Adobe® Integrated Runtime (AIR™) Quick Starts for Flex 3 (Beta 1)

Quick Starts for Flex

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA

Chapter 1: Building the quick-start sample applications with Flex

The sample applications contained in these quick-start topics are some of the most important or popular aspects of specific Adobe® Integrated Runtime (AIR™) features.

This chapter contains the following sections:

Set up, download, and install

1 Set up your development environment so that you can work with the sample code and compile and test the applications.

2 Download the sample application source files from the online location indicated in each of the quick-start topics.

Note: *This is a sample application provided as is, for instructional purposes.*

3 Install the sample applications using either of the following techniques (depending on which development environment you're using).

See also

"Setting up for Flex Builder" on page 12

"Setting up for Flex SDK" on page 14

Use the Flex SDK and the command-line tools to install sample applications

1 Unzip the sample application files into a folder on your computer. Create a new folder for each, if you like.

2 With the SDK properly installed and configured (the SDK added to your class path), you're ready to test the samples.

CLOSE PROCEDURE

Use Flex Builder 3 to install sample applications

1 Start Flex Builder and create a new AIR project (see "Create AIR projects with Flex Builder" on page 25).

2 Name the project the same name as the sample application's main MXML file (for example, for the sample that contains TextEditor.mxml, name the project TextEditor).

3 Close Flex Builder.

4 Copy the sample application zip file to the folder containing the new AIR project and then unzip the file, overwriting the contents of the project.

5 Open Flex Builder and select the project from the Project Navigator.

CLOSE PROCEDURE

Next steps

Follow the instructions in the quick-start topics to edit, run, and debug the applications.

See also

Flex Builder users:

- Developing AIR applications with Flex Builder in *Developing AIR Applications with Adobe Flex*

Flex developers using the Flex SDK:

- Compiling an AIR application with the amxmlc compiler in *Developing AIR Applications with Adobe Flex*
- Compiling an AIR component or library with the acompc compiler in *Developing AIR Applications with Adobe Flex*
- Debugging using the AIR Debug Launcher in *Developing AIR Applications with Adobe Flex*
- Packaging an AIR application using the AIR Developer Tool in *Developing AIR Applications with Adobe Flex*

Chapter 2: Building a text-file editor

The Text Editor sample application shows a number of features of working with files in Adobe Integrated Runtime (AIR), including the following:

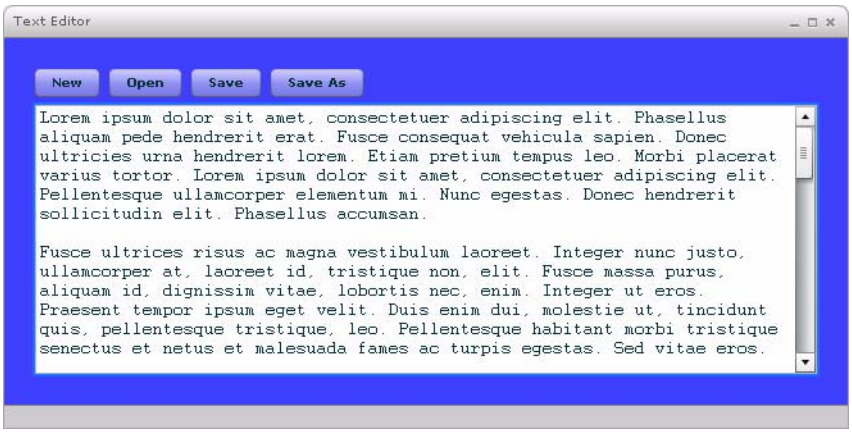
- Setting up a File object to point to a file path.
- Getting the system-specific path (`nativePath`) for the file.
- Using the FileMode and FileStream classes to read data from the file.
- Using the FileMode and FileStream classes to write data from the file.
- Using events to handle asynchronous processes.

Note: *This is a sample application provided, as is, for instructional purposes.*

This chapter contains the following sections:

Testing the application

The TextEditor application is intentionally simple. The intent is to show you the basics of how to work with files in Adobe AIR.



The installer (AIR) file for this sample application is available in the Samples directory included with the documentation at the AIR pre-release site.

The application is a straightforward, but simple, editor of plain text files. The application uses UTF-8 encoding for reading and writing all text files.

Files used to build the application

The application is built from the following source files:

File	Description
TextEditor.mxml	The main application file in MXML for Flex. Details of the code are discussed in "Understanding the code" on page 8.
TextEditor-app.xml	The AIR application descriptor file (see "The application descriptor file structure" on page 42)
icons/ApolloApp_16.png icons/ApolloApp_32.png icons/ApolloApp_48.png icons/ApolloApp_128.png	Sample AIR icon files.

To download the source files for this example, go at http://www.adobe.com/go/learn_air_flex3_qs.

See also

"Building the quick-start sample applications with Flex" on page 5

Understanding the code

This section contains the following topics:

This article does not describe all of the Flex components used in the MXML code for the file. For information on these, see the *Flex 3 Language Reference*.

Pointing a File object to a file

The `appCompleteHandler()` method sets the `defaultFile` File object to point to a pre-defined path:

```
defaultFile = File.documentsDirectory.resolve("test.txt");
```

This code sets the File object to point to the test.txt file in the user's documents directory. In Windows, this is the My Documents directory. In Mac OS, it is the /Users/*userName*/Documents directory.

The `defaultFile` file is later passed to Open and Save As pop-up windows, if the user has not yet selected a file path.

Reading a file

The `openFile()` method contains code that opens a file chooser dialog box. When the user selects a file and clicks the OK button. The `fileOpenSelected()` method is called. The method first initiates the `file` File object to point to the path specified by the user:

```
currentFile = event.file;
```

Next the `stream` FileStream object is closed (in case it currently has a file open) and the `stream` FileStream object are initialized:

```
if (stream != null)
{
    stream.close();
}
stream = new FileStream();
stream.openAsync(currentFile, FileMode.READ);
```

Note: *The `fileMode` parameter of the `openAsync()` method is set to `FileMode.READ`. This lets you read the file.*

Event handlers are set up to respond to the `complete` and `ioError` events:

```
stream.addEventListener(Event.COMPLETE, fileReadHandler);
stream.addEventListener(IOErrorEvent.IO_ERROR, readIOErrorHandler);
```

AIR begins reading the file asynchronously, and the runtime automatically starts reading the file in and firing events. Note that you can add these event listeners after calling the `openAsync()` method, because the runtime will complete executing this ActionScript code (the block of code that includes the calls to `addEventListener()` methods) *before responding* to any events.

Note: *This sample application shows how to read a file asynchronously. You can also read the file synchronously by calling the `open()` method when opening the file, rather than calling the `openAsync()` method. For more information, see "Working with the file system" on page 7.*

If the `stream` object dispatches an `ioError` event, the `readIOErrorHandler()` displays an error message for the end user.

When the file is read in fully, the `stream` object dispatches the `complete` event, and the `fileReadHandler())` method reads and processes the file.

The `readUTFBytes()` method of the `stream` object returns a string by reading UTF-8 characters from specified number of bytes. Since the `stream` object just dispatched `complete` event, the `bytesAvailable` property represents the total length of the file, and it is passed as the `length` property of the call to the `readUTFBytes()` method:

```
var str:String = stream.readUTFBytes(stream.bytesAvailable);
```

The following code replaces the line ending characters from the file with the "\n" newline character, which is used in a TextField object in a SWF file. It then assigns the string to the `text` property of the Text control:

```
var lineEndPattern:RegExp = new RegExp(File.lineEnding, "g");
str = str.replace(lineEndPattern, "\n");
mainTextField.text = str;
```

Writing data to a file

The `saveFile()` contains code that writes the text to the file.

