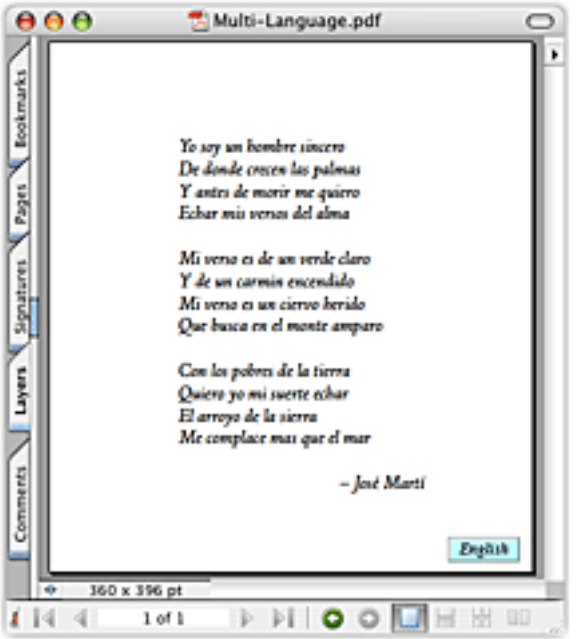
Table of Contents

Acrobat 6 Layers and Forms

Acrobat 6 introduced a concept new to the Acrobat and PDF world: *Layers*. This is a feature of long standing in design and page layout software, but is new to the PDF file spec. The introduction of layers makes possible a set of new and powerful abilities for people who design PDF files. In particular, layers make it extremely easy to hide and show collections of information on a PDF page.

For example, you may have several versions of an document's text (perhaps in different languages) placed on the page in different layers. When the reader of the document selects a language from a pop-up menu, the menu could make visible the layer that contains that version of the text. (There is an example of just such a document among the sample files for this month.)



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

This month we are going to see how layers work in Acrobat. We'll look at the Acrobat *Layers* panel and see how to create simple buttons that make specific layers visible. We shall also see how to manipulate layer visibility with JavaScript.

Next Page ->

Acumen Journal: Acrobat User 2

Creating Acrobat Layers

First the disappointing news: you cannot create layers within Acrobat itself. You must assign objects to layers within whatever application you use to create the PDF file.

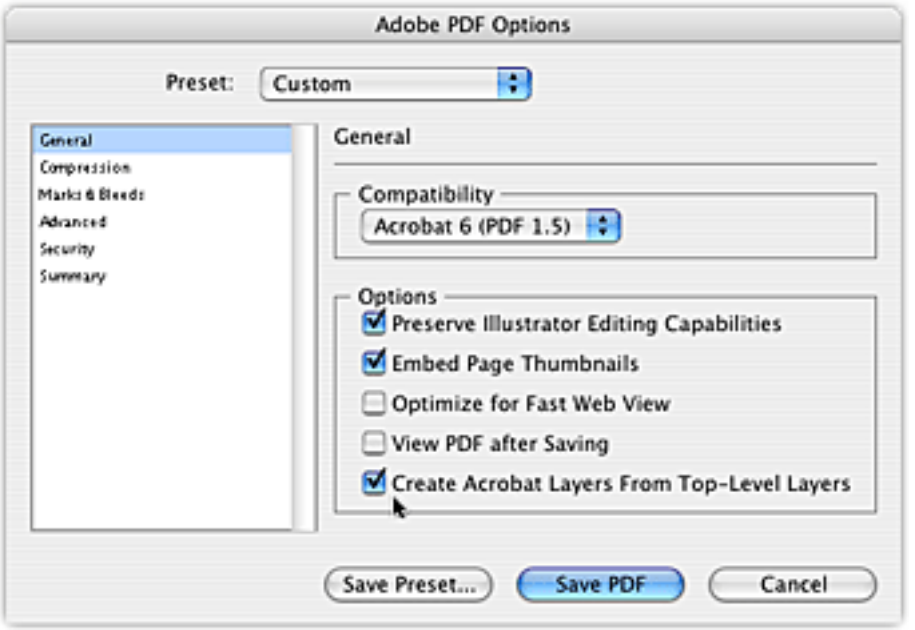
Many design and layout programs support a notion of layers, but few of them will preserve those layers when exporting to a PDF file. The applications that do so include the current ("CS") versions of *Adobe Illustrator* and *Adobe InDesign*.

Preserving Layers in Adobe Illustrator

For example, when you export an Illustrator document to PDF, Illustrator presents you with the *Adobe PDF Options* dialog box (at right).

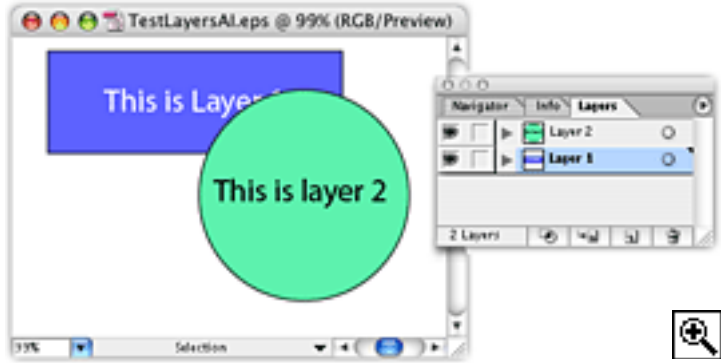
Simply select the *Create Acrobat Layers…* button in this dialog box and Illustrator will convert layers in the Illustrator document into Acrobat layers in the PDF file.

Next Page ->>



Sample Files

The two sample files for this article are on the Acumen Training Resources page; there are four PDF files packaged into the file *AcrobatLayers.zip*. Three of these sample files started life as an Illustrator document containing a square and a circle, each in its own layer.

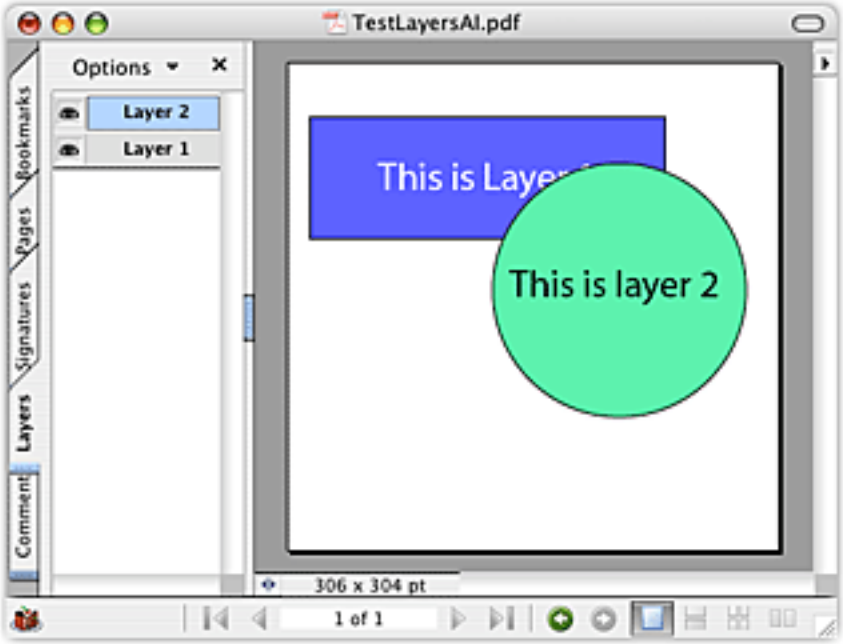


The *Layers* Pane

The original Illustrator document was exported to a PDF file, preserving the layers. The resulting Acrobat document, *Layers Demo 1.pdf,* at first looks like any other PDF file. The only evidence that this file contains layers is the cute little cake icon in the lower left corner of the window frame.

