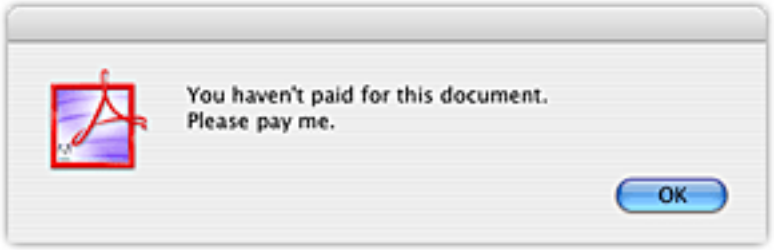
Table of Contents

JavaScript: Creating a Nagware Document, Part 1

I have always been a supporter of the shareware concept. Create your product, post it on-line, encourage people who like it to pay some reasonable—usually small—amount of money for it. I always pay my shareware donation for any piece of software that I find even occasionally useful.

In an effort to encourage us users to pay our shareware fees, many shareware programs will periodically display a dialog box reminding you that you haven't paid; this dialog box stops appearing once you register the software and type in your serial number. This category of shareware is often referred to as "nagware," since it continues to nag you until you finally cough up the fee.

Much the same principle can apply to documents. It is reasonable to distribute your book electronically, let people read it, and ask them, if they liked it, to send you money.

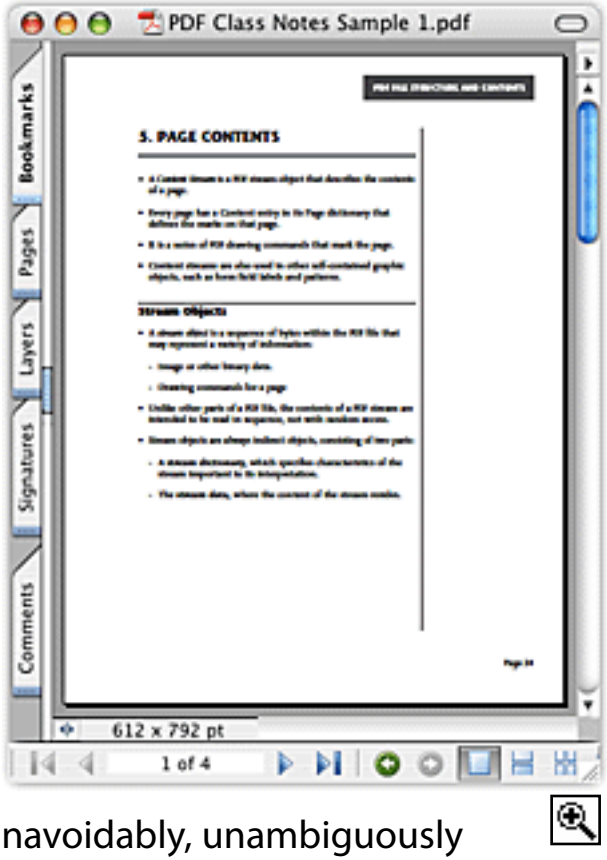


In this issue and the next, we shall see how to implement a "nagware" document, a PDF file that periodically reminds the reader to pay the author. So that we can see several approaches to this task, we are going to make a PDF file that becomes progressively more strident in demanding funds and that eventually renders itself unreadable.

Next Page ->>

The Project

Over the course of these two issues, we are going to turn an existing PDF file into a nagware file. If you want to follow along, you may use any PDF file you wish, although the Acumen Training Resources page has appropriate before-during-and-after files. (Look for the file "Nag_1.zip.")



This file, pictured at right, will open without complaint the first two times. On the third opening, it will display the dialog box on the previous page, asking for money. Each time the file is opened thereafter, it will present an increasingly obtrusive demand; eventually, the file will blank out the pages so that the user can no longer read the document.

We will finish by providing a "registration" button that lets the User enter a serial number and that will turn off the nagging.

Limitations

The techniques we describe in these articles do not provide complete document security; a determined, sophisticated user could circumvent them. The intent here is that of most shareware: to make it unavoidably, unambiguously clear that, a User who likes the article and finds it useful should pay for it.

