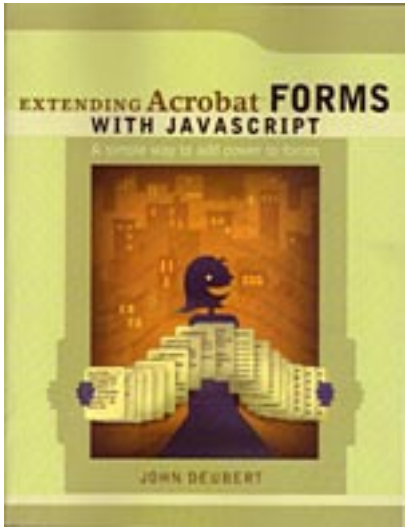
# Table of Contents

# JavaScripter: The Acrobat 6 Thermometer Object

Whenever you are carrying out a task in your JavaScript program that may take a very long time—counting the number of characters used in each of the document's fonts, say—it is polite to give the user some indication of how the task is progressing. (This will also let the user know that your script hasn't just frozen).

A typical way of doing this is to present the user with a progress bar of some flavor: a colored line, row of rectangles, or other gauge that grows as the task proceeds. Acrobat 6 adds to the Acrobat JavaScript interface a new *Thermometer object* that fulfills this function. This month, we shall see how to use it.

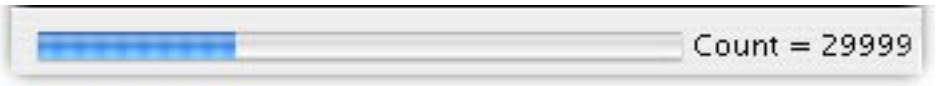

This month's article presumes you know at least minimal JavaScript and know how to attach a JavaScript to an Acrobat control. I shall assume you have at least the level of expertise resulting from reading my book *Extending Acrobat Forms With JavaScript*. (Well, go get it, then!)




You should also read the May 2003 Acumen Journal (available at acumentraining.com/AcumenJournal.html), to review how you create form fields in Acrobat 6 (and how to attach actions, including JavaScripts, to those fields).

Next Page ->

## Thermometer Objects




An Acrobat JavaScript Thermometer object is a classic horizontal progress bar that fills in from left to right as your task completes. This progress bar may include a label, indicating the current value.

### Appearance

The progress bar presented by the Thermometer object is actually that used by Acrobat itself. This makes it *very* easy to use, but limits the appearance and position of the progress bar to that used by Acrobat.


Specifically, the progress bar displayed by the Thermometer object always appears at the bottom of the current Acrobat window, replacing the document controls that normally appear there, as at right.

When the Thermometer is closed, the normal window controls reappear.

Also, the Thermometer is designed to count from zero to some positive number. If you want to count across some other range (-50 to 150, say), you will need to map those numbers into appropriate non-negative numbers before handing them to the Thermometer object.

Next Page ->

## The JavaScript Interface

The way you use the Thermometer object in JavaScript is very simple. There are a handful of properties you can set and a pair of methods that let you create and close the object.

### Methods

There are two methods that you must call when using the Thermometer object.

*begin()* The *begin* method initializes the thermometer, drawing it in the current Acrobat window. You must call this method before attempting to use the Thermometer object.

*end()* The *end* method closes down the Thermometer object, redrawing the window controls that normally appear at the bottom of a document window. You must call this method when you are finished with the thermometer object.

### Properties

The Thermometer object properties are what you mostly use to manipulate the object.

*duration* *integer*

This property is the maximum value of the progress bar. The progress bar fills in as the value proceeds from *0* to this number. The default *duration* is *100.*

Next Page ->

Acumen Journal: Acrobat User 4

*value* *number*

This is the current value of the Thermometer. When you set the value, the progress bar is colored in to reflect the proportion of this number to the *duration.*

*text* *string*


This string is printed to the right of the Thermometer object's progress bar. This property may to used to provide a text read-out of the current value.

