Table of Contents

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

The Acrobat User

JavaScript Reports

Acrobat JavaScript has a Report mechanism that lets you create a report of any data you choose. You have formatting tools similar to *printf;* primitive, but surprisingly useful.

PostScript Tech

PostScript User Paths

User paths are nearly unknown to most PostScript programmers. They are easy to construct and use and certainly *look* as though they should be useful.

Class Schedule

Aug-Sep-Oct

What's New?

JavaScript e-book is selling well. Makes me happy.
Also, I've been fussing around with the *Journal's* format.

Contact John at Acumen Training

Telephone number, email address, postal address

User Paths

User Paths are a curious part of the PostScript language. They're easy to construct and use and, by golly, they *look* as though they should be useful, but for the life of me I've never found a problem for which they are the best solution. This month, I thought I'd describe them in a bit of detail and see if anyone out there can suggest a circumstance that cries out for their use.

"What's a User Path?" You Ask

A user path is a procedure body that draws an unpainted path; that is, it contains calls to **moveto**, **lineto**, **arc**, etc., but no **stroke**, **fill** or other painting operators:

```
{
    0 0 85 81 setbbox    % We haven't talked about setbbox yet
    0 50 moveto
    85 50 lineto
    16 0 lineto
    42 81 lineto
    69 0 lineto
    closepath
}
```

To use this procedure, you hand it to one of the specialized user path operators: **ufill**, **ustroke**, or **uappend**:

```
{ ... } ufill
```

As you might imagine, **ufill** and **ustroke** paint the path onto the current page and **uappend** adds the path to the existing current path.

Easy.

A PostScript fragment that uses the above user path would look something like this:

```
/StarPath {
    0 0 85 81 setbbox    % We haven't talked about setbbox yet; patience.
    0 50 moveto
    85 50 lineto
    16 0 lineto
    42 81 lineto
    69 0 lineto
    closepath
} def

//StarPath ufill       % The // keeps the StarPath procedure from executing immediately.
```

This PostScript code draws a five-pointed start at the origin, as in **Figure 1.**

Figure 1. The user path we are discussing draws a five-pointed star at the origin.

User Path Requirements

There are a few requirements attached to the use of user paths.

- *Operator restrictions* - A user path does not have access to the entire PostScript language; in fact, it may contain only the operator calls listed in **Table 1.** This shouldn't cramp your style much, since the sole purpose of a user path is to construct a path.

  The operator calls may be in the form of the operator names or as operator objects, as would be the case if you called the **`bind`** operator when creating the user path procedure body:

```
/StarPath    {
    ...
} bind def
```

Table 1 User Path Legal Operators

ucache	lineto	arc
setbbox	rlineto	arcn
moveto	curveto	arct
rmoveto	rcurveto	closepath

- *No non-operator names* - A user path may not contain any names other than those listed in Table 1. This means you cannot perform calculations within the procedure body nor may you reference variables or procedures. The user path may contain only literal numbers and the 12 names listed in Table 1.

- *Must start with setbbox* - The first call in a user path procedure must be a call to the **setbbox** operator:

```
xll yll xur yur setbbox
```

This operator takes four numbers as its arguments, representing the *x* and *y* coordinates of the lower-left and upper-right corners of the path's bounding box, that is, the rectangle that encloses all of the points that go into defining the path (**Figure 2**). All of the points used in constructing the path—including Bezier control points—must lie on or within this rectangle.

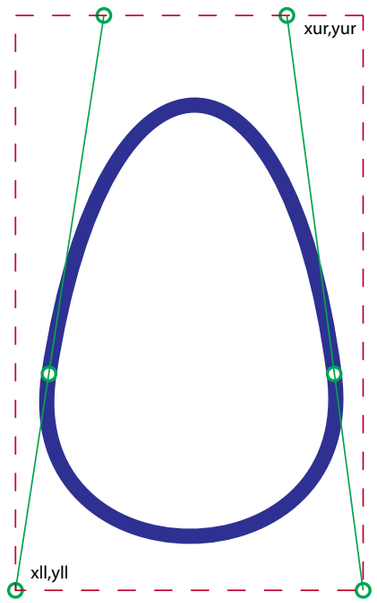


Figure 2. The user cache *bounding box* encloses all of the points that go into the path definition, including Bezier curve control points.

