Table of Contents

The Acrobat JavaScript *Global* Object

Variables and functions defined in Acrobat Document Javascripts are considered "global," in that they are accessible from within other scripts throughout the PDF document; any page or form field script can use those variables or functions. They are global within that document.

This month's article assumes you have read my book *Extending Acrobat Forms With JavaScript*. It does assume that that's *all* the experience you have, so experienced JavaScript programmers may find the pace a bit slow. Live with it.

However, Acrobat JavaScript has something called a *global object,* whose properties are global to Acrobat itself. The *global* object is a holder for data that is available to any PDF file open in Acrobat. You can add data to the global object by adding JavaScript properties to the object; the following will be true of those properties:

- They will be accessible within any script throughout the document.
- They will be accessible within any document currently open in Acrobat.
- They can be made *persistent,* so that their values will survive from one launch of Acrobat to the next.

The *global* object can contain strings (an account name, for example), booleans (indicating whether the user has registered for support), or numbers (an account balance).

In this article, we shall see how to add our own data to the *global* object, how to make that data persist from one launch of Acrobat to the next, and how to access that data from another Acrobat file.

Next page ->

Acumen Journal: Acrobat User 2

Review: Document Scripts

Most of the JavaScripts that set up global variables are most appropriate for use in a *Document JavaScript*. We discuss Document scripts in quite a lot of detail the *Extending Acrobat Forms…* book, but as a very brief reminder:

The Document JavaScripts in a PDF file are executed when that file is opened in Acrobat. Any variables or functions defined in a Document script are available in other scripts throughout the Acrobat file.

To define a document script, start with the document open in Acrobat Pro, and then do the following:

- Select *Tools* > *JavaScript* > *Document Scripts*.

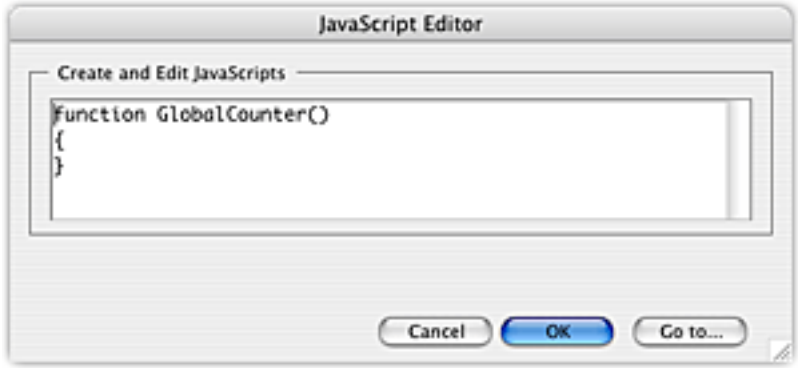
  Acrobat will present you with the *JavaScript Functions* dialog box (at right).

- Type a name for the script into the *Script Name* field and click the *Add* button.

Next page ->

Next page ->

Acrobat will present you with the *JavaScript Editor* dialog box. This contains a standard text editing pane into which you can type your script. This text field will start out with a template for a function whose name you supplied for the script. You can simply erase this initial template if you don't want to use it.