*cancelled* *boolean*

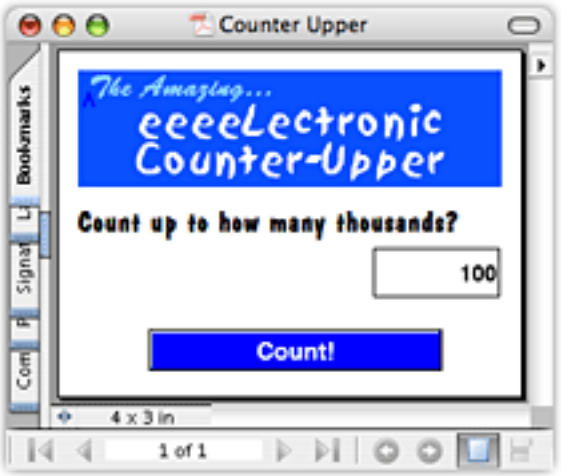
This is a property that you may monitor in your program to tell whether or not the user has requested the task be cancelled. This boolean will be set to *true* if the user has pressed the Escape key. On the Macintosh, the user may also press Command-period to indicate a cancellation request.

Next Page ->

## For Example

Let us add a JavaScript to the simple form at right. This form has two fields:

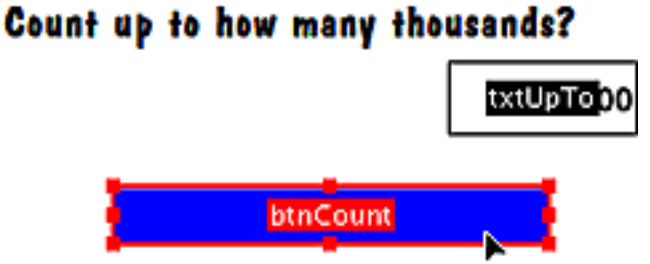
- A Text field named *txtUpTo,* formatted as a number.
- A button named *btnCount* that will trigger the form's activity.




As usual, the sample files for this article are available at the Acumen Training Resources page. Look for the file *ThermometerObj.zip.*

When the user clicks on the *Count* button, the form counts from 0 to however many thousands are indicated by the Text field.

We shall attach to the *Mouse Up* event for *btnCount* a JavaScript that does the following:




- Create a Thermometer object.
- Count from *0* to 1000 times the value of *txtUpTo,* setting the current value of the Thermometer object as we go.
- Close the Thermometer object.

*Required Reading*

I am assuming you know how to attach a JavaScript to a button. If not, read the May 2003 Acumen Journal article and (ahem) *Extending Acrobat Forms With JavaScript.*

Next Page ->

## The JavaScript

```
var i                                       // A counter variable
var fldUpTo = this.getField("txtUpTo")      // Get the text field
var maxI = fldUpTo.value * 1000             // maxI is maximum counter value
var therm = app.thermometer                 // Get a thermometer object

therm.duration = maxI                  // Set duration to our maximum counter val.
therm.begin()                          // Initialize the Thermometer object

for (i = 0; i < maxI; i++) {           // Start a for loop; counts from 0 to maxI
     therm.value = i                   // Set the thermometer's value to i
     therm.text = "Count = " + i       // Set the Thermometer's text
     if (therm.cancelled)  {           // Check to see if the user cancelled
          app.alert("Count cancelled!\n\n(Wimp.)") // Yes: cancel
          break
     }
}                                      // End of the loop; do it all again

therm.end()                            // Dispose of the Thermometer
```

Let's look at this code in detail.

Next Page ->

[Next Page ->]

**CounterUpper1.pdf**

This JavaScript is in the file *CounterUpper1.pdf* in this month's zip file.

**++ Notaion**

You may not have seen the notation "i++" before. The ++ operator increments the variable to which it is applied. "i++" is the equivalent of

```
i = i + 1
```

There is also a "--" operator that decrements the variable to which it is applied.

**Step By Step**

```
var i
var fldUpTo = this.getField("txtUpTo")
var maxI = fldUpTo.value * 1000
var therm = app.thermometer
```

We start by creating some variables that we need in the course of our script.