Cached User Paths

If you include the **ucache** operator in your user path procedure body, this instructs the painting operator (**ufill** or **ustroke**) to cache the painted path or fetch the painted path from the cache if it's already there. This promises to speed up the painting of repeatedly-used paths significantly.

If used, **ucache** must be the first operator in the user path; our StarProc path becomes:

```
/StarPath {
    ucache
    0 0 85 81 setbbox
    0 50 moveto
    ...
    closepath
} def

//StarPath ufill
```

Why is there a Double-Slash in our code?

You have certainly noticed that in our call to **ufill** we double-slashed the name **StarPath**:

```
//StarPath ufill
```

We needed to do this because **StarPath** is defined as a procedure body. If we had omitted the double-slash

```
StarPath ufill
```

PostScript would have looked up the name **StarPath** immediately and then executed the associated procedure; **ufill** would have found an empty stack and died with a **stackunderflow** error.

Double-slash, you remember (*yes*, you do), is the "evaluate immediately" delimiter; when a name is preceded by **//**, the interpreter (actually, the scanner) immediately looks up the name and pushes its value—a procedure body, in our case—on the operand stack.

There is an alternative to the double-slash

As an alternative, we could have defined **StarPath** as a literal array, rather than a procedure body (procedure bodies are executable arrays, you may recall). This doesn't particularly simplify our code:

```
/StarPath {
    ucache
    0 0 85 81 setbbox
    0 50 moveto
    ...
    closepath
} cvlit def

StarPath ufill
```

We got rid of the double-slash here, but we had to add a call to **cvlit** to the definition of the user path.

It might initially seem that we could make our user path as a literal array by simply replacing the braces with brackets:

```
/StarPath [
    ucache
    ...
    closepath
] def

StarPath ufill
```

However, remember that the PostScript interpreter is operating exactly as always in between brackets; if we used brackets, the interpreter would immediately execute all the bracketed PostScript code. (See the sidebar for a reminder of how the square brackets work to create an array.)

To get PostScript to defer execution of the operators in the user path definition, we need to enclose them in braces and then convert the resulting procedure body to a literal array with the **`cvlit`** ("convert to literal") operator.

Not better; not worse. Just different.

How Arrays are Made

Back in the *PostScript Foundations* class, we learned that the open and close brackets are actually a pair of PostScript operators that together create an array.

Open bracket pushes a mark object on the stack and that is all it does; the PostScript interpreter continues executing the incoming PostScript stream as normal.

Close bracket takes everything off the stack back to the topmost mark and constructs an array from those objects.

Thus, the following code creates a 2-element array:

```
[
   100
   200 300 add
]
```

The resulting array contains the numbers **`100`** and **`500`**; the **`add`** between the brackets executed as usual.

So, What's This Good For?

And that brings us back to my original question: what problem do user paths solve in today's World of PostScript?

In principal, the benefits that user paths bring to the table are:

- *Caching* - If you use the **`ucache`** operator, the painted path may be cached.

  Or may not, unfortunately; caching is effectively optional and may not be implemented in a particular PostScript interpreter. Also, if you are converting your PostScript to PDF with Acrobat Distiller, the user path disappears altogether; Distiller converts it to a series of repeated instances of **`moveto`**, **`lineto`**, **`curveto`**, etc. (or, rather, the PDF equivalents: **`m`**, **`l`**, **`c`**, etc.)

- *Dependability* - Since user paths can't have anything in them but numbers and native path operators, there is no chance that the results will be changed by redefinitions of those operators.

  This seems uncompelling, somehow. If I'm writing the PostScript program, then I'm perfectly capable of the degree of discipline it takes to keep from sabotaging my own code. If the path or painting operators have been redefined by someone else—by code pre-pended to my own PostScript, perhaps—then presumably there's a good reason for it; perhaps my PostScript document is being imposed or otherwise post-processed.

The alternatives to implementing a repeatedly-used path as a user path are:

- *A standard PostScript procedure* - Just define the same procedure body that constructs the path (minus the calls to **ucache** and **setbbox**) as a regular, ol' procedure body that you call as usual.

```
/StarPath {
    0 50 moveto
    ...
    closepath
} def

StarPath fill
```

One advantage to this method is that you can place the painting operator in the procedure (so it ends with a call to **stroke** or **fill**).

