource.hasFormat("items"))
                {
                    // Explicitly handle the dragDrop event.
                    event.preventDefault();

                    // Since you are explicitly handling the dragDrop event,
                    // call hideDropFeedback(event) to have the drop target
                    // hide the drop indicator.
                    // The drop indicator is created
                    // automatically for the list controls by the built-in
                    // event handler for the dragOver event.
                    event.currentTarget.hideDropFeedback(event);

                    // Get drop target.
                    var dropTarget:DataGrid =
                        DataGrid(event.currentTarget);

                    var itemsArray:Array =
                        event.dragSource.dataForFormat('items') as Array;
                    var tempItem:Object =
                        { label: itemsArray[0].label,
```

```

        data: itemsArray[0].data,
        date: new Date()
    });

    // Get the drop location in the destination.
    var dropLoc:int = dropTarget.calculateDropIndex(event);

    IList(dropTarget.dataProvider).addItemAt(tempItem, dropLoc);
    }
}
]]>
</mx:Script>

<mx:HBox>
    <mx:List id="srcList"
        dragEnabled="true"
        dragMoveEnabled="true"/>

    <mx:DataGrid id="destDG"
        dropEnabled="true"
        dragDrop="dragDropHandler(event);">
        <mx:columns>
            <mx:DataGridColumn dataField="label"/>
            <mx:DataGridColumn dataField="data"/>
            <mx:DataGridColumn dataField="date"/>
        </mx:columns>
    </mx:DataGrid>
</mx:HBox>
</mx:Application>

```

Example: Moving and a copying data for a nonlist-based control

The `dragComplete` event occurs on the drag initiator when a drag operation completes, either when the drag data drops onto a drop target, or when the drag-and-drop operation ends without performing a drop operation. The drag initiator can specify a handler to perform cleanup actions when the drag finishes; or when the target does not accept the drop.

One use of the `dragComplete` event handler is to remove from the drag initiator the objects that you move to the drop target. The items that you drag from a control are copies of the original items, not the items themselves. Therefore, when you drop items onto the drop target, you use the `dragComplete` event handler to delete them from the drag initiator.

To determine the type of drag operation (copy or move), you use the `action` property of the event object passed to the event handler. This method returns the drag feedback set by the `dragOver` event handler. For more information, see [“Example: Handling the dragOver and dragExit events for the drop target” on page 1072.](#)

In the following example, you drag an Image control from one Canvas container to another. As part of the drag-and-drop operation, you can move the Image control, or copy it by holding down the Control key. If you perform a move, the `dragComplete` event handler removes the Image control from its original parent container:

```
<?xml version="1.0"?>
<!-- dragdrop\DandDImageCopyMove.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="horizontal">

    <mx:Script>
        <![CDATA[
            import mx.managers.DragManager;
            import mx.core.DragSource;
            import mx.events.DragEvent;
            import flash.events.MouseEvent;

            // Embed icon image.
            [Embed(source='assets/globe.jpg')]
            public var globeImage:Class;

            // The mouseMove event handler for the Image control
            // functioning as the drag initiator.
            private function mouseOverHandler(event:MouseEvent):void
            {
                var dragInitiator:Image=Image(event.currentTarget);
                var ds:DragSource = new DragSource();
                ds.addData(dragInitiator, "img");

                // The drag manager uses the image as the drag proxy
                // and sets the alpha to 1.0 (opaque),
                // so it appears to be dragged across the canvas.
                var imageProxy:Image = new Image();
                imageProxy.source = globeImage;
                imageProxy.height=10;
                imageProxy.width=10;
                DragManager.doDrag(dragInitiator, ds, event,
                    imageProxy, -15, -15, 1.00);
            }

            // The dragEnter event handler for the Canvas container
            // functioning as the drop target.
            private function dragEnterHandler(event:DragEvent):void {
                if (event.dragSource.hasFormat("img"))
                    DragManager.acceptDragDrop(Canvas(event.currentTarget));
            }

            // The dragOver event handler for the Canvas container
            // sets the type of drag-and-drop
            // operation as either copy or move.
```

```
// This information is then used in the
// dragComplete event handler for the source Canvas container.
private function dragOverHandler(event:DragEvent):void
{
    if (event.dragSource.hasFormat("img")) {
        if (event.ctrlKey) {
            DragManager.showFeedback(DragManager.COPY);
            return;
        }
        else {
            DragManager.showFeedback(DragManager.MOVE);
            return;
        }
    }

    DragManager.showFeedback(DragManager.NONE);
}

// The dragDrop event handler for the Canvas container
// sets the Image control's position by
// "dropping" it in its new location.
private function dragDropHandler(event:DragEvent):void {
    if (event.dragSource.hasFormat("img")) {
        var draggedImage:Image =
            event.dragSource.dataForFormat('img') as Image;
        var dropCanvas:Canvas = event.currentTarget as Canvas;

        // Since this is a copy, create a new object to
        // add to the drop target.
        var newImage:Image=new Image();
        newImage.source = draggedImage.source;
        newImage.x = dropCanvas.mouseX;
        newImage.y = dropCanvas.mouseY;
        dropCanvas.addChild(newImage);
    }
}

// The dragComplete event handler for the source Canvas container
// determines if this was a copy or move.
// If a move, remove the dragged image from the Canvas.
private function dragCompleteHandler(event:DragEvent):void {
    var draggedImage:Image =
        event.dragInitiator as Image;
    var dragInitCanvas:Canvas =
        event.dragInitiator.parent as Canvas;

    if (event.action == DragManager.MOVE)
        dragInitCanvas.removeChild(draggedImage);
}
]]>
```

```
</mx:Script>

<!-- Canvas holding the Image control that is the drag initiator. -->
<mx:Canvas
    width="250" height="500"
    borderStyle="solid"
    backgroundColor="#DDDDDD">

    <!-- The Image control is the drag initiator and the drag proxy. -->
    <mx:Image id="myimg"
        source="@Embed(source='assets/globe.jpg')"
        mouseMove="mouseOverHandler(event);"
        dragComplete="dragCompleteHandler(event);"/>
</mx:Canvas>

<!-- This Canvas is the drop target. -->
<mx:Canvas
    width="250" height="500"
    borderStyle="solid"
    backgroundColor="#DDDDDD"
    dragEnter="dragEnterHandler(event);"
    dragOver="dragOverHandler(event);"
    dragDrop="dragDropHandler(event);">
</mx:Canvas>
</mx:Application>
```

