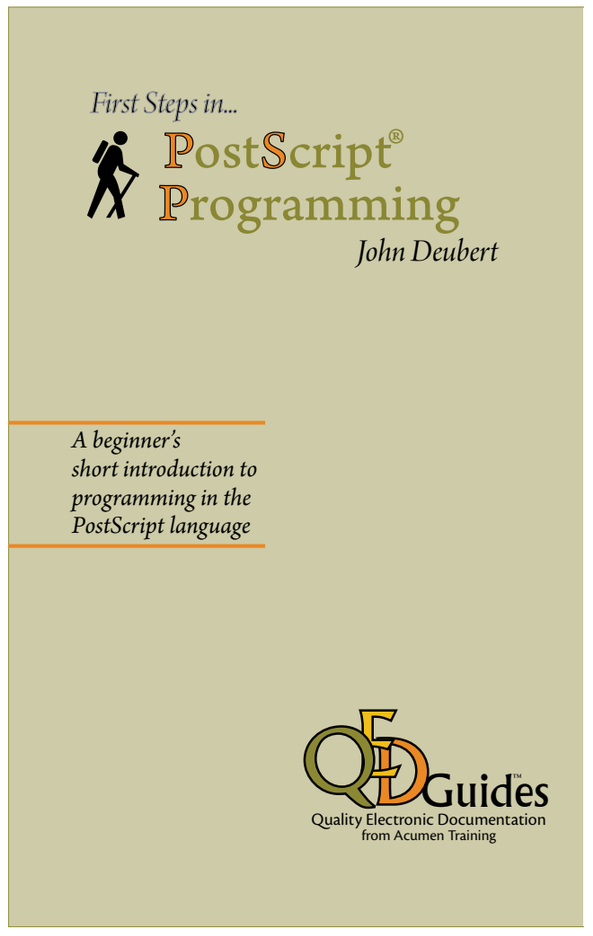
Table of Contents

Acumen Training & Services
Creative Expertise in PostScript, Acrobat, and PDF



Deep Copying PostScript Objects

As you know, arrays, dictionaries, and strings in PostScript are called *composite objects* because their values reside in VM; an array object on the stack, for example, contains a reference (think of it as a pointer into VM) to the array's actual value. This is as distinct from a simple object, such as an integer, that is self-contained, its value living within the object, itself (**Figure 1**).

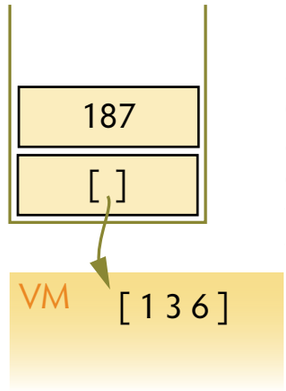


Figure 1. Simple objects, such as integers, are self-contained, whereas arrays and other composite objects contain a reference to the place in VM where their values reside.

When you do a `dup` of an array, what you are really duplicating is the array *object;* you now have two objects on the stack with pointers to the same place in memory (**Figure 2**). You can demonstrate this is so with the following code:

```
/array1 [ 1 3 6 ] def

/array2 array1 dup exch pop def        % array2 is a dup of array1

array2 1 16 put    % Replace one item in array2
array1 ==          % Examine the arrays
array2 ==
```

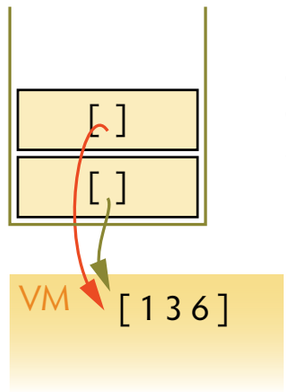


Figure 2. The `dup` operator produces two objects that point to the same place in memory.

The above code sends the following to stdout (the log file if you're using Acrobat Distiller):

```
[1 16 6]    ← Array 1's contents
[1 16 6]    ← Array 2's contents
```

Changing `array2` also changed `array1`, because they are references to the same array in VM.

In most cases, the fact that `dup` results in two references to the same value has no effect on your program. But it does occasionally happen that you need to make a *deep copy* (or *clone*, if you prefer) of the array, creating a new array whose value is separate from, but identical to, the original (**Figure 3**). This is simple enough in principle, but becomes a bit complicated if you need to make a deep copy of an array whose contents include other arrays that themselves need to be deep copied.

