Table of Contents

Glad to be back!

I'm pleased to be writing the *Acumen Journal* again. I've had to skip the past two issues while I was working on the *PDF File Content and Structure* class.

The class is done. The *Journal's* back!

Creating Your Own Rubber Stamp Annotations

One of the more fun features of Acrobats 4 through 6 is the "Stamp" annotation. These annotations are cousins of the sticky notes you could place on an Acrobat page, but are represented on the page by pictures; they are the electronic equivalent of the rubber stamps you can apply to paper documents. In Acrobat 4 and 5, the Adobe supplied a set of imaginative rubber stamps you could use right out of the box, like those at right. (These are screen shots, by the way; don't bother clicking on them.)

Acrobat 6 shows Adobe's recent focus on the corporate, rather than the designer, market. The supplied stamps are all very functional and useful, but not particularly fun, like that at left.

Making up for this, Acrobat 6 has made it very easy to create your own stamps to the stamp annotation tool. You could make custom stamps in Acrobat 5, also, but the process was amazingly tedious. (Check out the June 2001 issue of the *Journal* to see how this was done; you can get it from the Acumen Training Acumen Journal page.)

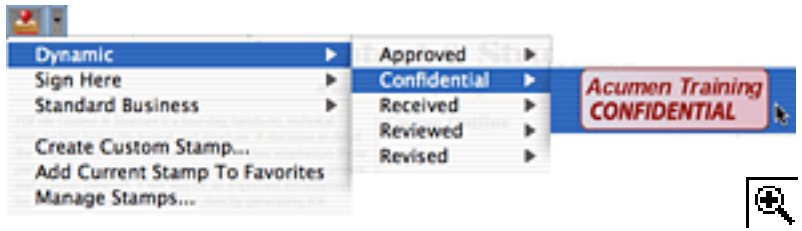
This month, we shall see how to add our own stamp to those supplied by Adobe in Acrobat 6.

Next Page ->>

Stamp Annotations in Acrobat 6

There are several ways of applying comments to a page in Acrobat 6. I usually use the Rubber Stamp tool in the *Commenting Toolbar.*

Click and hold on this tool to select from a set of hierarchically organized "rubber stamp" comments. (You can get to this same submenu by going to *Tools>Commenting>Stamp Tool* in the menu bar.)



The Adobe-supplied stamps are organized into three categories: *Dynamic, Sign Here,* and *Standard Business.* These contain a variety of very useful stamps which I will let you explore. The *Approved* category is rather interesting: these stamps incorporate information taken from Acrobat's *Identity* preferences, dynamically inserting your name, the date, etc.

When you select a rubber stamp from one of the category submenus, the Acrobat cursor turns into a little rubber stamp (as at left). Click on the Acrobat page (or click-and-drag to specify a size) and the stamp you selected will appear on the page.

We are going to add a new category ("Personal Stamps") and a new stamp ("Nyah-Nyah," at right) to this menu.

Next Page ->>

Making a New Rubber Stamp

Step 1: Create Your Artwork

The first step in creating your own rubber stamp annotation is to create the artwork that will represent your annotation on the page. The artwork must ultimately be in PDF format, so you can use any design application you wish—Illustrator, Photoshop, etc.—that exports to PDF.

Simply create your artwork and then export to PDF in whatever manner is used by your graphics application. This may involve saving directly to PDF in the application; it may entail printing to a PostScript file and then Distilling that file to PDF.

The page size of your PDF file doesn't matter, incidentally; the size of the annotation will be determined by only the artwork in the PDF file, not the size of the PDF page.

In my case, I made my "Nyah-Nyah" artwork in Adobe Illustrator 10, which does a nice job of exporting directly to PDF.