- *A PostScript Form* - Make a PostScript form that draws the painted path. This is a bit more work, but is more-or-less guaranteed to be cached on a PostScript device.

```
/StarForm    <<
    /FormType 1
    /FormBBox [ 0 0 85 81 ]
    /PaintProc  {
        pop
        0 50 moveto
        ...
        closepath
    }
    /Matrix [ 1 0 0 1 0 0 ]
>> def

StarForm execform
```

Both of these alternatives are easy and are far more commonly used.

So why use user paths?

That's the question I pose to you, my readers: have you ever used this mechanism and, if so, why? What made user paths the perfect solution to the problem you were solving?

As I said when we started, they certainly *look* as though they should be better than anything else at *something*.

But, what?

The JavaScript Report Object

Try this: type your name into the Text field at right and then click the button.

John | Click Me

If you are reading this with Adobe Acrobat or Reader, what *should* have happened is the software presented you with a page similar to that in **Figure 1,** containing a brief message that incorporates your name.

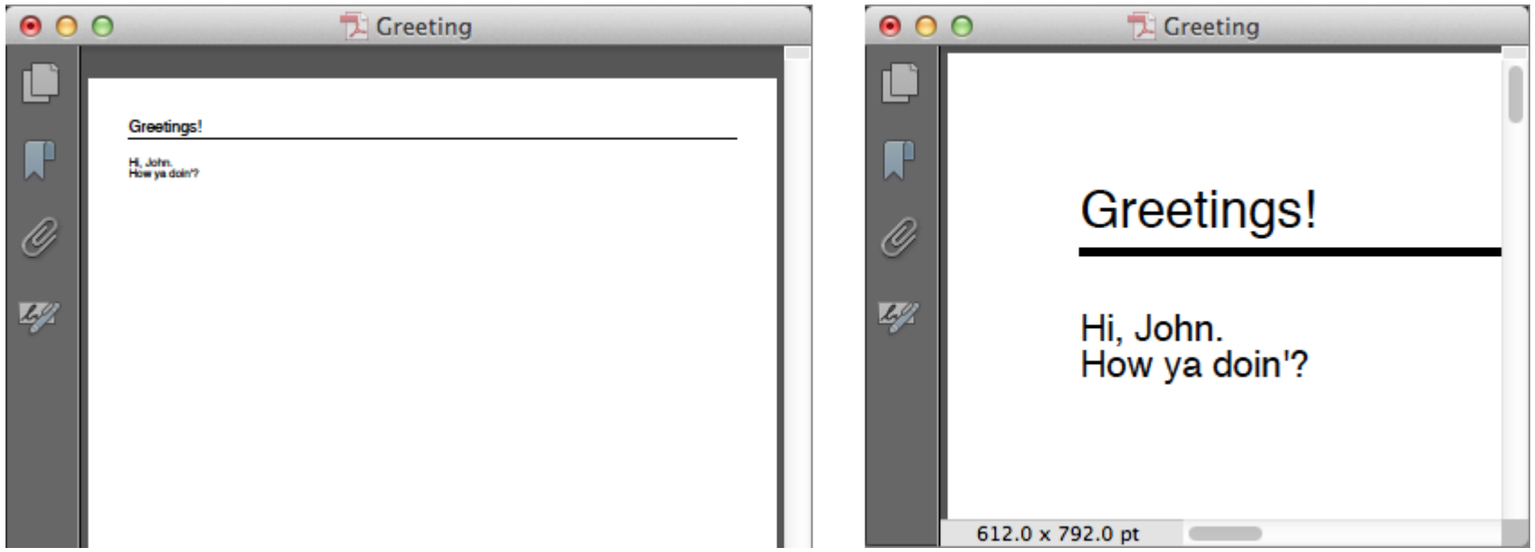


Figure 1. The button on this page uses a Report object to create a "Greetings" document.

Sometimes you need your JavaScript to produce a "written" document, that is, a separate PDF file that presents data that the reader can take away; an obvious example would be a "print a receipt" button.

Acrobat provides a way to do this with its JavaScript Report object. I talked about this object in my original JavaScript book, way back in 2002. I dropped it in my recent re-write of the book and have been feeling guilty about it, so let me make amends by talking about it here.

Using the Report Object

The JavaScript Report object lets you construct a multi-page PDF document and then open it in Acrobat or email it to a specified address. The way you do this is straightforward to the point of primitive. You use a handful of methods to print text, draw horizontal lines, indent, change color, and that's about it. Pages are assembled from top, down; you don't get to place items at arbitrary, randomly-specified positions on the page.