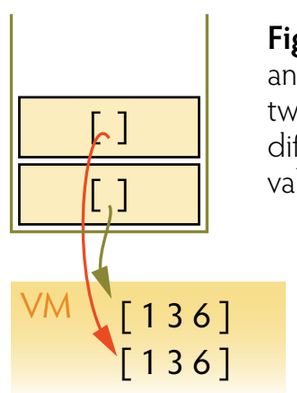


Figure 3. Deep copying an object produces two objects pointing to different, but identical, values in VM.

This recursive deep copy is what we'll be discussing in this article.

First Pass: The *copy* Operator

PostScript has an operator that will do what we want, up to a point. The `copy` operator takes two strings, arrays, or dictionaries and copies the contents of the first item into the second, leaving the second object (with its new contents) on the stack.

Thus, the simplest way of deep copying an array is simply:

```
% File: simple copy.ps
/array1 [ 1 3 6 ] def       % <== this is the array we want to copy
array1 dup length array     % make a new array the same length as the old
copy                        % copy the original array into the new one.
/array2 exch def
```

Again, we can confirm that `array2` is an independent copy of `array1` by changing the second array and seeing if the change carries into `array1`:

```
array2 1 16 put     % Replace item #1 in array2
array1 ==           % Examine the arrays
array2 ==
```

This time, we see the following text written to stdout:

```
[1 3 6]
[1 16 6]
```

The change we made to `array2` did not appear in `array1`; they are references to different objects.

The problem with this method is that it goes only one level deep. That is, if you have an array or dictionary that contains another array or dictionary, `copy` will copy only the reference to that inner object (**Figure 4**). Generally, if we are bothering to do a deep copy at all, we want to deep copy any descendant arrays or dictionaries, as well.

For that, we need to do a little more work.

The Sample Code

As always, the sample file for this article can be downloaded from the Acumen Training Resources page; look for *DeepCopy.zip*.

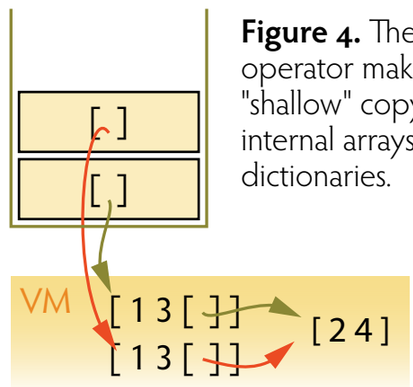


Figure 4. The copy operator makes only a "shallow" copy of any internal arrays, strings, or dictionaries.

Second Pass: A Recursive DeepCopy Procedure

We need to do a recursive deep copy that will examine each element in `array1` and, if that element is an array, do a recursive deep copy on it. For this, we'll need to define a procedure that we can call recursively:

```
% File: CopyArray.ps
/DeepCopyArray      % [ array ]  =>  [ array' ]
{
    [ exch          % Push a mark on the stack & bring original array to top
    {               % forall: for each element in the array...
        dup type /arraytype eq % is it an array?
        { DeepCopyArray } if       % yes: do a deep copy; otherwise, leave it alone
    } forall        % After the loop, all the new array's contents are on the stack.
    ]               % Finally, create the array
} bind def

/array1 [ 1 3 [ 2 4 ] ] def   % Let's try it out
/array2                       % Deep copy array1
    array1 DeepCopyArray
    def

array2 2 get 0 14 put      % Change the contents of array2's inner array

array1 ==                  % & confirm that array1's inner array is unaffected
array2 ==
```

As expected, this writes the following to stdout:

```
[1 3 [2 4]]
[1 3 [14 4]]
```

The two arrays have separate sub-arrays; changing one has no effect on the other.