Note: Because layers were introduced with Acrobat 6, you will need to use that version of Acrobat to open this month's sample files. Earlier versions of Acrobat ignore the layer information.

The individual layers become evident when you click on the *Layers* tab at the left side of the Acrobat window. This exposes the *Layers* pane, which lists all of the layers in this Acrobat file.

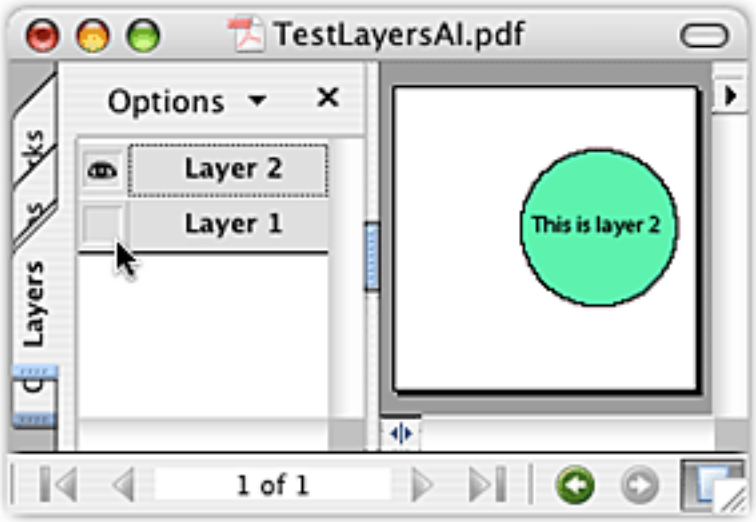


Next Page ->

Controlling Layer Visibility

To the left of each layer's name in the Layers palette is an eyeball icon. Clicking on this icon toggles the visibility of the corresponding layer.

This works well enough for experimentation, but an Acrobat form should control layers' visibility with, say, a set of buttons.



Layers and Buttons

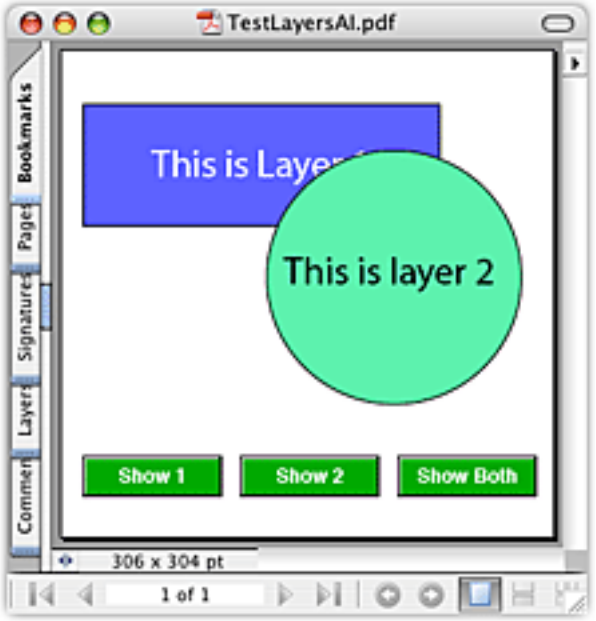
The zip file for this issue has three versions of this sample:

- *Layers Demo 1* is the Acrobat file with no buttons.
- *Layers Demo 2* has the buttons with actions attached to them.
- *Layers Demo 3* is the final form with functioning buttons.

Let's add three buttons to our Acrobat file: *Show 1, Show 2,* and *Show Both* that make visible Layer 1, Layer 2, and both layers, respectively. The final form will look like the illustration at right.

I am going to assume that you have skill with Acrobat forms sufficient to add the buttons, themselves, to the original Acrobat form-with-layers. (If not, you may want to read my book *Creating Acrobat Forms;* buy several copies!)

We are going to add appropriate actions to these buttons.



Next Page ->>

The *Set Layer Visibility* Action

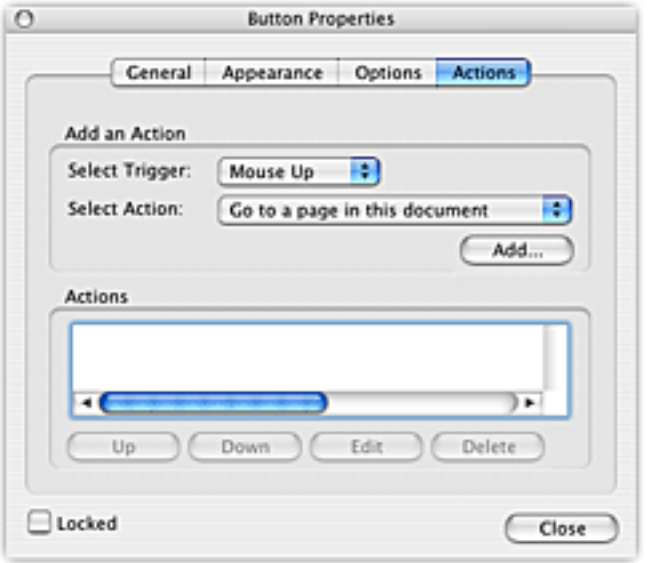
Acrobat 6 added a new action to those you may attach to a form field: *Set layer visibility.* This action sets the visibilities of all layers to whatever they were at the time you attached the action to the form field.

Thus, to create a *Show 1* button, we would do the following (assuming the basic button has already been created):

1. Expose the *Layers* pane.
2. Set the visibility of the document's layers to match what a button click should later do. For our *Show 1* button, we want to make Layer 1 visible and Layer 2 invisible.
3. Select the Button tool in the Forms toolbar.

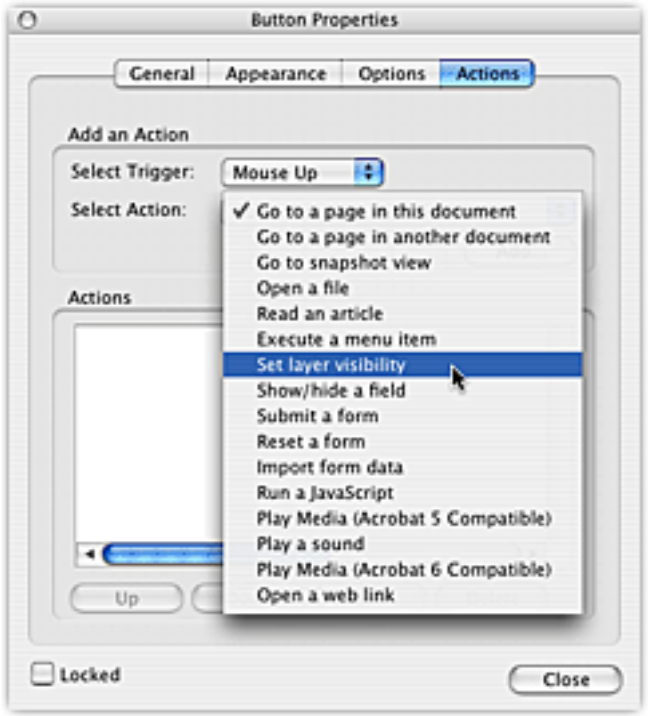
4 Double-click on the *Show 1* button. (In the sample file, this button is named "btnShow1.")

Acrobat will present you with the *Button Properties* dialog box (shown at right).



