Table of Contents

Locking Form Fields and Comments With *flattenPages*

JavaScript

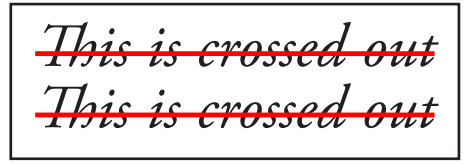
This month's article presumes you have some experience in using JavaScript with Acrobat. You'll need to know at least the basics of attaching a JavaScript to a button or other form field, about the equivalent of having read my book *Extending Acrobat Forms With JavaScript*.

Hint, hint.

A lawyer friend of mine was scanning some of his paper forms and converting them to PDF files, adding form fields in the places where he would normally handwrite clients' names, etc. His intent, of course, was to be able to type the information into the appropriate places on the forms and then send them to his clients as PDF files for signature.

A concern of his was he wanted to make sure that the information he typed in couldn't be changed by his clients. He could add password security to the files, but he was looking for something more streamlined.

What he decided to do was to *flatten* the comments and form fields in the PDF file. In this context, the term "flatten" has nothing to do with transparency. Rather, it refers to converting PDF annotations and form fields into regular drawn objects on the page. Once this is done, comments and form fields become immutable, since they have been transformed into lines, squares, text, and other non-interactive graphic objects.



For example, compare the two pieces of text at right. Both were crossed out using the Acrobat Text Annotation tools. In the top line, the original annotation is in place; you can click on the annotation within Acrobat and the strike through, you can double-click on the red line and see the text associated with the annotation.

The second line has been flattened; you cannot select, double-click, or otherwise manipulate the red line, because it is no longer an annotation; it is simply a red line.

Next Page ->

How Do We Do It?

Acrobat does not provide access to the *flattenPages* feature it its user interface; you must write a JavaScript, attached to a button, for example.

This can be a *very* simple JavaScript, consisting of a call to the Doc object's *flattenPages* method:

```
this.flattenPages(pageNum)
```

The *flattenPages* method takes an optional page number as its argument and flattens the comments and form fields on the specified page. If you omit the page number,

```
this.flattenPages()
```

then the method will flatten all of the pages in the PDF document.

This method was added to the Acrobat JavaScript interface in Acrobat 5, so it will be available on most of the Acrobat installations you are likely to encounter.

Let's use this in a specific example.

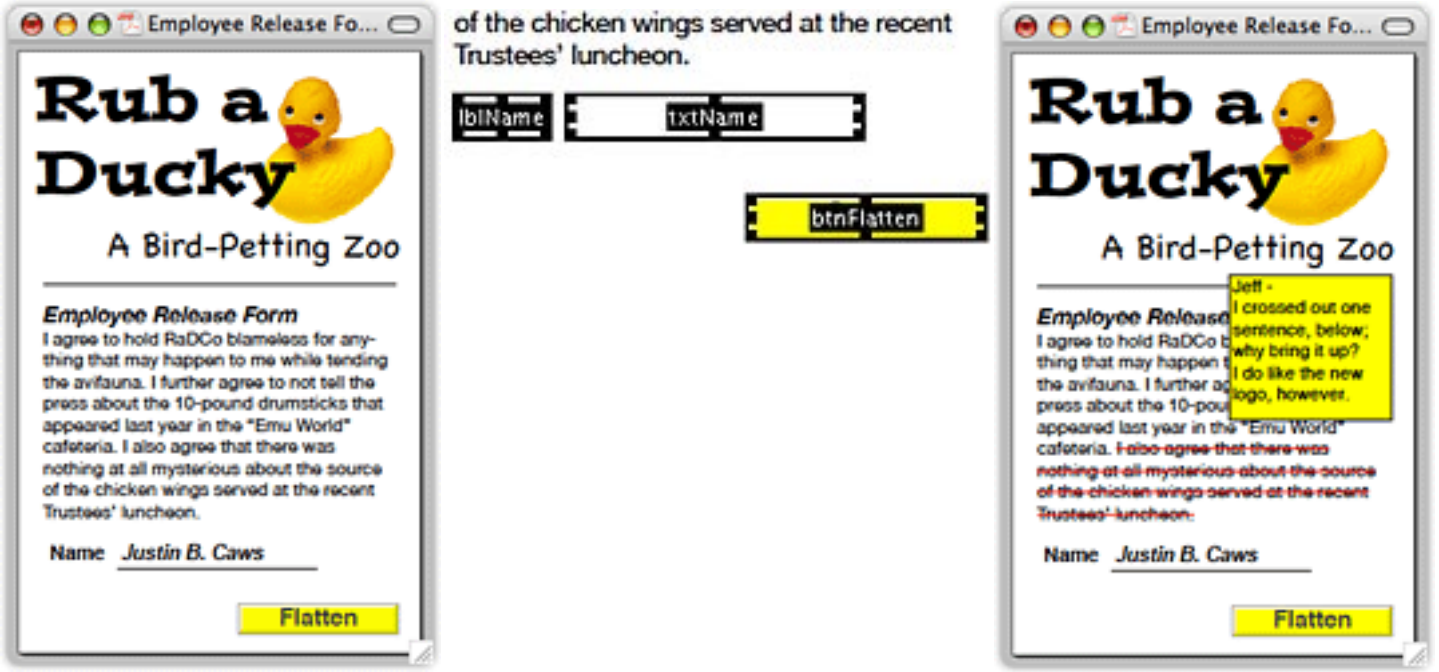
Next Page ->>

A *Flatten Doc* Button

Files on Web Site

As usual, the sample files for this month's article are on the Acumen Training Resources page among the Acrobat examples. Look for the file *Flattenpages.zip*.