Deep Copying Dictionaries

Doing a deep copy of a dictionary is almost exactly the same process:

```
% File: CopyDict.ps
/DeepCopyDict       % <<dict>> => <<dict'>>
{
    <<  exch              % Push a mark on the stack & bring the dictionary to the top
    {                           % forall: for each k-v pair
        dup type /dicttype eq   % is the value a dict?
         { DeepCopyDict } if    % yes: do a deep copy
    } forall          % The loop exits with the new dict's contents on the stack.
    >>                % Create the new dictionary
} bind def

/dict1 <<       % Try it out
    /a 1    /b [ 1 2 3 ]
    /c  <<
        /key1 4
        /key2 5
        /key3 6
    >>
>> def

/dict2 dict1 DeepCopyDict def  % Deep copy dict1

dict2 /c get /key1 26 put      % Change the inner dict in dict 2

dict1 /c get /key1 get ==      % Confirm dict1's inner dict is unchanged
dict2 /c get /key1 get ==
```

The stdout text from this program is

```
4
26
```

indicating that changing the internal dictionary in `dict2` didn't affect `dict1`'s internal dictionary.

However!

We still have a problem here. Note in the previous example that entry `b` in both dictionaries is an array; `DeepCopyDict` as written will correctly deep copy dictionaries, but will copy only the reference for any internal arrays.

We need a smarter procedure; one that will take *any* object and deep copy it as needed. We need a universal procedure that we can call blindly with any object and have it do the right thing.

A Universal DeepCopy Procedure

And here it is. The code below defines a DeepCopy procedure that takes an object of any sort from the stack and either does a recursive deep copy of it or does nothing at all, depending on the type of the object.

```
% File: DeepCopy.ps
/$CopyProcs     <<      % Procedures for each PostScript data type
    /arraytype  {
        [ exch { DeepCopy } forall ]
    } bind

    /dicttype   { << exch { DeepCopy } forall >> } bind

    /stringtype { dup length string copy } bind

    /default { }
>> def
```

```
/DeepCopy  % obj => obj'
{
    dup type                    % What kind of object do we have?
    $CopyProcs exch
    2 copy known                % does the typename exist in $CopyProcs?
    not { pop /default } if     % no: replace name with /default
    get                         % Get the corresponding proc from $CopyProcs...
    exec                        % ...and execute it
} bind def

% Try it out with an array...
/Array1
[ 1 2 3 [ (A) (B) (C) ] 4 << /z 4 /Gee (Whiz) >> ] def

Array1 DeepCopy
/Array2 exch def

Array2 3 get 0 (Mouse) put
Array1 ==
Array2 ==

% ... and with a dictionary
/Dict1
<<
    /A 1
    /B 2
    /C <<
        /D (Mouse)
```

```
        /E 17
    >>
    /D [ 1 2 3 ]
>> def


/Dict2 Dict1 DeepCopy def
Dict2 /C get /D (Elephant) put
Dict2 /D get 1 100 put

Dict1 /C get /D get ==
Dict2 /C get /D get ==
() =
Dict2 /D get 2 -17 put

Dict1 /D get ==
Dict2 /D get ==
```

This `DeepCopy` procedure uses the PostScript `type` operator to determine the kind of object that is on the stack.

```
obj   type   →   /typename
```

`Type` returns a predefined name on the stack, one of those listed in **Table 1**.

Let's step through the `DeepCopy` procedure in detail.

Table 1 Standard PostScript type names

arraytype	gstatetype	operatortype
booleantype	integertype	packedarraytype
dicttype	marktype	realtype
filetype	nametype	savetype
fonttype	nulltype	stringtype

Step by step

```
/$CopyProcs  <<      % Procedures for each PostScript data type
  /arraytype  { [ exch { DeepCopy } forall ] } bind
  /dicttype   { << exch { DeepCopy } forall >> } bind
  /stringtype { dup length string copy } bind
  /default { }
>>
```

We start by defining a dictionary, `$CopyProcs`, that holds a procedure for each data type we are going to deep copy. Each key is one of the standard type names from Table 1; the value associated with that key is a procedure that deep copies an object of that data type.

In our case, we have three of these procedures, associated with the names `/arraytype`, `/dicttype`, and `/stringtype`. These procedures are identical to those from earlier in this article except that they recursively call `DeepCopy`, rather than a type-specific procedure, like `DeepCopyArray`.

Our `stringtype` procedure simply calls `copy`, since string contents can only be characters, which don't require deep copying.