Next Page ->

Acumen Journal: Acrobat User 3

Reading Assignments

As is true of all the *Journal's* JavaScript articles, I assume you have some basic experience with Acrobat JavaScript, equivalent to having read my book *Extending Acrobat Forms With JavaScript*. In particular, this article depends upon the following topics:

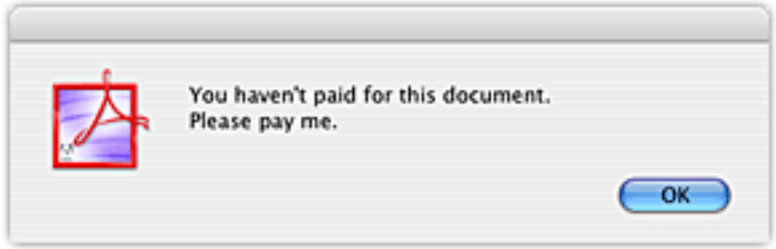
- *The Acrobat global object* – Actually, I did not discuss this in my book. However, it was the topic of the *August 2004 Journal* article, which you can get from the Acumen Training website.
- *Document JavaScripts* – The same August 2004 *Journal* reviews document JavaScripts, as well.
- *The JavaScript switch statement* – I'll be presenting a reminder in this article, but this is a mildly complicated operator that really does need a more in-depth discussion. *Extending Acrobat Forms…* covers it thoroughly, but any book on JavaScript should describe it reasonably well.

You may also want to get a copy of the *Acrobat JavaScript Object Reference (AJOR)* for reading afterward. This is available for the downloading from Adobe's website.

Next Page ->

Overview of the Process

Conceptual



Our approach to this task will be to keep track of the number of times our document has been opened. If the document has been opened only a couple of times, then we'll do nothing; let the User read unannoyed. The third time Users open the file, we'll present them with the dialog box asking for money; the fourth time, we'll add a message to the page; and so forth.

Each time a User opens the document he or she will get an additional nag of some form or other. Eventually, we'll just white out the page, so that they can no longer read the document.

An important design goal is that they should not be able to reset the nagging by just closing the document without saving it or downloading a fresh copy of the PDF file. The mechanism that decides whether they can see the file, and with what level of annoyance, should somehow be associated with the computer.

Secure Your Document

A perhaps obvious point is that the final Acrobat file you distribute should be password protected so that the user can't examine, change, or delete your JavaScripts.

The sample files have not been protected, because they are primarily teaching tools.

The January 2003 *Acumen Journal* describes how to apply password protection to a PDF file. (It's a bit dated, but most of what it describes has changed only cosmetically.)

Next Page ->>

JavaScript Overview

We are going to implement this nagging as a Document JavaScript that will execute each time the file is opened.

The first time our script runs, it is going create a new property of the Acrobat JavaScript *global* object. We shall use this property, which we'll call *nagCount,* to keep track of how many times the document has been opened. (Again, see the August 2004 *Journal* for a discussion of the *global* object and a review of document JavaScripts.)

Our script will do the following:

Create or increment global.nagCount

1. Check to see if the property *global.nagCount* exists.
2. If the property does not exist, create it with a value of *1*.
3. If the property does already exist, add *1* to its value.

Nag, as needed

4. If *global.nagCount* is *1* or *2*, do nothing.
5. If *global.nagCount* is 3 or more, display the "Pay me" dialog box.
6. If *global.nagCount* is *4*, add the "Pay me" message to the page.
7. etc.

Let's look at the code.

Next Page ->

Acumen Journal: Acrobat User 6

Code, First Pass

Our first version of our document JavaScript will carry out only steps 1 through 5 on the previous page: we'll create or increment *global.nagCount* and then either do nothing or display the nag dialog box, depending upon whether the document has been opened once, twice, or three times. You can see this script operating in the file *Nag 1.pdf,* among the sample files.

Resetting the Nags

Each of the sample files has a *Reset nagCount* button at the top. Clicking this button resets the value of *global.nagCount* to zero.

You should click this each time you finish using one of the example documents and are going to move on to the next. Otherwise, since all the example files use the same global property, it will be harder to see how the nagging progresses on successive openings.

```
// If global.nagCount doesn't exist,
// then create it.
if (global.nagCount == null)     {
     global.nagCount = 1
     global.setPersistent("nagCount", true)
}
else     {          // Otherwise: increment it
     global.nagCount++
}

if (global.nagCount > 2)     {    // Is nagCount greater than 2?
                                  // Yes: display alert
   app.alert("You haven't paid for this document.\n Pay me.")
}
```

Next Page ->

Step by Step

Create nagCount, if nec.

```
if (global.nagCount == null)     {    // Does nagCount exist?
     global.nagCount = 1                // No: create it
     global.setPersistent("nagCount", true)
}
```

The first time the User opens this PDF file, *nagCount* won't exist—that is, it will be *null*—and so we must create it, giving it a value of *1* (indicating this is the first time we have opened the file). This is exactly what our *if* statement does:

- Check to see if *global.nagCount* is null.
- If so, do the following:
  - Set *nagCount* to *1*. This creates the new global property.
  - Make it *persistent,* so that it will survive from one opening of the document to the next. (We don't want *nagCount* to disappear when the user closes the file.)

