JSFL IS JAVASCRIPT FOR LOVERS

As I mentioned all the way back in Chapter 2, Flash CS4 is not really an ideal environment for game development in the traditional sense. The way it manages assets, for example, can be very clunky. Adobe has greatly improved the way the library works in CS4, but there are still some basic features that would be really nice to have, such as batch renaming, autosorting assets into the correct folders, etc. In many other applications, users would simply have to wait until the next version in the hopes that features like these might get added.

Luckily, Adobe has created a way in which users can add their own custom tools to their environment by making the entire Flash interface accessible through an API. This API is known as JavaScript Flash (JSFL) and, as expected, is JavaScript based. It can be used in an almost unlimited number of ways to automate various aspects of the Flash IDE. Because is it so much like ActionScript, it is also very approachable. However, unlike AS3, it has no type enforcement or compile-time checking, so it can also be trickier to debug. In general, though, most scripts written in JSFL are pretty short. Because these scripts run once, linearly (from start to finish), it's a good idea to not have too much functionality in a single script. Adobe has also included the entire JSFL reference in the Flash CS4 help files, under Extending Flash. After you have finished reading this, I encourage you to read more about what JSFL can do.

Writing JSFL

To create a JSFL file, simply select File → New → Flash JavaScript file. Performing JSFL commands can be as simple

© 2010, Elsevier Inc. All rights reserved.

as writing some code in a *.jsfl file and executing it through the Commands → Run Command menu in Flash. For example, the following script will change the size of the Stage to 640 × 480 from the default 550 × 400:

```
fl.getDocumentDOM().width = 640;
fl.getDocumentDOM().height = 480;
```

This command format may seem odd at first, but basically these lines break down to the following logical steps:

- fl returns a reference to Flash itself.
- getDocumentDOM() returns a reference to the active document object.
- Width and height are simply properties of the document object, both integers.

These two lines of code require no interaction from the user to function. Some commands in the JSFL API are based around some type of user input. Here is an example that puts all of the selected items in the Library panel into a particular folder, even if the folder does not exist:

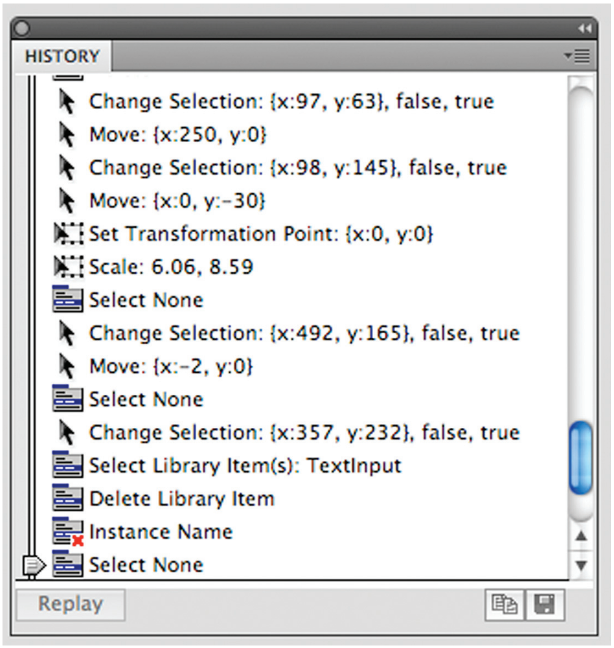
```
var selectedItems = fl.getDocumentDOM().library.
getSelectedItems();

var folderExists = fl.getDocumentDOM().library.
itemExists("Graphics");
if (!folderExists) fl.getDocumentDOM().library.
newFolder("Graphics");

fl.getDocumentDOM().library.moveToFolder("Graphics");
```

To test this example, simply import an item or two into an empty document. Select both items in the library and then run this script. Both items will be moved to a folder called "Graphics," which will be created if it does not exist. As you can see, this API can quickly become very powerful for organizing assets and automating other tasks inside of Flash.