Finally, we also define a procedure named `default`, which we'll use for objects other than arrays, dictionaries, and strings. Note that this default procedure does nothing at all; it simply lets the object remain on the stack.

```
/DeepCopy      % obj => obj'
{
```

We then define the `DeepCopy` procedure. This will take an object from the stack and return an object; the return value is either the unchanged original object or a cloned string, array, or dictionary.

```
dup type       % stack: obj /typename
```

The first thing `DeepCopy` does is duplicate its argument and then execute `type`, obtaining a type name for the object.

```
$CopyProcs exch  % stack: obj <<$CopyProcs>> /typename
```

We then push the `$CopyProcs` dictionary on the stack and then bring the typename back to the top.

```
2 copy    % stack: obj <<$CopyProcs>> /typename <<$CopyProcs>> /typename
```

Here we are using `copy` in its stack-operator guise; it takes a number as its argument and makes a copy of the top that-many items on the stack, not including its own argument (the 2, in this case). Thus, our call to `copy` duplicates the top 2 items on the stack: `$CopyProcs` and the typename.

```
known     % stack: obj <<$CopyProcs>> /typename bool
```

The `known` operator takes a dictionary and a key and returns a boolean, true if the key exists in the dictionary, false otherwise. In our case, this boolean will indicate whether or not the typename exists as a key in `$CopyProcs`.

```
not { pop /default } if % stack: obj <<$CopyProcs>> /name
```

If the boolean value returned by known is false, we'll discard the typename and replace it with the name `/default`.

```
get   % stack: obj {proc}
```

We get the procedure corresponding to our typename (or `default`) from `$CopyProcs`…

```
exec
```

…and then execute that procedure.

```
} bind
```

And that ends the `DeepCopy` procedure.

Cool, eh?

I won't step through the rest of the sample program, since it merely confirms that `DeepCopy` works as advertised.

So, When Would I Use This?

Admittedly, not very often. I routinely use DeepCopy when I'm doing a pseudo-object-oriented program design (such as I described in the previous *Journal).* I inevitably use dictionaries to represent my objects and creating a new object involves doing a deep copy of a generic template dictionary.

As is true with so many things in life, when you need it, you usually need it badly.

Signing Documents in Acrobat 11

Among the coolest things that have happened to Acrobat in the past few versions is the dramatic improvement in the ease of signing PDF documents.

In the early days, the only electronic signature built into Acrobat was Adone's "self-signed" security. To sign a document, you needed a fair amount of preparation; you needed to:

1. Create a digital ID in your copy of Acrobat.
2. Create a digial certificate from this ID.
3. Email this certificate to everyone to whom you expect to send a signed document.
4. Call those people and step them through the somewhat complicated process of importing the certificate to create a "trusted identity" on their computer. (This is not an easy task for a novice.)

Now, at last, you're finally ready to sign the document.

Acrobat 11 has seriously simplified the process of signing a document. The old way, outlined above, is still available, but there are a few additional signature types that are secure enough and are vastly simpler and quicker to use, and, crucially, don't require you to teach someone else how to carry out a complex task.

How secure is "secure enough?"

People get (understandably) paranoid about the security of electronic signatures. How can someone be sure that the signature on a document page is really yours?

For practical purposes, the standard of security to which electronic signatures should be compared is that associated with traditional ink-on-paper.

For centuries, a "signature" was a handwritten rendering of your name, something easily forged by other people. Not very secure, you might think, but commerce based on this method of identification is centuries old and we muddled through somehow.

So the question is: Is the new method of signing a document in Acrobat at least as verifiable as the traditional, handwritten signature?

The answer is "yes," as you'll see in this article.

The Sign Pane

The new signature mechanism is accessed through the new Acrobat Sign pane (**Figure 1**). This pane lets you select among four different ways of signing your document:

Add Text Type arbitrary text (a date, your printed name, etc.) onto the page.

Add Checkmark Add a checkmark character to the page, letting you "fill in" a printed checkmark box.

Place Initials Place your initials on the page; you may either type your initials or draw them with a graphics tablet.

Place Signature Place your signature on the page; this may be a scanned image of your handwritten signature.

Let's look at each of these in turn.

