

Student Guide

Introduction to Adobe Flex 3

Guide to Drag and Drop

ChikaraDev Tutoring

Trademarks

ChikaraDev and the ChikaraDev logo are trademarks of ChikaraDev. Such trademarks may be registered in the United States or in other jurisdictions, including internationally. This manual may include trademarks, service marks, or trade names of Adobe Systems Incorporated, Inc. and other companies. Such trademarks, service marks, or trade names may be registered in the United States or in other jurisdictions, including internationally.

Third-Party Information

This manual contains information such as links to third-party websites that are not under the control of ChikaraDev, and ChikaraDev is not responsible for the content on any linked site. If you access a third-party website mentioned in this manual, then you do so at your own risk. ChikaraDev has provided these links only as a convenience, and the inclusion of the link does not imply that ChikaraDev endorses or accepts any responsibility for the content on those third-party sites.

Copyright © 2009 ChikaraDev. All rights reserved.

The software described in this manual is provided under an agreement with Adobe Systems Incorporated, and such software can only be used in accordance with the terms of the agreement provided by Adobe Systems. Software code described and provided in this manual is provided under an agreement with ChikaraDev. The software can only be used in accordance with the terms of the agreement.

No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, photocopying, manual, optical, recording, or otherwise, outside the license agreement accompanying these materials, without the prior written permission of ChikaraDev. ChikaraDev claims copyright in this program and documentation as an unpublished work, revisions of which were first licensed on the date indicated in the foregoing notice. Claim of copyright does not imply waiver of other rights of ChikaraDev and its subsidiaries.

Information in this manual may change without notice and does not represent a commitment on the part of ChikaraDev.

NOTICE OF LIABILITY

This information in these training materials is distributed on an “AS IS” basis, without warranty of any kind, either express or implied. While every precaution has been taken in the preparation of these materials, neither ChikaraDev nor its licensors shall have any liability to any person or entity with respect to liability, loss, or damage caused or alleged to be caused directly or indirectly by the instructions contained in these materials or by the computer software and hardware products described herein.

First Edition: September 2009 - ChikaraDev - Cupertino, CA 95014 USA

Overview of Drag and Drop in Flex

Drag and drop involves clicking on an object to select it, dragging the mouse, and then releasing the mouse button to move or copy the dragged object to the new location. Some Flex controls support drag and drop by default, but you can add support for drag and drop to all Flex components.

There are three main stages to drag and drop operations: **initiation**, **dragging**, and **dropping**.

Initiation begins the drag and drop operation when the user selects a Flex component, or an item within a Flex component, with the mouse, and then moves the component or item while holding down the mouse button. The selected component or item is the **drag initiator**.

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

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

Drag-and-drop either **copies or moves data** from the drag initiator to the drop target. A single Flex component can act as both drag initiator and drop target, making it possible to move the component within its container. This makes it possible to rearrange items in a control.

Like many operations in Flex, drag and drop is event driven, requiring you to write event handler for specific events, such as the `dragDrop` and `dragEnter` events.

The following list-based Flex controls include built-in support for drag and drop:

DataGrid

HorizontalList

List

PrintDataGrid

TileList

Tree

Basic Example of Drag and Drop with List-Based Controls

The following code (**BasicDragDrop1.mxml**) illustrates a basic example of drag and drop with List controls:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  creationComplete="init();">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable] private var srcAC:ArrayCollection = new ArrayCollection();
      [Bindable] private var destAC:ArrayCollection = new ArrayCollection();

      private function init():void{
        srcAC = new ArrayCollection(['Fish', 'Meat', 'Chicken']);
        destAC = new ArrayCollection();
      }
    ]]>
  </mx:Script>
  <mx:HBox>
    <mx:VBox>
      <mx:Label text="Today's Menu" fontWeight="bold" fontSize="14"/>
      <mx:List id="menuList" width="200" dataProvider="{srcAC}"
        dragEnabled="true" dragMoveEnabled="true"
        allowMultipleSelection="true" fontSize="14"/>
    </mx:VBox>
    <mx:VBox>
      <mx:Label text="Your Order" fontWeight="bold" fontSize="14"/>
      <mx:List id="orderList" width="200" dataProvider="{destAC}"
        dropEnabled="true" fontSize="14"/>
    </mx:VBox>
  </mx:HBox>
  <mx:Button id="b1" label="Reset Order" click="init();"/>
</mx:Application>
```

The **dragEnabled** property allows the **menuList** component to be a drag initiator. The **dropEnabled** property allows the **orderList** component to be a drop target. The **dragMoveEnabled** property enables moving data between components. Notice that you can drag items from **menuList** to **orderList** but you can not do the opposite because only one of the lists has **dragEnabled** set and only one has **dropEnabled** set.

In this simple application the **dragMoveEnabled** property determines whether or not items will be moved or copied from one list to another. If **dragMoveEnabled** is set to true items can be moved but not copied. If it is set to false items can be copied but not moved. The default value of **dragMoveEnabled** is false, so items will be copied unless you set this property.

Also notice that you can drag items from **menuList** to **orderList** but you cannot drag and drop items within either list. You can drag items within **menuList** because is **dragEnabled** set to true, but you cannot drop items within **menuList**, as seen by the default **dragProxy** image (red circle with white X).

Example of Two-Way Drag and Drop with List-Based Controls

The next example (**BasicDragDrop2.mxml**) illustrates performing two-way drag and drop between List controls:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  creationComplete="init();">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable] private var srcAC:ArrayCollection = new ArrayCollection();
      [Bindable] private var destAC:ArrayCollection = new ArrayCollection();

      private function init():void{
        srcAC = new ArrayCollection(['Fish', 'Meat', 'Chicken']);
        destAC = new ArrayCollection();
      }

      private function displayData():void{
        trace("*****");
        trace("menuList dataProvider:");
        for each(var srcItem:Object in srcAC){
          trace("\t" + srcItem.toString());
        }
        trace("-----");
        trace("orderList dataProvider:");
        for each(var destItem:Object in destAC){
          trace("\t" + destItem.toString());
        }
      }
    ]]>
  </mx:Script>
  <mx:HBox>
    <mx:VBox>
      <mx:Label text="Today's Menu" fontWeight="bold" fontSize="14"/>
      <mx:List id="menuList" width="200" dataProvider="{srcAC}"
        dragEnabled="true" dropEnabled="true" dragMoveEnabled="true"
        allowMultipleSelection="true" fontSize="14"/>
    </mx:VBox>
    <mx:VBox>
      <mx:Label text="Your Order" fontWeight="bold" fontSize="14"/>
      <mx:List id="orderList" width="200" dataProvider="{destAC}"
        dragEnabled="true" dropEnabled="true" dragMoveEnabled="true"
        allowMultipleSelection="true" fontSize="14"/>
    </mx:VBox>
  </mx:HBox>
  <mx:Button id="b1" label="Reset Order" click="init();"/>
  <mx:Button label="Display Data Providers" click="displayData();"/>
</mx:Application>
```

In this example, **dragEnabled** and **dropEnabled** and **dragMoveEnabled** are set to **true** for both lists, making it possible to drag and drop freely to **move** items **between** both lists. You can even drag items **within** both lists to change their order in the lists.

Launch the application in debug mode (F11) and click the **Display Data Providers** button to see a trace of the data providers for both lists. This allows you to drag items within a list and verify that the order of the items in the list data provider is also changed.

Now set the **dragMoveEnabled** property for both lists to **false** and drag and drop items between or within the lists to copy the items instead of moving them. Do this multiple times for the same item. The items are copied, but when more than one instance of any item exists in either list, only the last instance of that item (highest index number in the data provider) can be selected and dragged in subsequent drag and drop operations.