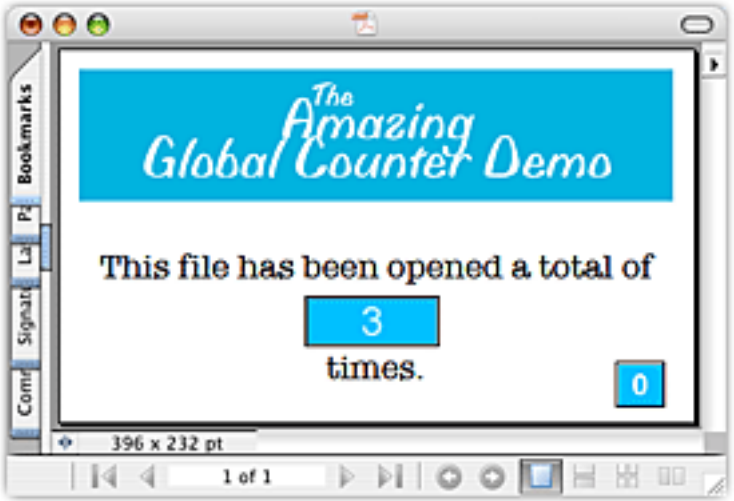


- Type your script into the edit pane and click *OK* repeatedly until you are back at your Acrobat document.

Next page ->>

A Global Counter

Consider the Acrobat document pictured at right, named "Counter.pdf." The blue text field in the lower middle of the window displays the number of times the document has been opened.



This month's sample files are packaged into the file *Global.zip,* available, as always, on the Acumen Training Resources page.

This first sample file is named *Counter.pdf.*

We shall implement this by creating a global variable named *counter.* Every time the document is opened, the value of *counter* will be incremented and then the new value will be displayed in the text field.

The "0" button in the lower right corner resets *counter* to zero.

4
times.

5
times.

In the nature of things, our *counter* variable must survive across openings of the document; that is, when we close the Acrobat file, *counter* must not disappear, as happens to most JavaScript variables. Furthermore, we want *counter* to survive even if we shut down the Acrobat application and then launch it again later.

This is exactly what the Acrobat JavaScript *global* object is for. Let's see how to do this.

Next page ->

Acumen Journal: Acrobat User 5

Creating a Global Variable

The Acrobat *global* object is a pre-existing object; you don't make a global object in your JavaScripts, rather you add new properties to the already-existing object.

You do this simply by referring to the new property:

```
global.counter = 0
```

If the *global* object doesn't already have a property named, in this case, *counter,* JavaScript will add that property to the object.

This new *counter* property will be accessible to all documents currently open in Acrobat, a characteristic we shall examine in more detail shortly.

In our case, it is more important that *counter* should be *persistent,* that is, it should retain its existence and value even if you shut down and then restart Acrobat. We must explicitly tell Acrobat when a property should persist by calling the *global* object's *setPersistent* method.

```
global.setPersistent("counter", true)
```

The *setPersistent* method takes two arguments:

- A string - the name of a property of the *global* object.
- A boolean - Indicates whether the property should be persistent (true) or not (false).

Next page ->

Acumen Journal: Acrobat User 6

Our Document Script

We create our global counter in a document script attached to the Acrobat file:

```
if (global.count == null) {      // If count doesn't exist...
     global.count = 1            // …Create it with a value of 1
     global.setPersistent("count", true) // & make it persist
}
else                             // If global.count does exist...
     global.count++              // ...then increment it

var f = this.getField("txtCount")    // Get the field "txtCount"
f.value = global.count               // Set its value to count
```

In broad outline, this script does the following:

- Check to see if the global property *count* already exists.
  - If not, create the property, setting its value to *1*, and then make it persistent.
  - If the property does exist, increment it.
- Set the text displayed in the text field (whose name is "txtCount") to the value of *count*.

Let's look at this in a little more detail.

Next page ->

Acumen Journal: Acrobat User 7

Step by Step

```
if (global.count == null) {
    global.count = 1
    global.setPersistent("count", true)
}
```

When we first open this document, we want to do one of two things:

- If the *global* object doesn't yet have a *count* property, meaning we have never before opened this PDF file, then we need to create the new property with an initial value of *1* and then specify that the property should persist.
- If the *global* object does have a *count* property, we need to increment it.

How do we check to see if *count* exists?

If the *count* property does not yet exist, then referring to it will yield a JavaScript *null* object. Therefore, our *if* clause tests for whether *count* exists by seeing if *global.count* is equal to *null*.