Broadly, there are three steps to creating a report in Acrobat JavaScript:

1. Create the Report object.
2. Draw on the page with repeated calls to the Report object's drawing methods.
3. Create the report's PDF document, either opening it in Acrobat or emailing it to a specified address.

Here's the script that created the report in Figure 1.

```
var rep = new Report()                          // Create a Report object
var name = this.getField("txtName").value       // Get the Text field's value

rep.size = 1.4                                  // Increase the point size
rep.writeText("Greetings!")                     // Draw the page title
rep.size = 1.0                                  // Revert to original point size
rep.divide(2)                                   // Draw a horizontal divider line
rep.writeText("\nHi, " + name + ".")            // Draw a two-line message
rep.writeText("How ya doin'?")
rep.open("Greeting")                            // Open the report as a PDF document
```

This script gets a new Report object and the value of the txtName Text field. It then prints the following:

- The string "Greetings" in a moderately large point size.
- A horizontal dividing line.
- A two-line text greeting.

The script then opens the report as a PDF document named "Greeting."

We'll look at this script in detail in a moment. First, let's review the tools the Report object makes available to us.

The Fuller Scoop

This article assumes you have at least minimal knowledge of Acrobat JavaScript. If you need to acquire this expertise, there are two documents you can read:

- If you have little or no programming experience, I recommend (of course) *Beginning JavaScript for Adobe Acrobat;* This is my own e-book, available at www.acumentraining.com/QEDGuides. This e-book will teach you how to add JavaScript-based features to your Acrobat forms and, along the way, teach you the principles of JavaScript programming.
- If you are an experienced JavaScript programmer, you should go right to Adobe's own, complete documentation by clicking here. This is a programmer's document that assumes you have good knowledge of programming and JavaScript.

Report Object Properties and Methods

The Report object supplies a collection of properties and methods that let us assemble a useful, if not terribly sophisticated, multi-page report. Below is a compendium of the most useful of these; there are a few others, but they can be safely ignored for the moment.

Report Object Properties

Table 1 lists the most useful Report object properties.

Table 1 Report Object Properties

Name	*Value*	*Meaning*
`color`	color	Text and divide color
`size`	number	Text size multiplier
`style`	string	Text style

color

Remember from the *Beginning JavaScript* book that JavaScript defines a Color object that supplies values such as `Color.red` that may be used to define color within your JavaScript code. The following bit of code would print a line of red text.

```
myReport.color = Color.red
myReport.writeText("Was my face red!")
```

size

This, surprisingly, is a multiplier for the "standard text size," which is not explicitly defined, as far as I can tell. It's a floating-point number, so you could make a larger-size subhead with code something like this:

```
myReport.size = 1.4
myReport.writeText("Manufacturer's Warning")
```

style

This lets you set your text to one of two pre-determined styles, specified as strings: `"DefaultNoteText"` or `"NoteTitle"`.

```
myReport.style = "NoteTitle"
myReport.writeText("I am entitled!")
```

Report Object Methods

Here are the methods you will most commonly use to create a report. Note that there are no methods for specifying drawing location on the page; as I said earlier, a report is drawn from the top, down.

Creating a Report Object

Report This is the class' constructor; you create a Report object with the line

```
var myReport = new Report
```

There are optional arguments that let you specify the page margins and paper size, but I'll let you research those on your own.

Drawing on the Page

writeText("text") Draws a string on the page.

indent(pts) Indents (that is, moves to the right) the text that follows by the specified amount, expressed in points (1/72-inch).

You may omit the argument, in which case the following text will be indented the default distance, which is 0.

outdent(pts) Outdents (that is, moves to the left) the text that follows by the specified amount, expressed in points (1/72-inch).

divide(width) Draws a horizontal line across the page with the specified line width, expressed in points.

breakPage Inserts a page break.

Finalizing

open("docName") Finalizes the report, opening it in Acrobat as a PDF file with the specified name.

mail(bool,"addr") Finalizes the report, emailing the document to a specified address.

If the Boolean is true, this method presents a dialog box to the user (**Figure 2**), allowing the use of either the system's default email client or a webmail client. Note that this Boolean has meaning and is heeded only if the method is being executed in "privileged" mode (which is the topic of a future article); under normal circumstances, Acrobat always treats the Boolean as true.