Otherwise, increment it

```
else     {
     global.nagCount++
}
```

If *global.nagCount* does exist (that is, our *if* statement returned *false)*, then we increment its value with the ++ operator. The *nagCount* property now reflects the number of times the document has been opened.

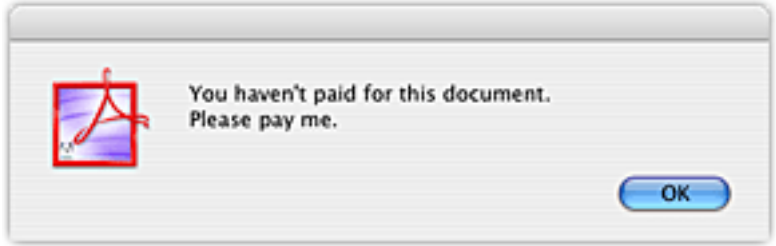
Next Page ->

Acumen Journal: Acrobat User 8

Nag, if appropriate

```
if (global.nagCount > 2)    {
   app.alert("You haven't paid for this document.\nPay me.")
}
```

If *global.nagCount* has a value greater than *2,* indicating the PDF file has been opened three times or more, then we display the alert box, asking for money.



Thus we implement our first nag.

A Second Nag

If the User opens our document a fourth time, we are going to add a message at the bottom of the first page, as at right.

We are going to do this by adding an annotation—a "comment," in Acrobat 6 terminology—to the bottom of the page. Our JavaScript will do this with a call to the Acrobat Doc object's *AddAnnot* method.

Next Page ->>

Doc.addAnnot

```
this.addAnnot({
     page: 0,                          // Add to first page
     type: "FreeText",                 // Use this annotation type
     rect: [ 206, 24, 406, 48 ], // Position on the page
     contents: "Slacker! I need my money!",  // Comment text
     ...                               // Other properties, as needed
})
```

The Doc object's *addAnnot* method creates a new annotation somewhere in the document. You must specify the type of annotation you want to create, the page on which the annotation resides, the position on the page, and the text that makes up the comment. You may also dictate a variety of properties for the resulting annotation, including color, font, text size, etc. These are all properties of the *annotation object* created by the *addAnnot* method.

In the example above, we are making use of a JavaScript syntax that allows you to specify named arguments for an method call. Generically, the syntax looks like this:

```
obj.methodName( {
     argumentName: value,
     argumentName: value,
     ...
} )
```

ArgumentName:value pairs must be separated by commas and their order is unimportant.

Next Page ->

Our Second Nag Code

For our second nag, we shall check to see if the document has been opened four times; If so, we'll use the *addAnnot* method to add a nag message to the first page of the document. The code below—and all our future code—omits the initial *if…else* block that creates or increments *nagCount*.

```
if (global.nagCount > 2)    {   // nagCount greater than 2?
   app.alert("You haven't paid for this document.\n Pay me.")
}
if (global.nagCount == 4)    {      // Fourth time opened?
    this.addAnnot(    {             // Yes: add annotation
        page: 0,                    // Put on first page
        type: "FreeText",           // Text on page
        rect: [ 206, 24, 406, 48 ], // Location on the page
        contents: "Slacker! I need my money!",  // The text
        fillColor: color.red,       // Background color
        width: 0,                   // No border
        textSize: 10,               // Text point size
        readOnly: true              // No user changes
    })
}
```

Note that we are calling the *addAnnot* method in the *this* object, which in this context is referring to the current document.

Next Page ->

Acumen Journal: Acrobat User 11

Annotation Properties

In our code, we set the following annotation properties:

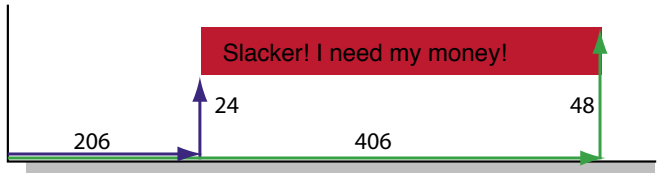
page: 0,

The *page* property specifies the page on which the annotation resides. Remember that within a JavaScript, pages are numbered starting with zero. Our annotation will be placed on page *0,* that is, the first page of the document.

type: "FreeText",

The *type* property indicates what kind of annotation we want to create. Acrobat JavaScript defines a set of standard names for the various annotation types ("Line", "Text", etc.). We are placing a "FreeText" annotation on the page, like the one at left; this is a text note that sits directly on the page, rather than within its own frame.

rect: [206, 24, 406, 48],



This identifies where the annotation should be located on the page. The four numbers are the *x,y* coordinates of the lower left and upper right corners of the annotation. These are measured in points ($^1/_{72}$ inch) from the lower left corner of the page.

contents: "Slacker! I need my money!",

This is the text that should appear in the annotation.