If *global.count* is null, we inialize a new *count* property to *1* and then call the *setPersistent* method, specifying that the property *count* should be persistent.

```
else
    global.count++
```

If *global.count* is not null, meaning *global* already has a *count* property, then we increment *count* with the ++ operator.

Next page ->

Acumen Journal: Acrobat User 8

```
var f = this.getField("txtCount")
f.value = global.count
```

Having either initialized or incremented our new *count* property, we get a reference to the *txtCount* Text field and then set its value (that is, the text that it displays) to the new value of *count*.

The global reoperty *count* is now exactly what we need it to be. It will retain its value indefinitely. We can open and close the document, close Acrobat and relaunch it, and *count* will still keep track of how many times the *Counter.pdf* document has been opened.

By the way... The *global* object and its properties reside on your computer, not within the Acrobat document. If you copy the PDF file to another computer, you will find that *count* is recreated and set to *1* when you open the document.

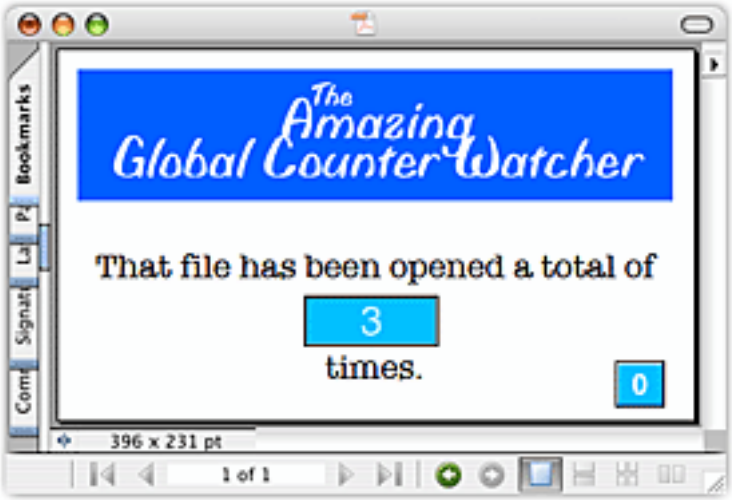
Talking Between Documents

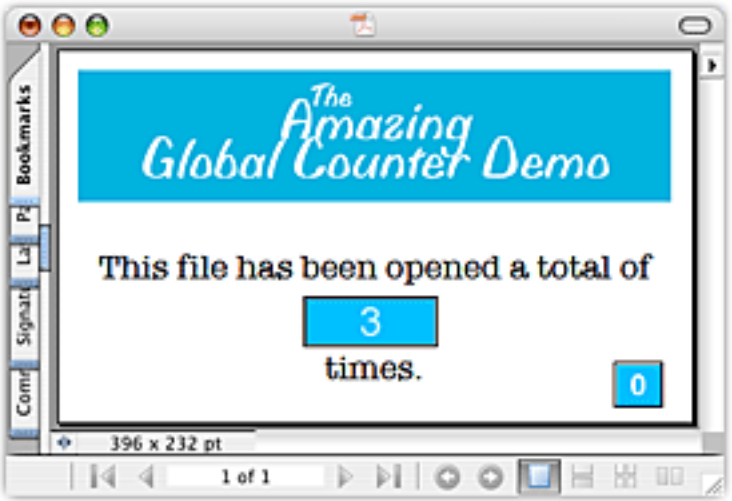
A significant characteristic of the *global* object—what makes it "global"—is that the object's properties are available to any PDF file, not just the file that created the property.

Thus, having added the *counter* property to *global,* any other Acrobat file, we can make a reference to *global.count* in any other PDF file's JavaScripts.

Next page ->>

For Example... Below left is another PDF file, *CounterWatcher.pdf*; this file displays the number of times that our *previous* PDF file (*Counter.pdf,* on the right) has been opened.