This behavior occurs because Flex provides default support for **moving** in drag and drop for these list-based components, but not copy. Because there are valid scenarios where more than one instance of an object exists in a list, you would need to write the code for your desired item copy behavior.

Note: the default value of the dragMoveEnabled property is false for all list classes except the Tree control. For the Tree control, the default value of the dragMoveEnabled property is true.

Enabling Drag and Drop within the Same Control

Although the previous example enabled drag and drop within the same control, it is worth mentioning again. This allows users to reorganize items in list-based controls by dragging and then dropping items within the same control. In this example the user reorganize the nodes of a Tree control using drag and drop. Set **dragEnabled** and **dropEnabled** to **true** for the Tree control. Remember that for all list classes except the Tree control the default value of the **dragMoveEnabled** property is false, but for Tree **dragMoveEnabled** defaults to **true**.

BasicDragDrop3.mxml

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:XML id="treeDP">
    <node>
      <node label="Saved">
        <node label="Expense Accounts - June 14, 2009"/>
      </node>
      <node label="New Items">
        <node label="Daily Report - Aug 3, 2009"/>
        <node label="Expense Accounts - Aug 2, 2009"/>
        <node label="Supply Review - Aug 2, 2009"/>
      </node>
      <node label="Trash">
        <node label="Expense Accounts - July 28, 2009"/>
        <node label="Supply Review - July 15, 2009"/>
      </node>
    </node>
  </mx:XML>
  <mx:Tree id="firstList" width="275" height="220" showRoot="false" labelField="@label"
    dataProvider="{treeDP}" dragEnabled="true" dropEnabled="true" allowMultipleSelection="true"/>>
</mx:Application>
```

Maintaining Object Type Information during Copy

In some situations data-type information can be lost when using list control built-in drag and drop support to copy data from one list-based control to another. This loss of type information can occur in these situations:

- when copying between two list-based controls (this does not occur during move operations)
- the copied item data-type is not a basic ActionScript data-type such as Date, Number, Object, or String
- the data-type of the copied item is not DisplayObject, or a subclass of DisplayObject

The solution is to use the **[RemoteClass]** metadata tag before the class declaration. This metadata tag registers the class with Flex so its type information is preserved copy operations. If you omit the **[RemoteClass]** metadata tag, type information is lost during copy.

The **[RemoteClass]** metadata tag can also be used to represent a server-side Java object in a client application.

Use **[RemoteClass(alias=" ")]** to create an ActionScript object mapping directly to the Java object. Specify the fully qualified class name of the Java class as the value of alias. For more information, see this LiveDocs page: http://livedocs.adobe.com/flex/3/html/help.html?content=data_access_1.html

This example presents a simple ActionScript class (**Contact.as**) whose type information is preserved using **[RemoteClass]**. Compile and run this program, then drag an object from one List to the other and click the button. The message indicates data-type information was retained during the copy. Now modify the class by removing the **[RemoteClass]**, re-compile and the data-type information is lost during the copy.

```
<?xml version="1.0"?>
package
{
    [RemoteClass]
    public class Contact extends Object{
        public var fname:String;
        public var lname:String;
        public var email:String;

        public function Contact(){
            super();
        }

        public function get label():String{
            var retVal:String = "";
            if(this.fname && this.lname && this.email){
                retVal = this.fname + " " + this.lname + " - " + this.email;
            }
            return retVal;
        }
    }
}
```

SaveTypeInfo.mxml

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns="*" verticalGap="25" creationComplete="init();">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      [Bindable] private var ac:ArrayCollection;

      private function init():void{
        var c1:Contact = new Contact();
        var c2:Contact = new Contact();
        var c3:Contact = new Contact();
        c1.fname = "Bob";
        c1.lname = "Smith";
        c1.email = "smith@xyz.com";
        c2.fname = "Tim";
        c2.lname = "Casey";
        c2.email = "casey@abc.com";
        c3.fname = "Mary";
        c3.lname = "Covey";
        c3.email = "covey@jkl.com";
        ac = new ArrayCollection([c1, c2, c3]);
      }

      public function checkType():void{
        if (myList2.dataProvider != null){
          messageTxt.text += myList2.dataProvider[0].label;
          if(myList2.dataProvider[0] is Contact){
            messageTxt.text += " *** Data type IS 'Contact'\n";
          } else{
            messageTxt.text += " *** Data type is NOT 'Contact'\n";
          }
        }
      }
    ]]>
  </mx:Script>
  <mx:HBox>
    <mx:List id="myList1" width="300" height="100" dataProvider="{ac}"
      dragEnabled="true" dragMoveEnabled="false"/>
    <mx:List id="myList2" width="300" height="100" dropEnabled="true"/>
  </mx:HBox>
  <mx:Button label="Check Data Type" click="checkType();"/>
  <mx:TextArea id="messageTxt" width="600" height="100"/>
</mx:Application>
```

Remember, data-type information is lost only when executing a copy, not when executing a move operation.

Drag and Drop Between Different List-Based Controls

The next example will illustrate enabling drag from a DataGrid to a List control to copy data. All that is necessary apart from setting the properties related to drag and drop is setting the “**labelField**” property of the List control to indicate which field from the DataGrid item to add to the List data provider. Data from ActionScript file **data1.as** will be made accessible using the “**include**” keyword.

```
import mx.collections.ArrayCollection;
```

```
[Bindable] private var ac:ArrayCollection = new ArrayCollection([
    {name: "Bob Smith", employee_number: "3396", department: "Accounting"},
    {name: "Tim Carey", employee_number: "7494", department: "Human Resources"},
    {name: "Susan Weston", employee_number: "1036", department: "Marketing"},
    {name: "Mary Gilbert", employee_number: "8834", department: "Human Resources"},
    {name: "Ellen Cranston", employee_number: "2285", department: "Sales"},
    {name: "Jim Metcalf", employee_number: "1046", department: "Accounting"},
    {name: "Eleanor Mayfield", employee_number: "5148", department: "Sales"},
    {name: "Fred Roberts", employee_number: "3382", department: "Sales"},
    {name: "Phillip Tsai", employee_number: "7745", department: "Marketing"},
    {name: "Ted Franconia", employee_number: "8291", department: "Shipping"},
    {name: "Erin Brocken", employee_number: "2837", department: "Accounting"}
]);
```