Next Page ->

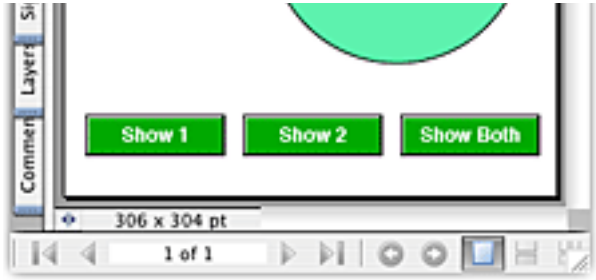
5. In the *Trigger* pop-up menu, select *Mouse Up*.
6. In the *Actions* panel, select *Set layer Visibility*.
7. Click on the *Add* button. (This is hidden by the *Action* menu in the illustration at right.
8. Click on the *Close* button.



That's it That's all we need to do. Return to the Acrobat Hand tool and you're done. Now, whenever you click on the *Show 1* button, Acrobat will return the visibility of the documents layers to what they were when you attached the action to the button: Layer 1 visible, Layer 2 hidden.

For practice, you may wish to start with the file *layers Demo 2* and add the appropriate actions to the three buttons.

(You can start with *Layers Demo 1* if you also want to create the buttons, themselves.)



Next Page ->

JavaScript, OCGs, and Layers

Your buttons can do more sophisticated things with layers using a JavaScript. The following discussion assumes you have at least basic knowledge of Acrobat JavaScript, comparable to having read my book *Extending Acrobat Forms With JavaScript*.

OCG Objects

In JavaScript, a layer is referred to as an *Optional Content Group,* that is, a collection of text and graphics that may or may not be presented to the reader of the Acrobat file. Acrobat JavaScript uses an *OCG* object to represent one of the OCGs—that is, one of the layers—in an Acrobat document.

OCG Object Properties

OCG objects have only two properties visible to JavaScript:

name (String) The label assigned to this layer. This will be the same as the name associated with this layer in the *Layers* pane.

state (Boolean) This indicates whether the layer is visible (true) or invisible (false). Changing this property changes the visibility of the layer.

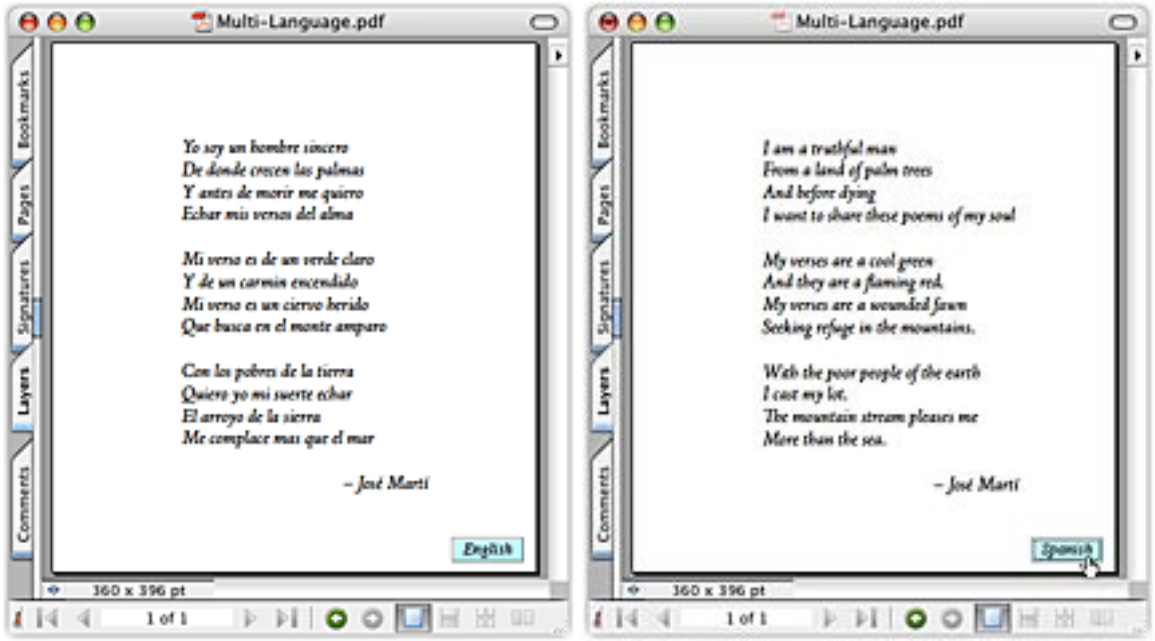
doc.getOCGs(pageNum)

You can find out what layers the current document contains with the Doc object's *getOCGs* method. This takes an optional page number as its argument and returns an array containing the OCG objects on that page; if the page number is omitted, the method returns all the OCG objects in the document. If there are no layers in the document, *getOCGs* returns a null.

Next Page ->

A Multi-Language Document

This document is included in this month's zip file as *Multi-Language.pdf*.



Let's use the Acrobat JavaScript OCG mechanism to implement the bilingual document pictured at right.

This document was created in Adobe Illustrator with two text blocks, each in its own layer (named "English" and "Spanish"), one atop the other.

I exported the document to a PDF file, preserving the layers, and then in Acrobat added the English/Spanish language button in the lower-left corner. Clicking on the button toggles between the English and Spanish layers.

We are going to look at the JavaScript attached to the button's *Mouse Up* event. This is the script that reverses the visibility of the document's two layers.

Next Page ->

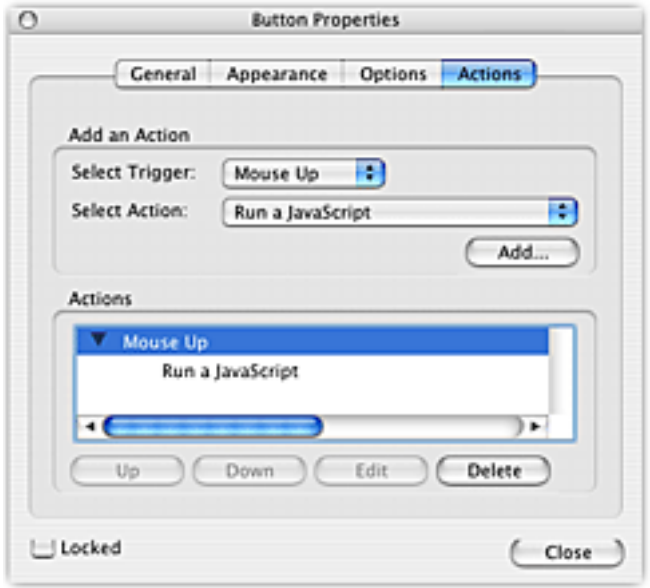
Getting to the JavaScript

We get to the button's JavaScript in the usual fashion:

- Select the button tool in the *Form* toolbar.
- Double-click on the button (whose name is *btnLanguage).*

This will yield the *Button Properties* dialog box, below right.

- Going to the dialog box's *Actions* pane, we can see that the *Mouse Up* event has a JavaScript attached to it.
- Select the "Run a JavaScript" entry associated with *Mouse Up* and click on the *Edit* button at the bottom of the panel.



Acrobat will present you with a text editing window containing the JavaScript. You may edit the code, resize the window, and otherwise treat it as though it a standard text editing window.

Next Page ->

The JavaScript

Here's the JavaScript that does the deed:

```
var OCGs = this.getOCGs()                    // Get array of OCGs

OCGs[0].state = !OCGs[0].state               // Reverse the OCGs' states
OCGs[1].state = !OCGs[1].state

// Now let's change the button's label; start by getting the form:
var btnLabel = event.target.buttonGetCaption()

if (btnLabel == "English")
     event.target.buttonSetCaption("Spanish")
else
     event.target.buttonSetCaption("English")
```

Step-by-Step

Let's look at this JavaScript in detail.

