
Using the file system

In this module you will learn how to access the file system of the host computer

Objectives

After completing this unit, you should be able to:

- Use the Flex 3 AIR file system components
- Work with the file system
- Read and write files to the file system

Topics

In this unit, you will learn about the following topics:

- Using the File class
- Enabling visual file system interaction
- Removing files from the file system
- Using file streams
- Working with binary data

Using the File class



- A `flash.filesystem.File` object
 - represents either a file or directory
 - may refer to an object which does not yet exist
- Some `File` methods have synchronous and asynchronous versions:
 - `copyTo()` and `copyToAsync()`
 - `deleteFile()` and `deleteFileAsync()`
 - `moveTo()` and `moveToAsync()`
- Synchronous method calls suspend code execution until completed
- Asynchronous method calls do not suspend code execution, but require that an event listener be added to handle results

Accessing common directories

- A `File` object exposes properties pointing to `File` objects representing pre-defined directories on the user's system
 - `applicationDirectory`: refers to a read-only directory of application assets relative to the project source directory
 - all assets in the project source directory will be available here at runtime, if they are included during Export Release Build
 - `applicationStorageDirectory`: refers to an automatically created, application-specific read/write directory intended for persistent, offline file storage
 - `nativePath`: the object's path on the local file system

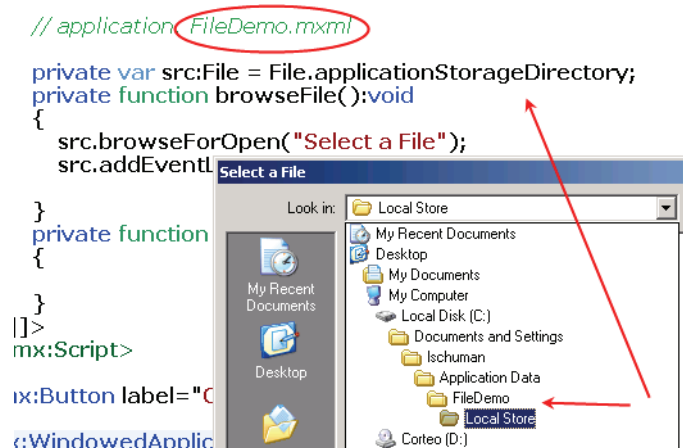


Figure 18: Application storage directory

- Common directories may also be referenced using aliases
 - `app:/` - refers to the application directory

-
- `app-storage:/` - refers to the application storage directory
 - `file:///` - refers to the root of the user's hard disk

Referencing a file or directory

- A `File` path may be set using pre-defined aliases

```
var dir:File = new File("app-storage:/myDirectory");
trace(dir.nativePath);
// displays C:\Documents and Settings\[user name]\Application Data\
[application id]\Local Store\myDirectory
```

```
var dir:File = new File("app:/myDirectory");
trace(dir.nativePath);
// displays [application install location]\myDirectory
```

```
var dir:File = new File("file:///newDirectory");
trace(dir.nativePath);
// displays \myDirectory
```

- A `File` path may also be set by assigning the `nativePath` property

```
var path:String = "app-storage:/test.txt";
var file:File = new File();
file.nativePath = path;
```

- The `resolvePath(path)` method returns a new `File` object relative to the `File` object through which it is called

```
var dir:File = new File("app-storage:/myDirectory");
var newDir:File = dir.resolvePath("newDirectory");
trace(newDir.nativePath)
// displays C:\Documents and Settings\[user name]\Application Data\
[application id]\Local Store\myDirectory\newDirectory
```

- A `File` path may be set by referring to a pre-defined application folder by its static `File` property

```
var newDir:File =
    File.applicationStorageDirectory.resolvePath("newDirectory");
```

- The `exists` property indicates if a file or directory exists yet at its path

```
if (file.exists) { ... }
```

- The `File` object's `parent` property refers to the directory which contains the current file or directory, also represented as a `File` object

```
var file:File = new File("file:///adobeTraining/demo.txt");
trace(file.nativePath) // displays \adobeTraining\demo.txt
trace(file.parent.nativePath) // displays \adobeTraining
```

Moving and copying file system objects

- A `File` object exposes methods to copy or move the content of its path to a new location, also specified by a `File` object
 - `copyTo(newFile, overwrite)`: copies the content at a `File` object's path to a path specified by a second `File` object
 - `moveTo(newFile, overwrite)`: moves the content at a `File` object's path to a path specified in a second `File` object

```
var src:File = new File("app:/assets/simple.txt");
var dest:File = new File("app-storage:/simple.txt");

src.copyTo(dest, true); // src.moveTo(dest, true);
```

- Asynchronous versions of these methods are also available:
`copyToAsync()`, `moveToAsync()`

Walkthrough 1: Using the File class



In this walkthrough, you will perform the following tasks:

- Understand the `applicationDirectory` and `applicationStorageDirectory`
- Create a new directory using a `File` object
- Display the `nativePath` of a `File` object
- Copy a file from one directory to another
- Delete a file
- Delete a directory and its contents

Steps

Create a new application-specific directory

1. Open the `FileSystem` project and close unrelated projects.
2. Open `FileSystem_wt1.mxml` from the `src` folder.
3. Review the existing code:
 - `init()` called in application `creationComplete` event
 - three `File` properties declared: `newDir`, `srcFile`, `destFile`
 - a `Button` object with `copyFile()` assigned as `click` handler
 - `copyFile()` and `init()` functions
4. In the `init()` function, assign `newDir` a reference to a `File` object pointing (resolved) to a directory named `files` in the `applicationStorageDirectory` for this application

```
newDir = File.applicationStorageDirectory.resolvePath("files");
```

Note: could alternately be written as:

```
newDir = new File("app-storage:/files");
```

5. Below this code, use `newDir` to create a new directory for its path.

```
newDir.createDirectory();
```

6. Trace the `nativePath` of this new directory to the console.

```
trace(newDir.nativePath);
```

7. Your `init()` method should look like this:

```
private function init():void
{
    newDir = File.applicationStorageDirectory.resolvePath("files");
    newDir.createDirectory();
    trace(newDir.nativePath);
}
```

8. **Debug** the application.