DataGridListDragDrop.mxml

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            include "data1.as";
        ]]>
    </mx:Script>
    <mx:HBox horizontalGap="20">
        <mx:VBox>
            <mx:Label text="All Employees:" fontSize="14" fontWeight="bold"/>
            <mx:DataGrid id="dg" dataProvider="{ac}" dragEnabled="true" dragMoveEnabled="false"
                width="300" rowCount="{ArrayCollection(dg.dataProvider).length}"
                alternatingItemColors="[0xc6f4f5,0xf6c9e3]">
                <mx:columns>
                    <mx:DataGridColumn dataField="name" headerText="Name" width="80"/>
                    <mx:DataGridColumn dataField="employee_number" headerText="Emp#" width="50"/>
                    <mx:DataGridColumn dataField="department" headerText="Department" width="100"/>
                </mx:columns>
            </mx:DataGrid>
        </mx:VBox>
        <mx:VBox>
            <mx:Label text="Selected Employees:" fontSize="14" fontWeight="bold"/>
            <mx>List id="list" labelField="name" dropEnabled="true"
                width="150" height="{dg.height}" alternatingItemColors="[0x66eef0, 0xf393cb]"/>
        </mx:VBox>
    </mx:HBox>
</mx:Application>
```

Manually Adding Drag and Drop Support

Manually adding support for drag and drop allows you to move beyond the default behavior of List based components. It also makes it possible to add drag and drop support to any Flex component.

This section presents the basics of manually adding support for drag and drop. Subsequent sections present more advanced examples of implementing custom UI behavior using drag and drop.

Manually adding or customizing drag and drop behavior means handling drag and drop events using three Flex classes, **DragManager**, **DragSource**, and **DragEvent**.

DragManager manages drag-and-drop operations. **DragSource** contains the data being dragged and provides additional drag management capabilities. **DragEvent** represents the event object for all drag-and-drop events.

Events Handled by Drag Initiator

mouseDown is dispatched when the user clicks a control with the mouse and holds down the mouse button.

mouseMove is dispatched when the user moves the mouse.

dragStart is dispatched and used internally by list-based components when a drag operation starts. You do not need to handle it when implementing drag and drop operations. Use the **mouseDown** or **mouseMove** events to control the start of a drag-and-drop operation.

dragComplete is dispatched when a drag operation has completed, either when dragged data is dropped onto a drop target, or when the drag-and-drop operation is aborted. For example, **dragComplete** can be used to perform final cleanup after drag-and-drop operations, such as deleting moved data from the drag initiator.

When adding drag-and-drop support to a component, you are required to implement an event handler for either the **mouseDown** or **mouseMove** event, and can optionally implement one for the **dragComplete** event.

Events Handled by Drop Target

dragEnter is dispatched when the drag proxy moves over the drop target. Components must define a **dragEnter** event handler to be a drop target. The event handler evaluates the data being dragged to determine if it is in an accepted format. The event handler calls the **DragManager.acceptDragDrop()** method to accept the drop and this method must be called for the drop target to receive the **dragOver**, **dragExit**, and **dragDrop** events. The appearance of the drop target can be changed in the **dragEnter** handler to provide visual cues to the user that the component can accept the drag operation.

dragOver is dispatched while the user moves the mouse over the target, after the **dragEnter** event has been dispatched. Handling this event allows you the opportunity to perform custom logic before allowing the drop operation, such as dropping the data to numerous locations within the drop target, or providing different types of visual cues based on the type of drag-and-drop operation.

dragDrop is dispatched when the user releases the mouse over the drop target. The drag data can be added to the drop target in the **dragDrop** event handler.

dragExit is dispatched when the user moves the drag proxy off the drop target without dropping the data onto the target. Use this event to restore the drop target to its original appearance if it was modified in response to a **dragEnter** event or any other event.

You must implement an event handler for the **dragEnter** and **dragDrop** events when adding drag-and-drop support to nonlist-based components. Implementing event handlers for the other events is optional.

Overview of Drag and Drop Operations

1) There are two ways a component can become a drag-and-drop initiator:

For list-based components, set the **dragEnabled** property is set to **true** and Flex automatically makes the component an initiator if the user clicks / moves the mouse on the component.

For non-list-based components, or list-based components with **dragEnabled** set to **false**, the component must explicitly become a drag initiator when it detects that the user has started a drag operation. The component will use the **mouseMove** or **mouseDown** event handler methods to create an instance of **mx.core.DragSource** containing the data to be dragged and specifying the format of the data. The component then calls **mx.managers.DragManager.doDrag()**, to initiate the drag-and-drop operation.

2) While still pressing the mouse button and moving the mouse around the application, Flex displays the drag proxy image (default image specified by the **DragManager.defaultDragImageSkin** property). Releasing the mouse button when the drag proxy is not over a valid drop target ends the drag-and-drop operation, Flex generates a **DragComplete** event on the drag initiator, and the **DragManager.getFeedback()** method returns **DragManager.NONE**.

3) Anytime the user moves the drag proxy over a Flex component, Flex dispatches a **dragEnter** event for the component. For List-based components with **dropEnabled** set to **true**, Flex checks to see if the component can be a drop target. For non-list-based components, or list-based components with **dropEnabled** set to **false**, the component needs to define an event handler for the **dragEnter** event to be a drop target.

The **dragEnter** event handler examines the **DragSource** to see if the data format is acceptable. To accept the drop, the event handler calls the **DragManager.acceptDragDrop()** method. **DragManager.acceptDragDrop()** must be called for the drop target to receive the **dragOver**, **dragExit**, and **dragDrop** events.

If the drop target does not accept the drop, the drop target parent chain is examined to see if any component in the parent 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.

- 4) The drop target can optionally handle the **dragOver** event. For example, the drop target can use this event handler to set the focus on itself.
- 5) If the user decides not to execute a drop moves the drag proxy outside of the drop target without releasing the mouse button, Flex dispatches a **dragExit** event. The drop target can optionally handle this event, perhaps to undo any actions carried out in the **dragOver** event handler.
- 6) If the user releases the mouse button while over the drop target, Flex dispatches a **dragDrop** event on the drop target. Flex automatically adds the drag data to the drop target for list-based components with **dropEnabled** set to **true**. You need to implement the event handler for the **dragDrop** event for a list-based control if the drag and drop operation is a copy operation.

For non-list-based components, or list-based components with **dropEnabled** set to **false**, the drop target needs to define an event listener for the **dragDrop** event handler to add the drag data to the drop target.

- 7) When the drop operation completes, Flex dispatches a **dragComplete** event, and the drag initiator can optionally handle this event, perhaps to delete the drag data from the drag initiator in the case of a move.

For move operations on list-based components with **dragEnabled** set to **true**, Flex automatically removes the drag data from the drag initiator.

For non-list-based components or list-based components with **dragEnabled** set to **false**, the drag initiator must carry out any required final processing. For move operations, the event handler must remove the drag data from the drag initiator.

Simple Example Adding Drag and Drop for a Non-List-Based Control

This example application (**ManualDragDrop1.mxml**) enables users to drag a small colored Canvas rectangle onto a larger Canvas to change the `backgroundColor` of the larger Canvas. No data is copied or moved, the dragged data is simply used to affect a property in the `dragTarget`.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
      import mx.core.DragSource;
      import mx.managers.DragManager;
      import mx.events.*;

      private function mouseDownHandler(event:MouseEvent):void {
        var dragInitiator:Canvas=Canvas(event.currentTarget);
```

```

        var dragColor:String = dragInitiator.getStyle('backgroundColor');
        var dragSource:DragSource = new DragSource();

        dragSource.addData(dragColor, 'color');
        DragManager.doDrag(dragInitiator, dragSource, event);
    }

    private function enterHandler(event:DragEvent):void {
        if (event.dragSource.hasFormat('color')) {
            var dropTarget:Canvas=Canvas(event.currentTarget);
            DragManager.acceptDragDrop(dropTarget);
        }
    }

    private function dropHandler(event:DragEvent):void {
        var data:Object = event.dragSource.dataForFormat('color');
        myCanvas.setStyle("backgroundColor", data);
    }
}]]>
</mx:Script>
<mx:Canvas>
    <mx:Canvas width="40" height="40" backgroundColor="0xFFFFFF" x="0" y="55"
        mouseDown="mouseDownHandler(event);"/>
    <mx:Canvas width="40" height="40" backgroundColor="0x000000" x="0" y="105"
        mouseDown="mouseDownHandler(event);"/>
    <mx:Canvas width="40" height="40" backgroundColor="0xFF0000" x="55" y="0"
        mouseDown="mouseDownHandler(event);"/>
    <mx:Canvas width="40" height="40" backgroundColor="0xFFFF00" x="105" y="0"
        mouseDown="mouseDownHandler(event);"/>
    <mx:Canvas id="myCanvas" width="100" height="100" x="50" y="50"
        backgroundColor="#FFFFFF"
        dragEnter="enterHandler(event);" dragDrop="dropHandler(event);"/>
    <mx:Canvas width="40" height="40" backgroundColor="0x00FF00" x="160" y="55"
        mouseDown="mouseDownHandler(event);"/>
    <mx:Canvas width="40" height="40" backgroundColor="0x00FFFF" x="160" y="105"
        mouseDown="mouseDownHandler(event);"/>
    <mx:Canvas width="40" height="40" backgroundColor="0xFF00FF" x="55" y="160"
        mouseDown="mouseDownHandler(event);"/>
    <mx:Canvas width="40" height="40" backgroundColor="0x0000FF" x="105" y="160"
        mouseDown="mouseDownHandler(event);"/>
</mx:Canvas>
    <mx:Label text="Drag a color to the main canvas" fontSize="16"/>
</mx:Application>

```

The smaller colored rectangles act as **drag initiators**, and initiate the drag and drop operation with a **mouseDown** event handler. The **drop target** implements handlers for the **dragEnter** and **dragDrop** events.