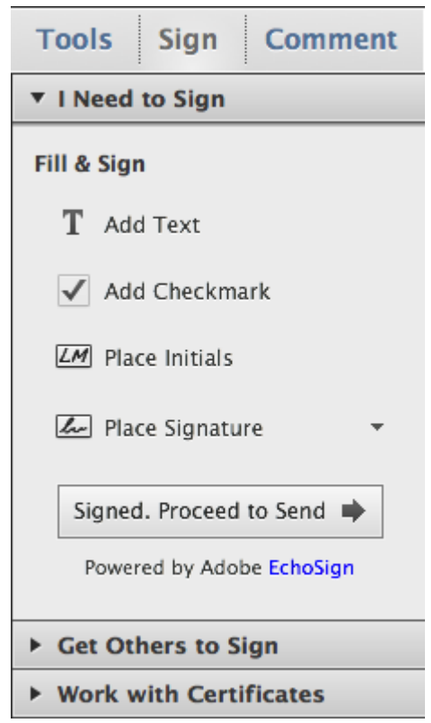


Figure 1. All of the Signature functions are accessed through the aptly-named Sign pane.

Add Text

This tool lets you place any text you like anywhere you like on the page; it's identical, so far as I can tell, to the Add Text tool in the Content Editing pane. It's not intended for a signature, but rather for ancillary text, such as a date or the "Print your name" line on a printed form.

When you select this tool, the cursor turns into a fancy version of the classic insert-text I-beam and a text palette appears (**Figure 2**), allowing you to specify font, point size, style, and other parameters. Again, this is identical in form and function to the Add Text tool. Just click on the page and start typing.

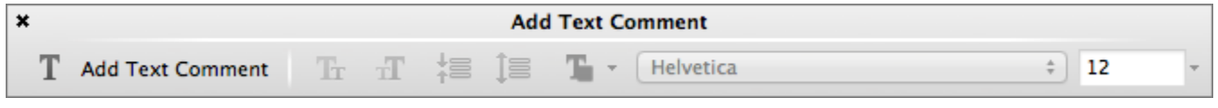


Figure 2. When you select the Add Text tool, you are presented with a floating pallet containing common formatting tools.

Add Checkmark

Little to tell here and no surprises at all. The mouse pointer turns into a translucent checkmark; click on the page and Acrobat will add a checkmark to the page. This checkmark is initially surrounded by a rectangle with handles that let you rotate and resize it as needed to match the pre-printed text on the page, as at right.

Place Initials

This option is a little more elaborate. When you click on this tool in the Sign pane, Acrobat presents you with a dialog box (**Figure 3**) that lets you specify how you want to place your signature. Your choices are two: type your initials or draw them (**Figure 4**, far right).

Figure 3. This dialog box lets you place your initials on the page.

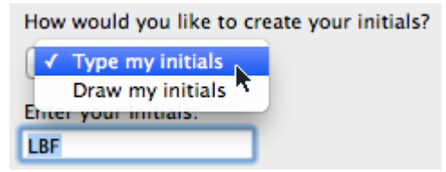


Figure 4. You can type your initials or try to draw them.

Typing

Typing your initials is just what you'd expect: type one or more characters into the text field; as you do so, the characters will appear in the graphics field that makes up of the bottom half of the dialog box, displayed in one of several script fonts, as in Figure 3. You can cycle through the available fonts by clicking on the "Change initials style" button.

Finally, click on the Accept button and Acrobat lets you click on a location on the page, applying the "handwritten" initials to the page. You are presented with handles that let you reposition, resize, and rotate the graphic as needed to match the page's background artwork (**Figure 5**).

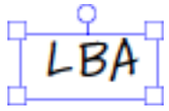


Figure 5. Your initials start out with handles that let you totate and resize them.

Drawing

If you decide to draw your own initials (by selecting "Draw my initials" in Figure 4's pop-up menu), you can draw directly into the graphic panel in the dialog box using your track pad or mouse. Presumably you would do this only if you have a graphics tablet or its equivalent, since trying to do this with a mouse (or even a finger on your trackpad) yields pretty unappealing results (**Figure 6**).

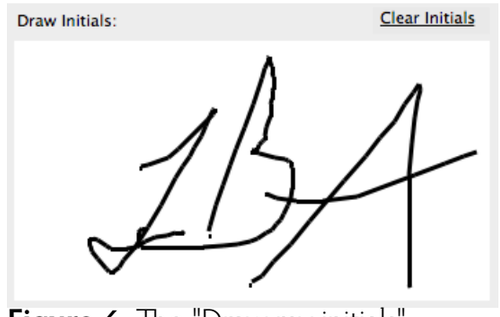


Figure 6. The "Draw my initials" feature yields unpleasant results unless you have a graphics tablet.