The **Console View** should display the path to a newly created directory.
Examine this path using your operating system file browser.

```
C:\Documents and Settings\[username]\
    Application Data\fileSystem\Local Store\files
```

Note: This value will vary by operating system. Notice that the new directory names "files" is created in "Local Store" under a directory named for the application itself: "fileSystem"

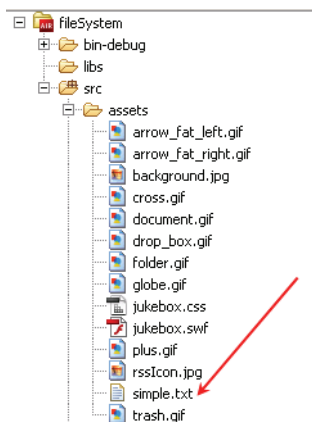
9. Close the AIR application.

Copy a file from the application directory to the newly created storage directory

10. In the `copyFile()` function, test whether the new directory exists.

```
if(newDir.exists)
{
}
```

11. In this condition, assign `srcFile` a reference to a `File` object resolved (pointing) to the `simple.txt` file in the `assets` folder of the `applicationDirectory` for this application (use the `app:/` alias).



```
srcFile = new File("app:/assets/simple.txt");
```

Note: The syntax above, using the `app:/` alias, is equivalent to the following:

```
srcFile =  
File.applicationDirectory.resolvePath("assets/simple.txt");
```

12. Next, trace the `nativePath` of `srcFile` to the console concatenated to a short `String` identifying the information displayed.

```
trace("srcFile path: " + srcFile.nativePath);
```

13. Resolve the `newDir` object to a new, matching file name of `simple.txt` and assign the resulting `File` object to `dstFile`.

```
dstFile = newDir.resolvePath("simple.txt");
```

14. Copy `srcFile` to `dstFile`, passing `true` as the second parameter to overwrite `dstFile` if it already exists.

```
srcFile.copyTo(dstFile, true);
```

15. Trace the `nativePath` of `dstFile` to the console, concatenated to a short `String` identifying the information displayed.

```
trace ("dstFile path: " + dstFile.nativePath);
```

16. Your `copyFile()` method should look like this:

```
private function copyFile():void  
{  
    if (newDir.exists)  
    {  
        srcFile =  
            File.applicationDirectory.resolvePath("assets/simple.txt");  
        trace("srcFile path: " + srcFile.nativePath);  
  
        dstFile = newDir.resolvePath("simple.txt");  
        srcFile.copyTo(dstFile, true);  
        trace("dstFile path: " + dstFile.nativePath);  
    }  
}
```

17. **Debug** the application.

In your file browser, examine the application storage folder. It should exist, and have a sub-folder names `/files`

Press the **Copy File** button. Examine the **Console View** to see the source and destination of the file. In your file browser, examine the `/files` folder, which should now contain `simple.txt`

18. **Close** the application.

19. **Close** `FileSystem_wt1.mxml`.

Enabling visual file system interaction



- A `File` object may open a native file system browser
- The Flex 3 framework includes new display components specific to AIR
 - File system browsing components
 - HTML rendering components
 - Windowing components

Allowing user to specify files or directories

- The `File` class exposes methods to open a system dialog with a specified `title`, and optionally filter by an array of permitted types
 - `browseForDirectory(title)`: select a directory
 - `browseForOpen(title, typeFilter)`: select a file from this `File` object's directory
 - `browseForOpenMultiple(title, typeFilter)`: select multiple files

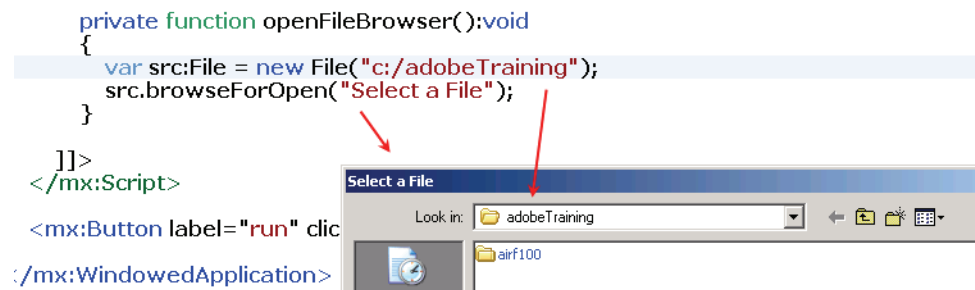


Figure 19: Browse for Open dialog

- A `SELECT`, `SELECT_MULTIPLE`, or `CANCEL` event is dispatched based on user's response to this dialog
- The target of a `SELECT` event refers to a `File` object representing the selected file or directory

Introducing the Flex 3 file browsing components

- File system browsing components display file system information retrieved from the operating system currently running the application, including
 - `<mx:FileSystemList />`
 - `<mx:FileSystemTree />`
 - `<mx:FileSystemDataGrid />`