First, the method checks to see if the main File object is set:

```
if (currentFile)
```

The `currentFile` object is undefined when the user first opens the application, and when the user clicks the `New` button. If there is no `currentFile` defined, then the `saveAs()` method is called (in the `else` block), which lets the user select a file path for saving the file.

Next the `stream` FileStream object is closed (if it is currently open) and then it is initialized:

```
if (stream != null)
{
    stream.close();
}
stream = new FileStream();
stream.openAsync(currentFile, FileMode.WRITE);
```

Note that the `mode` parameter of the `openAsync()` method is set to `FileMode.WRITE`. This lets you write to the file.

An event handlers is set up to respond to the any `ioError` events:

```
stream.addEventListener(IOErrorEvent.IO_ERROR, writeIOErrorHandler);
```

The following code replaces the "\n" newline character, which is used in a TextField object in a SWF file, with the line ending character used in text files in the file system (`File.lineEnding`) It then assigns the string to the `text` property of the Text control:

```
var str:String = mainTextField.text;
str = str.replace(/\r/g, "\n");
str = str.replace(/\n/g, File.lineEnding);
```

The `writeUTFBytes()` method of the `stream` object writes a string to the file in UTF-8 format and then closes the file:

```
stream.writeUTFBytes(str);
stream.close();
```

If the `stream` object dispatches an `ioError` event, the `writeIOErrorHandler()` displays an error message to the end user.

Chapter 3: Building a directory search application

The File Search sample application shows a number of features of working with files in Adobe Integrated Runtime (AIR), including the following:

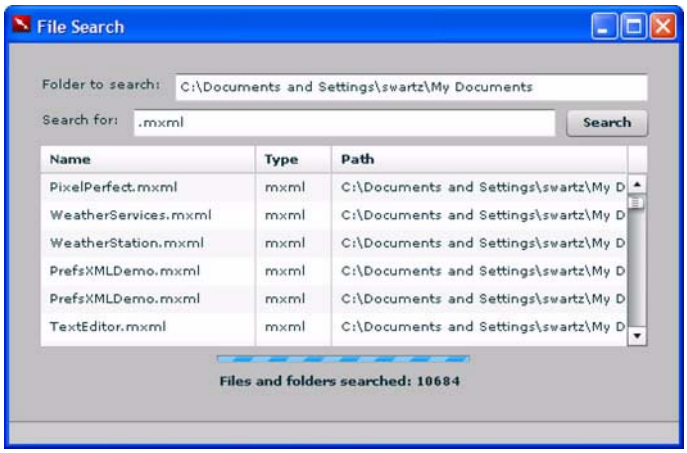
- Reading files asynchronously so that other ActionScript processes can take place as file data is read
- Getting the file extension from filenames
- Using the platform-specific `nativePath` property of a File object

Note: *This is a sample application provided, as is, for instructional purposes.*

This chapter contains the following sections:

Testing the application

The File Search application lets you search a directory—and its subdirectories—for files that have names matching the search pattern you specify:



Files used to build the application

The application includes the following source files:

File	Description
FileSearch.mxml	The main application file in MXML for Flex. Details of the code are discussed in "Understanding the code" on page 12.
FileSearch-app.xml	The Apollo application descriptor file (see "The application descriptor file structure" on page 42).
icons/ApolloApp_16.png icons/ApolloApp_32.png icons/ApolloApp_48.png icons/ApolloApp_128.png	Sample Apollo icon files.

You can download the source files for this example at http://www.adobe.com/go/learn_air_flex3_qs.

See also

"Building the quick-start sample applications with Flex" on page 5

Understanding the code

This section contains the following topics:

This document does not describe all of the Flex components used in the MXML code for the file. For information on these, see the Flex documentation.

Setting the root directory to search

The `init()` method sets the `text` value of the `folderPath` TextInput component to a pre-defined path (the user's documents directory):

```
folderPath.text=File.documentsDirectory.nativePath;
```

The `File.documentsDirectory` is a File object pointing to the user's documents directory (such as My Documents on Windows), and the `nativePath` property is a string value of the platform-specific path to the directory. For example, on Windows, this path could be:

```
C:\Documents and Settings\userName\My Documents
```

Whereas on Mac OS, it could be:

```
/Users/userName/Documents
```

The user can edit this value (in the TextArea component), and when the user clicks the Search button, the `search()` method checks to see if the path is valid, and displays an error message if it is not:

```
var dir:File = new File(folderPath.text);
if (!dir.isDirectory)
{
    Alert.show("Invalid directory path.", "Error");
}
```

Notice the difference between the `nativePath` property and the `url` property of a File, listed here for the same `directory` File object:

```
trace(directory.nativePath); // C:\Documents and Settings\swartz\My Documents
trace(directory.url); // file:///C:/Documents%20and%20Settings/swartz/My%20Documents
```

Searching the directory for matching files

The main search process lists directory contents asynchronously, one at a time. By doing this, other ActionScript-based processes (such as the progress bar animation and result listing) can take place as Apollo gets directory listing information (asynchronously) from the file system.

The `search()` method sets up event listeners for the `dirListing` event, which is dispatched when the `listDirectoryAsync()` process has completed:

```
dir.addEventListener(FileListEvent.DIRECTORY_LISTING, dirListed);
dir.listDirectoryAsync();
```

The `dirListed()` method processes the list of files in the current directory. This list is represented as the `files` property of the `dirListing` event. The method sets a `currentNodes` variable to this array:

```
currentNodes = event.files;
```

The `dirListed()` method iterates through each member of this array, to see if it is a directory. If it is a directory, the object is added to a list of current subdirectories (of the current directory being examined).

```
node = currentNodes[i];
if (node.isDirectory)
{
    currentSubdirectories.push(currentNodes[i]);
}
```

After the iteration is completed, the members of this array are added to a master array of directories to be searched, named `directoryStack`:

```
for (i = currentSubdirectories.length - 1; i > -1; i--)
{
    directoryStack.push(currentSubdirectories[i]);
}
```

At this point, now that processing of the current directory is complete, the application can call another asynchronous listing of the next directory in the stack (if there is one):

```
var dir:File = directoryStack.pop();
if (dir == null) {
    progress.visible = false;
    // There are no further directories to search. The search is completed.
} else {
    dir.addEventListener(FileListEvent.DIRECTORY_LISTING, dirListed);
    dir.listDirectoryAsync();
}
```

Displaying search results

As the `dirListed()` method iterates through each member of this array of contents of the current directory, it checks if the its name matches the Search pattern, and if so, it creates an object that is added to the ArrayCollection that is a data provider for the `resultsGrid` DataGrid component:

```
if (node.name.search(pattern) > -1)
{
    if (node.isDirectory)
    {
        fileExtension= "";
    }
    else
    {
        fileExtension = node.name.substr(node.name.lastIndexOf(".") + 1);
    }
    var newData:Object = {name:node.name,
                          path:node.nativePath,
                          type:fileExtension}
    resultData.addItem(newData);
}
```

Note: *The file extension, represented by the `fileExtension` variable, is extracted from the filename by using the `substr()` and `lastIndexOf()` methods of the `name` property of the File object. (The file extension is the string portion of the filename that follows the last occurence of the "." character in the filename.)*

Chapter 4: Reading and writing from an XML preferences file

This chapter shows the following Adobe Integrated Runtime (AIR) features:

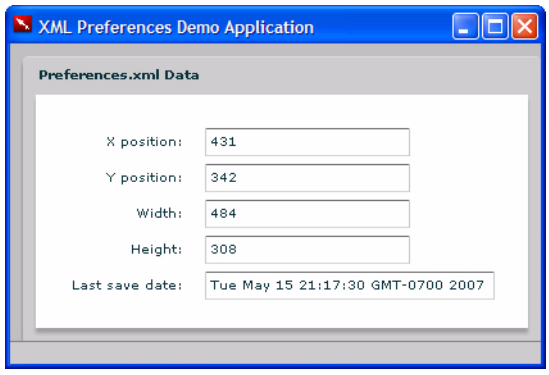
- Reading and writing to a text file
- Specifying a file in the Adobe AIR *application storage* directory
- Modifying the position, size, and visibility of an AIR application window
- Responding to the `closing` event dispatched by a Window object

Note: *This is a sample application provided, as is, for instructional purposes.*

This chapter contains the following sections:

Testing the application

The application is intentionally simple. It reads and writes data related to the position and size of the application, as well as the date saved. It also positions the window according to that data when the application opens:



The preferences are stored in a text file contains XML data. The application converts that data to an XML object, upon reading, and converts the XML data to text upon writing.

1. Open the application.
2. Resize and reposition the window.
3. Click the Save Preferences button.

The application writes the new data to the XML file.

4 Restart the application to see your preferences take effect.

Files used to build the application

The application is built from the following source files:

File	Description
PrefsXMLDemo.mxml	The main application file in MXML for Flex. Details of the code are discussed in "Understanding the code" on page 16.
PrefsXMLDemo-app.xml	The AIR application descriptor file (see "The application descriptor file structure" on page 42).
icons/ApolloApp_16.png icons/ApolloApp_32.png icons/ApolloApp_48.png icons/ApolloApp_128.png	Sample AIR icon files.

You can download the source files for this example at http://www.adobe.com/go/learn_air_flex3_qs.

For help on building this quick start sample application, see "Building the quick-start sample applications with Flex" on page 5.

Understanding the code

The following sections discuss how the code related to AIR works in the file.

This section does not describe all of the Flex components used in the MXML code for the file. For information on these, see the *Flex 3 Language Reference*.

Reading data from the XML file

The `appCompleteHandler()` method initializes the `prefsFile` File object to point to a pre-defined path and then call the `readXML()` method, which reads the data:

```
prefsFile = File.applicationStorageDirectory;
prefsFile = prefsFile.resolve("preferences.xml");
readXML();
```

`File.applicationStorageDirectory` points to the AIR application store directory, which is uniquely defined for each AIR application.

The `appCompleteHandler()` method also sets up an event handler to respond to the window closing (which saves the preferences data to the file):

```
stage.window.addEventListener(Event.CLOSING, windowClosingHandler);
```

The `readXML()` method sets up a File object and a FileStream object. The `fileMode` parameter of the call to the `open()` method is set to `FileMode.READ`, so that the FileStream object can read data from the file.

```
stream = new FileStream();
stream.open(prefsFile, FileMode.READ);
```

The `open()` method of the `stream` object opens the file synchronously and begins reading data into the read buffer.

The `processXMLData()` event method processes the XML data and closes the file. The `bytesAvailable` property of the FileStream object is the number of bytes in the read buffer, which is all of the bytes from the file (since the file is read synchronously):

```
prefsXML = XML(stream.readUTFBytes(stream.bytesAvailable));
stream.close();
```

Repositioning and resizing the application window

In the application.xml file, which defines properties of the application, the `visible` attribute of the `rootContent` property is set to `"false"`. The window is resized and repositioned before the window is made visible.

The `window` property of the Stage object contains properties of the AIR window.

The `processXMLData()` method resizes and repositions the window based on data in the XML preferences object (which was just read in from the preferences file):

```
stage.window.x = prefsXML.windowState.@x;
stage.window.y = prefsXML.windowState.@y;
stage.window.width = prefsXML.windowState.@width;
stage.window.height = prefsXML.windowState.@height;
```

Note: *This code sample uses E4X notation, which was introduced in ActionScript 3.0. For example, `prefsXML.windowState.@x` is the value of the `x` attribute of the `windowState` property of the `prefsXML` XML object. For more information, see the Working with XML (http://livedocs.adobe.com/flex/2/docs/00001910.html#118717) chapter of the Programming ActionScript 3.0 book.*

The `readXML()` method makes the window visible after the `processXMLData()` method returns:

```
stage.window.visible = true;
```

Writing XML data to the file

The `writeXMLData()` converts the XML data to a string, adds the XML declaration to the beginning of the string, and replaces line ending characters with the platform-specific line ending character (represented by the AIR `File.lineEnding` constant):

```
var outputString:String = '<?xml version="1.0" encoding="utf-8"?>\n';
outputString += prefsXML.toXMLString();
outputString = outputString.replace(/\n/g, File.lineEnding);
```

It then sets up and uses a FileStream object for writing the data. Note that `FileMode.WRITE` is specified as the `fileMode` parameter in the `open()` method. This specifies that the FileStream object will be able to write to the file (and will truncate any existing data before writing):

```
stream = new FileStream();
stream.open(prefsFile, FileMode.WRITE);
```

Next, the `writeUTFBytes()` method is called, which writes the string version of the XML data to the file (as UTF-8 data):

```
stream.writeUTFBytes(outputString);
```

Since this file was opened for synchronous operations (using the `open()` method) and the write method is included *within* the event handler for the Window object's `closing` event, the file writing *will* complete before the window (and application) actually close. This application uses synchronous read and write operations because the XML preferences file is relatively small. If you were to want to write a file asynchronously, you would want to cancel the `closing` event, and explicitly close the application by calling the `Shell.shell.exit())` method in an event handler for the `outputProgress` event dispatched by the FileStream object.

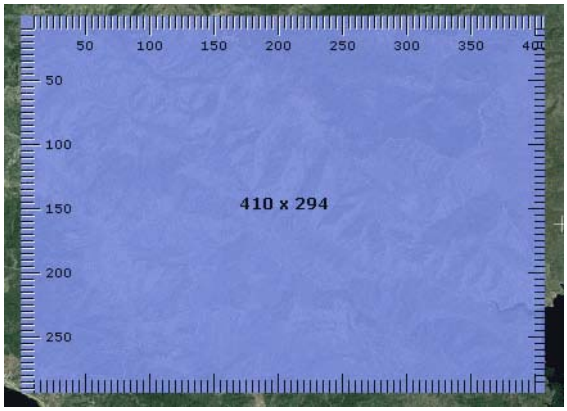
Chapter 5: Interacting with a window

The PixelPerfect sample application performs basic tasks on window objects. These tasks include creating, resizing, moving, and closing windows.

This chapter contains the following sections:

Using the PixelPerfect AIR application

The PixelPerfect sample application creates a transparent pixel ruler on the operating system desktop. The pixel ruler is represented by a window that can be moved, resized, and changed with respect to display state.



Note: *This is a sample application provided, as is, for instructional purposes.*

The PixelPerfect application demonstrates how you can use the AIR window API to:

- Open multiple windows
- Obtain a reference to a window.
- Use mouse events to trigger operations on a window.
- Use the `startResize()` method of the NativeWindow class to resize a window.
- Use the constants of the NativeWindowResize class to specify how a window resize operation is completed.
- Use the `startMove()` method of the NativeWindow class to move a window.
- Use the `close()` method of the NativeWindow class to close a window.
- Use full screen mode

See also

"Understanding the code" on page 20

"Building the quick start sample applications" on page 5

Create the PixelPerfect project as a Flex application

The application includes the following source files:

PixelPerfect.as An ActionScript class that launches a new ruler window and then exits.

