Table of Contents

Short Articles This Month

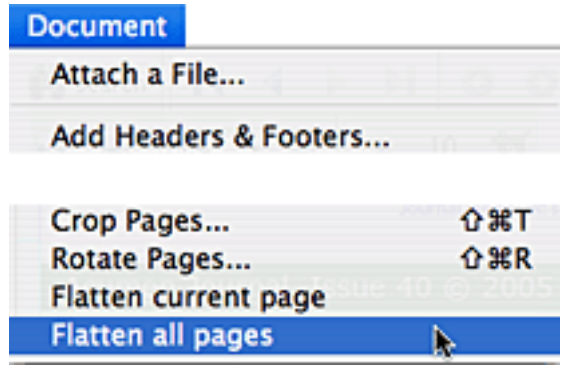
This month's articles are pretty short. I'm up to my ears working on the *PDF File Content and Structure 2* class.

I expect to have the new class ready to teach in November.

JavaScript: Adding Menu Items to Acrobat

Acrobat's JavaScript capabilities are remarkably powerful. One often-ignored tool that JavaScript puts into programmers' hands is the ability to modify Acrobat's menus. It is possible to add or remove menu items from Acrobat itself, allowing you to add your favorite JavaScripts to one of Acrobat's user interface.

This month, we shall see how to do this. Specifically, we shall add two commands to the *Document* menu, both based on last month's article on flattening PDF comments:



- *Flatten current page,* which will flatten all the comments on the currently-visible page.
- *Flatten all pages,* which will flatten all comments throughout the document.

You may wish to reread the previous *Journal* article on flattening comments, although this isn't actually necessary in order to follow this month's discussion. This article *does* assume you have minimal JavaScript skills, equivalent to having read my book *Extending Acrobat Forms With JavaScript*. (You have read it, haven't you?)

The JavaScript we write will be unusual in that it may be executed only as a *folder script,* a JavaScript that Acrobat is executes at startup time.

Let's see how we do this.

Next Page ->

App.addMenuItem

The JavaScript command with which you add items to an Acrobat menu is the App object's *addMenuItem* method:

```
app.addMenuItem({cName: "itemName",
                                cParent: "menuName", cExec: "fcnName"})
```

This method takes a relatively large number of named arguments, some of them optional; the most important are:

cName: "itemName"

This is the text of the item that should be put into the menu.

cParent: "menuName"

This is the name of the menu to which the item should be added.

cExec: "jsCode"

This is a snippet of JavaScript code that should be executed when the user selects this menu item. Often, this will simply call a JavaScript function defined elsewhere in the script.

nPos: itemPosition *(Optional)*

The position within the menu where the new item should be placed; if you don't supply this number, the item will be appended to the end of the menu.

There are additional arguments that you may supply to *addMenuItem;* I shall refer you to the *Acrobat JavaScript Object Reference* for the details on these.

Next Page ->

Acumen Journal: Acrobat User 3

Using *addMenuItem*

Here is a script that adds our Flatten Comments items to the *Document* menu:

```
// First define a function that flattens document pages; this is the
// function that carries out our menu items' itent.
function Flatten(boolCurrentPageOnly)     // Takes a boolean argument
{
     var i

     // Give them a chance to back out
     i = app.alert("Are you sure you want to do this?", 1, 2)

     if (i != 3)   {                  // If they didn't click "No"...
          if (boolCurrentPageOnly)    // Flatten either this page...
               this.flattenPages(this.pageNum)
          else
               this.flattenPages()    // ...or all pages
     }
}

// Now add our two menu items to the Document menu
app.addMenuItem({ cName: "Flatten current page",
                        cParent: "Document", cExec: "Flatten(true)"});
app.addMenuItem({ cName: "Flatten all pages",
                        cParent: "Document", cExec: "Flatten(false)"});
```

Next Page ->>

Discussion

This script has two parts:

- The definition of a function named *Flatten,* which takes a Boolean argument and flattens all of the comments on the current page or in the entire document, depending upon the Boolean's argument.
- A pair of calls to the App object's *addMenuItem* method that add two items to the Acrobat *Document* menu.