`var OCGs = this.getOCGs()`

The snippet starts by calling the *getOCGs* method in the current ("this") document. The resulting array of `OCG` objects is assigned to the variable named *OCGs*.

"OCGs" is pronounced "O-C-Geez," by the way.

The *OCGs* array will have two entries, representing the two layers in our document. We can access these layers as `OCGs[0]` and `OCGs[1]`, in usual array fashion.

Next Page ->

Reverse the OCG states

```
OCGs[0].state = !OCGs[0].state
OCGs[1].state = !OCGs[1].state
```

We now reverse the *state* property of each layer; the visible layer will become hidden and *vice versa*. We do this using the "!" operator (pronounced "not"). The *not* operator reverses the value of a Boolean; *true* becomes *false* and *false* becomes *true*.

Thus, the statement

```
OCGs[0].state = !OCGs[0].state
```

sets the value of the *state* property of the *0th* OCG in the *OCGs* array to the opposite of its current value, thus reversing the visibility of the corresponding layer.

We do this for both of the OCG objects in *OCGs,* reversing the visibility of both the document's layers.

Get the button label

```
var btnLabel = event.target.buttonGetCaption()
```

Now we must change the label of the choose-your-language button. If we just made the English text visible, we want to change the button's label to "Spanish," and *vice versa*.

We start by getting the language button's current label using the Acrobat Form object's *buttonGetCaption* method. This method returns a string containing the button's label text. Note that we can use *event.target* to refer to the language button, since the button was the source of the *Mouse Up* event that triggered this script. We place the label text into the variable *btnLabel*.

Next Page ->

Reverse the button label

```
if (btnLabel == "English")
     event.target.buttonSetCaption("Spanish")
else
     event.target.buttonSetCaption("English")
```

The script ends with an `if...else` block that reverses the button label: "English" becomes "Spanish" and *vice versa*.

The `if` clause looks to see if the value of *btnLabel* (the string variable containing the button label text) is "English." If so, we set the button's label (using the *buttonSetCaption* method) to "Spanish."

If the value of *btnLabel* is not "English," then we set the label to "English."

Finding a Particular Layer

The above script takes advantage of the fact that we know that our Acrobat document has only two layers. We didn't actually look for the layer named "English" or "Spanish"; we just reversed the state of the layers without regard to which was which.

Somewhat surprisingly, Acrobat JavaScript provides no way to directly ask for the OCG object representing a particularly-named layer; you need to obtain the array of all layers and then go through the OCG objects in the array, looking for the layer you want. For example, to find a layer named "Layer1," you would do something like this:

Next Page ->

```
var ocgs = this.getOCGs()
var theOCG = null

for (var i = 0; i < ocgs.length; i++)   {
    if (ocgs[i].name == "Layer1") { // Is ocgs[i] our layer?
        theOCG = ocgs[i]            // Yes: set theOCG...
        break                       // ...and then leave loop
    }
}
```

This script finishes with the variable *theOCG* being equal to either the named layer's OCG object or to *null,* if there is no such layer.

We are going to be brief in our examination of this snippet of JavaScript, but in broad outlines, here's how it works:

```
var ocgs = this.getOCGs()
var theOCG = null
```

Create a pair of variables:

- *ocgs* contains the array of OCG objects in this document, as in our earlier JavaScript.
- *theOCG* will be assigned the OCG object associated with our target layer, if we find it. Note that *theOCG* is initially set to *null.*

Next Page ->

```
for (var i = 0; i < ocgs.length; i++)  {
    ...
}
```

Start up a *for* loop that sets a variable *i* to *0* and repeats the loop as long as *i* is less than the length of the *ocgs* array. At the end of each time through the loop, we shall increment *i*.

We didn't talk about loops in the *JavaScript* book, so you will need to research the *for* loop, if you haven't seen it before. Any book on JavaScript for the Web will discuss the *for* loop and its relatives.

```
if (ocgs[i].name == "Layer1") { // Is ocgs[i] our layer?
    theOCG = ocgs[i]            // Yes: set theOCG...
    break                       // ...and then leave loop
}
```

Each time through the loop, we check to see if the name of entry *i* in the *ocgs* array is "Layer1." If it is, we set *theOCG* to the entry and then exit from the loop with the *break* command. Otherwise, *for* increments *i* and then executes this set of statements again.

By the time we exit from the loop, *theOCG* will be set to the OCG object whose name is "Layer1." If we never found such a layer, then *theOCG* will still be *null*.

A Caveat

By the way, a name can be shared by several layers in an Acrobat file. It is up to you, when you create the file in Illustrator or InDesign, to make sure that layer names are unique within that file.

Next Page ->

Acumen Journal: Acrobat User 15

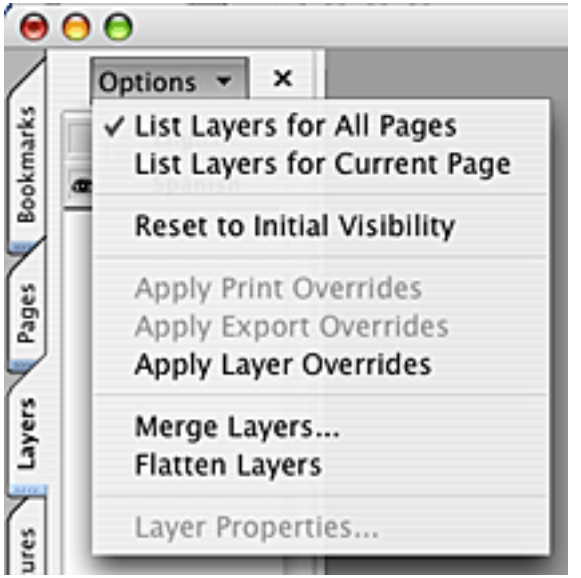
Layers R Good

Layers are one of my favorite new parts of Acrobat 6. They have many obvious uses and make easy a range of effects that were difficult before. And, as we have seen, it is relatively easy to control their visibilities from within an Acrobat form.

There are some subjects I haven't touched on in this article, such as the commands available through the menu attached to the Layers pane.

But, alas, we're out of space and time.

Later, perhaps.



Return to Main Menu

Binary Tokens

PostScript is usually generated as an ASCII language; a PostScript program is composed of a stream of ASCII characters. Adobe's decision to have PostScript be ASCII-encoded has had many benefits; in particular, it makes the language portable across all platforms and transmission methods. It has also made it possible to write PostScript programs by hand.

On the other hand, it does make the language relatively verbose. Sending a one-byte integer value to the interpreter can require transmitting up to four bytes of ASCII (as in "100" followed by a delimiter).

All PostScript Level 2 and Level 3 devices support a rarely-discussed binary encoding called *Binary Tokens*. This allows a PostScript program to be send to the interpreter as a stream of binary commands and values. A binary-encoded PostScript program is unreadable to the eye and you certainly could not write such a PostScript program by hand; these are definitely appropriate only to machine-generated PostScript.

On the other hand, binary tokens are very compact. A PostScript program expressed as a stream of binary tokens is, on average, the most compact form of that program, short of applying compression.

This month we shall look at how to use binary tokens in a PostScript program.

Next Page ->

Acumen Journal: PostScript Tech 17

PostScript Encodings

Modern PostScript supports three different methods by which a PostScript program may be expressed:

- *ASCII Encoding* - This is the most common encoding used for PostScript files. The program consists of a stream of ASCII characters, as in "100 100 400 100 rectfill".
- *Binary Tokens* - This is a compact version of the PostScript language that allows each PostScript object in the stream to be expressed as a one-byte "token" followed by one or more bytes of value.
- *Binary Object Sequences* - This interesting encoding allows a set of executable PostScript to be sent to the interpreter as a stream of eight-byte PostScript objects; in effect, the driver generates the actual PostScript objects that the interpreter actually uses internally.