*i* This is a variable we shall use as a loop counter. This is the variable that will count from zero to our maximum value.

*fldUpTo* This is a reference to the Text field *txtUpTo.*

*maxI* This variable holds the maximum value of *i.* Our variable *i* will count from *0* to *maxI.* Note that we are giving *maxI* an initial value of 1,000 times whatever number is the value of *fldUpTo.*

*therm* This is the reference to our Thermometer object. Note that we obtain a Thermometer object by invoking *app.thermometer.*

*Initialize the thermometer*

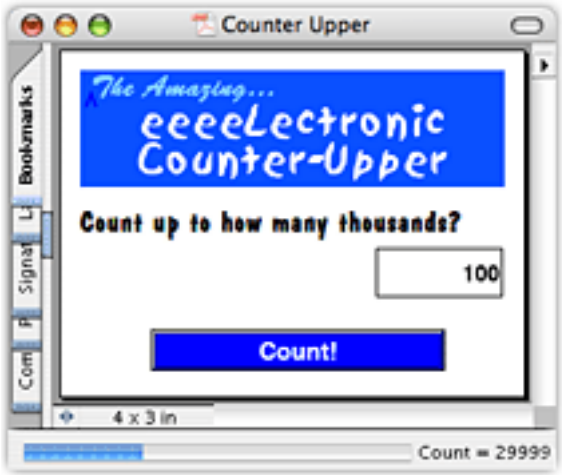
```
therm.duration = maxI
```

We initialize our Thermometer object by first setting its *duration* property to *maxI;* remember that we earlier set *maxI* to 1,000 times the value of our Text field.

Next Page ->

```
therm.begin()
```

We then call the *end* method, which draws the Thermometer at the bottom of the document window.




We are ready to start counting.

*Start our loop*

```
for (i = 0; i < maxI; i++)  {
      ...
}
```

If you are not familiar with *for* loops (my JavaScript book doesn't discuss them), this loop will repeatedly execute the JavaScript code in braces, each time setting the variable *i* to a successively higher value; as the loop proceeds, *i* will count from *0* to *maxI.*

Roughly translated into English, this *for* loop says:

1. Set *i* to zero.
2. If *i* is less than *maxI,* then execute the set of JavaScript statements in braces; otherwise, quit the loop.
3. Increment *i* by 1 and return to step 2.

We'll talk about JavaScript loops in the next *Acumen Journal* JavaScript article.

Next Page ->

*Within the loop*

Within the loop, we first increment our Thermometer's value and text:

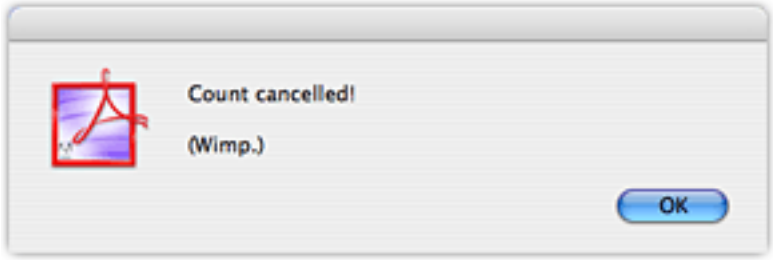
```
therm.value = i
therm.text = "Count = " + i
```

The first of these two lines sets the value of our Thermometer object to the current counter value. The second line then changes the object's *text* property to the text "Count = " followed by the current counter value. ("Count = 1", "Count = 2", etc.).

```
if (therm.cancelled)  {
     app.alert("Count cancelled!\n\n(Wimp.)")
     break
}
```

Now we must check to see if the user cancelled the count. The Thermometer object automatically sets its *cancelled* property to *true* if the user presses the Escape key or, on the Macintosh, Command-period.

We shall use an *if* statement to see if *therm.cancelled* is *true*. If so, we shall put up an alert and then exit from the loop. (The JavaScript *break* statement causes execution to drop out of a loop.)