The **drag initiator** event handler **initiating** the drag and drop operation is responsible for two things:

- Create a **DragSource** object and initialize it with the drag data and the data format.
- Call the **DragManager.doDrag()** method to start the drag-and-drop operation.

Drop targets respond in a **dragEnter** event handler when a **drag proxy** is moved over the target, and respond in a **dragDrop** event handler when the user releases the mouse button while the **drag proxy** is over the target.

Drag Initiator Event Handler Details

In the example application, the **drag initiator** registers a handler for the **mouseDown** event:

```
private function mouseDownHandler(event:MouseEvent):void {
    var dragInitiator:Canvas=Canvas(event.currentTarget);
    var dragColor:String = dragInitiator.getStyle('backgroundColor');
    var dragSource:DragSource = new DragSource();

    dragSource.addData(dragColor, 'color');
    DragManager.doDrag(dragInitiator, dragSource, event);
}
```

The **mouseDown** event handler gets a reference to the **drag initiator** object from the **currentTarget** property of the event object. It also uses a string **dragColor** to store the data to be transferred during the drag and drop operation, in this case the **backgroundColor** of the **drag initiator**.

The handler then creates a **DragSource** object, and calls that object's **addData()** method to hold the data to be transferred. **addData()** takes an object with the data and also takes a string indicating the **format** of the data, in this case "**color**". When implementing drag and drop support manually, this format string can be anything, such as **backgroundColor**, **rectColor**, **myTargetColor**, etc., but the **dragEnter** event handler will check the **DragSource** data format string, and will only respond appropriately upon finding an identically named format string.

List controls have predefined values for the **format argument**. For **non-Tree controls**, the format String is "**items**". For the **Tree control**, the format String is "**treeItems**".

Later examples will illustrate the use of the **DragSource addHandler()** method, which allows you to defer processing data to be dragged until it is accessed. This is useful if the drag data is complex or large.

Finally the handler initiates the drag and drop operation by calling the **DragManager.doDrag()** method, passing in as arguments the **drag initiator**, the **DragSource** object, and the **event** passed into the **mouseDown** handler.

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
```

Additional arguments to the method allow you to control the image used as the drag proxy, the x and y offsets of the drag proxy from the mouse pointer, the drag proxy image alpha, and whether or not the drop target is allowed to move the dragged data.

At this point the user may have moved the mouse anywhere within the application. The mouse might be over a drop target or might be over the application background, but when the user pressed the mouse button down, the mouseDown event handler initiated the drag and drop operation.

dragEnter Event Handler Details

When the user moves the **drag proxy** over any control Flex generates a **dragEnter** event, but to be a **drop target** a control must define a **dragEnter** event handler, and the handler will typically perform the following actions:

- Examine the **DragSource** object format string to see if the drag data **format** is acceptable to the **drop target**.
- Call the **DragManager.acceptDragDrop()** method if the data format is acceptable. This enables the user to drop the data on the drop target. If the event handler does not call this method, the user will not be allowed to 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**.

In the example application, the **dragEnter** event handler accesses the **dragSource** event property to call its **hasFormat()** method, and if an acceptable dragSource **format** is found, the **acceptDragDrop()** method is called, with the **dropTarget** passed as a method parameter.

```
private function enterHandler(event:DragEvent):void {
    if (event.dragSource.hasFormat('color')) {
        var dropTarget:Canvas=Canvas(event.currentTarget);
        DragManager.acceptDragDrop(dropTarget);
    }
}
```

This results in a **dragDrop** event being dispatched, which the **dropTarget** event handler listens for.

dragDrop Event Handler Details

The **dragDrop** event is dispatched when the user drops data onto a target, and the **dragEnter** event handler checks the data format and then calls the **DragManager.acceptDragDrop()** method to accept the drop.

In the example application, the **dragDrop** event handler calls the **DragSource.dataForFormat()** method to access the **drag data**, which in this case is a string representing the **backgroundColor** of a Canvas. It then uses this drag data color information to set the **backgroundColor** of the larger Canvas.

```
private function dropHandler(event:DragEvent):void {
    var data:Object = event.dragSource.dataForFormat('color');
    myCanvas.setStyle("backgroundColor", data);
}
```

Simple Example Customizing Drag / Drop for a List-Based Control

Flex automatically manages all drag-and-drop events for move operations when **dragEnabled** is set to **true** for a drag initiator or when **dropEnabled** is set to **true** for a drop target. For copy operations you need to define your own **dragDrop** event handler.

You also need to define your own event handlers to explicitly control other aspects of drag-and-drop operations for list-based controls, or to implement drag and drop for non-list controls, as in the previous example.

Another option is to use the built-in drag-and-drop handlers Flex provides in addition to defining your own event handlers. Your event handler executes first and then the default event handler executes. You can explicitly prohibit execution of the default event handler if necessary by calling the **Event.preventDefault()** method within your event handler. In some cases execution might differ if you call **Event.preventDefault()** before or after your custom event handler code.

Note: calling **Event.preventDefault()** in the event handler for the **dragComplete** or **dragDrop** events when dragging data from one **Tree** control to another prevents the drop.

Tree controls handle drag and drop differently from the other list-based controls. For Tree controls the **dragDrop** event handler only performs an action when you **move** or **copy** data within the same Tree control, or **copy** data to another Tree control.

To **move** data dragged between **Tree** controls, the **dragComplete** event handler adds the data to the destination Tree control, and removes the data from the source Tree control, rather than the **dragDrop** event handler. This is because Flex must remove the data from the source tree first in order to re-parent the data being moved.

This next example application (**ManualDragDrop2.mxml**) illustrates changing the default drag and drop behavior of list-based controls. Flex defines default event handlers for drag-and-drop events when you set **dragEnabled** or **dropEnabled** to true for list-based controls, but you can define your own event handlers to modify the default behavior for list-based controls.

Comment out the line with **hideDropFeedback(evt)** to see how **evt.preventDefault()** causes the drop feedback black line to remain after the drop, and how calling **hideDropFeedback(evt)** corrects that behavior.

You can remove the **dragDropHandler()** method and the reference to it in the **cartlist** List tag and still have drag and drop behavior, but because **dragMoveEnabled** is set to **false**, the data will be copied and it is possible to drag the same item more than once to the drop target. Our custom code in the **dragDropHandler()** method allows us to implement correct copy behavior, allowing data to be copied only if the data does not already exist in the drop target data provider.

Notice that our call to the **dataForFormat()** method specifies an argument "items". The list-based controls have predefined values for the **data** format of drag data, with all non-tree list controls having a format String of "items", and the Tree control having a format string of "treelItems". The return value of **dataForFormat()** is always an Array for list-based controls, even when dragging a single item.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
      import mx.events.DragEvent;
      import mx.collections.ArrayCollection;

      [Bindable] private var books:ArrayCollection = new ArrayCollection([
        {Author:'Samuel Phillips', BookTitle:'Winter in Tahoe', Price:28.00},
        {Author:'Robert McGaven', BookTitle:'Southwest Sunrise', Price:18.00},
        {Author:'Linda Thomas', BookTitle:'Lilies on the Pond', Price:19.50},
        {Author:'Richard Avondale', BookTitle:'Misty Morning', Price:32.00}
      ]);

      [Bindable] private var cartbooks:ArrayCollection = new ArrayCollection();

      public function dragDropHandler(evt:DragEvent):void {
        List(evt.currentTarget).hideDropFeedback(evt);
        evt.preventDefault();
        var dragObj:Array= evt.dragSource.dataForFormat("items") as Array;

        for (var i:uint = 0; i < dragObj.length; i++) {
          if(!cartbooks.contains(dragObj[i])){
            cartbooks.addItem(dragObj[i]);
          }
        }
      }
    ]]>
  </mx:Script>
  <mx:NumberFormatter id="nf" precision="2"/>
  <mx:HBox>
    <mx:VBox>
      <mx:Label text="Features books:"/>
      <mx:DataGrid id="booksgrid" dataProvider="{books}" allowMultipleSelection="true"
        dragEnabled="true" dragMoveEnabled="false">
        <mx:columns>
          <mx:DataGridColumn dataField="Author" width="100"/>
          <mx:DataGridColumn dataField="BookTitle" width="100"/>
          <mx:DataGridColumn dataField="Price" width="50">
            <mx:itemRenderer>
              <mx:Component>
                <mx:Label text="{outerDocument.nf.format(data.Price)}/>
              </mx:Component>
            </mx:itemRenderer>
          </mx:DataGridColumn>
        </mx:columns>
      </mx:DataGrid>
    </mx:VBox>
  </mx:HBox>
</mx:Application>
```

```

<mx:VBox>
  <mx:Label text="Your cart:"/>
  <mx:List id="cartlist" dropEnabled="true" width="300" height="{booksgrid.height}"
    dragDrop="dragDropHandler(event);" dataProvider="{cartbooks}">
    <mx:itemRenderer>
      <mx:Component>
        <mx:HBox>
          <mx:Label id="author" text="{data.Author}"/>
          <mx:Label id="title" text="{data.BookTitle}"/>
          <mx:Label id="price" text="{outerDocument.nf.format(data.Price)}/>
        </mx:HBox>
      </mx:Component>
    </mx:itemRenderer>
  </mx:List>
</mx:VBox>
</mx:HBox>
</mx:Application>