Below, left, is a simple PDF form with a single text box and a *Flatten* button, named *txtName* and *btnFlatten,* respectively, as shown below center. Below right, I have added a text annotation to the document and added a "strike-through" annotation over some of the text, crossing out the text.



We are going to add a JavaScript to the *Flatten* button that calls *flattenPages*.

Next Page ->

Attaching a JavaScript

Let's remind ourselves how to attach a JavaScript to an Acrobat Button form field. This description will be brief, because you will have already read how to associate JavaScripts with buttons in my JavaScript book. (Right?)

Sample Files

As it says at right, the file *RubADucky0.pdf* among this article's sample files contains the comments and form fields, but no JavaScript; you may add it yourself.

The file *RubADucky1.pdf* is the same file, but with the JavaScript already typed in. The *Flatten* button works in this one.

If you want to follow along, start with the file *RubADucky0.pdf,* among this issue's sample files. This document has the comments and form fields, but no JavaScripts.

With the PDF file open, do the following:

1. Select the *Select Object* tool in the *Advanced Editing* toolbar.

   All of the form fields in the document will become outlined in the usual rectangles-with-handles.

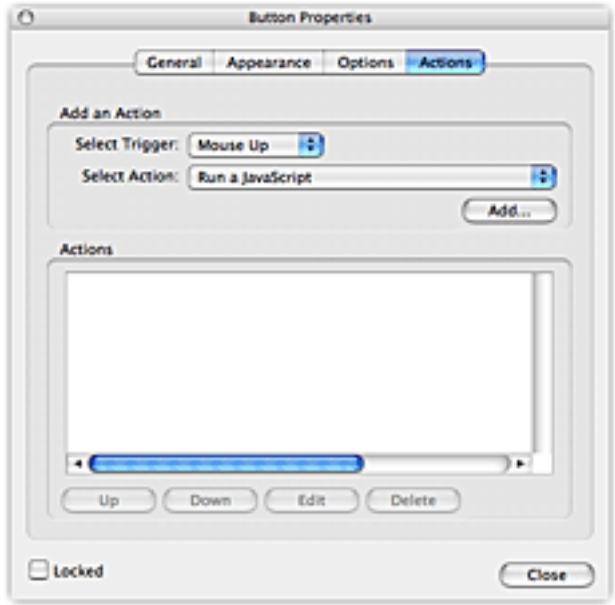
2. Double-click on the *Flatten* button.

   Acrobat will present you with the *Button Properties* dialog box, at right.

3. Click on the *Actions* tab and, in that panel, select *Mouse Up* for the trigger and *Run a JavaScript* for the action.

4. Click the *Add* button.

   Acrobat will present you with the *Edit JavaScript* dialog box (next page).

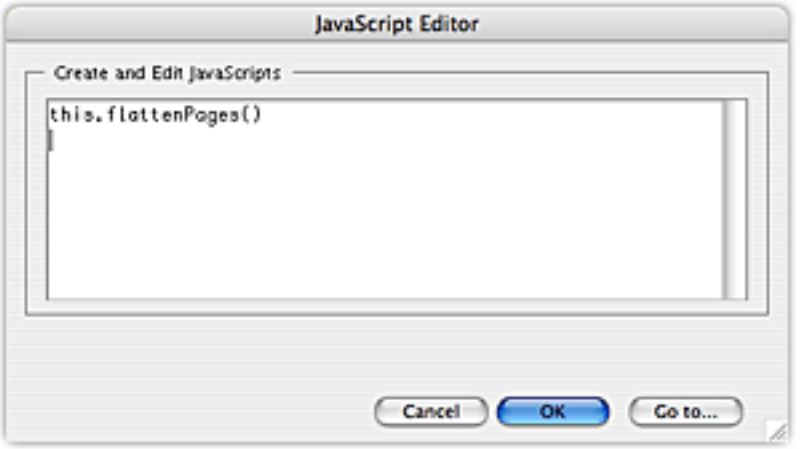


Next Page ->


5. Type the following JavaScript into the *JavaScript Editor* dialog box:

   ```
   this.flattenPages()
   ```

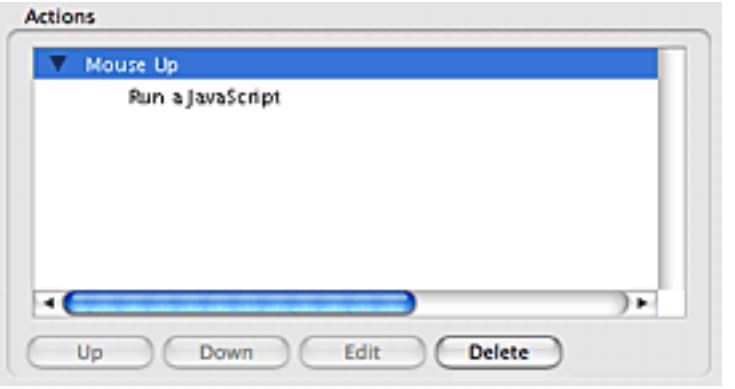
   This script calls the *flattenPages* method in the current document object. Since we are not passing a page number to the method, this JavaScript will flatten the entire document.



6. Click the *OK* button.

   Acrobat will return you to the *Button Properties* dialog box, now listing the JavaScript *Mouse Up* action.

7. Click the *Close* button in the *Button Properties* dialog box and we are done.



Done! We now have a functioning flatten button. When you click on this button, Acrobat will flatten all of the comments and form fields within the document.

So far, so good. There are a couple of improvements we can make, however.

Next Page ->

Acumen Journal: Acrobat User 6

Hide the Button

One annoyance is that the flattening process also flattens our *Flatten* button, converting it to a useless—and annoying—graphic object. This is an invitation to frustration, since it still looks like a button, but clicking on it doesn't do anything.