Ruler.as The ruler window class. Extends the NativeWindow class. Located in package: `com.adobe.pixelperfect`.

PixelPerfect-app.xml The AIR application descriptor file.

icons/AIRApp_128.png Sample AIR icon file.

Install the source files in Flex Builder

1 Open Flex Builder.

2 Select File > New, and then select AIR Project.

3 Enter `PixelPerfect` as the Project Name. Click Next.

4 Leave the Use Default Location and Use Default SDK options selected. Click Next.

5 Change the Main application file to `PixelPerfect.as` on the Set The Build Paths For The New AIR Project page,. Leave the other settings as is. Click next to advance to the Application XML Properties page.

6 Fill in the Name, Description, and Copyright text boxes, as desired and click Finish. Your project is created.

7 Copy the source files into the project folder (maintaining the package folder structure).

CLOSE PROCEDURE

Install the application

1 Double-click the PixelPerfect.air file.

2 Click Continue. AIR installs the application.

CLOSE PROCEDURE

Run the application

- In Windows, double-click the desktop shortcut for the application or select the file from the Start menu.
- In Mac OS, double-click the PixelPerfect application icon, which is installed in the Applications subdirectory of the user directory by default.

CLOSE PROCEDURE

Understanding the code

PixelPerfect presents a simple user interface that lets users access the NativeWindow API to modify application windows based on user input. PixelPerfect supports the following user interactions:

Context menu Right-click on the window to open a menu of window commands.

Drag corners or sides to resize The `mouseDown` event triggers application logic to run that calculates the mouse position and, if appropriate, makes a call to the `NativeWindow.startResize()` method.

Drag the middle to move The `mouseDown` event triggers application logic to run that calculates the mouse position and, if appropriate, makes a call to the `NativeWindow.startMove()` method.

Arrow keys move one pixel at a time The `keyDown` event triggers application logic that detects if an arrow key was pressed. If it was, the `NativeWindow.x` or `NativeWindow.y` property is either increased or decreased by 1, depending on whether the arrow that was pressed was the DOWN arrow key, the UP arrow key, the LEFT arrow key, or the RIGHT arrow key. This process occurs each time the user presses an arrow key.

Shift + arrow keys resize one pixel at a time The `keyDown` event triggers application logic that detects if the SHIFT key was pressed in combination with an arrow key. If such a key combination was pressed, the `NativeWindow.width` or `Window.height` property is either increased or decreased by 1, depending on whether the arrow that was pressed was the DOWN arrow key, the UP arrow key, the LEFT arrow key, or the RIGHT arrow key. This process occurs each time the user presses the SHIFT key in combination with an arrow key.

Mouse wheel changes opacity The event listener that was created to detect activity from the mouse wheel makes a call to an event handler that determines the extent and direction in which the mouse wheel is turned. These values are used to modify the alpha property, which affects the level of transparency that is applied to the background color.

Double-click to quit The `doubleClick` event triggers application logic that makes an explicit call to the `Window.close()` method.

Shift+n to create a new window The `keyDown` event triggers application logic that detects if Shift-N was pressed. If so, a new instance of the `Ruler` class (which extends `NativeWindow`) is created, displaying a new ruler window.

Creating a Window

To create a window, PixelPerfect instantiates a new `Ruler` object. The Ruler class extends the NativeWindow class. In the Ruler constructor, the NativeWindowInitOptions object is created and passed to the NativeWindow super class constructor:

```
public function Ruler(width:uint = 300,
                      height:uint = 300,
                      x:uint = 50,
                      y:uint = 50,
                      alpha:Number = .4){
    var winArgs:NativeWindowInitOptions = new NativeWindowInitOptions();
    winArgs.systemChrome = NativeWindowSystemChrome.NONE;
    winArgs.transparent = true;
    super(false, winArgs);

    //...
```

The rest of the constructor initializes the properties of the new window and its members.

Resizing the application window

The `startResize()` method receives a constant value defined by the `NativeWindowResize` class that indicates the edge or corner of the window from which the user is dragging the window to resize it. The `startResize()` method triggers a system-controlled resizing of the window.

The application also explicitly sets the window bounds in response to keyboard events. In this case, no resize events are involved. The following code snippet sets the window bounds inside a switch statement:

```
switch (e.keyCode)
    {
        case Keyboard.DOWN:
            if (e.shiftKey)
                height += 1;
            else
                y += 1;
            drawTicks();
            break;
        case Keyboard.UP:
            if (e.shiftKey)
                height -= 1;
            else
                y -= 1;
            drawTicks();
            break;
        case Keyboard.RIGHT:
            if (e.shiftKey)
                width += 1;
            else
                x += 1;
            drawTicks();
            break;
        case Keyboard.LEFT:
            if (e.shiftKey)
                width -= 1;
            else
                x -= 1;
            drawTicks();
            break;
        case 78:
            createNewRuler();
            break;
}
```

This code snippet also shows the key events used to move the window and to create new windows.

Moving the application window

The `startMove()` method is used to begin a system-controlled move of the window. The PixelPerfect sample application uses this method when the mouse device controls the move.

When the arrow keys control the parameters of the move, PixelPerfect changes the window bounds, one pixel at a time, by incrementing or decrementing the `x` or `y` property of the window. (The code is included in the `switch` statement in the preceding code snippet.)

Changing the transparency of the application window

The visible background of the window is a Sprite object that was added to the ruler stage. The application changes the transparency of the background by adjusting the alpha property of the background sprite in response to mouse wheel events:

```
private function onMouseWheel(e:MouseEvent):void
{
    var delta:int = (e.delta < 0) ? -1 : 1;
    if (sprite.alpha >= .1 || e.delta > 0)
    sprite.alpha += (delta / 50);
}
```

For a window to be transparent against the desktop, the window must be created with transparency enabled by setting `transparent` property to `true` in the `NativeWindowsInitOptions` object passed to the window constructor. The transparency state of a window cannot be changed once the window is created.

Closing the application window

When a `doubleclick` event is dispatched, PixelPerfect calls the `close()` method of the window.

```
private function onDoubleClick(e:Event):void
{
    close();
}
```

When a windows's `close()` method is called, the NativeWindow object does not dispatch a `closing` event. To allow cancellation of the close operation, dispatch the `closing` event in your event handler:

```
private function onDoubleClick(e:Event):void
    var closing:Event = new Event(Event.CLOSING,true,true);
    dispatchEvent(closing);
    if(!closing.isDefaultPrevented()){
        close();
    }
}
```

A `close` event is dispatched after the window is closed, signaling the completion of the close process. After a window is closed, it cannot be reopened. Closing the window explicitly frees the resources that are associated with the window so that they can be garbage collected. The application is responsible for closing its windows so that the associated resources can be freed. When the last window of an application is closed, the application also exits. This default behavior can be changed by setting the Shell object autoExit property to `false`.

For more information about the methods and properties of the AIR windowing API, see the AIR ActionScript 3.0 Reference.

See also

Working with windows

Chapter 6: Creating a transparent window application

The WeatherStation sample application illustrates the following Adobe Integrated Runtime (AIR) features:

- A window with a transparent background and a non-rectangular border
- Window control buttons for minimizing and closing the window
- Moving the window by responding to the "move" event

The WeatherStation sample also shows how some powerful Flash Player features can be used within an Adobe AIR application, including:

- Calling an HTTP service to retrieve weather forecast data
- Storing and retrieving application settings using a SharedObject

Note: *This is a sample application provided, as is, for instructional purposes.*

This chapter contains the following sections:

About transparent window applications

Adobe AIR supports both opaque and transparent application windows. Transparent windows let the operating system's desktop show through, while opaque windows obscure the desktop area behind them.