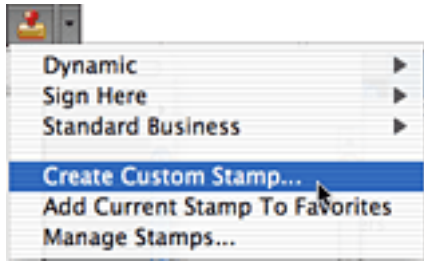
Next Page ->

2. Select *Create Custom Stamp*

Having saved your annotation artwork as a PDF file, you can now import that PDF art as a rubber stamp. Simply open Adobe Acrobat (you must have Professional version of Acrobat) and do the following:

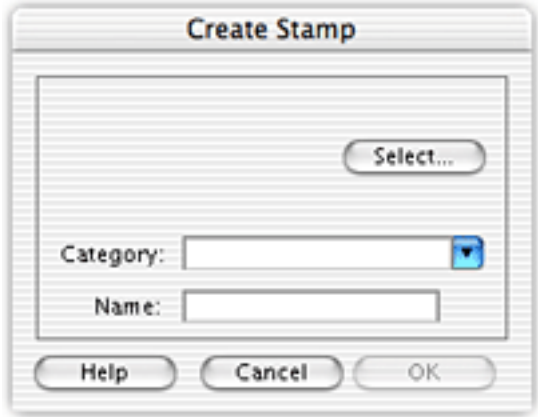
- Make the Commenting Toolbar visible, if necessary.

- Click and hold on the stamp tool (or click on the small, downward-pointing arrow next to the tool) and select *Create Custom Stamp* from the resulting drop-down menu.



Acrobat will present you with the *Create Stamp* dialog box, at right.

- Click on the *Select* button to select your PDF file.



Next Page ->

3. Select Your Artwork

When you click the *Select* button in the *Create Stamp* dialog box, Acrobat presents you with the *Select* dialog box, at right.

In the *File* field, type the complete UNIX-style pathname to the PDF file that contains your artwork.

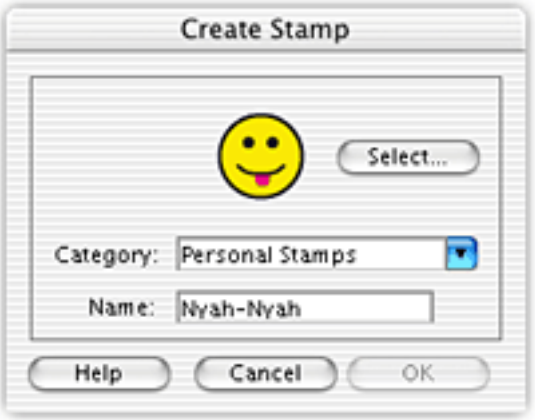
Just kidding.

Actually, just click the *Browse* button and select the PDF file in the resulting "Pick a file" dialog box.

The dialog box's *Sample* box will present you with a scrolling list of all the pages in the PDF file. Scroll to the page containing your artwork, then click the *OK* button.

4. Specify a Category and Name

The *Create Stamp* dialog box now displays your artwork. Select a category from the drop-down menu (or type in the name of a new category) and type in a name for your new annotation.



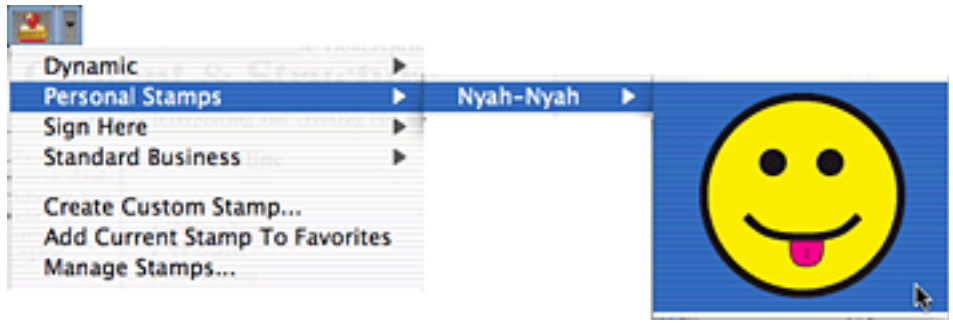
Note that you are not allowed to add a custom annotation to the Adobe-supplied default categories.

Next Page ->

5. Close it up. You're done

Click the *OK* button in the *Create Stamp* dialog box.