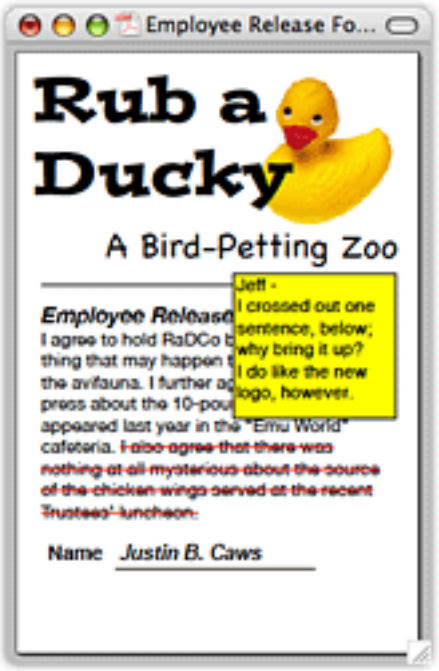
So, let's change our button's JavaScript so that, before we flatten the document, we hide the *Flatten* button.

```
var f = this.getField("btnFlatten")
f.hidden = true

this.flattenPages()
```

This code does three things:

- Get a reference to the current document's *btnFlattern* field, placing this reference in variable *f.*
- Set the field's *hidden* property to *true,* making the button invisible.
- Flatten the pages, as before.



Now, when we click *Flatten,* the button disappears and then everything is converted to flat graphics. (The button is still being flattened along with everything else, but since we can't see it anymore, we don't care.)

Next Page ->

Add an Alert

Finally, let's give the user a chance to back out of the flatten operation. Since the effect of flattening is broad and undoable, it seems only fair to warn the user and offer a way out.

Below is the final script as I would use it. It first presents the user with an "Are you sure?" dialog box (right). It flattens the file only if the user does not click the "No" key.


```
var i

i = app.alert("Are you sure you want to flatten the pages?", 1, 2)

if (i != 3)   {
     var f = this.getField("btnFlatten")
     f.hidden = true
     this.flattenPages()
}
```

Remember that the Application object's *alert* method takes a string, an icon code, and a "what buttons" code. It returns an integer indicating which button was clicked; a return value of *3* would indicate th euser clicked the *No* button.

Our script flattens the document only if the return value, *i,* is not *3.* Refer to my *JavaScript* book or the Acrobat JavaScript Objects Reference (downloadable from the Acumen Training Resources page) for more information on the App object's *alert* method.

Next Page ->

Final Notes

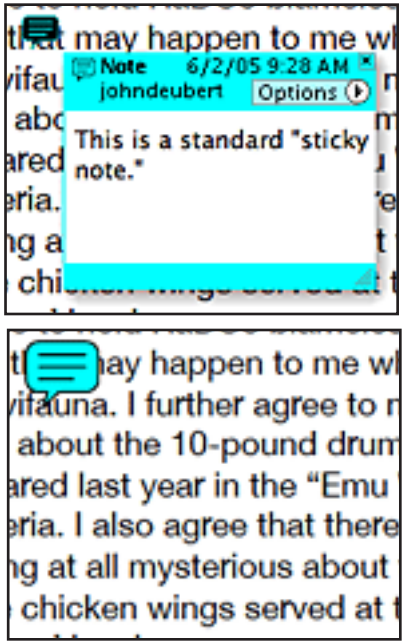
Appropriate Use

Flattening a PDF file is a remarkably useful ability under many circumstances. It isn't something you would normally use in an interactive form that you are going to send to someone else to fill out. However, it is something that is very useful in a form that you are going to fill out and then send to someone else as a customized, finished document (such as my friend's boiler-plate contract).

Caveat

The only caution I have to pass on is that when flattening comments within a document, the flattening process preserves only the graphic associated with the comment, not the pop-up text window.

Thus, a standard "sticky note" comment, as at top right, when flattened becomes just the icon; the comment text is discarded. This is fine for text edit comments, circles, arrows, etc.; it definitely is useless for sticky notes. (You'll notice that the text comment in the "RubADucky" file is actually a text *box* annotation.)



Return to Main Menu

Handling Errors in PostScript, Part 2

In the last issue, we saw how to override Acrobat's default error handling mechanism. We redefined the *handleerror* procedure in *errordict,* allowing us to emit an error message to stdout that included the name of the error, the offending command, and the contents of the operand stack.

Last month, we just duplicated the functionality of Distiller's default error handler. This month, we shall extend our error handler so that it will do two new things:

- Print the contents of the Dictionary stack.
- Print its error message on the current page, rather than sending the message to stdout.

This article presumes you have freshly read the first part, in the April 2005 issue of the *Acumen Journal.* You can get that issue from the Acumen Training web site, as usual.

You should also read the November 2003 *Journal,* which describes a *FindName* procedure that we shall be using in this month's project.

Next Page ->

The Story So Far...

When we last left our heros, we had written a replacement for the default *handleerror* procedure that resides in *errordict*. Here is the program we ended with last time:

Files on the Web

As usual, this file is on the Acumen Training Resources page. Look for the file *ErrorHandler 2.zip*.