An AIR window can use the borders, title bar, menu bar, and window control buttons (known collectively as *system chrome*) that are standard for the operating system. Your application uses the standard system chrome elements when the `windowMode` attribute of the <rootContent> element in the application descriptor file is set to systemChrome (the default setting). A window that uses system chrome is always opaque.

The Flex WindowedApplication component provides an alternative set of window chrome elements and a rectangular window frame. The WindowedApplication component's window is always opaque as well.

If you want your application to be transparent, it should not use system chrome or the WindowedApplication component. This means that the application must provide its own mechanisms for controlling the window and its background.

The WeatherStation sample application

The WeatherStation application window is partially transparent and it does not use system chrome, so it displays rounded and irregular borders instead of the usual rectangular frame. It also displays images that extend beyond the visual border of the application.

The application queries weather forecast data from the Yahoo! Weather service based on a U.S. ZIP code. Data is returned from the service as XML and the application then parses, formats and displays the data. The following screenshot shows the application in action:

Files used to build the application

The application includes the following source files:

File	Description
WeatherStation.mxml	The main application file in MXML for Flex 2. Details of the code are discussed in the Understanding the code section.
WeatherServices.mxml	Defines the HTTPService call and its parameters.
WeatherStation-app.xml	The application descriptor file.
assets/close_icon.png assets/minimize_icon.png	Images used for the custom window chrome buttons.
icons/ApolloApp_16.png icons/ApolloApp_32.png icons/ApolloApp_48.png icons/ApolloApp_128.png	Sample icon files.

To download the source files for this example, go to *http://download.macromedia.com/pub/labs/apollo/quickstart_apps/WeatherStation.zip* (395K).

See also

"Building the quick-start sample applications with Flex" on page 5

Understanding the code

This section contains the following topics:

"Minimizing and closing the application window " on page 27

"Casting a shadow on the desktop " on page 28

The WeatherStation application performs other interesting functions such as communicating with a remote HTTP service, parsing an XML result using the new ActionScript 3.0 E4X syntax, and storing and retrieving settings data using a SharedObject. For more information about these functions, see *Programming ActionScript 3.0* and the *ActionScript 3.0 Language Reference*.

This article does not describe all of the Flex components used in the MXML code for this application. For more information on the Flex components, see the *Flex 2.0 Language Reference*.

Setting the window transparency

The transparency of the application window is controlled in two places.

First, the application descriptor file for the application contains a `rootContent` element that has two attributes that affect window transparency: the `systemChrome` attribute and the `transparent` attribute. The `transparency` attribute must be set to `true`. In addition, set the `systemChrome` attribute to `none` or the standard system window title bar and buttons are used, and the application background will be opaque. The following example shows the `rootContent` element in the application descriptor file:

```
<rootContent systemChrome="none"
      transparent="true"
      visible="true">WeatherStation.swf</rootContent>
```

Second, the CSS styles that are applied to the Flex 2 Application component can affect how that component's background is displayed and whether it is opaque or transparent. To make sure the background is transparent, the WeatherStation.mxml file includes the following CSS style declaration:

```
<mx:Style>
   Application
   {
      background-color:"";
      background-image:"";
      padding: 0px;
   }

   /* additional style declarations... */
</mx:Style>
```

Moving the application window

A user can move the WeatherStation window around on the desktop by clicking anywhere in the background of the window, holding the mouse down, and dragging the window to another location. To trigger the window-moving process, the application monitors and responds to the `MouseEvent.MOUSE_DOWN` event.

First, the Application declaration specifies that the initApp() method will handle the creationComplete event.

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns="*"
    usePreloader="false"
    creationComplete="initApp()"
    layout="absolute"
    width="250"
    paddingRight="0"
    paddingLeft="0">
```

After the application is loaded, the `initApp()` method sets up mouseDown event listeners on the two visible VBox components. When the user presses the mouse button while the cursor is over one of these VBox components, the `onMouseDown()` method is loaded.

```
// adds mouseDown listeners so a click and drag on the background or the
// display area lets you move the window

this.bgBox.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
this.tempBox.addEventListener(MouseEvent.MOUSE_DOWN, onMouseDown);
```

The `onMouseDown()` method shows how simple it can be to start the window moving sequence:

```
private function onMouseDown(evt:MouseEvent):void
{
    stage.window.startMove();
}
```

Calling the `window.startMove()` method results in the window moving around the desktop in response to mouse movements until the mouse button is released. You don't need to write additional code to control the window moving and stopping sequence; it is handled automatically.

Minimizing and closing the application window

The WeatherStation application uses small Button components to let the user minimize or close the application.

Each button's click event is set to trigger an appropriate method, as follows:

```
<mx:Button id="minimizeBtn"
    icon="@Embed('assets/minimize_icon.png')"
    width="16"
    height="16"
    click="onMinimize(event)" />

<mx:Button id="closeBtn"
    icon="@Embed('assets/close_icon.png')"
    width="16"
    height="16"
    click="onClose(event)" />
```

When the minimize button is clicked, the `onMinimize()` method calls the `window.minimize()` method:

```
private function onMinimize(evt:MouseEvent):void
{
    stage.window.minimize();
}
```

When the close button is clicked, the `onClose()` method calls the `window.close()` method:

```
private function onClose(evt:MouseEvent):void
{
    stage.window.close();
}
```

The `window.close()` method terminates the application. It does so asynchronously, and a close event is started when the application is about to stop. The application can listen for the close event and perform various clean-up or housekeeping functions before the application stops.

Casting a shadow on the desktop

The WeatherStation window appears to cast a shadow on the desktop as if it were floating above it. In fact, the drop shadow is not applied to the window itself. It is applied to the `bgBox` VBox component that's used as a background element, and it casts a shadow on the applications window's transparent background. This makes it seem like the shadow is falling on the desktop instead.

To achieve this effect, the application first defines an instance of the `flash.filters.DropShadowFilter` class.

```
public var shadowFilter:DropShadowFilter;
```

The `initApp()` method then sets the parameters of the filter and applies the filter to the `bgBox` component and a number of other components, as follows:

```
// creates a generic drop shadow to use on components that don't accept CSS shadow styles
shadowFilter = new DropShadowFilter();
shadowFilter.color = 0x000000;
shadowFilter.alpha = 0.4;
shadowFilter.blurX = 5;
shadowFilter.blurY = 5;
shadowFilter.distance = 5;

// 'external' shadows
addShadow(this.bgBox);
addShadow(this.largeImage);

// 'internal' shadows
addShadow(this.locationTxt);
addShadow(this.tempTxt);
addShadow(this.conditionTxt);
addShadow(this.additionalTxt);
```

Because the `largeImage` Image object extends beyond the boundaries of the `bgBox` component's background, it also needs a shadow to make the illusion complete. The other display components are only given shadows because they seem to stand out better that way.

The same DropShadowFilter instance is applied to each of the components in the `addShadow()` method:

```
/**
 * Adds a standard drop shadow to a display object.
 */

public function addShadow(comp:DisplayObject):void
{
    comp.filters = [this.shadowFilter];
}
```

Each component is passed as an argument to this method and treated as a DisplayObject instance (so the same method could be used for DisplayObject instances that are not Flex `UIComponent` instances too). Each object's `filters` array is then set to an array containing the one `shadowFilter` object.

Chapter 7: Launching native windows

The Window Launcher sample application shows a number of features for opening native windows in Adobe Integrated Runtime (AIR), including the following:

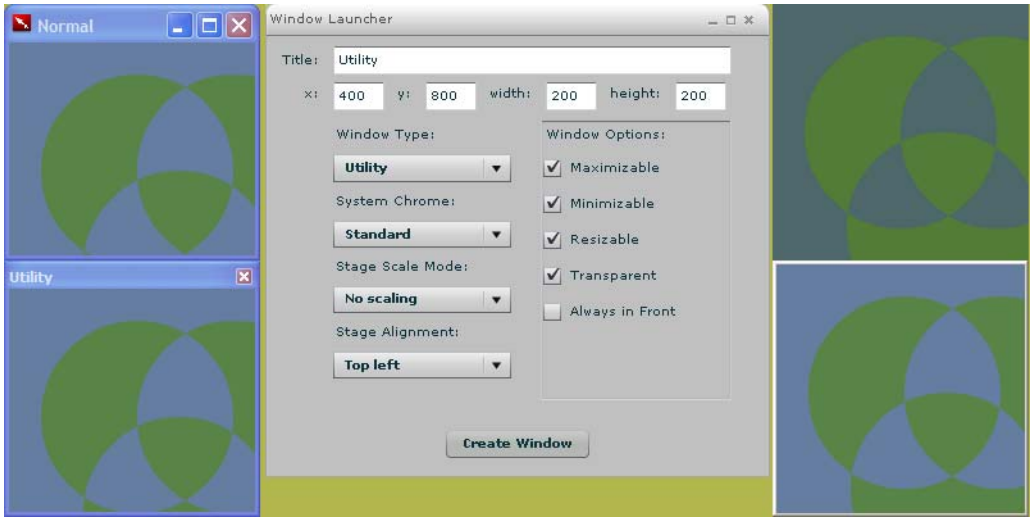
- Setting the window type.
- Setting the window chrome
- Turning on transparency
- Setting the window title
- Setting the window position and size
- Setting the whether a window is resizable, minimizable, and maximizable

Note: *This is a sample application provided, as is, for instructional purposes.*

This chapter contains the following sections:

Installing and testing the application

The WindowLauncher application is intentionally simple. The intent is to show you the basics of how to open new windows in Adobe AIR.



The Window Launcher sample application on Microsoft Windows, flanked by four different kinds of native windows.

Files used to build the application

The application is built from the following source files:

File	Description
WindowLauncher.mxml	The main application file in MXML for Flex. Details of the code are discussed in "Understanding the code" on page 31.
WindowLauncher-app.xml	The AIR application descriptor file.
icons/AIRApp_16.png icons/AIRApp_32.png icons/AIRApp_48.png icons/AIRApp_128.png	Sample AIR icon files.

To download the source files for this example, go to *http://www.adobe.com/go/learn_air_flex3_qs*.

See also

"Building the quick-start sample applications with Flex" on page 5

Understanding the code

This section contains the following topics:

This article does not describe the Flex components used in the MXML code for the file. For information on these, see the *Flex 2.0 Language Reference*.

Setting window initialization options

Before a new native window can be created, you must first create a NativeWindowInitOptions object and set its properties. These properties cannot be changed after the window is created. The following lines in the `createNewWindow()` method of the sample application create the `NativeWindowInitOptions` object and set its properties based on the MXML controls of the launcher window:

```
var options:NativeWindowInitOptions =
    new NativeWindowInitOptions();

options.maximizable = maximizableOption.selected;
options.minimizable = minimizableOption.selected;
options.resizable = resizableOption.selected;
options.transparent = transparentOption.selected;
options.systemChrome = Chrome.selectedItem.optionString;
options.type = windowType.selectedItem.optionString;
```

Note: *Transparency is not supported for windows that use system chrome. Likewise, system chrome can not be used for lightweight windows. These rules are enforced in Window Launcher by validating the settings as they are made.*

Creating the window

Create a native window with the NativeWindow class constructor. The first parameter determines whether the window will be visible when first created. This example sets the parameter to false so that the changes made to the window size, position, and contents are not visible. The second parameter is the NativeWindowInitOptions object created earlier. The following lines from the `createNewWindow()` method create the new window and set the properties based on the MXML controls of the launcher window:

```
var newWindow:NativeWindow = new NativeWindow(false,options);

newWindow.title = titleString.text;
newWindow.alwaysInFront = alwaysInFrontOption.selected;
newWindow.x = Number(xPosition.text);
newWindow.y = Number(yPosition.text);
newWindow.width = Number(widthValue.text);
newWindow.height = Number(heightValue.text);
```

These properties of the window can be changed at any time before a window is closed.

The width and height properties set the outer dimensions of the window, including the size of any system chrome. Thus a window without system chrome has a smaller client area than a window with chrome.

Adding content and setting stage properties

A native window is created with a stage that represents the client (drawable) area of the window. The stage is the root container of the display tree. Add content to the client area of a native window by adding a DisplayList object (or an object that inherits from DisplayList) to the stage or another DisplayObject already on the stage with the `addChild()` method. The stagescaleMode and align properties determine how the stage (and any objects on it) behave when the window is resized.

The stageScaleMode property sets how the stage scales and clips when a window is resized:

noScale The stage is not scaled. The size of the stage changes directly with the bounds of the window. Objects may be clipped if the window is resized smaller.

showAll The stage scales while maintaining the original aspect ratio. Objects in the original view area are never clipped. Padding is added to the stage area when the window is resized to a different aspect ratio.

noBorder The stage scales while maintaining the original aspect ratio. Objects in the original view are clipped if the window is resized to a different aspect ratio.

exactFit The stage scales without regard to the original aspect ratio. Objects are distorted when the window is resized to a different aspect ratio.

The stage align property sets the edge or corner to which the stage origin is anchored when the window is resized. For example, if you set the stage align property to bottomRight, when the window is resized larger from the bottom, right corner, the stage origin also moves down and to the right in relation to the window. To then place an object in the top, left corner, you use negative coordinates. The stage origin always starts in the top, lefthand corner of the window no matter which alignment value is chosen.

The following lines from the `createNewWindow()` method of the sample application set the stage properties based on the MXML control settings and draws some example content into the window by creating a Sprite and calling its graphics methods to draw a grid of rectangles.

```
newWindow.stage.align = stageAlignment.selectedItem.optionString;
newWindow.stage.scaleMode = ScaleMode.selectedItem.optionString;
var client:Sprite = new Sprite();
var rectSize:int = 40;
var rectSpace:int = 4;
with(client.graphics){
    lineStyle(1,0,1);
    beginFill(0x234578,.5);
    for(var i:int = 0; i <= Math.floor(newWindow.stage.stageWidth/rectSize); i++){
        for (var j:int = 0; j <= Math.floor(newWindow.stage.stageHeight/rectSize); j++){
            drawRoundRect(i*rectSize,j*rectSize,
                          rectSize-rectSpace,rectSize-rectSpace,10,10);
        }//j loop
    }//i loop
    endFill();
}
newWindow.stage.addChild(client);
```

Adding interactivity and making the window visible

The final lines of the `createNewWindow()` method add event handlers to the new window. Two event handlers are added to windows with no chrome to move and close the window.