-
- The components may also be instantiated through ActionScript 3
 - `mx.controls.FileSystemList`
 - `mx.controls.FileSystemTree`
 - `mx.controls.FileSystemDataGrid`
 - File system components display information supplied by a `File` object assigned to their `directory` property

Understanding the `FileSystemList` component

- Displays contents of a directory (files or more directories) as a scrolling list
- `selectedItem` is a `File` object pointing to the selected file or folder
- Clicking drills to next directory and dispatches `Event.SELECT`
- `target` of `SELECT` event object is the new `File` object for display

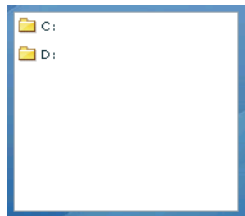


Figure 20: `FileSystemList` component

- `refresh()` re-reads the current directory and updates the display

Understanding the `FileSystemTree` component

- Displays the contents of a file system directory (files or additional directories) in a tree-based view, supporting drill-down behavior
- `selectedItem` is a `File` object pointing to the selected file or folder
- Clicking drills to next directory and dispatches `Event.SELECT`
- `target` of `SELECT` event object is the new `File` object for display
- `refresh()` re-reads the current directory and updates the display

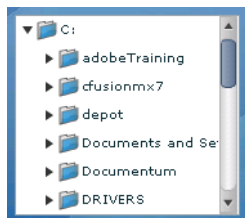
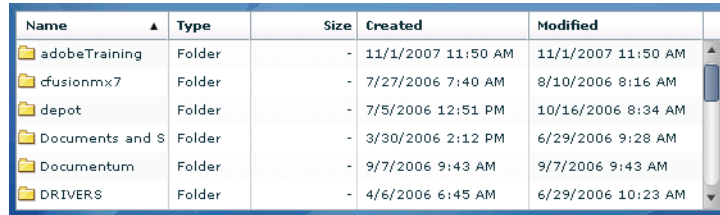


Figure 21: `FileSystemTree` component

- `refresh()` re-reads the current directory and updates the display

Understanding the FileSystemDataGrid component

- Displays specified data about the contents of a file system directory (files and/or additional directories) in configurable columns
- `selectedItem` is a `File` object pointing to the selected file or folder
- Clicking drills to next directory and dispatches `Event.SELECT`
- `target` of `SELECT` event object is the new `File` object for display



Name ▲	Type	Size	Created	Modified
adobeTraining	Folder	-	11/1/2007 11:50 AM	11/1/2007 11:50 AM
cfusionmx7	Folder	-	7/27/2006 7:40 AM	8/10/2006 8:16 AM
depot	Folder	-	7/5/2006 12:51 PM	10/16/2006 8:34 AM
Documents and S	Folder	-	3/30/2006 2:12 PM	6/29/2006 9:28 AM
Documentum	Folder	-	9/7/2006 9:43 AM	9/7/2006 9:43 AM
DRIVERS	Folder	-	4/6/2006 6:45 AM	6/29/2006 10:23 AM

Figure 22: `FileSystemDataGrid` component

- `refresh()` re-reads the current directory and updates the display

Walkthrough 2: Use file system browsing and display components



In this walkthrough, you will perform the following tasks:

- Use a File object native file system browser
- Use the Flex AIR file system components

Steps

Browse for a directory

1. Open the `FileSystem_wt2.mxml` file from the `src` folder at the root of the project.
2. Review the pre-written code:
 - `TextInput` component named `folderPath` with `selectDirectory()` assigned as a `click` event handler
 - `TextInput` component named `selectedPath`
3. In the `Script` block, create a `Bindable` private variable named `dir`, data typed as `File`.
4. Assign to it a new `File` object.

```
[Bindable]
private var dir:File = new File();
```

5. In the `selectDirectory()` method, using the `dir` object, invoke the `browseForDirectory()` method with a `String` argument of *Select a Directory*.

```
dir.browseForDirectory("Select a Directory");
```

6. Next, in the `selectDirectory()` method, add an `Event.SELECT` listener to the `dir` object which calls the pre-written `directorySelected` method.

```
dir.addEventListener(Event.SELECT, directorySelected);
```

7. Place a breakpoint on the last line of `directorySelected()`.

*Note: Add the breakpoint on the last line of `directorySelected()`, which runs **after** `selectDirectory()` displays the directory browser, not `selectDirectory()`, which opens the file browser.*

8. **Debug** the `FileSystem_wt2` application.

Click in the first Text Input and select a directory from the browse dialog. Switch to the debug perspective if necessary. Examine the `target` property of the `eventObj` in `directorySelected()` to view properties of the selected directory.

+	this	FileSystem_wt2 (@17b50a1)
+	eventObj	flash.events.Event (@1856e21)
	bubbles	false
	cancelable	false
+	currentTarget	flash.filesystem.File (@da04c1)
	eventPhase	2
+	target	flash.filesystem.File (@da04c1)
+	[inherited]	
	exists	true
+	icon	flash.desktop.Icon (@f9f6a1)
	isDirectory	true
	isHidden	false
	isPackage	false
	isSymbolicLink	false
	nativePath	"C:\\Documents and Settings\\lschuman\\My Documents_COURSES"
+	parent	flash.filesystem.File (@da0f41)
	url	"file:///C:/Documents%20and%20Settings/lshuman/My%20Documents/_COURSES"
	type	"select"

9. End the debug session, return to **Flex Development** perspective, and remove the breakpoint.

10. In the `directorySelected` method, assign the `nativePath` on the `target` object of the event to the `text` property of the `folderPath` text input component.

```
folderPath.text = eventObj.target.nativePath;
```

11. **Run the `FileSystem_wt2` application.**

Click in the first Text Input and select a directory from the browse dialog. You should see the selected directory path displayed.