The code at right is in the file *BasicErrorHandler.ps* within the zip archive.

```
errordict begin          % Put errordict on the dict stack
/handleerror             % Define handleerror:
{    $error begin        % Put $error on the dict stack
     newerror {          % Is neweerror true?
          (*** PostScript Error ***) =     % Yes: emit error msg
          (Error: ) print errorname =      % Print error name
          (Offending command: ) print      % Print offending command
          /command load ==
          (Operand stack: ) =              % Print a label
          clear                            % Clear the operand stack
          ostack aload pop                 % Unload ostack
          count { (\t) print == } repeat % Print stack contents
          flush                            % Flush %stdout
          /newerror false def              % Reset newerror
     } if
     end                         % Remove $error from the dict stack
} bind def                       % End of handleerror definition

/dhandleerror /handleerror load def      % accommodate Distiller
end                              % Remove errordict from dict stack
% Now let's try the new error handler
1 2 /3 (O'Leary) moveto          % moveto will yield a typecheck error
```

Next Page ->

Acumen Journal: PostScript Tech 11

For the remainder of this article, code listings will present only the *handleerror* definition; the other code, still necessary, will be omitted, to save space.

Displaying the Dictionary Stack

The first change we shall make to our error handler is to display the contents of the dictionary stack, in addition to the operand stack. This will be easy; we need only do with the *dstack* array what we have already done with *ostack*. Here's the revised error handler, with the new parts in red.

The code at right is in the file *BasicErrorHandler2.ps* within this issue's zip file.

```
/handleerror
{    $error begin          % Push $error on the stack
     newerror {            % Is newerror true? If so...
          (*** PostScript Error ***) =     % ... Print a title
          (Error: ) print errorname =      % Print the error name
          (Offending command: ) print /command load ==  % Print cmd
          (Operand stack: ) =              % Print the operand stack...
          clear ostack aload pop           % ...unpack the ostack array
          count { (\t) print == } repeat   %  & print each item
          (Dictionary stack: ) =           % Print the dict stack in the
          dstack aload pop                 % same way.
          count { (\t) print == } repeat
          flush
          /newerror false def              % Reset newerror
     } if
     end
} bind def
```

Next Page ->>

The only problem with this version of our program is that the double-equal operator does a very poor job of printing the Dictionary stack. Our error message looks like this:

```
*** PostScript Error ***
Error: typecheck
Offending command: --moveto--
Operand stack:
     (O'Leary)
     /3
     2
     1

Dictionary stack:
     -dict-
     -dict-
     -dict-
```

While this tells us that there were three dictionaries on the Dictionary stack, it doesn't tell us what they are. It would be more useful if our error message told us each dictionary's name, if it has one.

This is where our *FindName* procedure comes in.

Adding *FindName*

The November 2003 *Journal* presented the definition of a *FindName* procedure:

```
/FindName {   % stack:  PSObject  =>  /name true or false
     ....
} bind def
```

Next Page ->

Acumen Journal: PostScript Tech 13

This procedure takes a PostScript object as its argument and searches all dictionaries on the Dictionary stack for a key-value pair whose value is the argument object. It returns either a boolean *false,* if no such key-value pair exists, or the name associated with the object and a boolean *true,* indicating a match was found.

I am not going to step through the definition of *findName;* if you want to see its definition, click here.

We shall now use *FindName* to improve our error handler. Now, to print the Dictionary stack, we do the following; again, the new code is in red:

The code at right is in the file *BasicErrorHandler3.ps* within this issue's zip file.

```
(\nDictionary stack:) =
dstack aload pop                        % Unload the dstack array
count {                                 % For each item now on the stack
     (\t) print                         % Print a tab
     FindName                           % Look for the name of the object.
     { = }                              % Found: print the object's name
     { (<<Unnamed dict>>) = }           % Not found: print "unnamed dict"
     ifelse
} repeat
```

We have changed the printing of the *dstack* array so that, for each item, we execute *FindName*. If that procedure returns a *true* on the stack (with the dictionary's name beneath it), we simply call the *equal* operator, printing the name. If *FindName* returned *false,* indicating the dictionary being tested had no name associated with it, then we simply emit the phrase "<<Unnamed dict>>".

Next Page ->

Acumen Journal: PostScript Tech 14

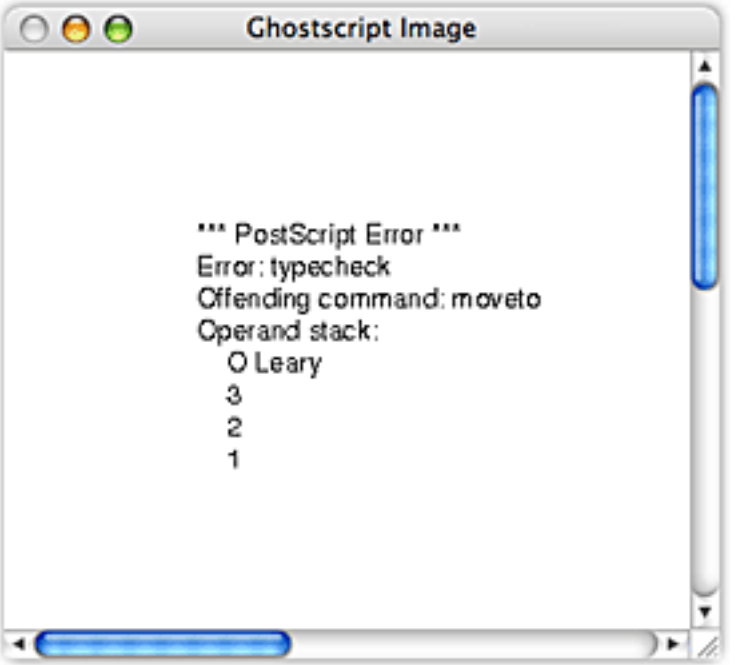
Now, our error message reports the dictionary stack contents like this:

```
Dictionary stack:
     userdict
     globaldict
     systemdict
```

You could extend this so that you print the names of objects on the operand stack, as well, but I shall leave that as an Exercise For the Student.