The History Panel

If you are trying to automate a specific, repetitive series of tasks, one of the best ways to create the JSFL for those commands (rather than looking it all up in the reference) is to use the History panel. This is a panel built into Flash that is accessible through the Window → Other Panels menu. It keeps a record of all of your actions inside of Flash. You can select one or all of these actions and Flash will automatically convert it to JSFL code.

As you can see from Figure C.1, the History panel's record is extremely detailed. Note, however, that the second to last action in the panel has a small red "x" over it. This represents an action for which JSFL code cannot be produced, for whatever reason. Usually this has to do with custom actions like naming an instance on Stage. Even though it won't generate the JSFL for those actions, you still have the correct order of operations to recreate the steps.



```
Change Selection: {x:97, y:63}, false, true
Move: {x:250, y:0}
Change Selection: {x:98, y:145}, false, true
Move: {x:0, y:-30}
Set Transformation Point: {x:0, y:0}
Scale: 6.06, 8.59
Select None
Change Selection: {x:492, y:165}, false, true
Move: {x:-2, y:0}
Select None
Change Selection: {x:357, y:232}, false, true
Select Library Item(s): TextInput
Delete Library Item
Instance Name
Select None
```

Figure C.1 The History panel contains a log of all of your performed actions in a document.

Custom Panels and MMExecute

Ultimately, even executing scripts with the Run command is not a particularly user-friendly workflow. Suppose you have a series of commands you want to run on items as they are imported into the library. This might include renaming them, moving them to a folder, putting them in sequential order on the timeline, etc. What's more, you might want the way you rename them and the folder where you want to move them to be custom every time. In a scenario like this, it becomes increasingly inefficient to just run single JSFL scripts over and over again, because you'd have to create custom files for every situation. For an instance such as this, we want to create a custom panel inside of Flash.

Custom panels are nothing more than SWFs that execute JSFL commands, so they are very simple to create. JSFL commands can actually be called from inside ActionScript using a global method called *MMExecute*. In fact, any SWFs running inside of Flash, either as panels or in test mode, can call JSFL. This is where the true power of JSFL is realized. To run a line of JSFL in ActionScript, you simple pass the JSFL code as a string to the

MMExecute method. For example, the following code changes the Stage height of a FLA the same way we saw earlier, just from inside a SWF:

```
MMExecute("fl.getDocumentDOM().height = 480;");
```

Before we go any further, there are a couple of "gotchas" to using MMExecute you should be aware of. The first is that no carriage returns are allowed—remember that this is simply a string in ActionScript. Because of this, it is of the utmost importance that semicolons are used at the end of every line (which I'm sure you were already doing anyway …). The second hitch is that any place where you may have used double quotes in your JSFL the quotes must be escaped with a backslash (\) character. Both of these are frustrating incompatibilities because they prevent you from simply creating a script and testing it in Run Command mode, then copying and pasting it into ActionScript. You must modify them first into what is basically an unreadable state.