Your artwork is now a rubber stamp in good standing in your copy of Adobe Acrobat. Both your new category and your new rubber stamp will appear in the Stamp Tool's drop-down menu.



Final Notes

Adobe has made it *much* easier to create your own rubber stamps. I have little to add to what we've discussed here. Two comments only:

Comment Window

Your hand-built rubber stamp comment has a pop-up window associated with it, so you may send along a text comment, not just a pretty picture. Just double-click on the annotation (try it at right!) and Acrobat will display a pop-up window containing the text.

Portability

The annotation artwork is embedded in each PDF file in which you use the rubber stamp, so the annotation will look correct on any machine that opens the PDF file. However, this means you shouldn't use large images or other bulky artwork for your stamp, since these will significantly increase the size of the PDF file.

Return to Main Menu

A Name Look-up Procedure

Here's a mild challenge that came up a while back: I had need of a procedure that would look up the name of the object on top of the operand stack. I ended up writing a procedure, *FindName,* that takes a PostScript object as its argument and searches the dictionary stack for a key-value pair that has that object as its value. *FindName* returns either a name and a boolean *true* (if the object is found on the dictionary stack) or only a boolean *false* (if the object is not found).

```
/FindName      % obj   =>   /name true   -or-   false
{ ... } bind def
```

The definition of this procedure entailed an uncommon use of the *stopped* operator.

Let's see how it works.

Next Page ->

The Procedure

Here's the definition of *FindName:*

```
/FindName {                  % obj => /name true -or- false
     /dictCount countdictstack def      % How many dictionaries?
     /dArray dictCount array def        % An array to hold the dicts
     dArray dictstack pop               % Get the dict stack's dicts

     {                                  % Begin "stopped" proc
          dictCount 1 sub -1 0          % Begin "for"
          {    dArray exch get          % Begin "forall" with dict # i
               {    2 index eq          % => obj key bool
                    { stop }            % Leave if object found
                    { pop }             % Else, discard name
                    ifelse
               } forall                 % Go to the next key-value pair
          } for                         % Go to next dictionary in dArray
     } stopped                % =>  obj false  -or-  obj /key true
     dup { 3 -1 roll}{ exch } ifelse    % Bring the obj to the top o' stack
     pop                                % Discard the original object
} bind def
```

Get the file

This PostScript code is available on the Acumen Training Resources page. Look for the file *FindName.ps.*

Try it out…

```
/x 12 def
12 FindName not {(Not Found)} if = % Print obj name or "Not Found"
/findfont load FindName not {(Not Found)} if =
(Testing) FindName not {(Not Found)} if =
```

Next Page ->

Acumen Journal: PostScript Tech 9

Step by Step Let's look at this in detail.

Get the Dictionary stack

```
/FindName {
```

Our procedure starts with a series of lines that copy the dictionaries on the dictionary stack into an array. This will allow us to conveniently step through each dictionary, looking for our target object.

```
/dictCount countdictstack def
```

The first line counts the number of dictionaries on the dict stack and stores the number as the variable *dictCount.*

```
/dArray dictCount array def
```

We then create an array, *dArray,* with room to hold *dictCount* objects.

```
dArray dictstack pop
```

Finally, we copy the contents of the dictionary stack into *dArray.* We do this using the PostScript *dictstack* operator.

[array] *dictstack* => [$dict_0$ $dict_1$ ... $dict_n$]

This operator takes an array argument and copies all the dictionaries on the dictionary stack into that array. It returns the same array, now full of dictionaries. (Note in our line of code that I didn't have immediate need of this return value, so I discarded it.)

Next Page ->

Start a "stopped" procedure

```
{
...
} stopped
```

we will execute the loop that searches for the target object inside a *stopped* procedure.

You may remember *stopped* from your PostScript class (you *did* take a PostScript class, didn't you?): it takes a procedure body from the stack and executes it. When the procedure returns, *stopped* returns a boolean value that will be *true* if the interpreter encountered the *stop* operator when executing the procedure body. The boolean will be *false* if the procedure reaches the end without a *stop.*

stopped Arguments

Although we are using *stopped* with a procedure body, the operator can actually take any executable object, including executable strings and (very commonly) file objects.