Next Page ->

Acumen Journal: Acrobat User 12

Other Properties

Annotation objects have several other properties that presumably can be set by *doc.addAnnot*. These are documented in the *Acrobat JavaScript Object Reference.*

fillColor: color.red,

This is the background color we want for the annotation. The value for this can be any valid Acrobat JavaScript color specification (see *Extending Acrobat Forms* or the *AJOR)*. In this case, I am using one of the predefined colors that reside as properties in the *color* class. Other predefined colors include *color.blue, color. magenta,* and *color.transparent*.

Had we wished (I'm not doing so here), we could have also defined a *strokeColor* property specifying the color of the annotation's border.

width: 0, This is the width of the border. Setting this to *zero* indicates we don't want a border for our annotation.

textSize: 10, Here we are specifying the point size we want for the annotation's text. We could also have specified the *textFont* property, setting it equal to the PostScript name of the font the annotation should use.

```
textFont: "Optima-Oblique",
```

readOnly: true

Finally and importantly, we set the annotation's *readOnly* property to *true,* preventing the User from selecting, moving, or otherwise changing the annotation.

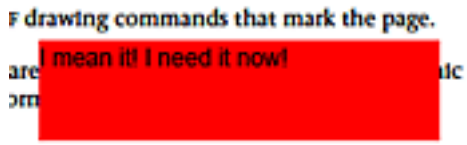
Notice that this property specification does not end with a comma, because it is the final property we are specifying.

Next Page ->

Third Nag

The fifth time the User opens the document, we shall display an annotation in a more prominent location on the page, so they can't ignore it. Our script now looks like this (abbreviating the parts we've already seen):

```
if (global.nagCount == 4)     {       // Fourth time opened?
        this.addAnnot(     {          // Yes: add annotation
            ...
        })
}
else if (global.nagCount == 5)  {     // Fifth time opened?
        this.addAnnot({                // Yes: add annotation
            page: 0,
            type: "FreeText",
            rect: [ 206, 500, 406, 550 ],
            contents: "I mean it! I need it now!",
            fillColor: color.red,
            textSize: 30,
            width: 0,
            readOnly: true
        })
}
```



if *global.nagCount* is not *4,* then the *else* clause checks to see if it is instead *5.* If so, we add an annotation in the middle of the page. Pretty straightforward.

Next Page ->

Acumen Journal: Acrobat User 14

The *Switch* Statement

Too many if…else if's

In our script we used an *if…else if* block to decide what type of nag to use:

```
if (global.nagCount == 4)     {      // Fourth time opened?
    ...
}
else if (global.nagCount == 5)  {    // Fifth time opened?
    ...
}
```

This works perfectly well for the two instances we have so far, but will quickly become unwieldy as we add more *nagCount* values to accommodate:

```
if (global.nagCount == 4)     {      // Fourth time opened?
}
else if (global.nagCount == 5)  {    // Fifth time opened?
}
else if (global.nagCount == 6)  {    // Sixth time opened?
}
else if (global.nagCount == 7)  {    // Seventh time opened?
}
```

JavaScript provides a somewhat more readable alternative to such a string of *if…else if* blocks: the *switch* statement.

Next Page ->

Acumen Journal: Acrobat User 15

switch...case

We discuss the *switch* statement in *Extending Acrobat Forms…*, but as a reminder:

The JavaScript *switch* statement allows you to specify a variable or property and a series of *case* statements, one for each possible value for that variable. Associated with each case is a set of JavaScript statements that should be executed if the variable has the corresponding value. Each of these blocks of JavaScript is terminated with a *break* command.

Thus, our earlier cascaded *if…else if* blocks now become:

```
switch (global.nagCount)     {    // Look at global.nagCount
    case 4:                         // Is it 4?
        …JS statements…             // If so, do this
        break
    case 5:                         // Is it 5?
        …JS statements…             // If so, do this
        break
    case 6:                         // Is it 6?
        …JS statements…             // If so, do this
        break
}
```

Overall, the *switch…case* block is much more readable than the set of *if…else if* statements.

Next Page ->

Nagging With Switch

Our current level of nagging, carried out using *switch*, becomes:

```
if (global.nagCount > 2)          // Still present the dialog box
   app.alert("You haven't paid for this document.\nPay me.")

switch (global.nagCount)     {    // Examine global.nagCount
    case 4:                       // If it's 4, do the following:
        this.addAnnot({
            page: 0,
            ...
        })
        break

    case 5:                       // If it's 5, do the following:
        this.addAnnot({
            page: 0,
            ...
        })
        break
}
```