```

Simple Example of Drag / Drop to a Container Drop Target

For a container to be a drop target, the **backgroundColor** property of the container must be set to a color, otherwise the background color of the container will be transparent, and the **DragManager** will be unable to detect that the mouse pointer is over a possible drop target.

This example application (**ContainerDropTarget1.mxml**) allows you to drag colored images to a large Canvas to add a smaller Canvas having a **backgroundColor** that matches the Image color.

Remove the **backgroundColor** attribute of the **workarea** Canvas and notice that drag and drop no longer works, because without a **backgroundColor** the Canvas is essentially transparent, and the **DragManager** is unable to detect that the mouse pointer is over a possible drop target.

You can get around this by setting a **backgroundColor** and also setting the **backgroundAlpha** to 0, which allows you to have a transparent background and still enable the container to respond to drag and drop events.

Notice the use of the Image **"name"** property to store the color hexadecimal value, to be used in the **moveHandler()** event handler and then in the **dragDropHandler()** event handler to set the **backgroundColor** of the small Canvas to the image color.

Also notice the use of **globalToLocal()** to convert mouse *x* and *y* coordinates (expressed in "global" application coordinates) to local (workarea Canvas) coordinates. Call **Canvas(evt.currentTarget).globalToLocal(globalPoint)**, because if you call **this.globalToLocal(globalPoint)**, the coordinates would not change, because local coordinates for "this" (the application), are the same as the global coordinates.

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  backgroundColor="0xFFFFFF" verticalGap="10">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.managers.DragManager;
      import mx.core.DragSource;
      import mx.events.DragEvent;
      import flash.events.MouseEvent;
      import mx.controls.Image;

      [Embed(source='assets/images/black_50_50.png')] public var blackImg:Class;
      [Embed(source='assets/images/blue_50_50.png')] public var blueImg:Class;
      [Embed(source='assets/images/brown_50_50.png')] public var brownImg:Class;
      [Embed(source='assets/images/gray_50_50.png')] public var grayImg:Class;
      [Embed(source='assets/images/green_50_50.png')] public var greenImg:Class;
      [Embed(source='assets/images/orange_50_50.png')] public var orangeImg:Class;
      [Embed(source='assets/images/purple_50_50.png')] public var purpleImg:Class;
      [Embed(source='assets/images/red_50_50.png')] public var redImg:Class;
      [Embed(source='assets/images/white_50_50.png')] public var whiteImg:Class;
      [Embed(source='assets/images/yellow_50_50.png')] public var yellowImg:Class;

      [Bindable] private var imagesAC:ArrayCollection = new ArrayCollection([
        {color: "0x000000", source: blackImg},
        {color: "0x0000FF", source: blueImg},
        {color: "0x754C24", source: brownImg},
        {color: "0xCCCCCC", source: grayImg},
        {color: "0x00FF00", source: greenImg},
        {color: "0xFF00FF", source: purpleImg},
        {color: "0xFF0000", source: redImg},
        {color: "0xFFFFFF", source: whiteImg},
        {color: "0xFFFF00", source: yellowImg},
        {color: "0xFF931E", source: orangeImg}
      ]);

      private function moveHandler(evt:MouseEvent):void{
        var colorData:String = evt.currentTarget.name;
        var dragInitiator:Image = Image(evt.currentTarget);
        var datasource:DragSource = new DragSource();
        datasource.addData(colorData, "color");
        DragManager.doDrag(dragInitiator, datasource, evt);
      }

      private function dragEnterHandler(evt:DragEvent):void {
        if (evt.dragSource.hasFormat("color")){
          DragManager.acceptDragDrop(Canvas(evt.currentTarget));
        }
      }

      private function dragDropHandler(evt:DragEvent):void {
        var imgCanvas:Canvas = new Canvas();
        imgCanvas.setStyle("backgroundColor", evt.dragSource.dataForFormat("color"));
        imgCanvas.width = imgCanvas.height = 50;
        var globalPoint:Point = new Point(this.mouseX, this.mouseY);
    