The document JavaScript for this file is very simple:

```
var f = this.getField("txtCount")

if (global.count == null)
     f.value = "--"
else
     f.value = global.count
```

We check to see whether *global.count* exists and, if so, display its value in the text field.

Next page ->

Acumen Journal: Acrobat User 10

Note *globalcounter.pdf* does not define *global.count;* that property was created by *counter.pdf* and is simply referenced here in the second file.

Actually, because we made *global.count* persistent, the original *counter.pdf* file doesn't even have to be open for our *counterwatcher.pdf* to work; persistent properties are always available to any PDF file at any time. Had we not made *count* persistent, it would have been available to any PDF file only as long as *counter.pdf* was open.

A Final Refinement: Instant Notification

One minor problem with *CounterWatcher.pdf* is that it checks the value of *global.count* and resets the number displayed in the Text field only when it is opened. If you close and then reopen *counter.pdf* while *Watcher* is open, the latter file will continue to display the old value.

The *global* object allows us to specify that a particular JavaScript function should be called whenever a particular global property changes value; such a function is referred to as a *callback function*.

For example, among the sample files for this *Journal* article is the file *CounterWatcher2.pdf*. This is identical to *CounterWatcher.pdf* except for an additional document script, listed on the next page.

Next page ->

Acumen Journal: Acrobat User 11

Creating a Callback

```
global.subscribe("count", CountChanged)

var fldTxtCount = this.getField("txtCount")

function CountChanged(newCount)
{
    fldTxtCount.value = newCount
}
```

The subscribe *method*

This script starts with a call to the *global* object's *subscribe* method:

```
global.subscribe("count", CountChanged)
```

Global.subscribe takes two arguments:

- A string containing the name of a global property ("count", in our case).
- The name of a JavaScript function (CountChanged, for us).

Whenever the property you specify changes its value, Acrobat will call the function, passing it the new value of the property.

Next page ->

Acumen Journal: Acrobat User 12

In our case, when *global.count* changes value, Acrobat will execute the *CountChanged* function, which will receive the new value of *count* as its argument.

```
function CountChanged(newCount)
{
    fldTxtCount.value = newCount
}
```

Our *CountChanged* function sets the value (the text displayed) of the Text field to the new value of *count*.

Note that we never directly call *CountChanged* in any of the JavaScripts in our PDF file; rather, we have instructed Acrobat to call the function automatically whenever *global.count* changes value.

A Minor Mystery

We'll finish our discussion with something I don't entirely understand about this final JavaScript.

Notice that our script gets a reference to *txtCount* in a separate line of JavaScript *outside* of the *CountChanged* function.

```
var fldTxtCount = this.getField("txtCount")
```

Normally, I would prefer this line to appear as part of the function definition, but here I have pulled it outside the function to work around a mysterious problem.

Next page ->

Acumen Journal: Acrobat User 13

The *this.getField* method always seems to fail inside a global variable callback function, such as *CountChanged*. Acrobat reports that *this.getField* doesn't exist. It appears that the JavaScript *this* keyword doesn't refer to the current Doc object ("this document") at the time the callback function is executed.

As a workaround, I get the reference to the text field ahead of time, outside of the callback function; this works perfectly well.

I should say that the Acrobat JavaScript documentation indicates that *this.getField* should behave normally inside a callback function, so I don't actually know what the problem is here. If anyone can give me more detail on why *this.getField* consistently fails inside a callback function, I'd very much like to hear it.

Return to Main Menu

Color Key Image Masking

A common technique in video is to photograph an actor against a blue (or other color) screen and then electronically replace the blue areas in the video with some other picture. This is how you get television weathermen superimposed against the weather map, for example.

PostScript languagelevel 3 allows you to do this with PostScript images. You can print an image as a *color-key masked image,* declaring that one color or range of colors (blue, in the example at right) shouldn't be printed, allowing the background to show.

This can be a very powerful technique and is worth examining.

Let's see how to do it.