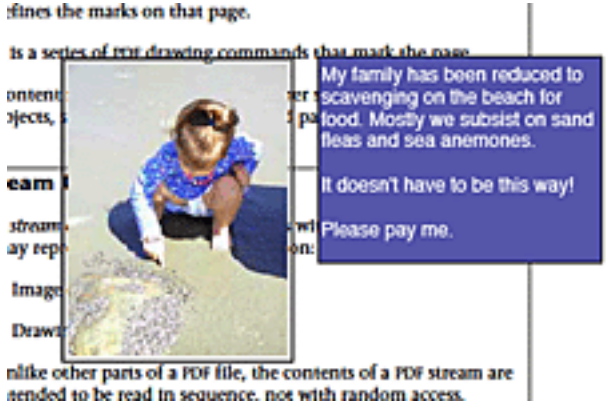
Now, to add new levels of nag to our file, we need simply add more *case* statements to our *switch* block. The current version of our JavaScript is in the sample file *Nag 3a.pdf*. I've also included it as a separate JavaScript file, *Nag 3a.js*.

Next Page ->

Out of Time

There are still techniques to discuss, but, alas, we are out of time for this month.

Next issue we'll finish up by doing the following:



- Add nag 4, which will display a pitiable picture of a small child searching the beach for food and ask the User if it's right that the author's family should be reduced to such extremes.
- Add a final nag, which will simply white out all of the pages in the document so it becomes unreadable.
- Add a "Register" button, which will let the User enter a serial number—which was presumably emailed to the User upon receipt of the cash—and then will disable all the nags and render the document usable from then on.

But that's all next issue.

See you then.

Return to Main Menu

Colorizing Images With the Separation Color Space

Files on Website

As always, the examples for this month's article are available on the Acumen Training website's Resources page. Look for the file *ColorizingImages.zip*.

I am asked about this task in class fairly often: if you have data for a grayscale image, how can you print the image using some other color? That is, how do you print the image as shades of green, say, rather than shades of gray?

There are several possible approaches to this problem. By far the easiest and most flexible uses the Separation color space.

This month we shall see how to use the Separation color space to carry out this task and a few variations, such as simulating print against a colored background. We'll also look at how to modify color images using the *DeviceN* colorspace.

Next Page ->>

Separation Color Space

The technique we are discussing uses the Separation color space to modify the interpretation of the grayscale image data. As you may recall from your PostScript class, Separation color space allows you to specify color in terms of a single named colorant. Although it was originally intended to support printers with highlight colors, it is routinely used in PostScript output to support spot colors.

You declare that you want to use Separation color space with a call to *setcolorspace,* of course:

```
[ /Separation
  /InkName
  /AltSpace
  { tint transform }
] setcolorspace
```

The four entries in the colorspace array are:

/Separation
: The name "/Separation", indicating the colorspace we want to use.

/InkName
: The name of the ink we want to use to specify color. This can be any arbitrary name and, in particular, does not need to correspond to inks available on the current printer. The colorant name entry may be either a string or a name object.

Next Page ->

Acumen Journal: PostScript Tech 20

If the current device does have an ink with this name, then PostScript will lay that ink onto the page. On the other hand, in the more common case in which there is no equivalent ink available, then the final two entries in the colorspace array provide a fallback strategy.

/AltSpace This specifies an alternative colorspace; if the ink named in the colorspace array is not available on the PostScript device (which is usually the case), then this colorspace will be used instead.

{ tint xform }
This "tint transform" procedure converts color values intended for our spot color into a color specification appropriate to the alternative colorspace.

Having specified our colorspace, the *setcolor* operator will expect a "tint value," a color value intended for the spot color.

```
.75 setcolor
```

If the ink is available, then we will paint the page with the specified tint of that ink; otherwise, *setcolor* will use the tint transform to convert the tint value into the alternative colorspace and we will paint the page with that alternative color.

For Example On the next page is a PostScript program that defines a "Turquoise" colorspace and then attempts to draw three Turquoise rectangles on the page. If Turquoise ink is not available on the PostScript device, then we shall substitute green. (Green doesn't look much like turquoise, I admit, but I'm trying to keep the code simple.) Next Page ->

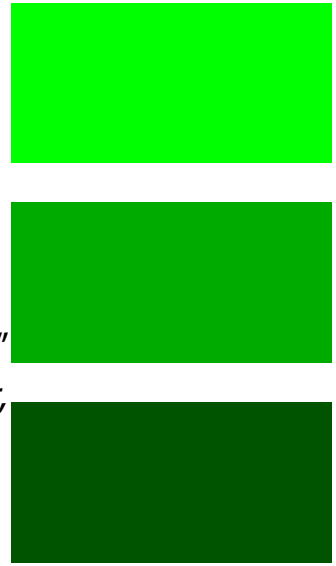
A Turquoise Colorspace

```
[    /Separation
     /Turquoise              % Use a "Turquoise" ink
     /DeviceRGB              % If no Turq available, use RGB instead
     { 0 exch 0 }            % Convert Turq values to RGB
] setcolorspace

1 setcolor                   % Turquoise rectangles, printing as green
100 600 200 100 rectfill

.67 setcolor
100 475 200 100 rectfill

.33 setcolor
100 350 200 100 rectfill
```