Let's look at the script in detail:

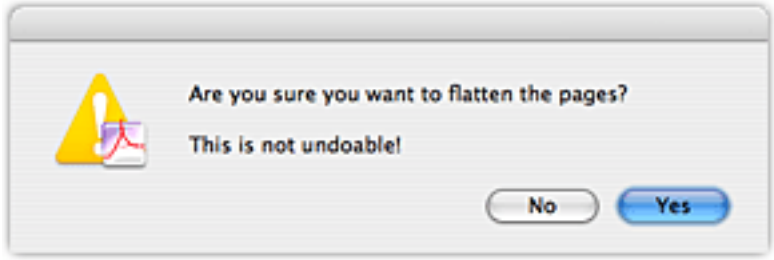
Step by Step

```
function Flatten(boolCurrentPageOnly)    // Takes a boolean argument
{
```

Our function, *Flatten,* will take a single Boolean argument. This argument dictates whether we flatten only the current page (true) or the entire document (false).

```
i = app.alert("Are you sure you want to do this?", 1, 2)
```

We display a warning that gives the user a chance to back out of the command. This is important since, once the comments have been flattened, they can't be put back again.



Remember that *app.alert* returns an integer indicating which button the user clicked in the dialog box.

Next Page ->

```
if (i != 3)  {
     ...
}
```

The *app.alert* method returns a value of *3 if* the user clicks the alert's *No* button; this value was placed in the variable *i*. We will want to flatten the comments only if *i* is not *3*.

flattenPages

See the June 2005 *Journal* for a discussion of the Doc object's *flattenPages* method.

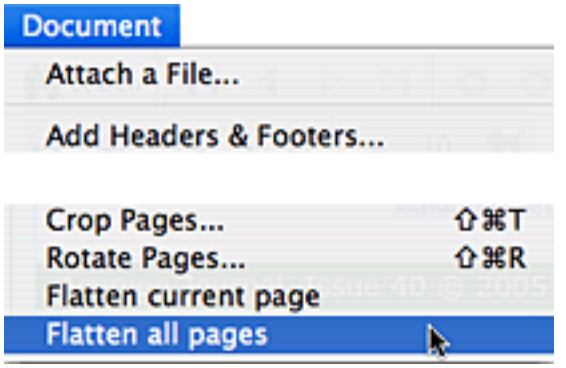
```
if (boolCurrentPageOnly)
     this.flattenPages(this.pageNum)
else
     this.flattenPages()
```

We then call the Doc object's *flattenPages* method either with or without a page number, depending on the value of the function's Boolean argument, *boolCurrentPageOnly*.

This conditional call to *flattenPages* ends our *Flatten* function.

```
app.addMenuItem({ cName: "Flatten current page",
                  cParent: "Document", cExec: "Flatten(true)"});
```

We now make our first call to *app.addMenuItem*, adding a menu item that will flatten the current page. We set three of the method's arguments, as follows:



cName: "Flatten current page"

This is the text that appears in the menu.

cParent: "Document"

We add the item to the *Document* menu.

Next Page ->

Acumen Journal: Acrobat User 6

cExec: "Flatten(true)"

If the user selects this menu item, we execute our *Flatten* function, handing it the argument value *true*. This will flatten comments on the current page.

```
app.addMenuItem({ cName: "Flatten all pages",
                  cParent: "Document", cExec: "Flatten(false)"});
```

Our second call to *app.addMenuItem* is similar to the first. We have a slightly different name for this menu item and we are passing *false* to the *Flatten* function causing that function to flatten all the comments in the document.

Folder Scripts

We can't use this script just anywhere

So now we have a script that creates our "flatten pages" menu items. Where do we put this script? Therein lies a serious, though reasonable, restriction: *app.addMenuItem* can only be executed when Acrobat starts up; it is ignored if we execute it when Acrobat is already running.

The javascripts folder

Upon startup, Acrobat looks inside a predefined folder named *javascripts* and executes any files in there that have a suffix of *.js*. To have our script executed at startup time, we simply have to place the *js* file into the *javascripts* folder. The trick is locating this folder.

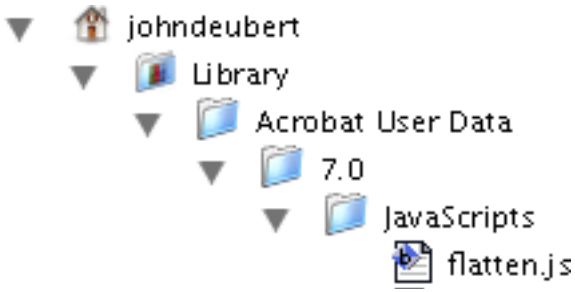
It's easy in Windows: the *javascript* folder resides in the same folder as Adobe Acrobat.