Use Flex file system components

12. Between the two `<mx:HBBox>` components, where indicated, insert a `<mx:FileSystemList>` component with an `id` of `fileComp`.

13. Assign its `itemClick` event to invoke the `pathSelected()` event listener method, passing `event` as the parameter.

```
<!-- Add Flex AIR Components here -->
<mx:FileSystemList id="fileComp" itemClick="pathSelected(event)" />
```

14. In the `pathSelected()` event listener, assign the `nativePath` of the `selectedItem` on the `target` object of the event to the `text` property of the `selectedPath` object.

```
selectedPath.text = eventObj.target.selectedItem.nativePath;
```

15. In the `directorySelected()` method, assign the `target` of the event object to the `directory` property of the `fileComp` component, casting it as a `File`.

```
fileComp.directory = eventObj.target as File;
```

16. **Run** the **FileSystem_wt2** application.

You should be able to browse the file system by double-click on directories, and see the path displayed for each in the `TextInput`.

17. **Close** the application.

18. In `FileSystem_wt2.mxml`, change the `<mx:FileSystemList>` component to `<mx:FileSystemTree>`.

19. **Run** the **FileSystem_wt2** application.

The application should use a tree structure to navigate the file structure.

20. **Close** the application.

21. In `FileSystem_wt2.mxml`, change the `<mx:FileSystemTree>` component to `<mx:FileSystemDataGrid>`

22. **Run** the **FileSystem_wt2** application.

The application should use a `DataGrid` to navigate the file structure.

*Note: The `DataGrid` component may exceed the default width of the application. Flex component layout and positioning is discussed in detail in the *Flex 3: Rich Client Applications* and *Flex 3: Extending and Styling Components* courses.*

23. **Close** the application.

24. **Close** `FileSystem_wt2.mxml`.

Removing files from the file system



- File objects may represent either a file or directory
- The object type must be known to remove it from the file system

Determining the File object type

- A File object may represent either a file or a directory
 - `isDirectory`: true if File object is a directory, otherwise false

```
var myFile:File = new File("c:\adobeTraining\test.txt");
// if myFile refers to a file and not a directory, do something
if (!file.isDirectory) { ... }
```

Recycling or deleting file system objects

- A File object exposes methods to recycle or delete its path content to a new location, also specified by a File object
 - `deleteFile()`: deletes the file represented by this object
 - `deleteDirectory(deleteContent)`: deletes the directory represented by this File object
 - if parameter is `true`, all contents are deleted with the directory
 - if parameter is `false`, an `IOException` exception is thrown if directory holds files or other directories

```
private function deleteFile(evtObj:Event, tree:FileSystemTree):void
{
    if(tree.selectedItem.isDirectory)
    {
        tree.selectedItem.deleteDirectory(true);
    }
    else
    {
        tree.selectedItem.deleteFile();
    }
    refreshLists();
}
```

- `moveToTrash()`: moves the file or directory represented by this File object to the system Trash or Recycle directory

```
private function trashFile(evtObj:Event, tree:FileSystemTree):void
{
    tree.selectedItem.moveToTrash();
    refreshLists();
}
```

}

- Asynchronous versions of these methods are also available:
`deleteFileAsync()`, `deleteDirectoryAsync()`,
`moveToTrashAsync()`

Handling missing file references with try/catch

- If any of these methods are called on a `File` object which does not exist, a catchable `Error` will be thrown due to the `null` object reference
 - Error objects expose a `message` property describing the error
- The `show(message)` method of the `mx.controls.Alert` class displays a popup message box to the user

```
import mx.controls.Alert
try
{
    tree.selectedItem.moveToTrash();
    refreshLists();
}
catch(err:Error)
{
    Alert.show("Please select an item to move to the trash");
    trace(err.message); // null object reference error
}
```

Walkthrough 3: Interacting with visually specified File objects



In this walkthrough, you will perform the following tasks:

- Assign explicit paths to a visual file browsing component
- Use visual components to select files and directories
- Conditionally copy or move files to a chosen directory
- Delete files or directories based on their type
- Move files or directories to the system trash

Steps

Assign a specific directory for display by a visual object

1. Open the `FileSystem_wt3.mxml` file.
2. Review the pre-written code:
 - `FileSystemTree` components named `sandbox1Files` and `sandbox2Files`
 - Eight `Button` components with icons and event handlers
 - `refreshLists()` method to refresh both `FileSystemTree` components
 - `init()` method called by application `initialize` event
3. In the declarations section of the `Script` block, create a `private` variable named `sandbox`, data typed as `File`, and set it equal to a new `File()`, passing `c:/adobeTraining/airf100/` to its constructor.

```
private var sandbox:File =  
    new File("c:/adobeTraining/airf100/_Sandbox/");
```

Note: If using Mac OSX, or if you've installed the course files to a different location, adjust this path as needed.

4. In the `init()` method, assign the `sandbox` object as the directory of the `sandbox1Files` and `sandbox2Files` components.

```
sandbox1Files.directory = sandbox;  
sandbox2Files.directory = sandbox;
```

5. Confirm that the `init()` method appears as follows:

```
private function init():void  
{  
    sandbox1Files.directory = sandbox;  
    sandbox2Files.directory = sandbox;  
}
```

6. Run the `FileSystem_wt3` application.

You should see the two `FileSystemTrees` populated with directory and file information from the specified course file directory.

Use visual component File references to copy/move files

Note: Moving files can cause severe damage to your operating system. Do not attempt to copy, move, or delete any critical files on the computer.