Next Page ->

*Close the thermometer* `therm.end()`

*Done!* That's it. Our loop will continue until either we have counted to *maxI* (specified by the text field *txtUpTo)* or the user cancels.

## Improving the Counter

One problem with our form as it stands is that it counts very slowly. It would take it several hours to count from 0 to 100,000; most of us will lose interest before then and cancel the count.

JavaScript actually counts very quickly; what takes time is waiting for the Thermometer object to update the progress bar and redraw the text. It seems a pity to waste time redrawing the bar every single time we increment *i;* all we really want from our Thermometer object is a general indication of our progress.

We can modify our loop so that we only update the Thermometer every 10,000 or so. The file *Counter2.pdf* (one of the two Acrobat forms in the zip file associated with this article) implements this improved loop; you will notice that it counts *much* faster than the first example.

Next Page ->

**Improved JavaScript**

Here's our improved *Mouse Up* JavaScript; I've marked the new lines in red.

**CounterUpper2.pdf**

This JavaScript is in the file *CounterUpper2.pdf* in this month's zip file.

```
var i                                       // A counter variable
var fldUpTo = this.getField("txtUpTo")      // Get the text field
var maxI = fldUpTo.value * 1000             // maxI is maximum counter value
var therm = app.thermometer                 // Get a thermometer object
var j = 10000                               // Indicates when to update therm.

therm.duration = maxI                  // Set duration to our maximum counter val.
therm.begin()                          // Initialize the Thermometer object

for (i = 0; i < maxI; i++) {           // Start our for loop
     if (--j == 0)   {                 // Decrement j; is it zero yet?
          therm.value = i              // Yes: set the thermometer's value & text
          therm.text = "Count = " + i
          j = 10000                    //       Then reset j to 10000
          if (therm.cancelled)   {     // And then check for user cancellation
               app.alert("Count cancelled!\n\n(Wimp.)")
               break
          }
     }
}
```

Next Page ->>

*What's happening here?*

I will not step through a detailed description of how this new script works. In short, however, as *i* counts from 0 to *maxI,* the Script uses a variable, *j,* to count down from 10,000 to zero . We change the Thermometer's value only when *j* reaches zero. (At that same time, we check to see if the user has cancelled and reset *j* to 10,000 again.)

Thus, instead of updating the progress bar every time through the loop, we do so every 10,000 times; this speeds up the script enormously.

**User Interface Courtesy**

User feedback is an important part of any good user interface. Letting a user know that your script is still working (and hasn't just frozen) is polite at the very least. The Acrobat 6 Thermometer object is easy to use and allows you to reassure the user that all is well.

*A Limitation*

One last word: the Thermometer object was not available before Acrobat 6. Your script should probably check to see what version of Acrobat the user is running (see the September 2002 *Acumen Journal* or the *Extending Acrobat Forms* book to see how to do this) and use the Thermometer object only if the version is *6.*

Return to Main Menu

# String Concatenation and *NullEncode*

It seems surprising to newcomers that PostScript has no operator for concatenating strings; you must write your own procedure to join two strings together.

Typically, such a procedure works something like this:

- Create a new string (call it the "target string") that is large enough to hold both the original strings (the "source strings").
- Use *copy* to copy the characters of the first source string into the target string.
- Use *putinterval* to copy the second source string into the target.

If you're curious to see what this code looks like, you can find it in the file *ConcatStrings0.ps* in the zip file associated with this article (*ConcatStrings.zip).*

This month, we're going to look at the definition of a *ConcatStrings* procedure that takes a somewhat different tack. We are going to attach the *NullEncode* filter to our target string and then write the source strings to the resulting filtered file object. This way of doing things has three benefits:

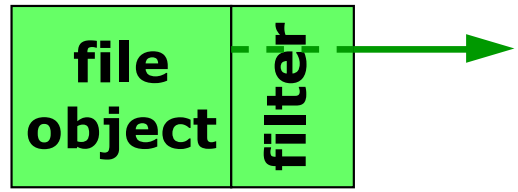
- It results in a slightly cleaner procedure definition.
- It is easily extended to concatenate an array of strings into a single string.
- It gives us a rare opportunity to use the *NullEncode* filter, one of life's very minor thrills.