Next page ->

Background: PostScript Images

As you remember from your PostScript class, images are printed with the PostScript *image* operator. In Level 2 and 3, this operator takes a single dictionary—an "image dictionary"—as its argument:

```
<<   /ImageType  1
     /Width      450
     /Height     338
     /BitsPerComponent   8
     /ImageMatrix [ 450 0 0 -338 0 338 ]
     /DataSource currentfile /ASCIIHexDecode filter
     /Decode     [ 0 1 0 1 0 1 ]
>> image
4C65BF4...
```

This article won't describe in detail the key-value pairs in the image dictionary; check your old student notes for complete coverage. Briefly, however:

`/ImageType  1`

This indicates what kind of image this is. Normal, scanned images are of type *1*.

`/Width      450`
`/Height     338`

These indicate the number of image pixels in each scanline and the number of scanlines in the image.

Next page ->

Acumen Journal: PostScript Tech 16

```
/BitsPerComponent   8
```

This is the number of bits associated with each color component in the image data. A value of *8* for an RGB image indicates that each red, green, and blue will consist of an 8-bit value.

```
/Decode      [ 0 1 0 1 0 1 ]
```

The *Decode* array defines the mapping of data values (varying from 0 to 255, say) to color values. The array contains a pair of color values for each color component in the image's data; an RGB image will have three pairs of numbers. Each pair of numbers indicates the color value that corresponds to the smallest and largest data values.

Thus, for an 8-bit image, a *Decode* pair *0 1* indicates that data values *0* to *255* should be mapped to RGB color values *0* to *1*.

```
/DataSource currentfile /ASCIIHexDecode filter
```

DataSource is the source of the image data. It may be a file object, a string, or a data acquisition procedure. (For the last, I refer you to your PostScript notes; it's a long story.)

In our case, the DataSource is *currentfile* with the ASCIIHexDecode filter attached; the *image* operator will read Hexadecimal image data in-line with our PostScript code.

Incoming image data isinterpreted in terms of the current color space. To print an RGB image, you would need to set the color space to *DeviceRGB* before calling *image*.

Next page ->

Acumen Journal: PostScript Tech 17

```
/ImageMatrix [ 450 0 0 -338 0 338 ]
```

ImageMatrix is a transformation matrix that specifies the size and position of the final printed image, transforming the printed image's position in User Space back to the original data in Image Space. It's a long story; look in your student notes.

As a help, the image matrix will almost always be:

```
[ width 0 0 –height 0 height ]
```

where *width* and *height* refer to the width and height of the image in pixels, not the size of the printed image.

Clipping an Image

Level 2 provided no support for masked images; if you want to mask out the sky in our image using only PostScript Level 2 commands, you will need to use the *clip* operator. This will be a bit tedious:

1. Construct a very complex current path that snakes between pixels in the image, enclosing those you want to appear on the page. This reqiores a *great* many *lineto's*.
2. Call the *clip* operator.
3. Call the *image* operator, painting the image on the page. Only those parts of the image that fall inside the clipping path will be painted.

Next page ->

Acumen Journal: PostScript Tech 18

The clipping path technique works perfectly well, though it can be very slow in some machines and I would expect it to be prone to *limitcheck* errors if memory is skinny.

Masked Images

PostScript LanguageLevel 3 introduced support for masked images; these contain not only image data, but information indicating what parts of the image should actually be printed.

PostScript supports two kind of masking: *explicit* and *color key* masking.

Explicit Masking

With explicit masking, the image is made up of two types of data:

- *Image data* - The data for the actual scanned image.
- *Mask data* - Information (usually 1-bit image data) indicating what part of the image data should actually be painted on the page.

The conceptually simplest version of explicit masking is the case in which the mask consists of a complete 1-bit image, completely unrelated to the scanned image. One of the colors in the mask (black or white, as you choose) indicates the area to be printed, as at right.

Image

Mask

Masked Image

This type of masking is more complicated than we can describe here, alas.