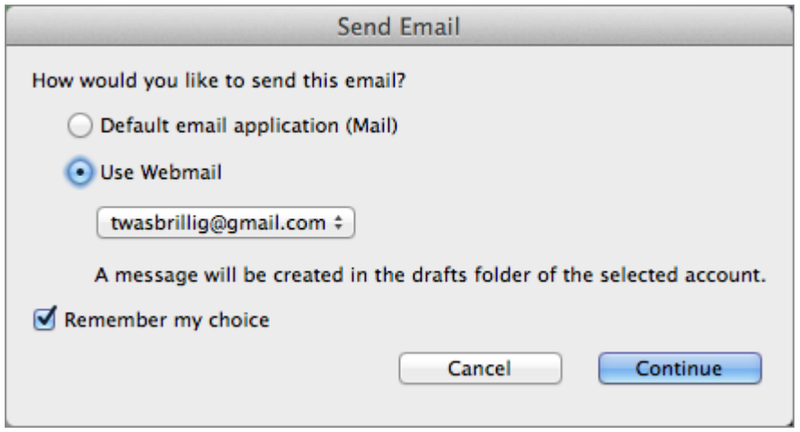


Figure 2. The mail method lets you email the report using your mail client or webmail.

Back to Our Sample

So, let's return to our sample script and examine each line in detail. As a reminder, here's the script:

```
var rep = new Report()                          // Create a Report object
var name = this.getField("txtName").value       // Get the Text field's value

rep.size = 1.4                                  // Increase the point size
rep.writeText("Greetings!")                     // Draw the page title
rep.size = 1.0                                  // Revert to original point size
rep.divide(2)                                   // Draw a horizontal divider line
rep.writeText("\nHi, " + name + ".")            // Draw a two-line message
rep.writeText("How ya doin'?")
rep.open("Greeting")                            // Open the report as a PDF document
```

Step by Step

`var rep = new Report()`

Here we create our Report object and assign it to the variable **`rep`**.

`var name = this.getField("txtName").value`

We get the value of the field txtName and assign the resulting string (containing a name) to the variable **`name`**.

Note that we are collapsing into a single line of JavaScript something that would be perhaps clearer if done in two steps:

```
var txtField = this.getField("txtName")  // Get a reference to the Text field
var name = txtField.value                 // Get the field's value
```

However, this entails creating an extra variable, **txtField**, that we never use again. Since the **getField** method returns a Field object, we can use the method call **this.getField** as though it were, itself, a Field object. Hence,

```
var name = this.getField("txtName").value
```

assigns to **name** the string value of the Text field txtName.

```
rep.size = 1.4
```

We set the size property of the Report object to 1.4. This is a multiplier; the text size will be set to 1.4 times the "standard text size," which is a curiously undefined value that looks, to my eyeballs, to be about 14 points.

```
rep.writeText("Greetings!")
```

We print some text on our report page. I intend the string "Greetings" to be my report title, which is why I am printing it 40% larger than the body text that will follow (**Figure 3**).

```
rep.divide(2)
```

We print a horizontal line that is intended as a divider, in this case separating the report title from its body.

Note that the method's argument is a line width; our divider will have a thickness of 2 points.

```
rep.size = 1.0
```

We set our linewidth back to the default text height. Remember that the number is a multiplier; we are setting the the text height to 1.0 times the standard text height.

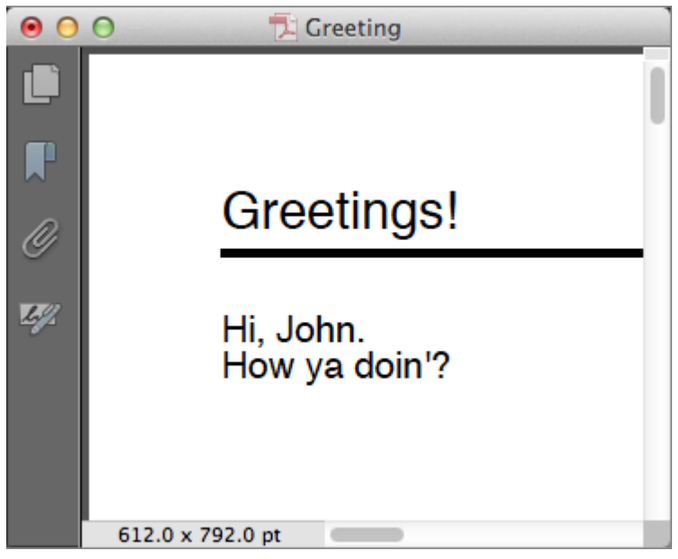


Figure 3. The text size of the "Greetings" title is 40% bigger than the body text beneath it. Also, notice the blank line before the "Hi" line, put there by the "**\n**" in the **writeText** string.

```
rep.writeText("\nHi, " + name + ".")
rep.writeText("How ya doin'?")
```

We print our two-line message. Note that these came out on separate lines on the page; each call to **writeText** starts a new line of text in our report.