Here our call to *setcolorspace* defines a colorant named "Turquoise." Since there is no turquoise ink available, every time we call *setcolor,* that operator pushes onto the stack the turquoise value we asked for and then executes the tint transform,

{ 0 exch 0 }

This pushes a zero onto the stack on top of the turquoise value, exchanges the two numbers (so that red is *0* and green is our turquoise value), and then pushes another zero on the stack to serve as our blue.

Next Page ->

Colorizing Gray Images

Let's apply this to grayscale images.

Here is some typical PostScript image code:

```
100 300 translate
396 293 scale
<<
    /ImageType  1
    /Width      396
    /Height     293
    /BitsPerComponent   8
    /Decode     [ 0 1 ]
    /ImageMatrix [ 396 0 0 -293
0 293 ]
    /DataSource currentfile /ASCIIHexDecode filter
>> image
6869686969686A6A6A6B6A6A6968686A686A61485F716F70717373737
374665...
...38080787E807E8B6B63524457413C3B4447
>                                   % ASCIIHexDecode end-of-data marker
showpage
```

Next Page ->>

For the purpose of this article, the most important characteristic of the *image* operator is that it interprets image data in terms of the current colorspace. Because the default imagespace is *DeviceGray,* the *image* operator will interpret its data as grayscale information unless otherwise informed. If we were printing an RGB image, we would need to precede our call to *image* with `/DeviceRGB setcolorspace`.

Using *Separation*

So let's mess with the *image* operator's mind a little; let's tell it that the incoming image data is turquoise data:

```
[    /Separation      % This is our earlier Turquoise colorspace
     (Turquoise)
     /DeviceRGB
     { 0 exch 0 }
] setcolorspace

396 293 scale
<<   /ImageType  1        % Our call to image is unchanged
     /Width      396
     ...
     /DataSource currentfile /ASCIIHexDecode filter
>> image
6869686969686 86A6A6A6B6A...
```

Next Page ->

Neither call to the *image* operator nor the image data have changed. What we have done is told the *image* operator to treat the image data in terms of our Turquoise colorspace. Each byte is now treated as a tint value that will be converted to green by the colorspace's tint transform.

Mimicking a colored background

It's cute enough for its own sake to print a gray image in shades of green. However, this has a very real application: simulating printing the image on colored paper.

This is a common requirement for people who print directories, such as the Yellow Pages, that are printed on a colored background. These directories were once printed on colored paper; nowadays they are often printed on white paper, each page of the directory being completely tinted with yellow (or whatever) ink.

To maintain the traditional look of a directory printed on colored paper, images must be converted from grayscale to shades of whatever color is used for the directory paper.

For Example

On the next page, we simultate printing our image against a yellow background. The background tint is a 35% yellow.

Next Page ->

Acumen Journal: PostScript Tech 25

```
[    /Separation
     (YandB)    % "Yellow and Black"
     /DeviceCMYK
     { 0 0 .35 4 -1 roll }
] setcolorspace

396 293 scale
<<
     /ImageType  1
     /Width      396
     /Height     293
     /BitsPerComponent   8
     /Decode     [ 1 0 ]
     /ImageMatrix [ 396 0 0 -293 0 293 ]
     /DataSource currentfile /ASCIIHexDecode filter
>> image
6869686969686A6A6A6B6A6A...
```

This time, we are defining a spot color named "YandB" (for "Yellow and Black") that maps into CMYK. To mimic printing a grayscale image against a yellow background, I am using the original gray data for the black value and adding a 35% yellow component to each pixel. That is, the original gray value (call it *g)* becomes a CMYK value of `[ 0 0 .35 g ]`.

Next Page ->>

What About Color Images?

DeviceN ColorSpace

You can do something similar with color images, simulating printing against a colored background, using *Separation* colorspace's younger brother, *DeviceN*.

DeviceN is a LanguageLevel 3 generalization of *Separation,* allowing you to specify color in terms of an arbitrary number of named colorants. The call to *setcolorspace* looks like this:

```
[ /DeviceN
  [ /InkName0 /InkName1 ... ]
  /AltSpace
  { tint transform }
] setcolorspace
```

In this case, the array elements are:

/DeviceN The name of the colorspace.

[/InkName$_0$ ...]

An array of ink names (the "colorant array") that we shall use to specify color. The *setcolor* operator will later expect one argument on the operand stack for each ink named in this array.