Enter the JSFLConverter

Luckily, I have put together a class and applet to help called JSFLConverter. It loads a JSFL file and processes it to strip out line breaks and escape double quotes. It even wraps the entire script in an *MMExecute("<>");* command and sends it to the clipboard so all you have to do after running it is paste the result in your ActionScript. Because this class also uses regular expressions to perform its task, let's review it quickly. It is associated with the JSFLConverter FLA, which has Button and TextArea components on the Stage:

```
package {

  import fl.controls.TextArea;
  import fl.controls.Button;
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.events.MouseEvent;
  import flash.net.*;
  import flash.system.System;

  public class JSFLConverter extends Sprite {

    public var outputText:TextArea;
    public var selectFileButton:Button;

    private var _file:FileReference = new FileReference();

    public function JSFLConverter() {
      selectFileButton.addEventListener(MouseEvent.CLICK,
selectFile, false, 0, true);
    }

    private function selectFile(e:MouseEvent):void {
      _file.browse([new FileFilter("JSFL Scripts", "*.jsfl")]);
      _file.addEventListener(Event.SELECT, fileSelected, false,
0, true);
    }

    private function fileSelected(e:Event):void {
      var urlRequest:URLRequest = new URLRequest(_file.name);
      var urlLoader:URLLoader = new URLLoader(urlRequest);
      urlLoader.addEventListener(Event.COMPLETE, fileLoaded,
false, 0, true);
    }

    private function fileLoaded(e:Event):void {
      var urlLoader:URLLoader = e.target as URLLoader;
      var jsfl:String = urlLoader.data;
      jsfl = jsfl.split(String.fromCharCode(13)).join("");
      var re:RegExp = new RegExp("\"","g");
      jsfl = jsfl.replace(re,"\\\"");
      outputText.text = "MMExecute(\"" + jsfl + "\");";
      System.setClipboard(outputText.text);
    }
  }
}
```

The class relies on the FileReference class to allow you to browse to any particular JSFL file and load it. Once the file has been loaded (note that it uses a URLLoader rather than the FileReference class's *load()* method), it is processed in two ways. The first is to use the *split* method to break the string into an array of lines (the carriage return character in Flash is character code 13) and then the *join* method to reconnect all of the lines without breaks. Next, we create a regular expression that will search for any instances of double quotes and then replace them with an escaped double quote (\"). Once this has been processed, the output is displayed in the TextArea on the Stage and copied to the clipboard. This allows you to perform a quick, one-step conversion to prep your JSFL script for use in a panel. You're welcome.

The Move Items Panel

To demonstrate how to create a custom panel, we'll use the simple JSFL script we looked at earlier to move the selected items to a particular folder, except that we'll allow the panel to define the name of the folder. You can see the result of this example in the Move Items.fla file in the Appendix C examples folder.

First, we'll create a new copy of the moveItems.jsfl script and name it moveItemsPanel.jsfl. Next, we'll do a Find and Replace to change all of the instances of the string "Graphics" to some identifier—in this case, "FOLDER-NAME-HERE." What results is the following new JSFL code:

```
var selectedItems = fl.getDocumentDOM().library.getSelectedItems();
var folderExists = fl.getDocumentDOM().library.
itemExists("FOLDER-NAME-HERE");
```

```
if (!folderExists) fl.getDocumentDOM().library.
newFolder("FOLDER-NAME-HERE");
fl.getDocumentDOM().library.moveToFolder("FOLDER-NAME-HERE");
```

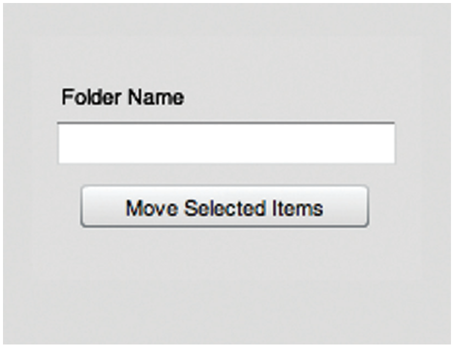


Figure C.2 The layout of our Move Items panel.

We'll set this code aside for the moment and create the panel itself. If you open Move Items.fla, you will see a few components on the Stage: a text input box, a label, and a button. This is shown in Figure C.2.