Next page ->

Acumen Journal: PostScript Tech 19

Color Key Masking

Color key masking allows you to specify a color or range of colors in the image data that should not be painted. This type of masking is most useful for pictures of some subject taken against a background of some known color.

Simple Example

To start our discussion of color key masking, consider the following code that draws a regular, unmasked image against a light blue background:

Sample Files

As usual, this month's examples are available on the Acumen Training Resources page. Look for the file *ColorKeyMasking.zip*.

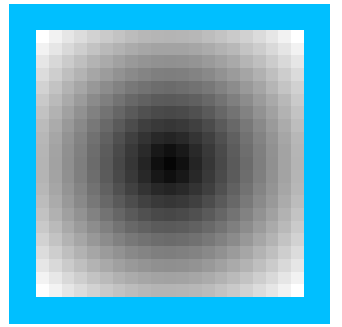
Not Complete Code

This code sample is not complete. The full program, on the *resources* page, starts by filling a string, *dataStr*, with data for the radial gradient seen in the illustration.

```
0 .75 1 setrgbcolor
90 90 120 120 rectfill

/DeviceGray setcolorspace
100 100 translate
100 100 scale
<<   /ImageType      1
     /Width          21
     /Height         21
     /BitsPerComponent 8
     /Decode         [ 0 1 ]
     /ImageMatrix    [ 21 0 0 21 0 0 ]
     /DataSource     dataStr % The code that loaded dataStr with
>> image                    % data was omitted for brevity

showpage
```



Next page ->>

Some Details

Here's what this program does, in light detail:

```
0 .75 1 setrgbcolor
90 90 120 120 rectfill
```

We start by drawing the rectangular background, filled with greenish-blue.

```
/DeviceGray setcolorspace
```

We set the colorspace to *DeviceGray,* to match the type of image data we have. Note that the earlier call to *setrgbcolor* set the colorspace to *DeviceRGB* as a side effect.

```
100 100 translate
100 100 scale
```

These two operators specify the position and size of the printed image. The *translate* operator specifies the position of the lower left corner of the image; *scale* specifies the size of the printed image, in PostScript units.

```
<<
    ...
>> image
```

Finally, we make the call to the *image* operator, in every way similar to our earlier example.

Next page ->

Acumen Journal: PostScript Tech 21

ImageType 4

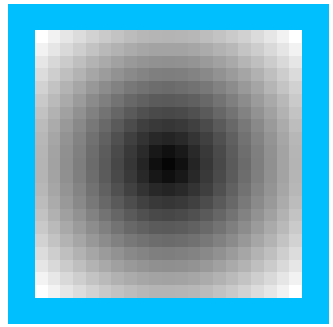
To apply color key masking to this image, we need only make two changes to the image dictionary in our preceding example:

- Change *ImageType* to *4*
- Add a *MaskColor* entry that specifies what color should be unpainted.

Omitting a Color

Our call to *image* now looks like this:

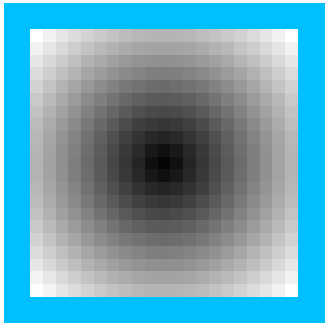
```
<<   /ImageType   4
     /MaskColor   [ 109 ]
     /Width       21
     /Height      21
     /BitsPerComponent 8
     /Decode      [ 0 1 ]
     /ImageMatrix [ 21 0 0 21 0 0 ]
     /DataSource dataStr
>> image
```



The *MaskColor* entry is an array whose contents make up a color specification in the current color space; this is the color that will be left unpainted when the image is printed. In our case, this array has only one number in it, since the image is made up of grayscale data; had this been an RGB image, the array would contain three values.