Next Page ->

altSpace The alternative colorspace. This is the colorspace that will be used if *any* of the colorants named in the colorant array are missing.

{tint xform} A tint transform procedure that converts a color specification intended for our named colorants into color values appropriate to the alternative colorspace.

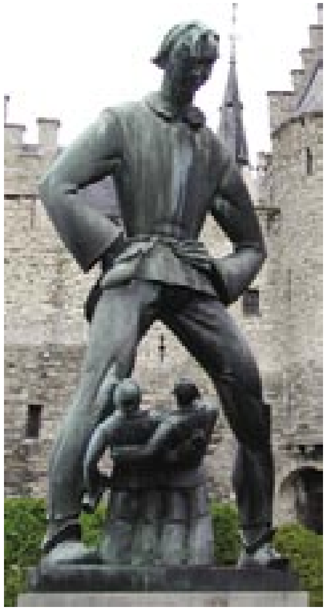
Now we can play the same trick with a CMYK color image that we did earlier with a grayscale image: add ink to the yellow channel.

An Example

Here's our original CMYK image, omitting routine items like the *scale* and *translate:*

```
/DeviceCMYK setcolorspace

<<   /ImageType 1
     /Width 181
     /Height 343
     /BitsPerComponent 8
     /Decode [ 0 1 0 1 0 1 0 1 ]
     /ImageMatrix [ 181 0 0 -343 0 343 ]
     /DataSource currentfile
         /ASCII85Decode filter /LZWDecode filter
>> image
J,g]g3$]7K#D>EP:q1$o*=mro@So+\<\5,H7U....
```



Next Page ->

Colorizing

Here's the code that prints the image against a simulated 35% yellow background:

```
[    /DeviceN                    % Our colorspace name
     [ /c /m /y /k ]             % Stand-in colorant names
     /DeviceCMYK                 % Use CMYK as the alternative
     { exch .35 add exch }       % Add .35 to the yellow value
] setcolorspace

% The call to image is unchanged
<<   /ImageType 1
     /Width 181
     ...
>> image
J,g]g3$]7K#D>EP:q1$o*=mro@So+\<\5,H7U....
```

Granted, the resulting image is no longer as pleasing, aesthetically; I'm being very simple-minded in my simulation. But the result *does* look as though it's been printed against a yellow background.

I'll leave it to you to come up with a better tint transform.

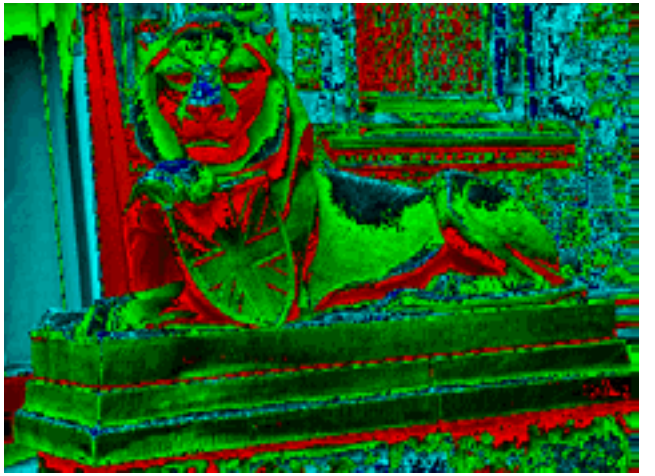
Next Page ->

Acumen Journal: PostScript Tech 29

Separation: Not Just Spot Colors

The moral of this article is that the Separation and DeviceN colorspaces have uses beyond just supporting spot colors and hifi separations. You can use them with images to colorize, apply color correction, strip alpha channels, and a wide variety of other tasks.

Return to Main Menu



Schedule of Classes, Oct 2004 - Jan 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		Nov 8–11	
PostScript Foundations	Oct 11–15		Jan 10–14
Variable Data PostScript			
Advanced PostScript			Jan 24-28
PostScript for Support Engineers			Dec 13–17
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

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?

New PDF Class

Nothing too new this month. I am still in the early stages of laying out the second PDF File Content and Structure class. The topic list is below; as before, if you think something should be added to or dropped from this list, send an email to john@acumentraining.com.

Preliminary Topic List

- Overprinting
- CID Fonts
- Halftones
- Marked Content
- Rendering Intents
- Smooth shading
- Reference XObjects
- Cross reference streams
- BX & EX
- File Spec
- Masked Images
- Digital Signatures
- AcroForm
- Transfer Functions
- Shape dictionaries
- Layers
- Name Dictionaries
- Patterns
- Composite Fonts
- Linearized PDF
- Stroke Adjustment
- Halftones
- Text Knockout
- Object streams
- More on data structures

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 yearn for the simpler, easier days of your youth?