When a user puts a name inside this text box, it will be used when the folder is created by the JSFL. If it is left blank, the code will not run. Because the functionality of this panel is so simple, we won't bother to create an entire class for it (though you should for more complex panels). Instead, in the frame we'll add the following script:

```
moveButton.addEventListener(MouseEvent.CLICK, moveItems,
false, 0, true);

function moveItems(e:MouseEvent) {
  if (folderName.text == "") return;
}
```

Now, when the Move button is clicked, the *moveItems* method will be called. This is where we want to add our JSFL. Fire up the trusty JSFLConverter applet and load the new moveItemsPanel.jsfl file. It will be correctly converted and ready to paste into the code:

```
MMExecute("var selectedItems =
fl.getDocumentDOM().library.getSelectedItems();var folderExists =
fl.getDocumentDOM().library.itemExists(\"FOLDER-NAME-HERE\");if
(!folderExists) fl.getDocumentDOM().library.newFolder(\"FOLDER-NAME-
HERE\");fl.getDocumentDOM().library.moveToFolder(\"FOLDER-NAME-
HERE\");");
```

Now all that is left to do is to substitute the FOLDER-NAME-HERE string with the name specified in the text box. Simply pull up the Find and Replace window of the Actions panel (Ctrl–F on Windows, Cmd–F on Mac), and replace FOLDER-NAME-HERE with "+ folderName.text +." Flash will find three instances of this and replace them. The resulting code will look like the following:

```
moveButton.addEventListener(MouseEvent.CLICK, moveItems,
false, 0, true);

function moveItems(e:MouseEvent) {
  if (folderName.text == "") return;
  MMExecute("var selectedItems =
fl.getDocumentDOM().library.getSelectedItems();var folderExists =
fl.getDocumentDOM().library.itemExists(\"" +
folderName.text + "\");if (!folderExists)
fl.getDocumentDOM().library.newFolder(\"" + folderName.text
 + "\");fl.getDocumentDOM().library.moveToFolder(\"" +
folderName.text + "\");");
}
```

When this SWF is run inside of Flash, it will now execute the JSFL commands. In order to have the panel actually show up inside of the Flash menu system, you'll need to copy the SWF to a special folder called WindowSWF. This folder is in a different place depending on your OS, so it's easiest to just do a search for it. There are usually two copies of it—one in a subfolder of your Flash CS4 installation that is shared by all users and another in a subfolder of your user folder. Once you've placed the SWF file there, restart Flash and you'll see it as an option under the Window → Other Panels menu. Figure C.3 shows how the panel looks when active inside of Flash.

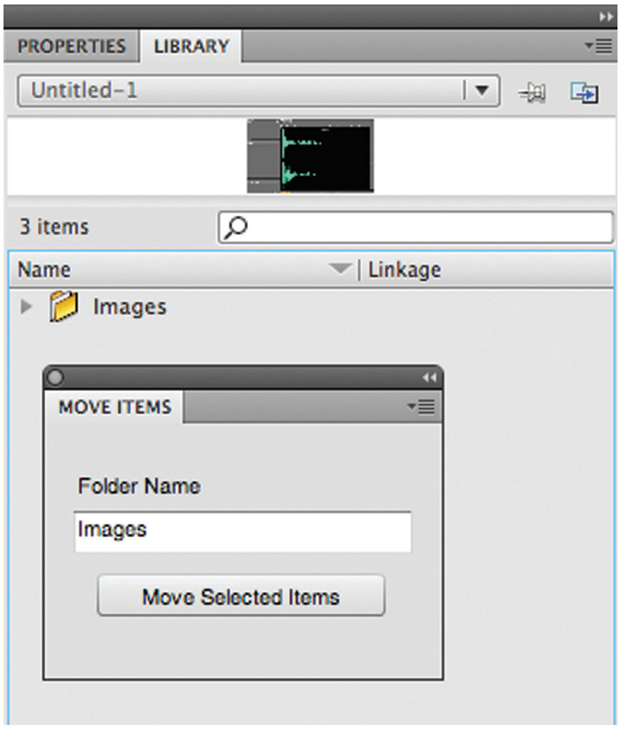


Figure C.3 The Move Items panel active inside of Flash.

Conclusion

Now that you know how to generate JSFL code and embed it in a custom panel, go nuts! Any time you have a repetitive task or want to streamline functionality in the Flash IDE, JSFL will quickly become your best friend. Scripts and panels like these can mean hours of saved time in productivity.