7. Locate the `copyFile()` method in the `Script` block.
8. In this method, create a local variable named `srcFile`, data typed as `File` and assign to it a new `File` object.
9. Pass the `File()` constructor the `selectedPath` of the source argument passed to this method.

```
var srcFile:File = new File(source.selectedPath);
```

10. Below this code, create another local variable. Name this one `destFile`, data typed as `File` and assign to it a new `File` object.
11. Pass the `File()` constructor the `selectedPath` of the `destination` argument passed to this method.

```
var destFile:File = new File(destination.selectedPath);
```

12. Below this code, write a condition using the `isDirectory` property of `srcFile` and `destFile`, logically testing whether `srcFile` is a file (`isDirectory` is false), and `destFile` is a directory (`isDirectory` is true).

```
if (!srcFile.isDirectory && destFile.isDirectory)
{
}
```

13. In the condition written above, assign to `destFile` the `File` object returned by the `resolvePath()` method on the `destFile` object, resolving the file name of the `srcFile`.

```
destFile = destFile.resolvePath(srcFile.name);
```

14. Invoke the `copyTo()` method on the `srcFile`, passing the `destFile` and `true` as the arguments.

```
srcFile.copyTo(destFile, true);
```

15. Then call the pre-written `refreshLists()` function to cause both `FileSystemTree` components to re-read their target directories and update their view.

```
refreshLists();
```

16. Confirm that the `copyFile()` method appears as follows:

```
private function copyFile(source:FileSystemTree,
destination:FileSystemTree):void
{
    if (!srcFile.isDirectory && destFile.isDirectory)
    {
        destFile = destFile.resolvePath(srcFile.name);
        srcFile.copyTo(destFile, true);
        refreshLists();
    }
}
```

17. Review the pre-written code in the `moveFile()` method, noticing the only difference is the use of `moveTo()` in place of `copyTo()`.

18. Run the `FileSystem_wt3` application.

Open a directory and select a file from the left-side tree, and a directory from the right-side tree, then click the **copy** button. The file should be copied, and appear in the right-side tree. Confirm the result using your local file browser.

Select a file from the right-side tree, and a directory from the left-side tree, then click the **move** button. The file should be moved, and appear only in the left-side tree. Confirm the result using your local file browser.

Determine File object type, and delete or move to trash

Note: Deleting critical files can permanently damage your operating system. Practice these techniques with non-critical files only.

19. In the `deleteFile()` method, above the call to the pre-written `refreshLists()` method, insert a condition that tests that the `selectedItem` from the `tree` argument passed to this method is a directory using `isDirectory`. Add an else block.

```
if(tree.selectedItem.isDirectory)
{
}
else
{
}
```

-
20. If it is a directory, invoke the `deleteDirectory()` method on the `selectedItem` of the `tree` argument, passing `true` as a parameter. Else, invoke the `deleteFile()` method on the same object.

```
if(tree.selectedItem.isDirectory)
{
    tree.selectedItem.deleteDirectory(true);
}
else
{
    tree.selectedItem.deleteFile();
}
```

21. Confirm that the `deleteDirectory()` method appears as follows:

```
private function deleteFile(eventObj:Event,
tree:FileSystemTree):void
{
    if(tree.selectedItem.isDirectory)
    {
        tree.selectedItem.deleteDirectory(true);
    }
    else
    {
        tree.selectedItem.deleteFile();
    }
    refreshLists();
}
```

22. In the `trashFile()` method, above the call to `refreshLists()`, invoke the `moveToTrash()` method on the `selectedItem` of the `tree` argument.

```
tree.selectedItem.moveToTrash();
```

23. **Run the `FileSystem_wt3` application.**

Before proceeding, empty your computer recycle bin to make it easier to spot files move to the trash.

Select a file and click the **Trash** button. Open your system trash folder. **Restore** the file.

With same file selected, click the **Delete** button. Notice it is not in your system trash folder.

Select the **clothing** directory and press the **Trash** button. After this folder disappears, and without selecting any other file or directory, press the Trash button a second time. This error should appear: **Cannot access a property or method of a null object reference.**

Restore the **clothing** directory from your system trash folder.

24. **Close** the application.

Using try/catch to manage errors

25. In `FileSystem_wt3.mxml`, in the `Import Classes` section, import the `mx.controls.Alert` class.

```
import mx.controls.Alert;
```

26. In the `trashFile()` method, wrap the code in a `try/catch` block for an `Error` object named `err`.

```
try
{
    tree.selectedItem.moveToTrash();
    refreshLists();
}
catch(err:Error)
{
}
```

27. In the `catch` block, use `Alert.show()` to display message asking the user to select a file or directory.

```
Alert.show("Please select a file or directory");
```

28. The `trashFile()` method should now look like this:

```
private function
trashFile(eventObj:Event, tree:FileSystemTree):void
{
    try
    {
        tree.selectedItem.moveToTrash();
        refreshLists();
    }
    catch(err:Error)
    {
        Alert.show("Please select a file or directory")
    }
}
```

29. Run the **FileSystem_wt3** application.
Select the **clothing** directory and press the **Trash** button. After this folder disappears, and without selecting any other file or directory, press the **Trash** button a second time. An Alert box should appear instructing the user to select a file or directory.
30. **Close** the application.
31. **Close** `FileSystem_wt3.mxml`.