```

```

        var localPoint:Point = Canvas(evt.currentTarget).globalToLocal(globalPoint);
        imgCanvas.x = localPoint.x - 25;
        imgCanvas.y = localPoint.y - 25;
        evt.currentTarget.addChild(imgCanvas);
    }
    ]]>
</mx:Script>
<mx:HBox width="590" height="65" backgroundColor="0xf59fb6"
    horizontalAlign="center" verticalAlign="middle">
    <mx:Repeater id="rp" dataProvider="{imagesAC}">
        <mx:Image name="{rp.currentItem.color}" source="{rp.currentItem.source}"
            mouseMove="moveHandler(event);"/>
    </mx:Repeater>
</mx:HBox>
<mx:Canvas id="workarea" width="590" height="590" borderStyle="solid"
    backgroundColor="0xf59fb6" dragEnter="dragEnterHandler(event);"
    dragDrop="dragDropHandler(event);" borderColor="0xe22052"
    borderThickness="4"/>
</mx:Application>

```

Specifying a Custom Drag Proxy

The **drag proxy** is the image that follows the mouse cursor during drag and drop operations. In the event handler initiating drag and drop operations, you can specify a custom drag proxy when calling the **DragManager.doDrag()** method, instead of using the default drag proxy image Flex provides.

The **doDrag()** method takes the following optional arguments related to drag proxy properties:

- | | |
|------------------|--|
| dragImage | The drag proxy image , which can be an image such as a JPEG , PNG , etc. specified by the image path such as myImage.jpg , or it can be a Flex component , where you create an instance of the control or container, configure and size it, and then pass it as an argument to the doDrag() method. |
| xOffset | The number of pixels from the upper-left corner of the drag initiator to offset the x property of the dragImage. The xOffset represents pixels from the left edge of the drag proxy to the left edge of the drag initiator. Positive numbers actually move the drag proxy to the left upon drag, and negative numbers move it to the right , so this value usually has a negative value. |
| yOffset | The number of pixels from the upper-left corner of the drag initiator to offset the y property of the dragImage. The yOffset represents pixels from the top edge of the drag proxy to the top edge of the drag initiator. Positive numbers actually move the drag proxy up upon drag, and negative numbers move it to the down , so this value usually has a negative value. |

imageAlpha

A number specifying the **alpha transparency** of the drag proxy image. By default Flex uses an alpha value of 0.5. A value of **0** makes the drag proxy **fully transparent** and a value of **1.0** makes it **fully opaque**.

The drag proxy image will not appear unless you specify a width and height. The following example (**CustomProxy1.mxml**) uses a 25 by 25 pixel Canvas control as the drag proxy to represent dragging a 50 by 50 pixel Canvas drag initiator. Notice the effect of the -10 pixel x and y offsets and the .5 alpha.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    backgroundColor="0xFFFFFF" verticalGap="10">
    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            import mx.managers.DragManager;
            import mx.core.DragSource;
            import mx.events.DragEvent;
            import flash.events.MouseEvent;
            import mx.controls.Image;

            [Embed(source='assets/images/black_50_50.png')] public var blackImg:Class;
            [Embed(source='assets/images/blue_50_50.png')] public var blueImg:Class;
            [Embed(source='assets/images/brown_50_50.png')] public var brownImg:Class;
            [Embed(source='assets/images/gray_50_50.png')] public var grayImg:Class;
            [Embed(source='assets/images/green_50_50.png')] public var greenImg:Class;
            [Embed(source='assets/images/orange_50_50.png')] public var orangeImg:Class;
            [Embed(source='assets/images/purple_50_50.png')] public var purpleImg:Class;
            [Embed(source='assets/images/red_50_50.png')] public var redImg:Class;
            [Embed(source='assets/images/white_50_50.png')] public var whiteImg:Class;
            [Embed(source='assets/images/yellow_50_50.png')] public var yellowImg:Class;

            [Bindable] private var imagesAC:ArrayCollection = new ArrayCollection([
                {color: "0x000000", source: blackImg},
                {color: "0x0000FF", source: blueImg},
                {color: "0x754C24", source: brownImg},
                {color: "0xCCCCCC", source: grayImg},
                {color: "0x00FF00", source: greenImg},
                {color: "0xFF00FF", source: purpleImg},
                {color: "0xFF0000", source: redImg},
                {color: "0xFFFFFF", source: whiteImg},
                {color: "0xFFFF00", source: yellowImg},
                {color: "0xFF931E", source: orangeImg}
            ]);

            private function moveHandler(evt:MouseEvent):void{
                var colorData:String = evt.currentTarget.name;
                var dragInitiator:Image = Image(evt.currentTarget);
                var datasource:DragSource = new DragSource();
                datasource.addData(colorData, "color");

                var canvasProxy:Canvas = new Canvas();
                canvasProxy.height = 25;
```

```

        canvasProxy.width = 25;
        canvasProxy.setStyle("backgroundColor", colorData);
        DragManager.doDrag(dragInitiator, datasource, evt,
            canvasProxy, -10, -10, .5);
    }

    private function dragEnterHandler(evt:DragEvent):void {
        if (evt.dragSource.hasFormat("color")){
            DragManager.acceptDragDrop(Canvas(evt.currentTarget));
        }
    }

    private function dragDropHandler(evt:DragEvent):void {
        var imgCanvas:Canvas = new Canvas();
        imgCanvas.setStyle("backgroundColor", evt.dragSource.dataForFormat("color"));
        imgCanvas.width = imgCanvas.height = 50;
        var globalPoint:Point = new Point(this.mouseX, this.mouseY);
        var localPoint:Point = Canvas(evt.currentTarget).globalToLocal(globalPoint);
        imgCanvas.x = localPoint.x - 25;
        imgCanvas.y = localPoint.y - 25;
        evt.currentTarget.addChild(imgCanvas);
    }
}]]>
</mx:Script>
<mx:HBox width="590" height="65" backgroundColor="0xf59fb6"
    horizontalAlign="center" verticalAlign="middle">
    <mx:Repeater id="rp" dataProvider="{imagesAC}">
        <mx:Image name="{rp.currentItem.color}" source="{rp.currentItem.source}"
            mouseMove="moveHandler(event);"/>
    </mx:Repeater>
</mx:HBox>
<mx:Canvas id="workarea" width="590" height="590" borderStyle="solid"
    backgroundColor="0xf59fb6" dragEnter="dragEnterHandler(event);"
    dragDrop="dragDropHandler(event);" borderColor="0xe22052"
    borderThickness="4"/>
</mx:Application>

```

Specifying a SWF Symbol as a Custom Drag Proxy

You can specify a symbol within a SWF file, perhaps created in Flash CS4, as the drag proxy, as in the following example (**CustomProxy2.mxml**). This example allows you to drag two Text controls to a Canvas to copy the text.

When dragging the “**Smiley Face**” Text control the drag proxy is symbol **smily1** in swf file **smily1.swf**, and when dragging the “**Smiley Animation**” Text control the drag proxy is symbol **smily2** in the same swf file.

smily2 is a symbol with the background color of the smiley face changing from yellow to orange and then back to yellow each second. The swf file **smily1.swf** was created in **Flash CS4**. Creating the symbols in Flash is not discussed in this tutorial, but you must ensure the “**Export for ActionScript**” checkbox is checked for the symbols in Flash when creating the SWF. The Flex Builder project accompanying this tutorial includes the SWF file.

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  backgroundColor="0xFFFFFF" verticalGap="10">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.managers.DragManager;
      import mx.core.DragSource;
      import mx.events.DragEvent;
      import flash.events.MouseEvent;
      import mx.controls.Image;

      [Embed(source="assets/swf/smily1.swf", symbol="smily1")]
      [Bindable] public var smileyImage:Class;

      [Embed(source="assets/swf/smily1.swf", symbol="smily2")]
      [Bindable] public var smileyAnimation:Class;

      private function moveHandler(evt:MouseEvent):void{
        var textData:String = evt.currentTarget.text;
        var dragInitiator:Text = Text(evt.currentTarget);
        var datasource:DragSource = new DragSource();
        datasource.addData(textData, "text");

        var imageProxy:Image = new Image();
        imageProxy.source = textData == "Smiley Face" ? smileyImage : smileyAnimation;
        DragManager.doDrag(dragInitiator, datasource, evt,
          imageProxy, -20, -20, 1);
      }

      private function dragEnterHandler(evt:DragEvent):void {
        if (evt.dragSource.hasFormat("text")){
          DragManager.acceptDragDrop(Canvas(evt.currentTarget));
        }
      }

      private function dragDropHandler(evt:DragEvent):void {
        var text:Text = new Text();
        text.setStyle("fontSize", 20);
        text.setStyle("fontWeight", "bold");
        text.text = evt.dragSource.dataForFormat("text").toString();
        var globalPoint:Point = new Point(this.mouseX, this.mouseY);
        var localPoint:Point = Canvas(evt.currentTarget).globalToLocal(globalPoint);
        text.x = localPoint.x - 25;
        text.y = localPoint.y - 25;
        evt.currentTarget.addChild(text);
      }
    ]]>
  </mx:Script>
  <mx:HBox horizontalGap="100">
    <mx:Text text="Smiley Face" fontSize="20" fontWeight="bold"
      mouseMove="moveHandler(event);" selectable="false"/>
    <mx:Text text="Smiley Animation" fontSize="20" fontWeight="bold"
      mouseMove="moveHandler(event);" selectable="false"/>
  </mx:HBox>

```