```
//Add handlers to move and close the window if there is no chrome.
if(options.systemChrome == NativeWindowSystemChrome.NONE){
    newWindow.stage.doubleClickEnabled = true;
    newWindow.stage.addEventListener(KeyboardEvent.KEY_DOWN,
        function(e:Event):void{e.target.stage.window.close();});
    newWindow.stage.addEventListener(MouseEvent.MOUSE_DOWN,
        function(e:Event):void{e.target.stage.window.startMove();})
}
```

After the window has been created, content added, and event handlers attached, the final step is to make the window visible:

```
newWindow.visible = true;
```

Chapter 8: Dragging, copying, and pasting data

The Scrappy sample application shows a number of features of the drag-and-drop and copy-and-paste APIs in Adobe Integrated Runtime (AIR), including the following:

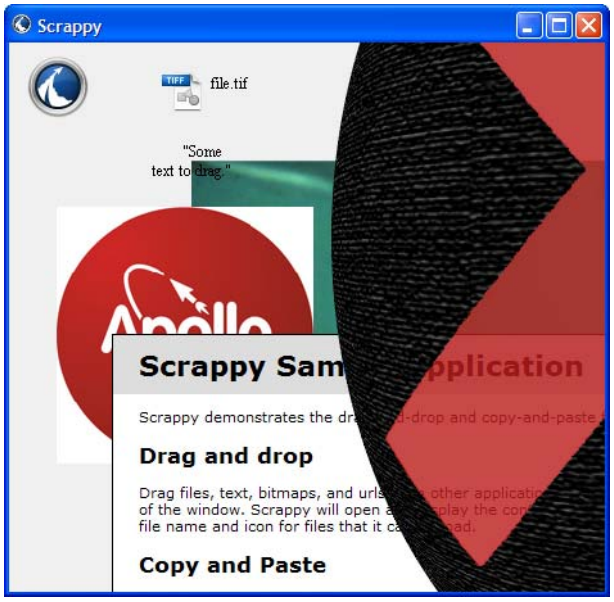
- Creating a TransferableData object to contain the information or objects to be transferred through drag and drop or the system clipboard.
- Starting a drag operation.
- Receiving dropped information or objects.
- Rendering the standard data formats using Adobe AIR and Flash objects.
- Writing to the system clipboard.
- Reading from the system clipboard.

Note: *This is a sample application provided, as is, for instructional purposes.*

This chapter contains the following sections:

Installing and testing the application

The Scrappy application is intentionally simple. The intent is to show you the basics of how to transfer data using drag and drop and copy and paste in Adobe AIR.



To use Scrappy, drag files, bitmaps, URLs, and text to the window. The application accepts dragged items in any of the standard formats supported by AIR. You can also paste (v) items into the window, as well as cut (x), copy (c) and delete (Del) existing items.

Note: *Not all applications post bitmap data to the clipboard in a format supported by AIR.*

Files used to build the application

The application is built from the following source files:

File	Description
Scrappy.as	The main application file in ActionScript. Details of the code are discussed in "Understanding the code" on page 36.
scraps/Scrap.as	The base class for displaying the dragged or pasted information.
scraps/TextScrap.as	Extends Scrap to display text.
scraps/BitmapScrap.as	Extends Scrap to display images.
scraps/HTMLScrap.as	Extends Scrap to display HTML and XML documents.
scraps/FileScrap.as	Extends Scrap to load and display files.
AboutWindow.as	Opens a window to display the about.html file.

File	Description
about.html	Describes this sample application and provides some items to drag into the main window.
Scrappy-app.xml	The AIR application descriptor file (see "The application descriptor file structure" on page 42)
icons/AIRApp_16.png icons/AIRApp_32.png icons/AIRApp_48.png icons/AIRApp_128.png	Sample AIR icon files.

To download the source files for this example, go to *http://www.adobe.com/go/learn_air_flex3_qs*.

For help on building this quick start sample application, see "Building the quick-start sample applications with Flex" on page 5.

Understanding the code

The following sections discuss how the code related to AIR works in this sample application.

This article does not describe all of the ActionScript classes used in the application. For information on these, see the *Flex 3 Language Reference*.

Note: *This article demonstrates the use of the flash.desktop.DragManager class in an ActionScript project. If you are building a Flex application, you should use the Flex mx.managers.DragManager class.*

Creating the canvas

Scrappy uses a full-window sprite that acts as the window backdrop and the target for drag-and-drop operations. To allow a Sprite, or other display object to act as a receiver for a drag gesture, you must listen for the NativeDragEvents dispatched from that object.

The Scrappy class defines the following members for creating the window, the drop target sprite and the event listeners:

```
public var dragTarget:Sprite = new Sprite();
public var about:NativeWindow;

public function Scrappy():void{
    super();
    addEventListener(Event.ADDED_TO_STAGE, onStaged);
    stage.align = StageAlign.TOP_LEFT;
    stage.scaleMode = StageScaleMode.NO_SCALE;
    stage.stageFocusRect = false;

    dragTarget.focusRect = false;
    addChild(dragTarget);
}

public function onStaged(event:Event):void{
    dragTarget.addEventListener(NativeDragEvent.NATIVE_DRAG_ENTER,onDragIn);
    dragTarget.addEventListener(NativeDragEvent.NATIVE_DRAG_DROP,onDrop);
    dragTarget.addEventListener(NativeDragEvent.NATIVE_DRAG_EXIT,onDragExit);

    stage.addEventListener(MouseEvent.MOUSE_DOWN,onMouseDown);
    stage.addEventListener(KeyboardEvent.KEY_DOWN,onKeyDown);

    stage.window.addEventListener(NativeWindowBoundsEvent.RESIZE, onResize);
    stage.window.addEventListener(Event.CLOSE,onClose);
}
```

The important events for drag-and-drop are nativeDragEnter, which tells you when a drag gesture enters within the display object area; nativeDragExit, which tells you when a drag gesture leaves the display object area; and nativeDragDrop, which tells you that the drop has occurred on the display object. You can also listen for the nativeDragOver event instead of the nativeDragEnter event if you need to track the position of the mouse over the target. The target will dispatch nativeDragOver events whenever the mouse moves (as well as on a short interval)

The keyDown event on the stage is used to check for the paste command. The event listener is attached to the stage because a sprite will only receive keyboard events when it has focus.

Dropping data onto the canvas

To allow a user to drop data onto a target, you must:

1. Check the format of the data.
2. Accept the drag gesture with DragManager.acceptDrop().
3. Handle the nativeDragDrop event.

Checking the dragged data

To check the data being dragged into the target, access the transferable property of the NativeDragEvent object:

```
public function onDragIn(event:NativeDragEvent):void{
    var transferable:TransferableData = event.transferable;
    if(transferable.hasFormat(TransferableFormats.BITMAP_FORMAT) ||
        transferable.hasFormat(TransferableFormats.FILE_LIST_FORMAT) ||
        transferable.hasFormat(TransferableFormats.TEXT_FORMAT) ||
        transferable.hasFormat(TransferableFormats.URL_FORMAT) ||
        transferable.hasFormat("SCRAP")){
            DragManager.acceptDragDrop(dragTarget);
}
}
```

The Scrappy application can accept each of the four standard formats. It also defines a custom format "SCRAP", which contains a reference to a Scrap object created within the Scrappy application itself.

In some situations, you might also want to check the allowedActions property of the event object before accepting the drop.

Accepting the gesture

The call to DragManager.acceptDrop() allows the passed in display object to receive the drop. The DragManager will change the display cursor to indicate that the drop is possible. If the user releases the mouse button while over the same display object (and before a nativeDragExit event occurs), then the designated display object will dispatch a nativeDragDrop event.

Taking the drop

After the drop occurs on an eligible display object, you can access the data through the transferable property of the nativeDragDrop event object. You may also set a drag-and-drop action to indicate which of the three actions, copy, move, or link should be taken. The chosen action is communicated back to the drag initiator object in the native-DragComplete event.

Scrappy passes the TransferableData object from the nativeDragDrop event to the Scrap class factory method, which creates new Scrap objects based on the data format, and adds the returned display object to the stage.