Using file streams



- A `flash.filesystem.FileStream` object is used to read or write files to the file system
- Files may be opened and written two ways
 - synchronous: application pauses while data is read or written
 - asynchronous: application does not pause, data is buffered, `PROGRESS` and `COMPLETE` events are dispatched during read/write

Reading and writing files

- To read or write a file from the file system
 1. Create a `FileStream` object
 2. Create `File` objects for the directory and file to read or write
 3. Open the file stream for synchronous or asynchronous use
 - synchronous: AIR pauses as read/write occurs
 - asynchronous: AIR continues, dispatching `OPEN`, `PROGRESS` and `COMPLETE` events as read/write occurs
 4. If asynchronous, listen for `PROGRESS` or `COMPLETE` events
 5. Read or write data to the file stream
 - `getBytes()`, `readUTF()`, `readObject()`, etc.
 - `writeBytes()`, `writeUTF()`, `writeObject()`, etc.
 6. Close the file stream
 - `close()`

Creating FileStream and File objects

- The `File` object represents the file to read or write

```
var file:File = new File("app-storage:/demo.txt");
```

- The `FileStream` object manages interaction with the file

```
var stream:FileStream = new FileStream();
```

Creating and working with directories

- `File` objects may represent either a file or a directory
- To create a new directory
 1. Create a `File` object representing the desired directory path
 2. Invoke `createDirectory()` to create the directory and its parents

```
var dir:File = new File("app-storage:/myNewFolder");  
dir.createDirectory();
```

-
- To refer to a new `File` object in a newly created directory, resolve the desired file name based upon the newly created path

```
var dir:File = new File("app-storage:/userData");
dir.createDirectory();
var file:File = dir.resolvePath("demo.txt");
```

Specifying the operation type

- The type of file operation is specified using constants from the `flash.filesystem.FileMode` class passed as a parameter when the file stream is opened
 - `FileMode.READ`: open file to read data
 - `FileMode.WRITE`: open file to write new data
 - if file does not exist, create it
 - if file does exist, delete current contents
 - `FileMode.APPEND`: open file to write additional data
 - if file does not exist, create it
 - if file exists, begin writing from end of current data
 - `FileMode.UPDATE`: open file for random read or write access
 - if file does not exist, create it
 - only bytes written to the current position are modified (the file remains otherwise unchanged)

Opening a file stream

- `FileStream` objects may be opened for synchronous or asynchronous operations
 - `open(file, fileMode)`: pauses operation until open

```
var file:File = new File("app-storage:/demo.txt");
var stream:FileStream = new FileStream();
stream.open(file, FileMode.WRITE);
```

- `openAsync(file, fileMode)`: dispatches `OPEN`, `PROGRESS`, and `COMPLETE` events while opening the file

```
var file:File = new File("app-storage:/demo.txt");
var stream:FileStream = new FileStream();
stream.openAsync(file, FileMode.READ);
stream.addEventListener(Event.OPEN, readOpenFile);
```

Note: Asynchronous approaches support more robust architectures, and improve the user experience when opening large files.

Reading data through a file stream

- `FileStream` objects may read various data formats from a file, including
 - `readBytes(byteArray, offset, length)`: beginning from the `offset` position, read `length` bytes into the `byteArray`
 - `readUTFBytes(bytesAvailable)`: read and return the specified bytes as a UTF-8 (Unicode) `String` from the file stream
 - `bytesAvailable` is a read-only property identifying how many bytes are currently available in the file stream

```
stream.open(file, FileMode.READ);
var fileData:String = stream.readUTFBytes(stream.bytesAvailable);
displayArea.text = fileData;
```

Writing data through a file stream

- `FileStream` objects may write various data formats to a file, including
 - `writeBytes(byteArray, offset, length)`: from the `offset` position, write `length` bytes from the `byteArray` to the file
 - `writeUTF(value)`: write UTF-8 (Unicode) `String` value to a file

```
stream.open(file, FileMode.WRITE);
stream.writeUTFBytes("some string of text");
```

Note: Numerous data type specific read and write methods are available, and described in the documentation.

Closing a file stream

- `FileStream` objects should be closed when read/write is complete
 - `close()`: closes the file stream and dispatches a `CLOSE` event

```
stream.writeUTFBytes("some string of text");
stream.close();
```

- Closing the application closes all file streams, but no `CLOSE` event occurs

Walkthrough 4: Create a directory or UTF-8 (Unicode) text file



In this walkthrough, you will perform the following tasks:

- Modify a custom component to create a specified file or directory
- Create a directory with a user-specified name and location
- Create a file with a user-specified name and location
- Use a custom component as a custom popup window

Steps

Create a popup window to create a file

1. Open the `NewForm_wt4.mxml` file from the **comp** directory.
2. In the declarations section, create a `public Bindable` property for this component named `file`, data typed as `File`, below the pre-written `public Bindable` type property.

```
[Bindable]
public var file:File;
```

3. In the `add()` method, above the existing code calling `cancelForm()`, create a local variable named `path`, data typed as `String`.
4. In a conditional statement testing the `file.isDirectory` property, assign to `path` the `nativePath` of the `file` object if it is a directory and the `nativePath` of the parent of `file` if it is not a directory.

```
var path:String;
if(file.isDirectory)
{
    path = file.nativePath;
}
else
{
    path = file.parent.nativePath;
}
```

-
5. After the `if` statement, create a `switch` statement testing the `bindable`, `public type` property pre-declared within the code, with two cases; *Directory* and *File*.

```
switch(type)
{
  case "Directory":
  {
    break;
  }
  case "File":
  {
    break;
  }
}
```

6. In the `Directory` case, before the `break` statement, create a local variable named `newDir`, data typed as `File` and assign to it a new `File` object, passing the value of `path` concatenated with a forward slash (`/`), concatenated with the `name` argument.

```
var newDir:File = new File(path + "/" + name);
```