```
<mx:Canvas id="workarea" width="590" height="590" borderStyle="solid"
  backgroundColor="0xf59fb6" dragEnter="dragEnterHandler(event);"
  dragDrop="dragDropHandler(event);" borderColor="0xe22052"
  borderThickness="4"/>
</mx:Application>
```

Handling Drop Target `dragOver` and `dragExit` Events

Two events you can optionally handle for the drop target are the **`dragOver`** event, which is dispatched continuously as the user moves the mouse over a drop target whose **`dragEnter`** handler has called the **`DragManager.acceptDragDrop()`** method, and the **`dragExit`** event, which is dispatched when the user drags the drag proxy off the drop target but does not execute a drop.

The **`dragOver`** event is often used to specify visual feedback to the user while the mouse is over a drop target. You might use the **`DragManager.showFeedback()`** method to specify the drag-feedback indicator displaying within the drag proxy image. **`showFeedback()`** can take one of four constant values as an argument, to control which drag-feedback indicator displays:

Argument value	Icon
<code>DragManager.COPY</code>	Green circle with a white plus sign indicating you can perform a drop.
<code>DragManager.LINK</code>	Grey circle with a white arrow sign indicating you can perform a drop.
<code>DragManager.MOVE</code>	Plain arrow indicating you can perform a drop.
<code>DragManager.NONE</code>	Red circle with white x indicating a drop is prohibited. This same image is used when the user drags over an object that is not a drag target.

The feedback indicator is often used to give visual cues when keys affecting drag and drop are pressed by the user. The event object for the **`dragOver`** event contains the Boolean properties **`ctrlKey`** and **`shiftKey`** indicating if the Control or Shift keys were pressed. If no key was pressed it indicates a move, if the Control key was pressed it indicates a copy, and if the Shift key was pressed it indicates a link. After detecting the key that was pressed you call **`showFeedback()`** with the argument appropriate for the key pressed.

The **`showFeedback()`** method is also used to set the value of the `DragEvent` object `action` property for the `dragDrop`, `dragExit`, and `dragComplete` events. The `action` property of the `DragEvent` will always be set to `DragManager.MOVE` if you do not call the **`showFeedback()`** method in the `dragOver` event handler.

The `dragExit` event is dispatched when the user drags the drag proxy off the drop target without dropping data onto the target. This event can be used to restore any visual changes made to the drop target in the `dragOver` event handler.

The following example (**CustomFeedback1.mxml**) illustrates providing customized visual feedback to the user during drag and drop operations.

, 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:

Additional Details on Moving & Copying Data with Drag and Drop

Move operations add data to the drop target (**dragDrop event**) and delete the data from the drag initiator (**dragComplete event**).

List-based controls already support moving data during drag and drop, and handle all required processing, but for a non-list based **drag initiator** you must implement the **dragComplete** event handler to **delete** the drag data, and you have to implement the **dragDrop** event handler for non-list based **drop target** to **add** the data.

When copying data rather than moving data, you must explicitly handle the **dragDrop** event for the list-based drop targets. You are always required to write a **dragDrop** event handler for non-list based controls, for both move and copy operations.

The reason Flex does not provide default support for copying data during drag and drop operations is because copying data in an object-oriented environment often involves objects having references to other objects that themselves may contain references to other objects. Implementing copy drag and drop operations with such objects requires an intimate knowledge of the objects and their references.

Sometimes it makes sense to implement a clone method in your objects making it easier to create a byte copy of the object, but often you just copy individual fields of the source object to the destination object.

Example of Copying Data from One List Control to Another

The next example (**CopyBetweenLists.mxml**) lets you copy items from one List control to another, and duplicates are not allowed.

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, and we need to call `Event.preventDefault()` to explicitly prohibit the default **dragDrop** event handler provided by Flex from executing.

We do this because the **drop target** is a List control, one of the list-based controls defining a default **dragDrop** handler that does not handle copy operations properly. For a non-list based component, you do not have to call **Event.preventDefault()** because only list-based controls define default drag-and-drop event handlers.

The **dropEnabled** property is set to true in the **drop target** is so Flex automatically handles all the other drop target events (`dragEnter`, `dragOver`, and `dragExit`).

The default value of the **dragMoveEnabled** property is **false**, so you can only copy elements from one List control to the other, you can't move them. Set **dragMoveEnabled** to **true** in the drag initiator to move and copy elements (hold down the Control key during the drag-and-drop operation to copy).

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" creationComplete="init();">
  <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 init():void {
        listOne.dataProvider = new ArrayCollection([
          {label:"Study Flex", priority:"1"}, {label:"Cook Dinner", priority:"3"},
          {label:"Feed Dog", priority:"2"}, {label:"Exercise", priority:"2"},
        ]);
        listTwo.dataProvider = new ArrayCollection([
          {label:"Wash Car", priority:"2"}, {label:"Clean Bathroom", priority:"3"},
          {label:"Yard Work", priority:"2"}, {label:"Backup Hard-drive", priority:"1"},
        ]);
      }

      private function dragDropHandler(event:DragEvent):void {
```

```

if (event.dragSource.hasFormat("items")){
    event.preventDefault();
    event.currentTarget.hideDropFeedback(event);

    var dropTarget:List=List(event.currentTarget);
    var dropInitiator:List=List(event.dragInitiator);
    var itemsArray:Array = event.dragSource.dataForFormat("items") as Array;

    for each(var obj:Object in itemsArray){
        var templtem:Object = {label: obj.label, priority: obj.priority};
        var dropLoc:int = dropTarget.calculateDropIndex(event);

        // For some reason the following code does not work, so need to loop through dataProvider:
        //if(!ArrayCollection(dropTarget.dataProvider).contains(templtem)){
        //    IList(dropTarget.dataProvider).addItemAt(templtem, dropLoc);
        //}
        var found:Boolean = false;
        for each(var tmp:Object in dropTarget.dataProvider){
            if(tmp.label == templtem.label && tmp.priority == templtem.priority){
                found = true;
            }
        }
        // Add the dragged item if it does not exist in the dropTarget dataProvider.
        if(!found){
            IList(dropTarget.dataProvider).addItemAt(templtem, dropLoc);
        }else{
            // The dragged item does exist in the dropTarget dataProvider, so see if it
            // was dragged to a different position.
            if(dropInitiator == dropTarget){
                var ac:ArrayCollection = dropTarget.dataProvider as ArrayCollection;
                for each(var tmp2:Object in ac){
                    if(tmp2.label == templtem.label && tmp2.priority == templtem.priority){
                        if(dropLoc != ac.getItemIndex(tmp2)){
                            ac.removeItemAt(ac.getItemIndex(tmp2));
                            if(dropLoc < ac.length){
                                ac.addItemAt(tmp2, dropLoc);
                            }else{
                                // This is necessary to avoid an index out or range error.
                                ac.addItemAt(tmp2, dropLoc-1);
                            }
                        }
                    }
                }
            }
        }
    }
}
]]>
</mx:Script>
<mx:HBox>
    <mx:VBox>
        <mx:Label text="TODO Today:" fontSize="14"/>
        <mx:List id="listOne" dragEnabled="true" dropEnabled="true"
            width="200" height="200" dragDrop="dragDropHandler(event);"/>
    </mx:VBox>
</mx:HBox>

```

```

</mx:VBox>
<mx:Spacer width="40"/>
<mx:VBox>
  <mx:Label text="TODO Tomorrow:" fontSize="14"/>
  <mx:List id="listTwo" dragEnabled="true" dropEnabled="true"
    width="200" height="200" dragDrop="dragDropHandler(event)"/>
</mx:VBox>
</mx:HBox>
<mx:Button id="b1" label="Reset Lists" click="init()"/>
</mx:Application>

```

Also note in this example application that **dragEnabled** and **dropEnabled** are set to **true** for both lists. This makes it possible to copy items between the two lists, and also to drag items within a list.