Next Page ->>

It is less straightforward to find this folder on the Mac. Through Acrobat 5, the *javascript* folder was located in the Acrobat folder, as in Windows. In both Acrobat 6 and 7, Adobe made the situation a bit more complex:

- In Acrobat 6, folder javascripts should go into *~/Library/ Acrobat User Data/javascripts*, where "~" signifies your home directory, as usual.
- In Acrobat 7, the javascript folder is *~/Library/Acrobat User Data/7.0/javascripts*.

This wasn't just random madness on Adobe's part; the goal was to restrict the scripts to the computer's current user, rather than being globally forced on all users. This is an important security feature, given the broad capabilities of JavaScript in Acrobat.



Note that your file must be suffixed *.js* for Acrobat to recognize it as containing a JavaScript.

Other Interface Modification

The Acrobat JavaScript interface allows you to make considerable modification to Acrobat's interface. It's particularly easy to add submenus *(app.addSubMenu)* and even your own toolbar buttons (*app.addToolButton)*.

But I'll let you explore those on your own.

Return to First Page

Drawing Dotted Lines in PostScript

PostScript lets you stroke paths with dashed lines of reasonable complexity, as in the box at right. You do this by calling the *setdash* operator, which specifies the details of the dashed line you want *stroke* to use.

What is trickier to achieve is to stroke a path with a *dotted* line, made up of little round dots running along the path, as at right.

This month's short article will demonstrate a trick by which you can produce this effect. The trick is based upon the behaviors of two PostScript operators: *setdash* and (seemingly unrelated) *setlinecap.*

We'll start with *setlinecap.*

Next Page ->>

setlinecap

The *setlinecap* operator allows a PostScript program to specify how the endpoints of stroked lines should be rendered. The operator takes an integer code as its argument:

```
n setlinecap
```

This integer may have values *0, 1,* or *2,* each specifying a different type of line ending, as follows:

0 — *Butt caps* - Stroked lines are simply chopped off at the endpoints. This is PostScript's default behavior.

1 — *Round caps* - PostScript draws semicircular caps at each endpoint, as illustrated at right.

2 — *Extended caps* - This is identical to butt caps, but the stroked line extends half the line width beyond the ends of the current path.

The current linecap resides in the graphics state and so is subject to *gsave* and *grestore*. Also, the linecap is applied at *stroke* time, not when you construct the path, so if you call *setlinecap* several times, *stroke* will apply whichever linecap is current.

Next Page ->>

setdash

The *setdash* operator specifies the dash pattern that *stroke* should apply to the current path. The default pattern, of course, is a solid line; but the *setdash* operator allows you to specify quite elaborate patterns of painted and unpainted segments for the stroked path.

The *setdash* operator takes two arguments, an array and a number:

```
[ dash array ] offset  setdash
```

The dash arrays specifies the details of what the dashed line should look like; the offset indicates how the dash pattern should be positioned on the path.

Let's look at these in detail.

Dash Array

The dash array contains a series of numbers, indicating the lengths of the alternating painted and unpainted parts of the dashed line. Thus, the following call to *setdash*

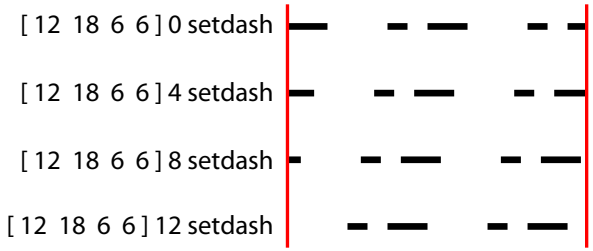
```
[ 12  18  6  6 ] 0 sedash
```

would yield the dashed line illustrated above right, made up of the following pattern: paint 12 units, skip 18 units, paint 6 units, skip 6 units, repeat to the end of the path.

Next Page ->>

Offset

The offset argument specifies how much of the pattern should be skipped at the starting endpoint.



At right, we have the same 12-18-6-6 dash pattern with a variety of offsets. When the offset is *0,* the beginning of the path corresponds to the start of the first dash pattern. When the offset is *6,* the iniitial endpoint starts 6 units into the pattern; we are missing half of the first 12-point dash in the painted line.

So, Dots?