7. Complete the case by invoking the `createDirectory()` method on the `newDir` object.

```
newDir.createDirectory();
```

8. In the `File` case, before the pre-written `break` statement, create a local variable name `newFile`, data typed as `File` and assign to it a new `File` object with the value of `path` concatenated with a forward slash (`/`), concatenated with the `name` argument, concatenated with the literal string `.txt`.

```
var newFile:File = new File(path + "/" + name + ".txt");
```

9. Below this code, create a new local variable named `stream`, data typed as `FileStream` and assign to it a new `FileStream` object.

```
var stream:FileStream = new FileStream();
```

10. Still in the `File` case, using the `open()` method on the `stream` object, pass the `newFile` and the static property `WRITE` on the `FileMode` class as parameters.

```
stream.open(newFile, FileMode.WRITE);
```

11. Using the `writeUTFBytes()` method on the `stream` object, pass the string `AIR` as the parameter, to write this value into the stream.

```
stream.writeUTFBytes("AIR");
```

12. Finally close the `stream` object with the `close()` method.

```
stream.close();
```

13. Confirm that the `add()` method appears as follows:

```
private function add(name:String):void
{
    var path:String;
    if(file.isDirectory)
    {
        path = file.nativePath
    }
    else
    {
        path = file.parent.nativePath;
    }

    switch(type)
    {
        case "Directory":
            {
                var newDir:File = new File(path + "/" + name);
                newDir.createDirectory();
                break;
            }
        case "File":
            {
                var newFile:File = new File(path + "/" + name + ".txt");
                var stream:FileStream = new FileStream();
                stream.open(newFile, FileMode.WRITE);
                stream.writeUTFBytes("AIR");
                stream.close();
                break;
            }
    }
    cancelForm();
}
```

Instantiate popup to specify the file or directory to create

14. Open `FileSystem_wt4.mxml`.

-
15. Uncomment and review the `createFileObj()` method, which uses the `PopUpManager` class to pop up an instance of the `NewForm_wt4` component modified above.

```
private function
  createFileObj(tree:FileSystemTree, type:String):void
{
  var formObj:NewForm_wt4 =
  NewForm_wt4(PopUpManager.createPopUp(this, NewForm_wt4, true));
  formObj.file = tree.selectedItem as File;
  formObj.type = type;
  formObj.addEventListener(Event.REMOVED, refreshLists);
}
```

Note: Detailed instruction of the `PopUpManager` class is outside the scope of this course. Please see the [Flex 3 documentation](#) for further information.

16. Run the `FileSystem_wt4` application.

Select a directory or file and click the **Create Directory** and **Create File** buttons to create new file objects.

Note: Both `FileSystemTree` components are displaying the same directory, so files and directories created through either will appear in both.

Working with binary data



- Files may be manipulated as a set of bytes
- ActionScript 3 can represent sets of bytes as a `ByteArray` objects

Using the `ByteArray` class

- `flash.utils.ByteArray` objects may be used as binary data containers by `FileStream` objects, when reading and writing file data

```
var byteArray:ByteArray = new ByteArray();
```

Reading binary data

- `readBytes(byteArray, offset, length)` is a `FileStream` method to read data from a `FileStream` into a `ByteArray`
 - `byteArray`: a `ByteArray` object into which data should be read from the `FileStream` object
 - `offset`: the offset position within the `FileStream` object from which to begin reading data into the `ByteArray` object
 - `length`: the number of bytes, beginning at the offset, to read into the `ByteArray` object
- Data from the `FileStream` is read into the `ByteArray` object

```
var file:File = new File("app:/demo.txt");
var stream:FileStream = new FileStream();
var byteArray:ByteArray = new ByteArray();

stream.open(file, FileMode.READ);
stream.readBytes(byteArray, 0, stream.bytesAvailable);
```

Writing binary data

- `writeBytes(byteArray, offset, length)` is a `FileStream` method to write data from a `ByteArray` into a `FileStream`
 - `byteArray`: a `ByteArray` object from which data should be written to the `FileStream` object
 - `offset`: the offset position within the `FileStream` object from which to begin reading data into the `ByteArray` object
 - `length`: the number of bytes, beginning at the offset, to read into the `ByteArray` object
- Data from the `FileStream` is read into the `ByteArray` object

```
var file:File = new File("app-storage:/demo.txt");
var stream:FileStream = new FileStream();
var bArray:ByteArray = new ByteArray();

stream.open(file, FileMode.WRITE);
stream.writeBytes(bArray, 0, stream.bytesAvailable);
```

Walkthrough 5: Read, modify, and write a binary file to the desktop



In this walkthrough, you will perform the following tasks:

- Read a file asynchronously into a byte array
- Modify the binary data while writing it to the desktop as a new file

Steps

Read a file asynchronously in binary

1. Open the `FileSystem_wt5.mxml` file.
2. In the declarations section of the `Script` block, create a `private` variable named `readStream`, data typed as a `FileStream`
3. Assign to `readStream` a new `FileStream` object.

```
private var readStream:FileStream = new FileStream();
```

4. Below this, declare a `private` variable `fileToModify` typed as `File`;

```
private var fileToModify:File;
```

5. Locate the pre-written `selectFile()` method in the `Script` block.
6. In `selectFile()`, assign `fileToModify` a new `File` object with the `nativePath` of the `selectedItem` on the `tree` parameter as its path.

```
fileToModify = new File(tree.selectedItem.nativePath);
```

7. Next, add an event listener to the `readStream` object that listens for the static property `COMPLETE` on the `Event` class and calls `modifyFile`.

```
readStream.addEventListener(Event.COMPLETE, modifyFile);
```