```
public function onDrop(event:NativeDragEvent):void{
    DragManager.dropAction = "copy";
    var scrap:Scrap =
        Scrap.createScrap(event.transferable,false,event.stageX,event.stageY);
    addChild(scrap);
    scrap.grabFocus();
}
```

Displaying information: The Scrap class

The scrap package classes act as containers for information dropped or pasted into the Scrappy window. The base Scrap class extends Sprite to add function for handling user interaction with scrap objects, including dragging out of the window, deleting, and copying. The Scrap class also implements a static factory method for creating new scrap objects.

The scrap subclasses extend Scrap to handle the standard formats of data that can be transferred. To render the data, these classes add child display objects to the scrap, such as a TextField or an HTMLControl. They also handle adding data in the correct format to the TransferableData object when a scrap is dragged out of the application. The Scrap subclasses are:

TextScrap Uses a TextField to display text data.

BitmapScrap Uses a Bitmap object to display bitmap data.

HTMLScrap Uses an HTMLControl to load and display URLs (including file URLs).

FileScrap Loads and displays the file contents, or displays the file name and icon. For the handful of file types Scrappy understands, the FileScrap constructor loads the data and creates one of other types of Scrap object to render it.

Scrap Factory

The scrap factory class, Scrap.createScrap(), takes a TransferableData object, checks the data formats available, and then creates the appropriate type of scrap object to display the data.

```
public static function createScrap(
        data:TransferableData, makeCopy:Boolean, placeX:int, placeY:int):Scrap{

    var scrap:Scrap;
    if(data.hasFormat("SCRAP")){
        scrap = Scrap(data.dataForFormat("SCRAP",TransferableTransferMode.ORIGINAL_ONLY));
    } else if(data.hasFormat(TransferableFormats.BITMAP_FORMAT)){
        scrap = new BitmapScrap(BitmapData(data.dataForFormat(
                                TransferableFormats.BITMAP_FORMAT)));

    } else if(data.hasFormat(TransferableFormats.URL_FORMAT)){
        scrap = new HTMLScrap(String(data.dataForFormat(
                                TransferableFormats.URL_FORMAT)));

    } else if(data.hasFormat(TransferableFormats.TEXT_FORMAT)){
        scrap = new TextScrap(String(data.dataForFormat(
                                TransferableFormats.TEXT_FORMAT)));

    } else if(data.hasFormat(TransferableFormats.FILE_LIST_FORMAT)){
        var dropfiles:Array =
            data.dataForFormat(TransferableFormats.FILE_LIST_FORMAT) as Array;
        for each (var file:File in dropfiles)
        {
            scrap = new FileScrap(file);
        }
    }
    scrap.x = placeX + offset.x;
    scrap.y = placeY + offset.y;
    return scrap;
}
```

If the TransferableData object contains the format, “SCRAP”, then the drag must have originated from the Scrappy application itself (or from another AIR application that uses the “SCRAP” format name). This can happen if, for example, the user drags a scrap out of the application window and then back into the window. By setting the transfer mode to originalPreferred, a reference to the original scrap will be returned when available. This avoids duplicating the scrap object when it originates in the current application. The originalOnly transfer mode should not be used in this case, because there is a chance the original object will be deleted in the middle of a cut-and-paste operation.

Note: *Copying serialized data is not implemented in this version of Scrappy. Because Scraps are complex objects, they can not be simply deserialized and used as is. An additional step is required to rebuild a valid Scrap object from the properties of the anonymous object returned by the deserializer.*

Dragging data out of the canvas

To allow a user to drag an object out of an application, you must:

1 Respond to a mouseDown or mouseMove event.

2 Create a TransferableData object containing the data or object to drag.

3 Start the drag operation from the mouseDown or mouseMove event handler by calling DragManager.doDrag().

Scrappy uses two separate drag APIs. As long as a drag gesture stays within the bounds of the main window, Scrappy uses the Sprite drag API, which provides a somewhat smoother operation for positioning sprites. When the drag gesture leaves the window, Scrappy transitions the drag gesture to the DragManager API, which allows data to be dragged to other applications.

Preparing to drag

When the Scrap object being moved leaves the window, it dispatches a rollOut event. The handler for this event stops the Sprite-controlled drag and gets a TransferableData object by calling the function, addTransferableData(). The addTransferableData() function creates a new TransferableData object and adds a reference to the current Scrap object by using a custom format name. The addTransferableData function is overridden in the Scrap subclasses to add the data appropriate to that subclass in addition to the reference format.

```
protected function onDragOut(event:Event):void{
    parent.removeEventListener(MouseEvent.ROLL_OUT,onDragOut);
    this.stopDrag();
    offset.x = -mouseX;
    offset.y = -mouseY;
    var transferObject:TransferableData = addTransferableData();
    DragManager.doDrag(this,transferObject,getImage(),offset);
}
protected function addTransferableData():TransferableData{
    var transfer:TransferableData = new TransferableData();
    transfer.addData(this,"SCRAP",true);
    return transfer;
}
```

Starting the drag operation

The call to DragManager.doDrag(), starts the DragManager-controlled part of the drag gesture. The initiator parameter of the method designates the display object that begun the drag gesture, and, more importantly, the object that will dispatch the nativeDragComplete event when the user releases the mouse button. The transferable parameter is the TransferableData object. Once the doDrag() is called, the object can only be accessed in the event handlers of the NativeDragEvents.

You can supply an image to display while the drag gesture is in progress. Scrappy creates an image by drawing the sprite into a BitmapData object. The offset point lets you offset the image from the mouse hotspot.

Completing the drop

When the drop occurs, the object passed to DragManager.doDrag() as the initiator will dispatch a nativeDragComplete event. The action selected by the drop target is reported in the dropAction property of the event. Scrappy only supports the copy action for external transfers, so no action needs to be taken.

Copying and pasting data

Compared to drag and drop, copy and paste are very simple operations. The key point to remember about the clipboard is that you can only access it within the scope of the function passed to the Clipboard manager. Once that function returns, accessing the Clipboard.data property or the TransferableData object it references will generate a runtime error.

Note: *In this Beta release, the traditional key combinations used for copy, cut, and paste commands, Ctrl-c, Ctrl-x, and Ctrl-v, are handled internally by the runtime and do not generate keyboard events.*

Paste

Scrappy implements a paste command on the main window. In response to a paste command (v), the application passes the clipboard data to the factory method Scrap.createScrap(), which is also used to create scraps for drop operations.

```
public function doPaste():void{
    ClipboardManager.accessClipboard(function():void{
        addChild(Scrap.createScrap(
            ClipboardManager.data,stage.stageWidth/4,stage.stageHeight/4));
    }); //end of closure
}
```

The pasted scrap is always placed in the same spot on the stage.

Copy

Copy is implemented by the Scrap base class. The function creates a TransferableData object, then, within a closure, copies the object to the system clipboard represented by ClipboardManager.data property.

```
public function doCopy():void{
    var transferObject:TransferableData = addTransferableData();
    ClipboardManager.accessClipboard(function():void{
        ClipboardManager.data = transferObject;
    }); //end of closure
}
```

Cut

Cut just copies the scrap object to the clipboard, then deletes it.