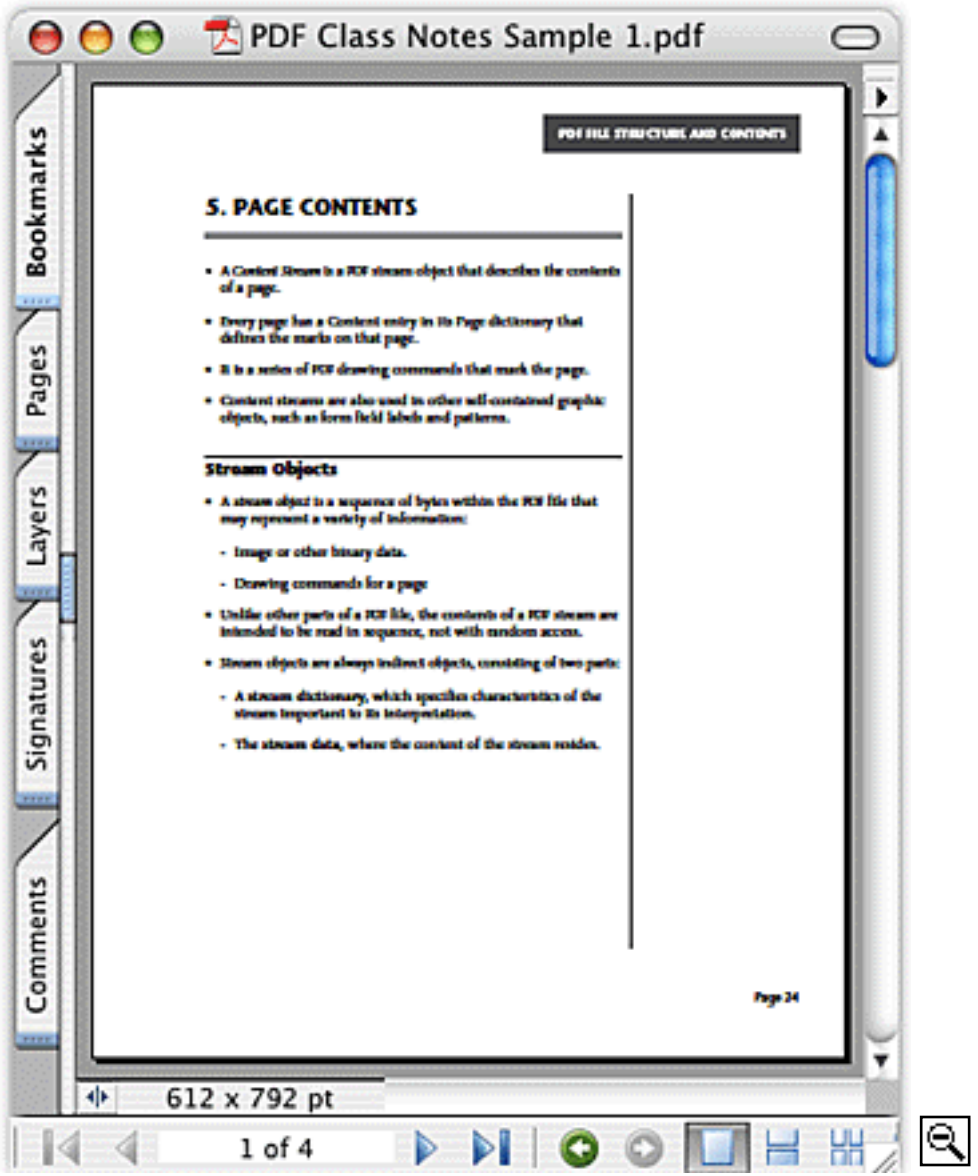
Suggestions for articles. Each Journal issue contains one article each on PostScript and Acrobat. What topics would you like me to write about?

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


PDF Class Notes Sample 1.pdf
Bookmarks
Pages
Layers
Signatures
Comments
PDF FILE STRUCTURE AND CONTENTS
5. PAGE CONTENTS
• A Content Stream is a PDF stream object that describes the contents of a page.
• Every page has a Content entry in its Page dictionary that defines the marks on that page.
• It is a series of PDF drawing commands that mark the page.
• Content streams are also used in other self-contained graphic objects, such as form field labels and patterns.
Stream Objects
• A stream object is a sequence of bytes within the PDF file that may represent a variety of information:
- Image or other binary data.
- Drawing commands for a page
• Unlike other parts of a PDF file, the contents of a PDF stream are intended to be read in sequence, not with random access.
• Stream objects are always indirect objects, consisting of two parts:
- A stream dictionary, which specifies characteristics of the stream important to its interpretation.
- The stream data, where the content of the stream resides.
Page 24
612 x 792 pt
1 of 4