Note also the **\n** in the first line of text. This is a "metacharacter" that wll be interpreted by Acrobat as a newline indicator; that is, it tells Acrobat to move to the beginning of the next line of text. In our case, since the **\n** is at the beginning of the text, we will see a blank line above the "Hi." (Take a look at Figure 3.)

Note finally that we assembled the first line of text by concatenating the string "Hi, " with the text taken from the txtName Text field, using the **+** sign, the JavaScript concatenation command. That is, if **name** has the string value "Amstel", then

```
"Hi, " + name + "."
```

will print as

```
Hi, Amstel.
```

Incidentally, you can also use the metacharacter **\t** within a string to denote a tab; the text that follows it in the string will be indented to a predetermined (and, again, undocumented) position .

```
rep.open("Greeting")
```

Finally, we end our report, opening it in Acrobat as a PDF document with the name Greeting.

If we had wished, we could have emailed the report as a PDF document by calling the mail method (Figure 4):

```
rep.mail(true,"hoozit@whatzit.com")
```

The **true** says that Acrobat should hand the mail off to your mail client. As I said earlier, you've no choice in the matter, since Acrobat always mails your PDF report using your mail client unless you're running in "privileged" mode.

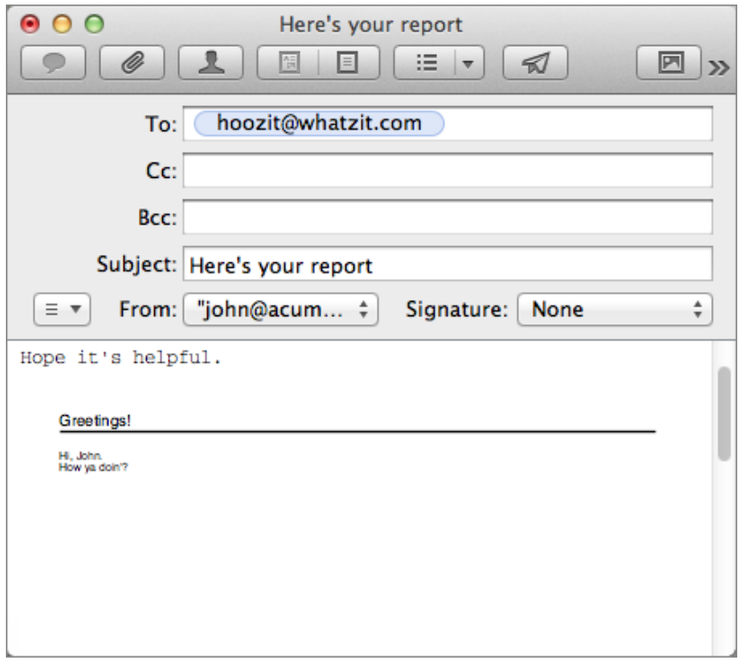


Figure 4. We could have mailed our finalized report to a list of email addresses.

Surprisingly Useful

Even though it's very primitive in its formatting capability, ther Acrobat JavaScript Report object is a very useful mechanism to know about. It doesn't come up often, but when it does, it is indispensable.

Schedule of Classes, August 2012– October 2012

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

Classes cost $2,000 per student, with the following exceptions:

- *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	Aug 20–23		Oct 15-18
PDF 2: Advanced File Content			
Support Engineers' PDF		Sept 6–7	

PostScript Classes

PostScript Foundations		Sept 17–21	
Advanced PostScript			
Variable Data PostScript			
Troubleshooting PostScript		Sept 3–5	