Binary Object Sequences aren't used, as far as I know. They are inconvenient to generate and don't offer much of a performance benefit.

Mixing Encoding

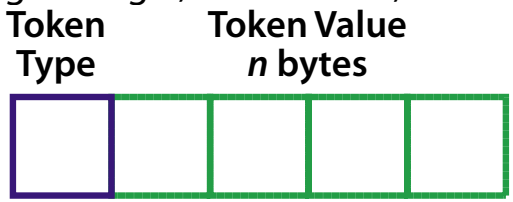
A PostScript stream can contain any mixture of the three encodings that is useful to the driver. A program can start with ASCII, continue with some binary tokens, follow with a binary object sequence or two, followed by more ASCII, etc. The PostScript scanner keeps it all straight according to the values of the incoming bytes in the stream:

- Incoming byte values 0-127 indicate ASCII PostScript
- 128-131 specify different types of Binary Object Sequences
- 132-149 indicate different binary tokens

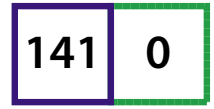
Next Page ->

Binary Tokens

A binary token represents a single PostScript object: a single integer, real number, string, name, etc. Each binary token starts with a one-byte "Token Type" that indicates what kind of PostScript object this token represents. The token type is followed by one or more bytes that indicate the value of the object.



A binary token is usually much more compact than its ASCII equivalent. For example, in ASCII PostScript, the boolean object *false* is represented by six bytes: the characters f-a-l-s-e and a trailing delimiter. As a binary token, *false* is represented by two bytes: a byte of value 141 (the token type that indicates a boolean object) followed by a byte of value zero (indicating *false)*.



Next Page ->

Token Types

The possible token type values—and the data types they specify—are as follows:

132, 133	*32-bit integer,* high/low-byte first. The token is followed by the integer data.
134, 135	*16-bit integer,* high/low-byte first.
136	*8-bit signed integer*
137	*16- or 32-bit fixed point number.* The bytes that follow indicate the details, such as the number of bits and the position of the decimal point.
138, 139	*IEEE real number,* high/low byte first
140	*Native real number.* This is for PostScript implementations that reside on the same system that is generating the PostScript code.
141	*Boolean*. The token is followed by a single byte indicating the boolean value. A zero indicates *false,* as usual.
142	*Short string* (fewer than 256 characters). The token is followed by a one-byte string length and then the string's characters.
143, 144	*Long strings* (up to 64k characters). The token is followed by a two-byte length (high byte/low byte first) followed by the string's characters.
145, 146	*Literal/Executable name* taken from the System Name Table (see below).
149	*Homogeneous number array.* (See below.)

Next Page ->

Acumen Journal: PostScript Tech 20

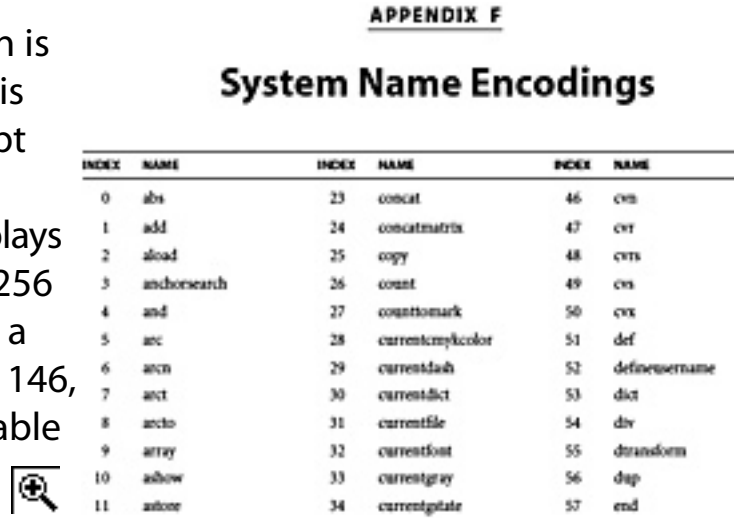
Thus, the number 2,000 can be sent to the PostScript interpreter as a three-byte sequence consisting of the byte *133* (indicating a two-byte integer), followed by the actual two-byte value for 2,000. This is half the size of the ASCII "2000" plus delimiter.

Most of the entries (integers, strings, etc.) in the preceding page's table are reasonably clear in their use and interpretation. Names and homogeneous number arrays require some additional discussion, however.

Names as Binary Tokens

Token types145 and 146 denote names. The former indicates a *literal name,* that the interpreter will put on the operand stack; in ASCII PostScript, you would indicate this with a preceding slash, as in "/Times-Roman." Token type 146 specifies an *executable name;* the interpreter will look this name up on the dictionary stack and take some appropriate action.

In both cases, the byte following the token is an index into the *System Name Table*. This is a predefined table of names that PostScript maintains internally. Appendix F in the PostScript Language Reference Manual displays the contents of this table. Any of the first 256 entries in this table can be represented as a two-byte binary token: token type 145 or 146, followed by the one-byte index into the table indicating the name you want.



APPENDIX F

System Name Encodings

INDEX	NAME	INDEX	NAME	INDEX	NAME
0	abs	23	concat	46	cvn
1	add	24	concatmatrix	47	cvr
2	aload	25	copy	48	cvrs
3	anchorsearch	26	count	49	cvs
4	and	27	counttomark	50	cvx
5	arc	28	currentcmykcolor	51	def
6	arcn	29	currentdash	52	definename
7	arct	30	currentdict	53	dict
8	arcto	31	currentfile	54	div
9	array	32	currentfont	55	dtransform
10	ashow	33	currentgray	56	dup
11	astore	34	currentgstate	57	end

Next Page ->

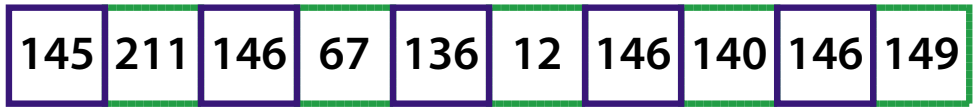
Acumen Journal: PostScript Tech 21

For Example... Consider the following common piece of PostScript code, amounting to 43 characters of ASCII, including the trailing delimiter:

```
/Times-Roman findfont 12 scalefont setfont
```

This could be expressed as the following stream of binary tokens:

So far, I'm being completely diagrammatic in depicting our binary tokens. Patience. We'll see some actual binary-encoded PostScript code shortly.



145 This is the token type for a literal name; the next byte must be an index into the System Name Table.

211 This is the position of the name */Times-Roman* in the System Name Table.

146 This is the token type for an executable name.

67 This is the position of the name *findfont* in the System Name Table.

134 12 The token type indicates an 8-bit integer. The byte following the token type is the value of that integer, *12*.

146 140 The executable name *scalefont*.

146 149 The executable name *setfont*.

This series of binary tokens is 10 bytes long, less than ¼ the size of the ASCII version.