The first secret to producing a dotted line is that PostScript applies the current linecap to each of the painted segments in the dashed line. At right is a dashed line with a linecap of *0* (top) and *1* (bottom). Note that each of the dashes has in the second line has rounded caps on either end.

The second secret is that a dash length of *0* is perfectly acceptable within a dash array; those segments will have lengths of zero. The surprise is that these zero-length dashes will have the current linecap applied to them; a zero-length dash with rounded caps is a dot.

Next Page ->

For example So, to stroke a rectangle with a dotted line, we could do the following:

```
8 setlinewidth
1 setlinecap
[ 0 16 ] 0 setdash      % 0-length dash; 16-pt gap

.5 1 .75 setrgbcolor
100 600 100 50 rectstroke

showpage
```


Easy, isn't it?

Return to First Page

Schedule of Classes, October-December 2005

Following are the dates of Acumen Training's upcoming PostScript and PDF Technical classes. Clicking on a class name below will take you to the description of that class on the Acumen training website.

These classes are taught in Orange County, California and on corporate sites world-wide. See the Acumen Training web site for more information.

Technical Classes

PDF File Content and Structure 1		Nov 7–10	
NEW! PDF File Content and Structure 2			Dec 12–15
PostScript Foundations			Jan 9–13, '06
Variable Data PostScript	Oct 3–7		
Advanced PostScript			
PostScript for Support Engineers		*On-site only*	

Course Fee The PostScript and PDF classes cost $2,000 per student.

Registration Info

Acrobat Class Schedule

These classes are taught occasionally in Costa Mesa, California, and on corporate sites. Clicking on a course name below will take you to the class description on the Acumen Training web site.

Acrobat Essentials — *No Acrobat classes scheduled for this quarter.* See the Acumen Training website regarding setting up an on-site class.

Interactive Acrobat

Creating Acrobat Forms

Acrobat Class Fees — *Acrobat Essentials* and *Creating Acrobat Forms* (½-day each) cost $180.00 or $340.00 for both classes. There is a 10% discount if three or more people from the same organization sign up for the same class.

Registration ->

Return to Main Menu

Contacting John Deubert at Acumen Training

For more information

For class descriptions, on-site arrangements or any other information about Acumen's classes:

Web site: http://www.acumentraining.com **email:** john@acumentraining.com

telephone: 949-248-1241

mail: 24996 Danamaple, Dana Point, CA 92629

Registering for Classes

To register for an Acumen Training class, contact John any of the following ways:

Register On-line: http://www.acumentraining.com/registration.html

email: registration@acumentraining.com

telephone: 949-248-1241

mail: 24996 Danamaple, Dana Point, CA 92629

Back issues

All issues of the *Acumen Journal* are available at the Acumen Training website:
http://www.acumenjournal.com/AcumenJournal.html

Return to First Page

What's New at Acumen Training?

PDF File Content & Structure 2 Scheduled

The first *PDF File Content and Structure 2* class is scheduled for December 12. The course description page for this class will be up and running before the end of the month. The topic list is still subject to change, but the following are very likely to be covered in class:

Overprinting	File Specification	Multibyte fonts
Masked Images	Halftones	Linearized PDF
Marked Content	AcroForms	Rendering Intents
Transfer Functions	Functions dictionaries	Smooth shading
Shape dictionaries	Xref streams	Object streams
Name Dictionaries	More on data structures	

The prerequisite for this class is the *PDF File Content and Structure 1* class.

If you are curious about the flavor of *PDF File Content & Structure 2* class, I have posted a sample chapter from the student notes on the Acumen Training Resources page. Look among the PDF resources for "PDF FC&S Sample.pdf." The chapter included is that on PDF file specification. This may take a couple of days for me to get up on the site.

Return to First Page

Journal Feedback

If you have any comments regarding the *Acumen Journal,* please let me know. In particular, I am looking for three types of information:

Comments on usefulness. Does the Journal provide you with worthwhile information? Was it well written and understandable? Do you like it, hate it? Did it make you remember fondly your last root canal?

Suggestions for articles. Each Journal issue contains one article each on PostScript and Acrobat. What topics would you like me to write about?

Questions and Answers. Do you have any questions about Acrobat, PDF, or PostScript? Feel free to email me about. I'll answer your question if I can. (If enough people ask the same question, I can turn it into a Journal article.)

Please send any comments, questions, or problems to:

journal@acumentraining.com

Return to Menu