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

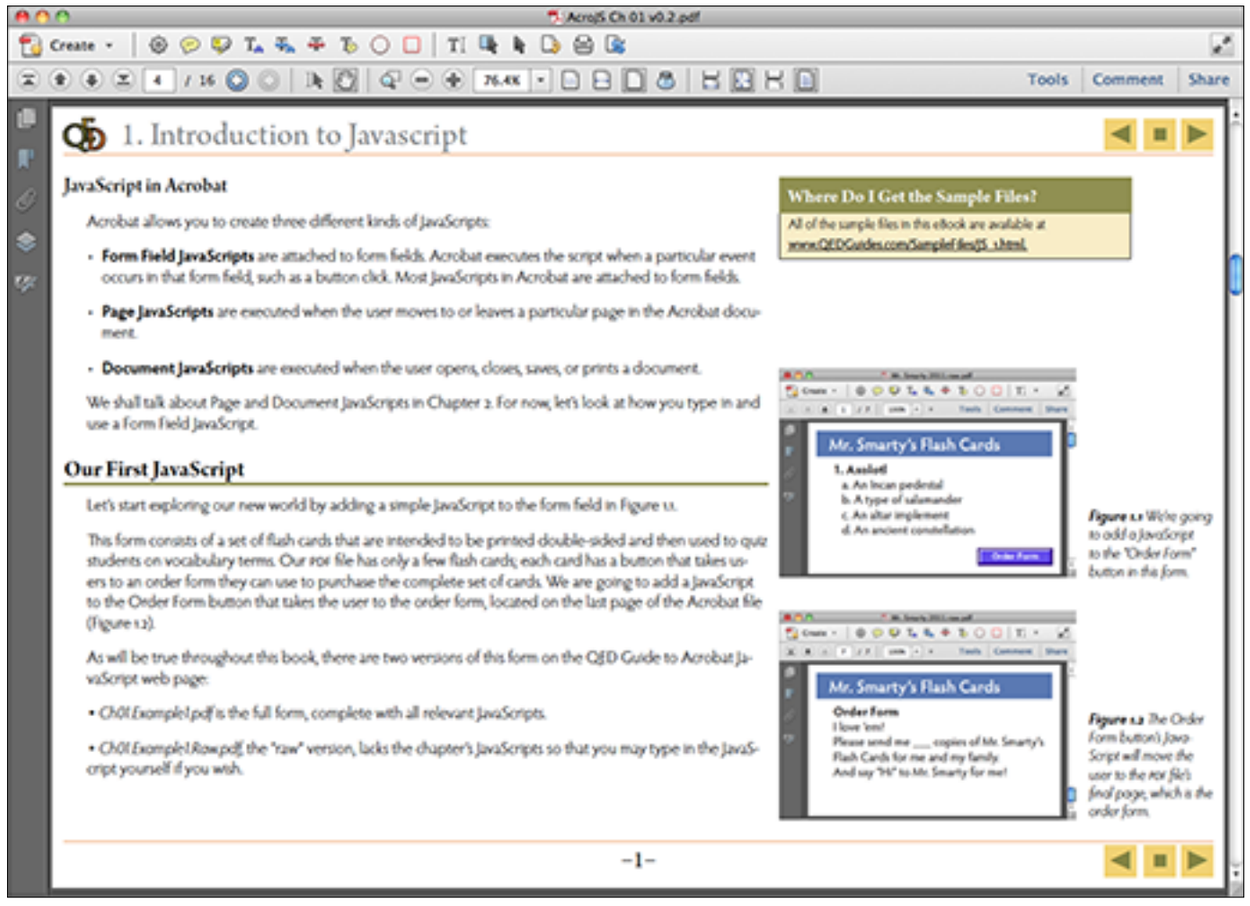
All issues of the *Acumen Journal* are available at the Acumen Training website: www.acumenjournal.com/AcumenJournal.html

What's New at Acumen Training?

The E-book is Selling Well, Hooray

Beginning JavaScript for Adobe Acrobat is out and selling well, I'm pleased to say. Do pick up a copy if you need to learn how to extend the capabilities of your Acrobat forms.

A description of the book, its table of contents, and a free, 2-chapter sampler are available on the Acumen Training website; click here.



Expanded PostScript Consulting

The infrastructure for the PostScript and PDF consulting services is now in place. If you need help with on a PostScript- or PDF-related project, I provide a range of consulting services, from email-based question-and-answer to planning and implementing a complete project.

More information is here on the Acumen Training website.

New Journal Design

As you noticed if you're a regular *journal* reader, I've been fussing around with its layout and design. I'm trying to make it more consistent with the layout and color scheme of the e-books and web site. I'm not a designer by trade, so my main goal has been simply to avoid a color combination that makes readers' teeth ache.