The *stop* operator, for its part, causes execution to immediately drop out of the procedure executed by *stopped* and provokes the latter operator to return a *true.*

Usually, *stopped* is used in trapping errors, since the PostScript error handler executes *stop* when a PostScript error takes place. Thus, a typical use would be:

```
{ ... questionable code ... } stopped { errorstuff } if
```

The *stopped* operator executes the questionable code and returns a *true* if there was a PostScript error, *false* otherwise. We can then report the error with our *errorstuff* procedure.

In our case, we shall search the dictionary stack for our target object inside a *stopped* procedure. If we find the object, we shall execute *stop.* Not only will this drop us out of our search, but also *stopped* will place a convenient boolean on the stack that will be *true* if we found the object and *false* otherwise.

Next Page ->

Acumen Journal: PostScript Tech 11

Step through the dict stack

```
dictCount 1 sub -1 0
{ ... } for
```

We now start a *for* loop that steps through the contents of our array of dictionaries, examining each dictionary in *dArray* in turn.

Note that the loop steps through the array from the back to the front. I want to search for our target starting with the dictionary at the top of the dictionary stack, which is the last item in the array.

Step through the key-value pairs

```
dArray exch get
{ ... } forall
```

Remember how *for* works: for each value of the counter, *for* puts the counter value on the stack and then executes the procedure body. In our case, the procedure pushes *dArray* onto the operand stack, reverses the array and the loop counter, and then executes *get,* fetching a dictionary from the array.

We then push another procedure body onto the stack and execute *forall.* The *forall* procedure steps through each key-value pair in the dictionary, pushing the key and then the value onto the stack, and then executing the procedure.

Our procedure must examine the value, on top of the stack, to see if it matches our target. If so, we shall execute *stop.*

Next Page ->

Examine a key-value pair

```
2 index eq
```

When executed, the procedure handed to *forall* will find two objects on top of the stack: a key (below) and its associated value (on top). In addition, the target object for which we search is still at the bottom of the stack; we shall need to preserve the target object in the design of our loop procedure.

Thus, at the beginning of each execution of our *forall* procedure, the stack looks like this (the bottom of the stack is to the left, as usual):

```
tgt  /key  value
```

We want to compare the value to our target, so we do a *2 index,* which brings a copy of the target to the top of the stack:

```
tgt  /key  value  tgt
```

We then execute *eq,* the "equal-to" operator. This removes the target and the key-value pair's value, compares them, and returns a boolean value which will be *true* if the two were equal.

```
tgt  /key  bool
```

Next Page ->>

Does the key match our target?

```
{ stop }{ pop } ifelse
```

If the boolean returned by *eq* is *true,* we have found the object and our search is finished; we execute *stop,* which causes execution to drop out of the *stopped* procedure. The *stopped* operator will push a boolean *true* onto the stack, which now looks like this:

```
tgt   /key   true
```

On the other hand, if the key-value pair's value did not match our target, we shall need to go on to the next pair. We discard the current key (with a *pop)* and continue with the loop.

```
} forall
```

Thus endeth the *forall* loop. The loop examines each key-value pair in the current dictionary until we find one with our target value (causing us to execute *stop)* or until we run out of key-value pairs.

```
} for
```

Here's the end of our *for* loop; this loop will have stepped us through each dictionary on the dictionary stack (reading them from our *dArray* array).

Next Page ->

Exit stopped `} stopped`

Our *stopped* will conveniently return *true* if we found a match (because we executed *stop* in that case) or *false* otherwise. The operand stack, upon returning from *stopped* will have one of two possible sets of contents:

- If we found a match:

  ```
  tgt /key true
  ```

  Beneath the boolean, we shall have the key associated with our target object. We shall also still have the copy of our target that we were so careful to preserve in our nested loops.

- If no match was found:

  ```
  tgt false
  ```

  In this case, beneath the boolean will be only the copy of our target.