Next page ->>

Note that the color values in the *MaskColor* array are actual data values. Since our image is made up of 8-bit data, our grayscale value in *MaskColor* must be a number between *0* and *255*.

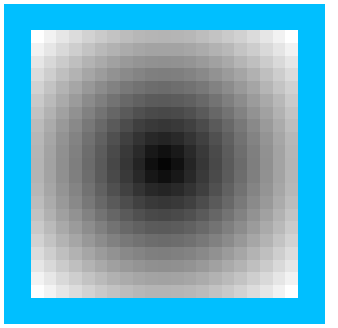


Our particular *MaskColor* array contains the value *109,* so all pixels that had a data value of 109 were left unpainted. At right you can see the blue background showing through the masked-out pixels.

Masking a Range of Colors

MaskColor can supply a range of color values that should be masked out. You need simply provide a pair of numbers for each color components, indicating the beginning and end of the set of values that should be unpainted.

```
<<   /ImageType   4
     /MaskColor   [ 100 125 ]
     /Width       21
     /Height      21
     /BitsPerComponent 8
     /Decode      [ 0 1 ]
     /ImageMatrix [ 21 0 0 21 0 0 ]
     /DataSource dataStr
>> image
```



Now, any pixels whose data lies on the range 100–125 will be unpainted, as above.

Next page ->

Acumen Journal: PostScript Tech 23

Color-Masking a Photograph

Let's see how this applies to the task of masking out the blue sky in the photograph at right.

In a real situation, you would have likely taken a picture of your subject against a backdrop of a known color. The minor challenge in our photograph is that the sky is made of a range of colors, different shades of blue. Our *ColorMask* array must specify the range of RGB values that look blue.

Not too realistic

This is probably not a very realistic example. The type of image you with which you would normally use color-key masking would be a picture taken specifically for the purpose, such as a model against a blue backdrop.

Practically speaking, this means we want to remove all pixels that have a relatively large amount of blue and relatively little red and green. Determining what *MaskColor* values give the best results in this case requires a little trial and error, but isn't particularly difficult.

The masked version of this image is on the next page.

Next page ->>

```
<<   /ImageType 4
     /MaskColor [ 0 150 0 150 150 255 ]
     /Width 450
     /Height 338
     /BitsPerComponent 8
     /ImageMatrix [ 450 0 0 -338 0 338 ]
     /DataSource currentfile /ASCIIHexDecode filter
     /Decode [ 0 1 0 1 0 1 ]
>>
image
```

As before, our *MaskColor* values are intended for the 8-bit RGB data that makes up our image. In this case, a pixel will not be painted if its red and green values are 150 or below and if its blue value is 150 or above.

Caveat The only real disadvantage to color masked images is that they require PostScript LanguageLevel 3; you should not use them in any PostScript code that needs to be used with a broad range of printers, since there are still many PostScript Level 2 devices out there.

Otherwise, color masked images are a powerful, easy tool in the PostScript toolbox.

Return to Main Menu

Schedule of Classes, Aug - Oct 2004

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 30–Sep 2		Oct 11–15
PostScript Foundations	Aug 2–6		
Variable Data PostScript	Aug 9–13		
Advanced PostScript	Aug 16–20		
PostScript for Support Engineers		Sep 20–24	
Jaws Development		*On-site only*	

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

Registration Info

Acrobat Classes

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

I am in the planning stages of a second four-day PDF class, provisionally *PDF File Content & Structure 2*. Like the first course in the series, this class will be an engineer's class in the structure and content of a PDF file. It will presume you have taken the first class or have equivalent knowledge. Also like the first class, this course will restrict itself to those parts of the PDF specification that apply to printed documents. Thus, I will not be talking about anything having to do with animation, sound, etc.

A preliminary list of topics, in no particular order, are below. Not all of these topics may make it to the final class and I am actively seeking comments on topics that should be added to this list; if you have strong feelings that 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

LONDON