Printing the Errors

The final version of our error handler will print its error message on the current page. For simplicity, we shall revert to the earlier error message, consisting of the name of the error, the offending command, and the contents of the operand stack. A typical error page would look like the illustration at right.



Note that I'm using GhostScript, rather than Distiller, to view this file. Distiller does not seem to like printing error messages to the current page. (This isn't unreasonable; it's hard to imagine a "real world" circumstance in which one would want a PDF file with an error message on the page.)

Next Page ->

Approaching the Task

The new *handleerror* definition is very similar to that presented in the file *BasicErrHandler.ps,* with which we ended in the previous *Journal*. There are two broad changes between that earlier version and our present definition.

Reset Grapics State

We can know nothing about the state the PostScript machine will have at the time the error takes place. Therefore, the first thing our error handler will have to do is set the graphics state to its initial condition. We can do this with a call to *initgraphics*.

Replace "=" with "show"

In our earlier error handlers, we used double-equal to send the pieces of our error message to stdout. In our new version, we'll convert each object to a string and print the string on the current page with *show*. Each time, we'll then move the current point to the beginning of the next line.

For convenience and readability, we'll also define some variables and a couple of procedures.

Here's the code listing:

The Code

The code at right is in the file *BasicErrorHandler4.ps* within this issue's zip file.

```
/lm 72 def                  % Left margin for our error message
/str 100 string def         % A scratch string for use in string conversions
/lnHt 12 def                % The distance between lines of test on the page

/nl                         % Move currentpoint to the next line
{ lm currentpoint exch pop lnHt sub moveto } bind def
```

Next Page ->

Acumen Journal: PostScript Tech 16

```
/showObject       % psObj => ---     Convert object to string & print it
{ str cvs show } bind def

errordict begin    % Put errordict on dictionary stack
/handleerror       % Begin our handleerror definition
{    $error begin  % Put $error on dict stack
     newerror {    % If newerror is true...
          initgraphics  % Initialize graphics state
          /Helvetica 10 selectfont    % Set current font
          lm 720 moveto      % Move to initial position on the page
          (*** PostScript Error ***) show nl   % Print label
          (Error: ) show errorname showObject nl    % Print error
          (Offending command: ) show
          /command load showObject nl      % Print offending command
          (Operand stack: ) show nl        % Print operand stack
          clear
          ostack aload pop
          count {                  % For each object on the stack:
               (     ) show showObject nl   % Print obj. on the page
          } repeat
          showpage                 % Print the error page
          /newerror false def      % Set newerror to false
     } if
     end                           % Remove $error from dict stack
} bind def                         % End of our handleerror definition
end                                % Remove errordict from dict stack

1 2 /3 (O'Leary) moveto
```

Next Page ->

Acumen Journal: PostScript Tech 17

Let's look at the new parts of this error handler in detail. (You may want to review the code step-through in the April 2005 *Journal* before reading this, since I'm not going to look in detail at code identical to our earlier programs'.

Step by Step: Variables & Procedures

```
/lm 72 def
/str 100 string def
/lnHt 12 def
```

We start by defining three variables.

- *lm* will be the left margin, the starting point of each line in our error message.
- *str* is a scratch string we'll use when we convert objects to strings (using *cvs)* so that we can print them with *show.*
- *lnHt* ("line height") is the distance between successive lines of text in our error message. Going from one line of text to the next, the current point moves down by *lnHt.*

```
/nl
{ lm currentpoint exch pop lnHt sub moveto } bind def
```

The *nl* procedure moves the current point to the beginning of the next line; specifically, it moves the current point down by *lnHt* and back to the left margin, *lm*. You have seen this procedure before if you have taken any of my PostScript classes, but briefly, here's what the procedure does:

Next Page ->>

lm
Push our left margin onto the operand stack; this will be our new *x* coordinate.

currentpoint
Execute the *currentpoint* operator, which pushes the current *x* and then the current *y* onto the operand stack. Our operand stack now looks like this:

```
72 x y
```

We want to decrement *y* by *lnHt* and discard *x* altogether, replacing it with the *72*.

exch pop
We discard the current *x* value...

lnHt sub
...and subtract *lnHt* from *y*. This leaves on the operand stack the *x* and *y* values of the beginning of the next line.

moveto
We move to that position. The current point is now at the start of the next line.

```
/showObject       % psObj => ---
{ str cvs show } bind def
```

The *showObject* procedure takes a PostScript object from the operand stack, converts that object to a string representation and then prints that string on the current page with *show*.

Next Page ->

Acumen Journal: PostScript Tech 19

Changes to handleerror

Our *handleerror* procedure is very like our earlier versions. There are only a few differences:

```
newerror {
     initgraphics
```

The first thing we do after establishing that *newerror* is true is to call *initgraphics*. This resets everything in the graphics state back to their original values. This will include the coordinate system, color, and clipping path; this will *not* erase the current page, so our error page will still have whatever marks were on it when the error took place.

```
/Helvetica 10 selectfont
lm 720 moveto
```

We set the current font to a 10 point Helvetica and then move the current point to an initial position on the page.

```
(*** PostScript Error ***) show nl
```

Any constant string we want to print will now be printed with a *show,* rather than double-equal. At the end of each line within our error message, we shall call our *nl* procedure.

```
ostack aload pop
count {
     (     ) show showObject nl
} repeat
```

We shall precede each object on the operand stack with a series of printed spaces; this allows us to indent the stack contents in the absence of a tab function. We then print the stack item with our *showObject* procedure and do a newline.

Next Page ->

`showpage`

Finally, having printed the contents of the operand stack, we do a *showpage,* ejecting the current page with its error message.

That's it

We now have a functioning error handler that will print an error message to the current page. Since we didn't erase the page before printing our message, it will have on it all of the marks drawn before the error took place, which is very useful for diagnosing the source of the error.