Next Page ->

# Background

## PostScript Filters

You may remember from your PostScript classes that a *filter* is something you can attach to a PostScript file object. The resulting file-plus-filter, called a *filtered fileobject*, behaves in every way as though it were a normal file.




In particular, you can read data from or write data to the filtered fileobject using all the normal file operators (*readstring,* etc.); the data passing through the filter is modified in some way specific to the filter: compressed, converted between ASCII and binary, etc.

**For More Information**

This review of PostScript filters is necessarily *very* brief. For more information, see the *PostScript Language Reference Manual,* the June 2002 *Acumen Journal*, or (of course) take an Acumen Training class.

You attach a filter to a file with the *filter* operator:

```
fileobj parameters /filtername filter   =>   filtered-fileobj
```

The *filter* operator takes as arguments a file object, a filter name, and (in between them) an optional set of parameters; it returns on the operand stack the filtered object representing the file with attached filter.

## *Filters and Strings*

Surprisingly, you can also attach a filter to a string:

```
(str) parameters /filtername filter   =>   filtered-fileobj
```

Reading data from the resulting filtered fileobject reads bytes sequentially from the string; writing to the filtered fileobject streams bytes into the string.

Our *ConcatString* procedure will be attaching a filter to a string,  as we shall see.

Next Page ->

Acumen Journal: PostScript Tech 15

*The NullEncode Filter*

The set of filters available in PostScript is reasonably large, allowing you access to a variety of compression and ASCII-binary conversions. Of interest to us in this article is the *NullEncode* filter.

*NullEncode* has two characteristics that are important to us:

- It is a null filter; it does not change the data passing through it.
- It is an *Encode* filter; it is a filter to which you write data. (The opposite is a *Decode* filter, from which you read data.)

If you attach the NullEncode filter to a string, you can stream data into the string just by writing them to the filtered file object using *writestring* or other PostScript file operator. This is exactly what we shall do in our *StringConcat* procedure.

**Filters & Acumen Training Courses**

The *Advanced PostScript* and *Variable Data PostScript* classes discuss filters in depth; the *PDF File Content and Structure* class discusses PDF filters, which work identically to PostScript filters.

Next Page ->>

## *ConcatString*

Here's our definition of *ConcatString:*

```
/ConcatStrings      % (s1) (s2) => (s1s2)
{
      dup length           % => (s1) (s2) s2len
      2 index length       % => (s1) (s2) s2len s1len
      add string           % => (s1) (s2) (s1s2)
      dup /NullEncode      % => (s1) (s2) (s1s2) (s1s2) /NullEncode
      filter               % => (s1) (s2) (s1s2) fobj
      dup 5 -1 roll        % => (s2) (s1s2) fobj fobj (s1)
      writestring          % => (s2) (s1s2) fobj
      3 -1 roll            % => (s1s2) fobj (s2)
      writestring          % => (s1s2)  fobj
      closefile            % => (s1s2)  Q.E.D.
} bind def

(ABC) (DEF) ConcatStrings ==
```

**File on the Website**

You can find this file on the Acumen Training Resources page. Look for *ConcatStrings.zip,* which contains three files; this one is *ConcatStrings.ps.*

In broad outline, this procedure does the following:

- Create a target string the size of the combined source strings.
- Attach the *NullEncode* filter to a copy of the destination string.
- Write the two source strings in succession to the NullEncode filtered fileobject.

We shall manage the stack so that this operation leaves the target string on the stack.

Next Page ->

Acumen Journal: PostScript Tech 17

**The Code in Detail**

```
/ConcatStrings      % (s1) (s2)  =>  (s1s2)
```

This procedure will expect the two source strings on the stack; call them *s1* and *s2.*

*Make the Target String*

```
dup length                  % => (s1) (s2) s2len
2 index length              % => (s1) (s2) s2len s1len
add string                  % => (s1) (s2) (s1s2)
```

We start by creating our target string. We do the following:

- Calculate the sum of the lengths of *s1* and *s2.*
- Use the *string* operator to create a string with room for that many characters; this will be our target string.