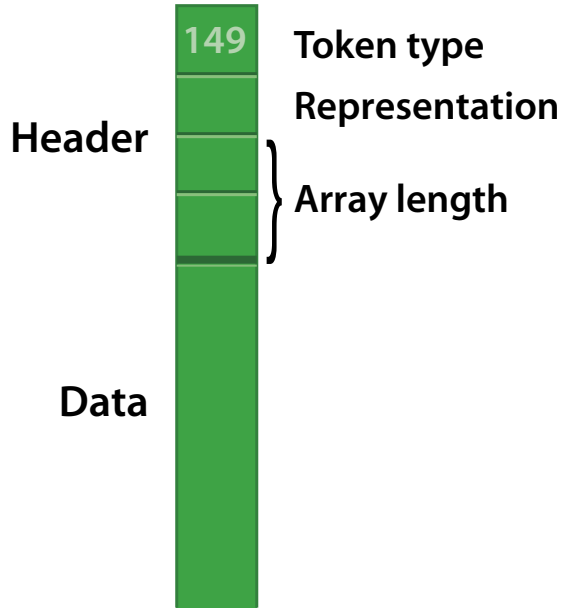
Next Page ->

Acumen Journal: PostScript Tech 22

Homogeneous Number Arrays

Token type 149 specifies that the bytes that follow are a *homogeneous number array;* this is the binary token representation of an array of numbers, all of which are the same type: fixed point numbers, integers, real numbers.

A homogeneous number array ("HNA" from now on) is made up of a four-byte header followed by the numeric data for the array's contents.



HNA Header

The header contains three components:

- The one-byte token type *149,* indicating that what follows is a homogeneous number array.
- A one-byte code, called the *representation code,* that specifies the type of numbers that make up the array.
- A two-byte length indicating the number of entries in the array.

Next Page ->

Representation Byte The second byte in the HNA header is a code that indicate what kind of number makes up the array. The values this byte may have are:

0 – 31 *32-bit fixed point number.* The exact value on the range 0–31 indicates the position of the decimal point; *0* indicates the decimal point is to the right of bit *0,* that is, the number is a 32-bit integer.

32 - 47 *16-bit fixed point number.* As with the 32-bit values, the specific value indicates the position of the decimal point. A representation value of *32* indicates the array will be made up of 16-bit integers.

48 *IEEE real number.*

49 *Native real number.* As before, this is for cases where the PostScript interpreter and the PostScript source share an underlying architecture that has some native idea of a real number.

128–177 These values duplicate the meanings of *0-49,* but indicate the numbers in the array will all be low-byte-first.

We shall see an example of a homogeneous number array in a moment.

Next Page ->>

Trying It Out

Binary tokens are not human-readable, so they would normally be appropriate only for PostScript generated by a driver. Still, it would be nice to play with them a bit.

Here's one way to do it: express the binary tokens as ASCII hex and execute them through the ASCIIHexDecode filter.

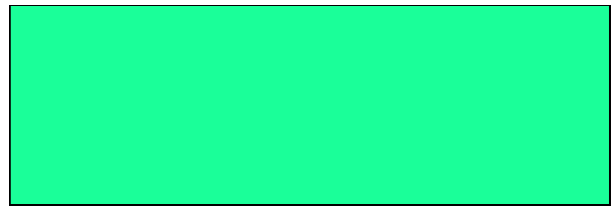
ASCII-Encoded Tokens

For example:

```
.1 1 .6 setrgbcolor
currentfile /ASCIIHexDecode filter cvx exec
95200004 0064 0258 012c 0064
9238 9280
8800 9296
9281 >              % The ">" marks end-of-data for ASCIIHexDecode
showpage            % Here we are back to normal, ASCII PostScript
```

The ASCII-encoded binary tokens above are the equivalent of the following ASCII PostScript:

```
[ 100 600 300 100 ]
dup rectfill
0 setgray
rectstroke
```

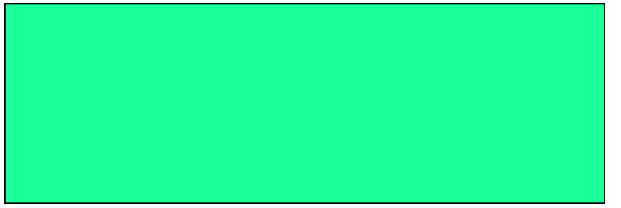


This produces a filled, stroked rectangle, as above right.

Next Page ->

Step-by-Step

Here's what's happening, in detail:



```
.1 1 .6 setrgbcolor
```

Set the current color to a light green. I did this in ASCII PostScript just because I didn't want to hand-code a set of IEEE reals.

```
currentfile /ASCIIHexDecode filter cvx exec
```

Attached the ASCIIHexDecode filter to *currentfile,* convert the resulting filtered fileobject to executable, and execute it. We shall begin executing *currentfile* through the ASCIIHexDecode filter; whatever follows *exec* will be converted from hexadecimal to binary and then handed to the PostScript interpreter.

```
95200004 0064 0258 012c 0064
```

This is an ASCII-encoded homogeneous number array. The contents are:

95	This is the token type for an HNA. (This is 149 expressed as hex.)
20	The representation byte. Value *32* (that is, hexadecimal *20)* indicates an array of 16-bit integers
0004	This is the two-byte array length; we have an array of four numbers.
0064…	The HNA header is followed by four 16-bit integers, the values *100, 600, 300,* and *100,*expressed in hex.

Next Page ->

Acumen Journal: PostScript Tech 26

9238 9280 The HNA is followed by two executable names: *dup* and *rectfill*. Note that "92" is token type *146*, indicating an executable name. "38" and "80" are the hexidecimal indices into the System Name Table for *dup* and *rectfill*.

8800 9296 This is the call to *setgray*. "88" is token type 136, indicating an 8-bit integer. The byte following, "00," specifies a value of *0*. "9296" indicates the executable name *setgray*.

9281 This is the executable name *rectstroke*.

> The greater-than symbol is the end-of-data marker for the ASCIIHexDecode filter. This causes us to resume executing *currentfile* directly, rather than through the filter.

showpage The program ends with a call to *showpage* expressed in ASCII PostScript.

Next Page ->>

Mixed Formats

I would like to demonstrate that PostScript can correctly interpret a combination of ASCII PostScript and binary tokens. To do so, we'll need to write a computer program that creates a mixed-format PostScript stream; I am going to do this using, as my programming language, PostScript, itself.

Below is a PostScript program that creates raw binary tokens mixed with ASCII PostScript, writing the resulting mixed-encoding PostScript to %stdout:

```
(.5 1 .7 setrgbcolor) =          % Write some ASCII PostScript

% Read some ASCII-encoded binary tokens
currentfile /ASCIIHexDecode filter 50 string readstring
8864 860258 86012c 8864 9280>
pop
=                                % Write raw binary tokens to stdout

(1 .5 .3 setrgbcolor)=           % Write some more ASCII PostScript
(200 550 50 200 rectfill)=
(showpage)=
```

The above call to *readstring* reads our ASCII-encoded binary tokens through the ASCIIHexDecode filter. The operator returns a string containing the raw binary tokens (and the usual boolean on top).

We discard the boolean with a *pop* and then print the binary data to stdout with =.

Next Page ->

Acumen Journal: PostScript Tech 28

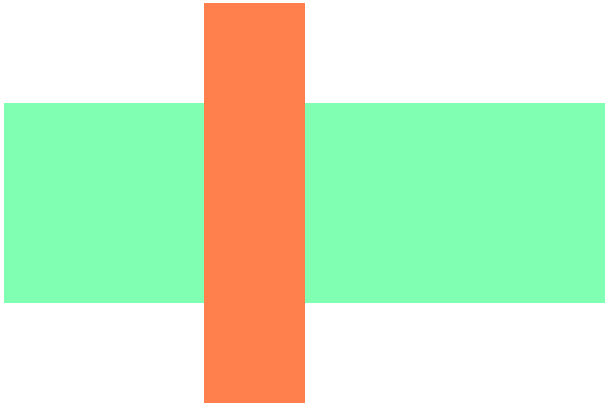
The binary tokens in this case encode a call to *rectfill:*

```
100 600 300 100 rectfill
```

The resulting PostScript code sent to stdout is:

```
.5 1 .7 setrgbcolor
àdÜXÜ,àdíÄ
1 .5 .3 setrgbcolor
200 550 50 200 rectfill
showpage
```

When executed, this results in two filled rectangles painted on the current page, as at right.