For More Information

If you are curious how to add more functions to the error handler, you might look at the PostScript code for the Acumen Errorhandler on the Acumen Training Resources page. This is a fully-functional error handler that prints the contents of the operand, dictionary, and execution stacks and the *ErrorInfo* array, in addition to the things printed by this issue's basic error handler. It also demonstrates how to carry out different reporting procedures based on the type of data on the stack and a variety of other useful techniques.

Be aware that the Acumen Errorhandler is not intended primarily as a teaching tool and, therefore, the documentation and commenting are fairly minimal.

Return to Main Menu

Schedule of Classes, July - September 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		Aug 8–11	
PostScript Foundations	July 18–22		Sept 19-23
Variable Data PostScript			
Advanced PostScript			
PostScript for Support Engineers			
Jaws Development		*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?

International Classes

A number of people have contacted me in recent weeks asking if I teach classes outside of the United States. The answer is, of course, "Yes." In fact, in the past twelve months, I have taught more classes outside the U.S. than inside in cities including Munich, Amsterdam, Antwerp, Guadalajara, Cambridge, Ottowa, Scarborough, and Delhi.

My PostScript and PDF on-site classes have the same pricing stucture regardless of location. For details, go to the Acumen Training On-site page; there is a link on that page for additional information about classes outside the U.S.

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?

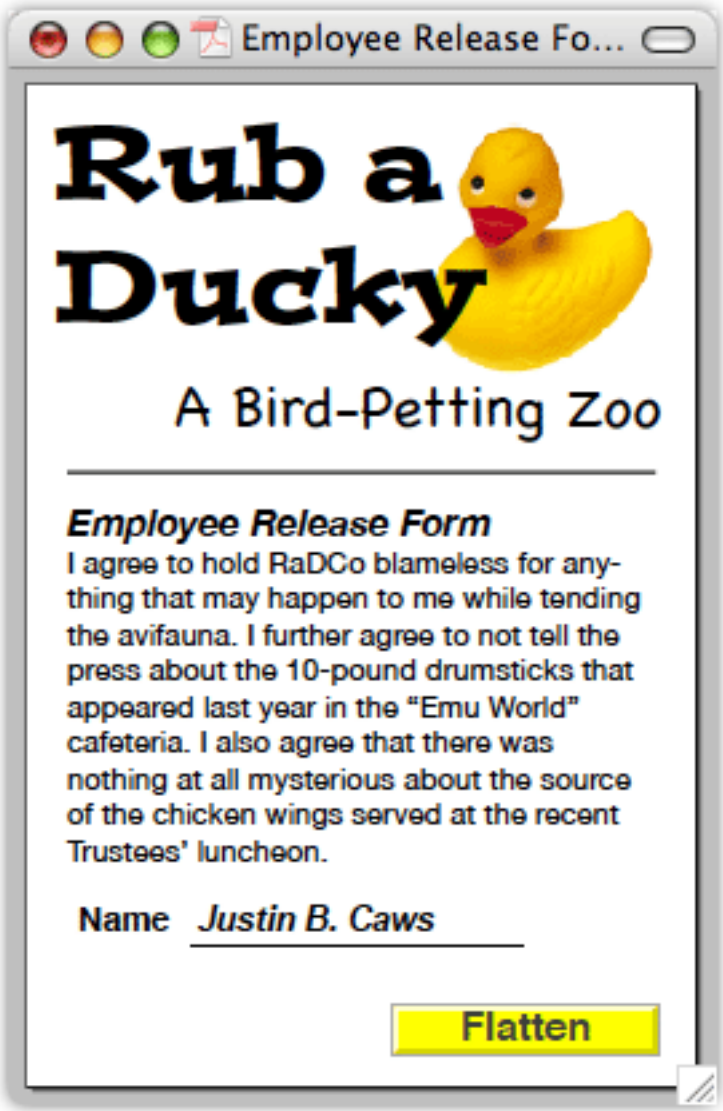
Suggestions for articles. Each Journal issue contains one article each on PostScript and Acrobat. What topics would you like me to write about?

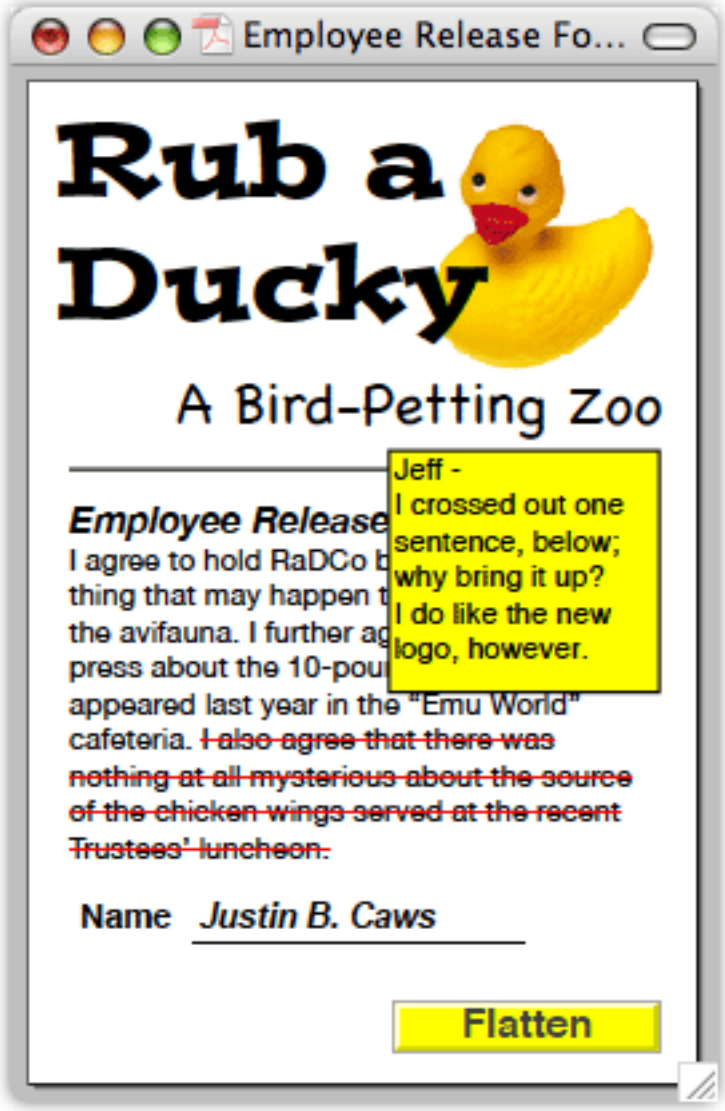
Questions and Answers. Do you have any questions about Acrobat, PDF, or PostScript? Feel free to email me about. I'll answer your question if I can. (If enough people ask the same question, I can turn it into a Journal article.)