As before, clicking Accept allows you to place your initials on the page.

Clear Saved Initials

Once you've created a set of initials, by either typing them or drawing them, Acrobat saves that artwork internally so that you can use those same initials repeatedly on any PDF document that you open with that particular copy of Acrobat.

To change the artwork, you need to select "Clear saved initials" in the drop-down menu that appears once

you have saved initials (**Figure 7**). The next time you try to place initials on the page, Acrobat will again present you with the Place Initials dialog box (Figure 3).

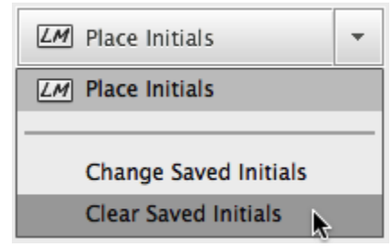


Figure 7. A drop-down menu becomes available once you have created your initials artwork

Place Signature

Place Signature works almost exactly the initials function. When you click on the tool, Acrobat presents you with the Place Signature dialog box, which looks like the twin of our earlier Initials box (**Figure 8**).

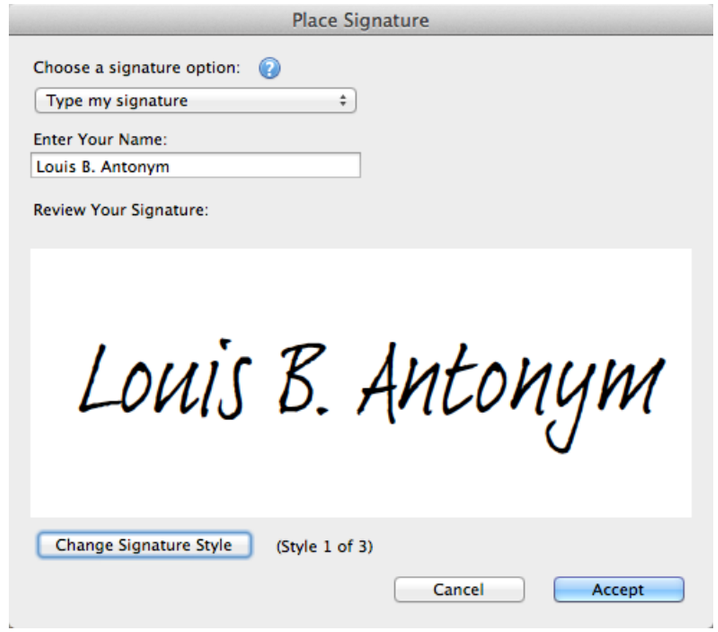


Figure 8. The Place Signature dialog box lets you create the artwork for your signature on the PDF page.

The Place Signature dialog box adds two methods of creating a signature (**Figure 9**) to the two available for making initials.

Use an image This lets you scan your actual, handwritten signature and the result as your Acrobat signature. (This is what I do.)

Use a certificate This let you apply a previously-created Adobe Self-Sign electronic signature to the Acrobat page. I'm not going to discuss these here, though I do feel a future Journal article coming on. Check out pretty much any how-to book on Adobe Acrobat (including my own *Quickstart Guide to Adobe Acrobat X)* for details on how to set these up and why you'd want to.

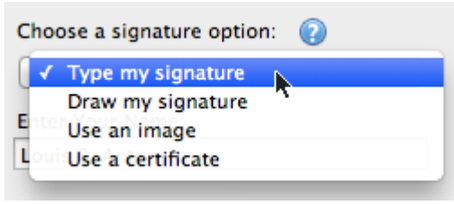


Figure 9. You can create signature artwork four different ways.

As with initials, Acrobat remembers your selection so that you can apply the same signature to future PDF files with minimal effort.

Transparency

By the way, when you choose an image to serve as your signature artwork, Acrobat places that image on the PDF page with a transparent background (**Figure 10**). This is greatly important, of course, since it allows the signature to be placed on top of the page's lines, labels, and other background artwork in a completely natural-looking way.

Figure 10. Signature and initials artwork is placed on the PDF page with the Background set to "transparent."

Once You've Signed the Document

Having signed the document, the simplest thing to do is close the Sign pane, save the document, and email it to the appropriate person.