Note that this PostScript code never explicitly switches between ASCII and binary token "modes"; the encoding of the incoming PostScript is implicit in the numeric range in which each incoming byte lies.

Next Page ->>

What's it good for?

Binary tokens are not at all useful for handwritten PostScript (except for experimentation, as we do above). They can be very appropriate for driver output because they are very compact. Some years back, I wrote an ASCII-to-binary-token converter that allowed me to convert driver output from various sources to binary tokens. On average, the binary token version of the PostScript output was about half the size of the original. (Unfortunately, I can no longer find the source code or the executable for that converter, so you'll need to take my word on it.)

The only instance I know of binary tokens being used in a driver dates back to the early 90s, when I was told by someone on the Adobe driver team that the *AdobePS* driver for the old Macintosh OS generated binary tokens when you printed directly to a PostScript device. If you printed to a PostScript file, the driver converted the binary tokens to normal ASCII PostScript, which was saved to disk. In the nature of things, I've not been able to confirm this directly, but it seems a reasonable approach to take.

I'd be curious to know if any other drivers—modern ones, preferably—are generating binary tokens. Anyone know?

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

Sorry there aren't a lot of classes scheduled through July. I've an extremely busy on-site schedule, so I can't conduct many open 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

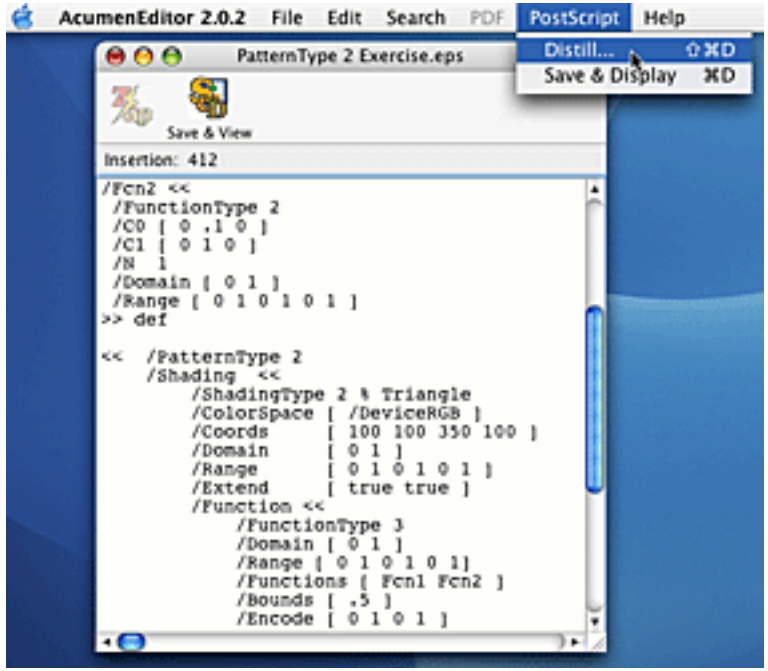
What's New at Acumen Training?

Acumen Editor Does PostScript

The current version of *Acumen Editor* now has features that let it be used in Acumen Training PostScript classes. In particular, the editor will pass your PostScript file to *Acrobat Distiller, Jaws PDFCreator,* or *GhostScript,* displaying the resulting PDF file, if any.

There have also been some new features (such as Block Comment) and the usual round of bug fixes (and new bug insertions, no doubt).

All future Acumen Training PostScript classes are being taught using Acumen Editor.



The current version of *Acumen Editor* is available on the Acumen Training Resources page. The software requires 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? Did reading it reduce your mental acuity so that you no longer finish your sentences when speaking?

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:

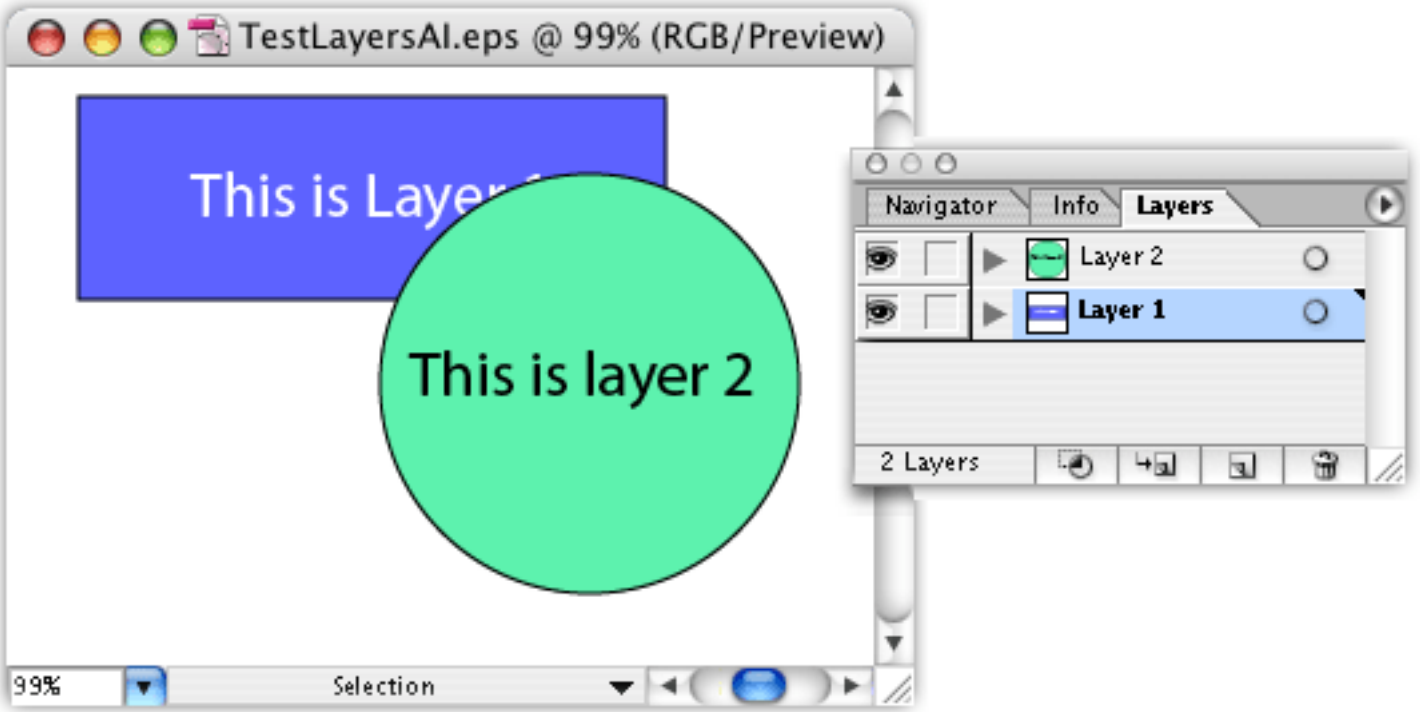
journal@acumentraining.com

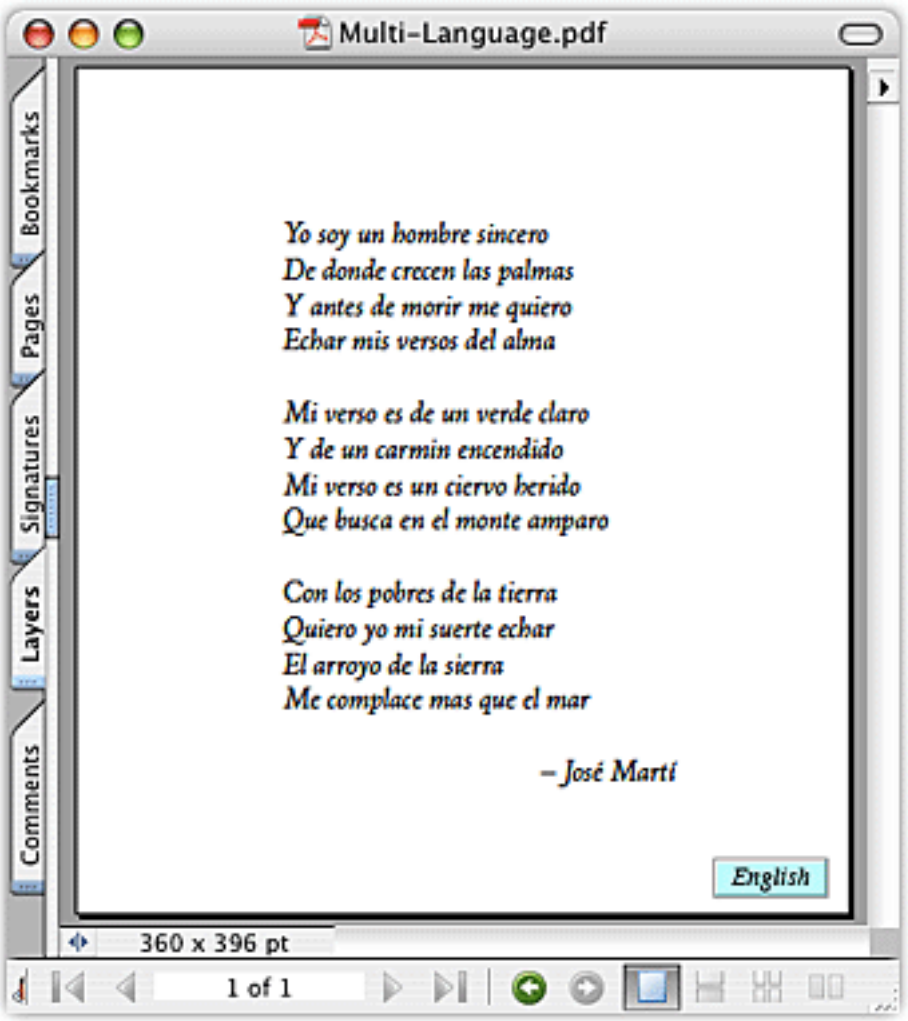
Return to Menu

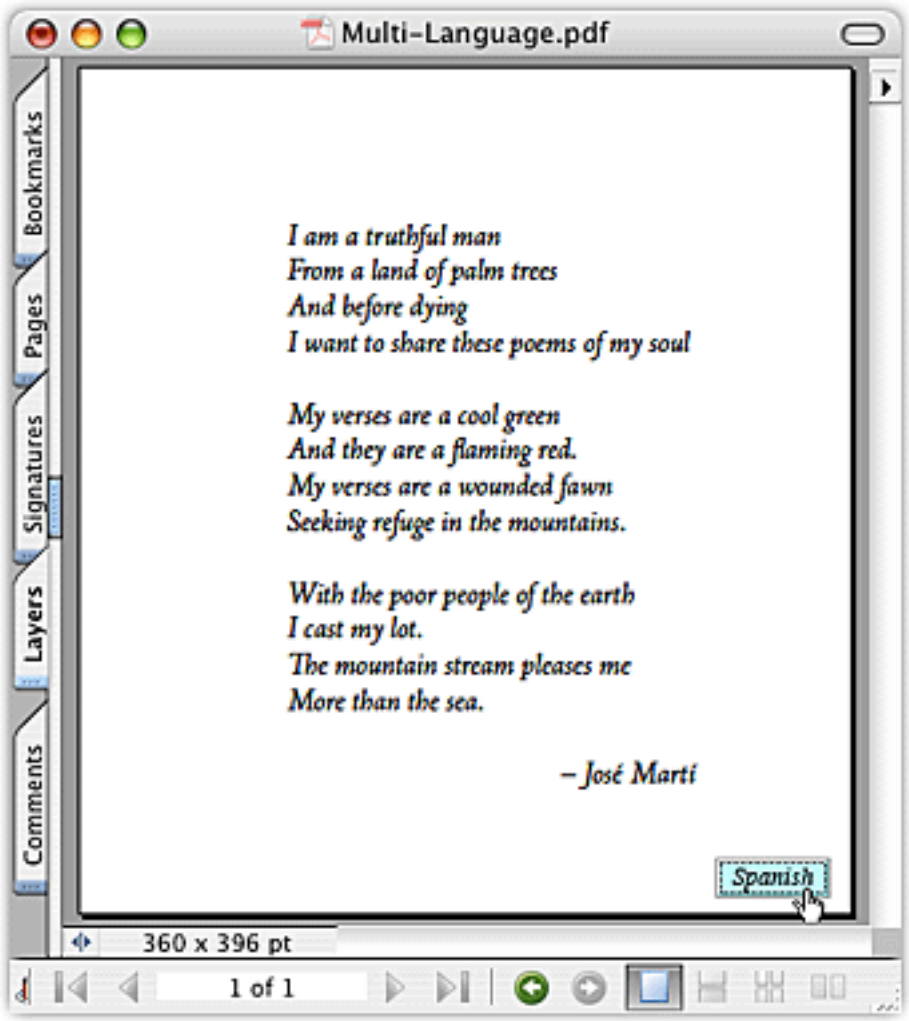
APPENDIX F

System Name Encodings

INDEX	NAME	INDEX	NAME	INDEX	NAME
0	abs	23	concat	46	cvn
1	add	24	concatmatrix	47	cvr
2	aload	25	copy	48	cvrs
3	anchorsearch	26	count	49	cvs
4	and	27	counttomark	50	cvx
5	arc	28	currentcmykcolor	51	def
6	arcn	29	currentdash	52	defineusername
7	arct	30	currentdict	53	dict
8	arcto	31	currentfile	54	div
9	array	32	currentfont	55	dtransform
10	ashow	33	currentgray	56	dup
11	astore	34	currentgstate	57	end


TestLayersAI.eps @ 99% (RGB/Preview)
This is Layer
This is layer 2
99%
Selection
Navigator
Info
Layers
Layer 2
Layer 1
2 Layers


Multi-Language.pdf
Bookmarks
Pages
Signatures
Layers
Comments
Yo soy un hombre sincero
De donde crecen las palmas
Y antes de morir me quiero
Echar mis versos del alma
Mi verso es de un verde claro
Y de un carmin encendido
Mi verso es un ciervo herido
Que busca en el monte amparo
Con los pobres de la tierra
Quiero yo mi suerte echar
El arroyo de la sierra
Me complace mas que el mar
– José Martí
English
360 x 396 pt
1 of 1


Multi-Language.pdf
Bookmarks
Pages
Signatures
Layers
Comments
I am a truthful man
From a land of palm trees
And before dying
I want to share these poems of my soul
My verses are a cool green
And they are a flaming red.
My verses are a wounded fawn
Seeking refuge in the mountains.
With the poor people of the earth
I cast my lot.
The mountain stream pleases me
More than the sea.
– José Martí
Spanish
360 x 396 pt
1 of 1