Additional code enables the user to change the position of items when they are moved within a list.

Example of Copying Data Between Non-List Based Controls

In this example (**CopyMoveBetweenCanvas.mxml**) the user is able to copy and move items between two Canvas containers, which do not by default support drag and drop operations.

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;
      import mx.managers.DragManager;
      import mx.core.DragSource;
      import mx.events.DragEvent;
      import flash.events.MouseEvent;
      import mx.controls.Image;
      import mx.core.Repeater;

      [Embed(source='assets/images/black_50_50.png')] public var blackImg:Class;
      [Embed(source='assets/images/blue_50_50.png')] public var blueImg:Class;
      [Embed(source='assets/images/brown_50_50.png')] public var brownImg:Class;
      [Embed(source='assets/images/gray_50_50.png')] public var grayImg:Class;
      [Embed(source='assets/images/green_50_50.png')] public var greenImg:Class;
      [Embed(source='assets/images/orange_50_50.png')] public var orangeImg:Class;
      [Embed(source='assets/images/purple_50_50.png')] public var purpleImg:Class;
      [Embed(source='assets/images/red_50_50.png')] public var redImg:Class;
      [Embed(source='assets/images/white_50_50.png')] public var whiteImg:Class;
      [Embed(source='assets/images/yellow_50_50.png')] public var yellowImg:Class;

      [Bindable] private var ac1:ArrayCollection = new ArrayCollection([
        blackImg, blueImg, brownImg, grayImg, greenImg,
        orangeImg, purpleImg, redImg, whiteImg, yellowImg
      ]);

      [Bindable] private var ac2:ArrayCollection = new ArrayCollection();
    ]]>
  </mx:Script>
</mx:Application>

```

```

private function mouseMoveHandler(event:MouseEvent):void{
    var dragInitiator:Image=Image(event.currentTarget);
    var ds:DragSource = new DragSource();
    ds.addData(dragInitiator.source, "imgClass");

    var imageProxy:Image = new Image();
    imageProxy.source = dragInitiator.source;
    imageProxy.height=20;
    imageProxy.width=20;
    DragManager.doDrag(dragInitiator, ds, event, imageProxy, -15, -15, 1.00);
}

private function dragEnterHandler(event:DragEvent):void {
    if(event.dragSource.hasFormat("imgClass")){
        DragManager.acceptDragDrop(Canvas(event.currentTarget));
    }
}

private function dragOverHandler(event:DragEvent):void{
    if (event.dragSource.hasFormat("imgClass")) {
        if (event.ctrlKey) {
            DragManager.showFeedback(DragManager.COPY);
            return;
        }else {
            DragManager.showFeedback(DragManager.MOVE);
            return;
        }
    }
    DragManager.showFeedback(DragManager.NONE);
}

private function dragDropHandler(event:DragEvent):void {
    if (event.dragSource.hasFormat("imgClass")) {
        var dropCanvas:Canvas = event.currentTarget as Canvas;
        var cls:Class = event.dragSource.dataForFormat("imgClass") as Class
        var tmpAC1:ArrayCollection = rp1.dataProvider as ArrayCollection;
        var tmpAC2:ArrayCollection = rp2.dataProvider as ArrayCollection;
        if(rp1.container == dropCanvas){
            if(!tmpAC1.contains(cls)){
                // Remove item from other AC if moving, because if we
                // do it in dragComplete handler its too late, the
                // AC item was dragged to already has the item added,
                // and we need to check for existance.
                if(DragManager.getFeedback() == DragManager.MOVE){
                    if(tmpAC2.contains(cls)){
                        tmpAC2.removeItemAt(tmpAC2.getItemIndex(cls));
                    }
                }
                tmpAC1.addItem(cls);
            }
        }else if(rp2.container == dropCanvas){
            if(!tmpAC2.contains(cls)){
                // Remove item from other AC if moving, because if we
                // do it in dragComplete handler its too late, the
                // AC item was dragged to already has the item added,

```

```

        // and we need to check for existence.
        if(DragManager.getFeedback() == DragManager.MOVE){
            if(tmpAC1.contains(cls)){
                tmpAC1.removeItemAt(tmpAC1.getItemIndex(cls));
            }
        }
        tmpAC2.addItem(cls);
    }
}
}
}
]]>
</mx:Script>
<mx:Canvas id="canvas1" width="557" height="62" borderStyle="solid" backgroundColor="0xEFCAD4"
    horizontalScrollPolicy="off" verticalScrollPolicy="off"
    dragEnter="dragEnterHandler(event);" dragOver="dragOverHandler(event);"
    dragDrop="dragDropHandler(event);">
    <mx:Repeater id="rp1" dataProvider="{ac1}">
        <mx:Image source="{rp1.currentItem}"
            x="{((rp1.currentIndex) * 55) + 5}" y="5"
            mouseMove="mouseMoveHandler(event);"/>
    </mx:Repeater>
</mx:Canvas>
<mx:Spacer height="10"/>
<mx:Canvas id="canvas2" width="557" height="62" borderStyle="solid" backgroundColor="0xEFCAD4"
    horizontalScrollPolicy="off" verticalScrollPolicy="off"
    dragEnter="dragEnterHandler(event);" dragOver="dragOverHandler(event);"
    dragDrop="dragDropHandler(event);">
    <mx:Repeater id="rp2" dataProvider="{ac2}">
        <mx:Image source="{rp2.currentItem}"
            x="{((rp2.currentIndex) * 55) + 5}" y="5"
            mouseMove="mouseMoveHandler(event);"/>
    </mx:Repeater>
</mx:Canvas>
</mx:Application>

```

The **dragComplete** event occurs on the **drag initiator** when the 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. Often the drag initiator uses a **dragComplete** handler to perform cleanup when the drag finishes, such as removing objects moved to the drag target from the drag initiator. The **dragComplete** event can also be used to perform necessary cleanup when the target does not accept the drop.

You can determine the type of drag operation (copy or a move) using the **action** property of the event object passed to the event handler. The **DragManager.getFeedback()** method returns the drag feedback set by the **dragOver** event handler, and you can use it as we have in the example application for conditional logic.

Notice that in this example application instead of performing cleanup in a **dragComplete** handler, it is performed in the **dragDrop** event handler. This is necessary because our application does not allow duplicates, and if the user is moving an item from one Canvas to the other Canvas, that item should be removed from the drag initiator Canvas only if the item does not already exist in the drop target Canvas. If the drop target already has

the dragged item, then the move operation should not be completed, and the item should not be removed from the drag initiator Canvas.

If the code removing the item from the drag initiator Canvas were placed in a **dragComplete** handler, the item would never be deleted from the drag initiator during valid move operations, because the **dragComplete** event is dispatched **after** the **dragDrop** event. Thus by the time we check if the drag target already contains the item, for valid move operations the item has already been moved, thus our check will return a false positive and we will not remove the item from the drag initiator Canvas.

Resources on Drag and Drop in Flex

http://livedocs.adobe.com/flex/3/html/help.html?content=dragdrop_1.html

<http://www.switchonthecode.com/tutorials/simple-flex-drag-and-drop>

http://www.adobe.com/devnet/flex/quickstart/adding_drag_and_drop/

http://livedocs.adobe.com/flex/3/html/help.html?content=dragdrop_6.html

<http://blog.everythingflex.com/2007/06/18/simple-drag-and-drop-air/>