You can also click on the Proceed to Send button in the Sign pane (**Figure 11**); this will take you to Adobe's EchoSign service, which will forward the document to your recipient. EchoSign is actually a pretty useful product; it's free for up to five signed documents per month, which is adequate for many very small businesses (including my own). You should look into it.

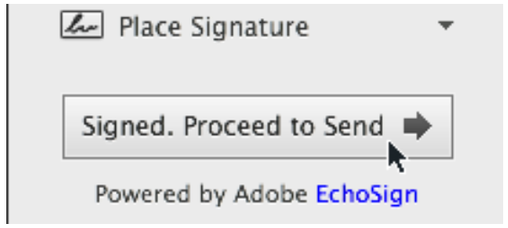


Figure 11. If you wish, you can send your document to its receiver using Adobe's EchoSign service.

Automatic Form Detection

When you open a PDF file, Acrobat may present you with a banner at the top of the document page saying that it has detected signature lines (**Figure 12**) and offering to open the Sign pane.

Clicking on the leftmost graphic will collapse the banner into a Navigation icon, as in the figure.

You can make this go away permanently in the Preferences by selecting *Preferences>Forms>Always hide forms document message bar.*

You're welcome.

The Net Result

I use Adobe Acrobat for very nearly all the forms I receive, both printed and electronic. Crucial to this is the ability to sign or initial a page in a form that the people who sent the form will accept. The current incarnation of Acrobat has made this as effortless as it's ever been.

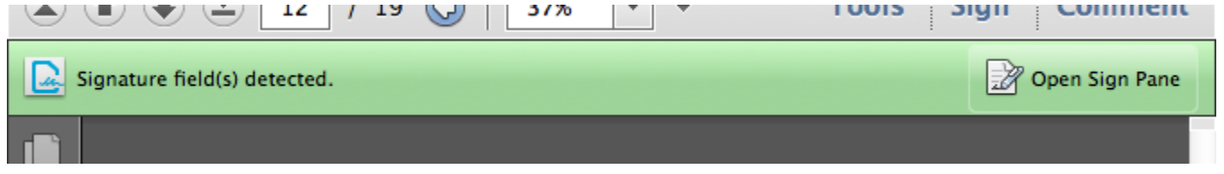


Figure 12. Acrobat tries to detect when a document needs to be signed and tells you about it. Clicking on the icon collapses the banner, as at left.

Schedule of Classes, September – November 2013

At right are the dates of Acumen Training's upcoming classes in Orange County, California. Click on a class name to see the description of that class on the Acumen Training website.

O.C. and On-Site

These classes are taught in Orange County, California and on-site at corporate sites world-wide.

Please see the Acumen Training web site for more information, including an up-to-date schedule.

Class Fee

Class fees are as follows:

- *PostScript Foundations* $2,000
- *PDF 1:* $2,000
- *Troubleshooting PostScript* $1,500
- *Support Engineers' PDF* $1,000

There is a 10% discount for signing up three or more students.

Note that if you have four or more students that need to take a class, it will almost certainly be cheaper to arrange an on-site class.

PDF Classes

PDF 1: File Content and Structure	Sept 16-19	Oct 21-24	
PDF 2: Advanced File Content			Nov 11–14
Support Engineers' PDF		Oct 17-18	

PostScript Classes

PostScript Foundations	Sept 9-13		Nov 4–7
Advanced PostScript			
Variable Data PostScript			
Troubleshooting PostScript		Oct 14-16	

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: 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: www.acumentraining.com/register.html

email: john@acumentraining.com

telephone: 949-248-1241

mail: 24996 Danamaple, Dana Point, CA 92629

On-Site Classes

Information regarding classes on corporate sites is available at www.acumentraining.com/Onsite.html. These courses are taught throughout the world; for additional information on classes outside the United States, go to www.acumentraining.com/OnsitesWorldWide.html.

Back issues

All issues of the *Acumen Journal* are available at the Acumen Training website: www.acumenjournal.com/AcumenJournal.html

What's New at Acumen Training?

New First Steps Guides in the Works: PostScript and PDF

The next two volumes in the First Steps series of booklets are in the works, covering PostScript programming and the pdf file format. Expect to see them toward the end of the year!

They'll each be a steal at $1.99!