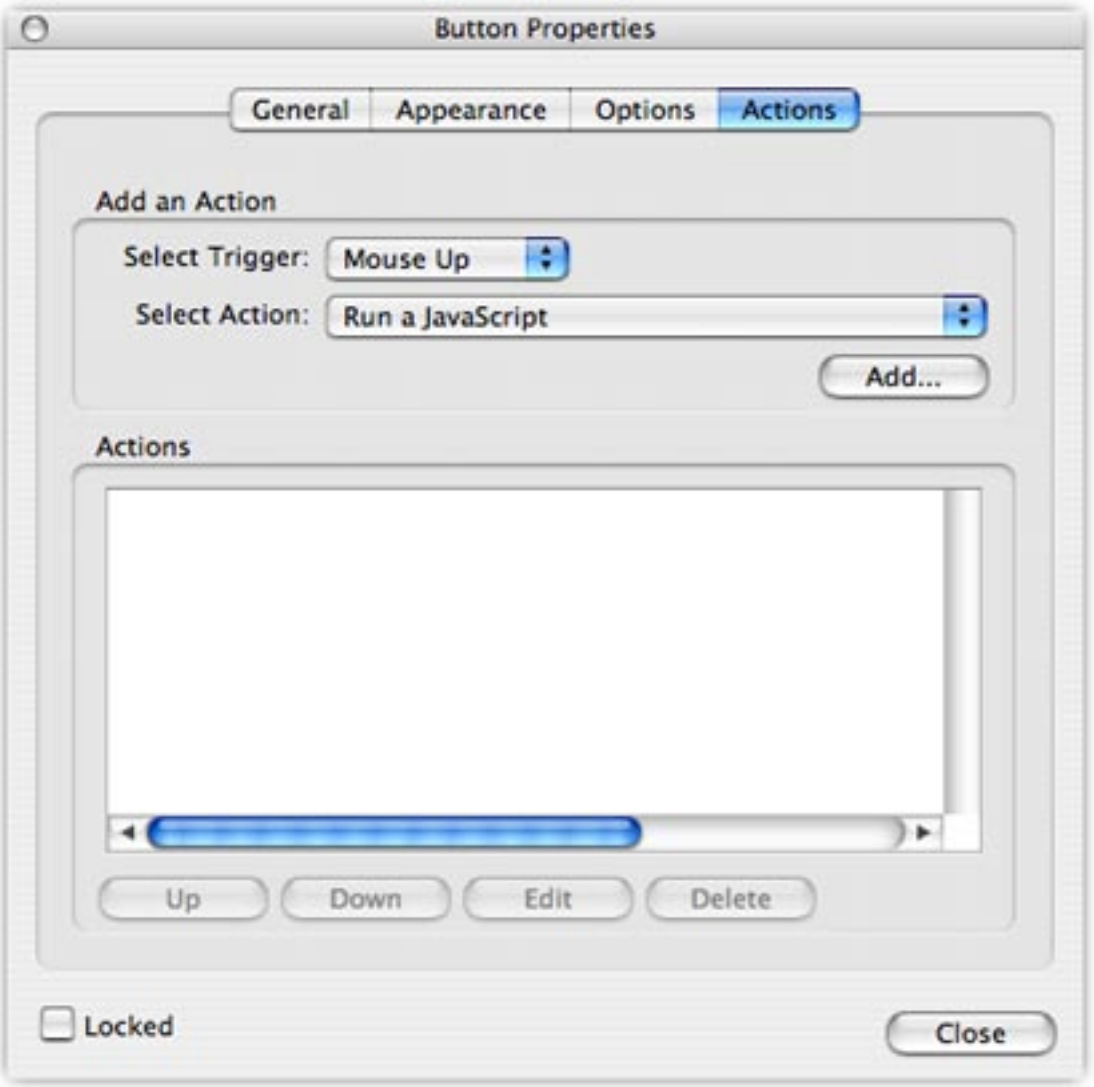
Please send any comments, questions, or problems to:

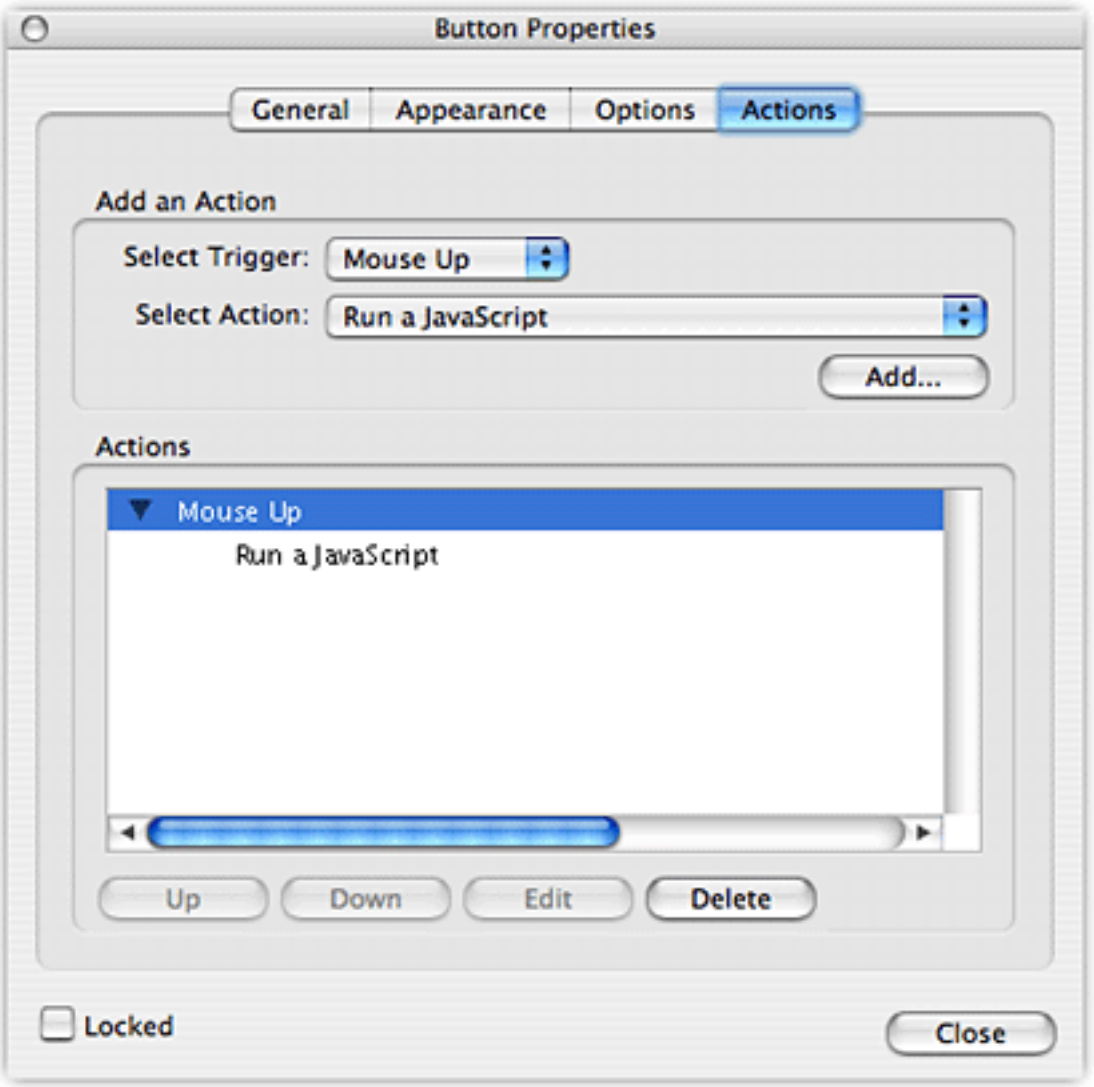
journal@acumentraining.com

Return to Menu


Employee Release Fo...
Rub a Ducky
A Bird-Petting Zoo
Employee Release Form
I agree to hold RaDCo blameless for anything that may happen to me while tending the avifauna. I further agree to not tell the press about the 10-pound drumsticks that appeared last year in the "Emu World" cafeteria. I also agree that there was nothing at all mysterious about the source of the chicken wings served at the recent Trustees' luncheon.
Name Justin B. Caws
Flatten


Employee Release Fo...
Rub a Ducky
A Bird-Petting Zoo
Jeff -
I crossed out one sentence, below; why bring it up?
I do like the new logo, however.
Employee Release
I agree to hold RaDCo b
thing that may happen t
the avifauna. I further ag
press about the 10-pou
appeared last year in the "Emu World" cafeteria. I also agree that there was nothing at all mysterious about the source of the chicken wings served at the recent Trustees' luncheon.
Name Justin B. Caws
Flatten


Button Properties
General
Appearance
Options
Actions
Add an Action
Select Trigger:
Mouse Up
Select Action:
Run a JavaScript
Add...
Actions
Up
Down
Edit
Delete
Locked
Close


Button Properties
General
Appearance
Options
Actions
Add an Action
Select Trigger:
Mouse Up
Select Action:
Run a JavaScript
Add...
Actions
Mouse Up
Run a JavaScript
Up
Down
Edit
Delete
Locked
Close

I'm not going to step through the definition of this procedure here; the comments should help you remember how this works. See the November 2003 *Journal* article for full details.

```
/FindName          % obj => /name true -or- false
{
     /dictCount countdictstack def    % How many items on the dict stk?
     /dArray dictCount array def      % Make an array that big
     dArray dictstack pop             % Load array with dict stk contents
     {                                % Begin "stopped" proc
          dictCount 1 sub -1 0        % for loop, counting backwards
          {    dArray exch get        % Get dictionary from dArray
               {    2 index eq        % forall: look for value match
                    { stop }{ pop } ifelse % Found: execute stop
               } forall
          } for
     } stopped                        % stack: val false -or- val /key true
     dup
     { 3 -1 roll} { exch } ifelse     % Bring orig. object to top
     pop                              % Toss the original target value
} bind def
```

Click the "minus" sign at right to return to the article.