8. Open the stream asynchronously with the `openAsync()` method and pass the `fileToModify` and the static property `READ` on the `FileMode` class as arguments.

```
readStream.openAsync(fileToModify, FileMode.READ);
```

9. Confirm that the `selectFile()` method appears as follows:

```
private function selectFile(tree:FileSystemTree):void
{
    fileToModify = new File(tree.selectedItem.nativePath);
    readStream.addListener(Event.COMPLETE, modifyFile);
    readStream.openAsync(fileToModify, FileMode.READ);
}
```

Write a file in binary

10. Locate the pre-written `modifyFile()` method.

11. In the pre-written `modifyFile()` method, declare a local variable *bArray*, typed as `ByteArray` and assign it a new `ByteArray` object.

```
var bArray:ByteArray = new ByteArray();
```

12. Load all bytes available in `readStream` into `bArray` at an offset of *0*.

```
readStream.readBytes(bArray, 0, readStream.bytesAvailable);
```

13. Create a local variable named *destFile*, data typed as `File`.

14. You will save the modified file to the desktop, so assign to `destFile` the `File.desktopDirectory` object, and use the `resolvePath()` method to specify the name on the `fileToModify` object as its name.

```
var destFile:File =
    File.desktopDirectory.resolvePath(fileToModify.name);
```

15. Create a new `FileStream` named *writeStream* and assign to it a new `FileStream` object.

```
var writeStream:FileStream = new FileStream();
```

16. Use `writeStream` to open `destFile` to be written into.

```
writeStream.open(destFile, FileMode.WRITE);
```

17. Create a `Date` object named *date*.

```
var date>Date = new Date();
```

18. Create a `String` object named *stamp*, and assign it the literal string value *FILE MODIFIED*, concatenated to the current date as a string (`date.toString()`), concatenated to two line return (`\n`) characters.

```
var stamp:String = "FILE MODIFIED: " + date.toString() + "\n\n";
```

-
19. Write the `stamp` string into the `writeStream` object using `writeUTFBytes()` method.

```
writeStream.writeUTFBytes(stamp);
```

20. Next, use the `writeBytes()` method to write into `destFile` (the file currently opened by `writeStream`), at an offset of 0, as many bytes as were read into `bArray` (`readStream.bytesAvailable`).

```
writeStream.writeBytes(bArray, 0, readStream.bytesAvailable);
```

21. Finally, close both file streams.

```
writeStream.close();  
readStream.close();
```

22. Confirm that the `modifyFile()` method appears as follows:

```
private function modifyFile(eventObj:Event):void  
{  
    var bArray:ByteArray = new ByteArray();  
    readStream.readBytes(bArray, 0, readStream.bytesAvailable);  
    var destFile:File =  
        File.desktopDirectory.resolvePath(fileToModify.name);  
  
    var writeStream:FileStream = new FileStream();  
    writeStream.open(destFile, FileMode.WRITE);  
  
    var date:Date = new Date();  
    var stamp:String = "FILE MODIFIED: " + date.toString() + "\n\n";  
  
    writeStream.writeUTFBytes(stamp);  
    writeStream.writeBytes(bArray, 0, readStream.bytesAvailable);  
  
    writeStream.close();  
    readStream.close();  
}
```

23. Run the `FileSystem_wt5` application.

Select a text file from either tree. Press the **Copy to Desktop** button pre-built in the interface (below the `FileSystemTree`). Open the copied file from your desktop. The date stamp string should appear inserted at the top of the file, followed by an empty line of text.

Note: UTF-8 (Unicode) text files are easy to modify in binary form. Each binary type requires a format-specific approach. A rich binary image manipulation library is available in the Flash Player and AIR environments.

24. Close the application.

25. Close `FileSystem_wt5.mxml`.

Summary



- A `File` object is used to create references to a physical file or directory on a local file system
- `File` supports synchronous and asynchronous methods to copy, move, and delete local files and directories
- `File.resolvePath()` set a `File` object to point to a file or folder
- `File.browseForDirectory()` opens a local file browser
- `FileSystemList`, `FileSystemTree`, and `FileSystemDataGrid` components provide pre-built file system browsing functionality
- `try/catch` blocks allow for robust handling of file objects which may change or move at runtime
- `File.applicationDirectory` points to the project source folder
- `File.applicationStorageDirectory` points to a user specific directory intended for offline application storage
- A `FileStream` object is used to read and write binary data to a `ByteArray` and the local file system
- A `ByteArray` serves as a runtime container for binary data
- `FileStream` objects support methods for reading and writing a variety of data types to the file system
- `FileStream` objects may be opened to `READ`, `WRITE`, `UPDATE`, or `APPEND`
- Call the `close()` method of a `FileStream` object to close the associated file

Review



16. The AIR components for Flex 3 allow you to:
 - a. Load HTML
 - b. Browse the filesystem
 - c. Create native windows
 - d. all of the above

17. The `File` class represents a:
 - a. `FileReference`
 - b. Physical file object
 - c. Path to a file or directory
 - d. all of the above

18. Which class is used for writing data to disk:
 - a. `FileReference`
 - b. `FileMode`
 - c. `File`
 - d. `FileStream`

19. The application source directory may be referred to through:
 - a. `File.applicationResourceDirectory`
 - b. `File.source`
 - c. `File.applicationDirectory`
 - d. `File.applicationStorageDirectory`
 - e. `new File("app:/");`

20. Using asynchronous read, write, or delete methods requires:
 - a. adding event listeners
 - b. a `FileStream` object
 - c. adding error checking
 - d. a `FileMode` object