Next Page ->>

Discard the Target

We are very nearly done. We need to discard the copy of the target, since we are finished using it; we also shall preserve the boolean returned by *stopped* because it exactly matches the boolean return value we declared for our *FindName* procedure (*true* if found, *false* if not).

```
dup { 3 -1 roll} { exch } ifelse
```

We first do a *dup* to preserve the *stopped* operator's boolean return value. Our operand stack now looks like one of the following cases:

- Name found: `tgt /key true true`
- Name not found: `tgt false false`

To discard the original target, we bring it to the top of the stack by doing a *roll* if the boolean is *true* or a simple *exch* if the boolean is *false.* (Remember that *ifelse* consumes the topmost boolean.)

```
pop
```

Finally, we do a *pop,* removing the target object from the stack and leaving only *FindName's* return values: a boolean and perhaps a name.

Next Page ->

Done!

```
/x 12 def

12 FindName not {(Not Found)} if =
/findfont load FindName not {(Not Found)} if =
(Testing) FindName not {(Not Found)} if =
```

That's it. To use our new *FindName* procedure, we shall see if it correctly recovers three test items: the name of a variable, *x,* that we define ourselves; the procedure body that implements *findfont;* a string that is not actually defined anywhere in the dictionary stack.

In each case, we execute *FindName* and print either the name of the object or the string "Not Found," depending on the results.

The output, sent to *stdout,* looks like this:

```
x
findfont
Not Found
```

Comment

The reason I trotted out this example was to demonstrate an alternative use of *stopped.* As I said earlier, we mostly use *stopped* to trap errors. Here we used the operator to drop us out of nested loops. As a side effect, we used *stopped's* boolean return value to conveniently determine whether our search was successful.

Next Page ->

Acumen Journal: PostScript Tech 17

Extra Credit for Next Month

FindName searches only the dictionaries immediately on the dictionary stack. It would be useful if it could also recursively search any dictionaries it encounters in the course of its examination. Thus, if *x* is defined inside *myDict,* itself defined in *userdict,* our search would still correctly find the name:

```
/myDict <<
    /x 12
>> def

12 FindName    % Should correctly return "x"
```

If you find yourself with a great bundle of spare time, you might want to rewrite *FindName* so that it does this.

We'll look at my solution in next month's Journal.

Return to Main Menu

Schedule of Classes, Nov 2003 – Jan 2004

Following are the dates and locations 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. The Acrobat class schedule is on the next page.

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

Technical Classes

New!	PDF File Content and Structure	Dec 8–11		Feb 23–26
	PostScript Foundations		Jan 12–16	
	Variable Data PostScript	Dec 15-19		
	Advanced PostScript			Feb 9–13
	PostScript for Support Engineers		Jan 26–30	
	Jaws Development		*On-site only*	

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

Registration Info →
Acrobat Classes →

Acrobat Class Schedule

These classes are taught quarterly 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. *Troubleshooting With PitStop* (full day) is $340.00. In all cases, 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: 25142 Danalaurel, 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: 25142 Danalaurel, Dana Point, CA 92629

Back issues

Back 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 and Structure launched!

The new *PDF File Content and Structure* course has been unleashed and is doing well. On-site classes have been taught at **Lexmark** and **Océ-Netherlands** and an on-site class is scheduled for mid-November at **Adobe** in San José. Seems to be popular, I'm pleased to say!

You can see the final topic list for the class at the Acumen Training website: www.acumentraining.com/Descr_TechPDF.html. This list of topics will probably change somewhat as the class continues to mature. (For that matter, even the long-standing PostScript Foundations class is still changing, though slowly.)

Acumen Editor™

I shall be putting the specialized text editor used in the class on the Acumen Journal Resources page. This is a fairly minimal-functional text editor that allow you to write hand-composed PDF files. Though its text-editing functions are not too extensive yet, it will automatically calculate and insert the lengths of streams and create the *xref* table required by a PDF file.

It's not too robust, yet, but it will improve over time.

Acumen Editor requires either Mac OS X 10.2 or later or Windows 98 or later.

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? Does it make you contemplate the futility of life?

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

Selecting Rubber Stamps