*Attach the NullEncode filter*

```
dup /NullEncode             % => (s1) (s2) (s1s2) (s1s2) /NullEncode
filter                      % => (s1) (s2) (s1s2) fobj
```

We duplicate our target string (called "(s1s2)" in the code comments) and push the name */NullEncode* onto the stack.

We then execute *filter,* attaching the *NullEncode* filter to our target string. When we write to the resulting filtered fileobject, the data will be streamed into the target string.

Next Page ->

*Write s1 to the target string*

We next want to write *s1* to the filtered fileobject, streaming its contents into the target string. We shall do this with the *writestring* operator:

```
fileobj  (data)  writestring  =>  ---
```

This operator takes a fileobject and a string from the operand stack and writes the bytes within the string to the file. In our case, the string will be our source strings, one at a time, and the fileobject will be the *NullEncode* filter attached to our target string.

```
dup 5 -1 roll              % => (s2) (s1s2) fobj fobj (s1)
```

We first duplicate the fileobject and then roll *s1* to the top of the stack. This leaves the top two items on the stack being exactly what *writestring* needs as arguments: a fileobject and a string.

```
writestring                % => (s2) (s1s2) fobj
```

We can now call *writestring,* writing *s1's* contents to our target string.

```
dup 4 -1 roll              % => (s1s2) fobj fobj (s2)
writestring                % => (s1s2)  fobj
```

In a similar manner, we again duplicate the fileobject, roll *s2* to the top of the stack, and then write that string to our target.

This leaves on the operand stack the filtered fileobject and our target string, now holding the concatenated contents of the original source strings. (Remember that writing to the filtered fileobject actually writes to our target string.)

Next Page ->

Acumen Journal: PostScript Tech 19

*Tidy Up*

```
        closefile               % => (s1s2)
} bind def
```

We finish our definition of *ConcatString* by closing the filtered fileobject, since we are done with it. This leaves our target string, the procedure's return value, on the stack.

*Try it out!*

```
(ABC) (DEF) ConcatStrings ==
```

To see if our procedure is working, we hand it a couple of strings and print the result to the back channel with ==. What should appear in the interpreter's log file (or wherever *stdout* is on your PostScript implementation) is "(ABCDEF)".

Next Page ->

## Concatenating an Array of Strings

One virtue of this method of concatenating strings is that it is very easily extended to concatenate an array of strings, rather than just two strings.

Here's the definition of a *ConcatStringArray* procedure:

```
/ConcatStringArray        % [ (s1) (s2) (s3) ... ] => (s1s2s3...)
{
      dup 0 exch          % => [ ] 0 [ ]
      { length add }
      forall              % => [ ] len
      string exch         % => (s) [ ]
      1 index             % => (s) [ ] (s)
      /NullEncode filter  % => (s) [ ] fobj
      exch                % => (s) fobj [ ]
      {                   % => (s) fobj (src)        | Beginning of forall loop
            1 index exch  % => (s) fobj fobj (src)
            writestring   % => (s) fobj
      } forall            % => (s) fobj
      closefile           % => (s)
} bind def


[ (ABC) (DEF) (GHI) (JKL) ] ConcatStringArray ==
```

Next Page ->

Acumen Journal: PostScript Tech 21

In broad outline, the *ConcatStringArray* procedure takes an array of strings and does the following:

- Fire up a *forall* loop that adds together the lengths of all the strings in the array.
- Make a target string of that length.
- Attach the *NullEncode* filter to the target string
- Use a *forall* loop to write each string in the array to the *NullEncode* filtered fileobject.
- Close the filtered fileobject

The procedure manages the operand stack so that the target string is left on the stack as a return value.

**It's Been Fun, But...**

It is fun to see the *NullEncode* filter in action; it doesn't get used much.

Unfortunately, some primitive performance testing indicates that this implementation of string concatenation is slower than the more common method that uses *copy* and *putinterval.* (The traditional version is perhaps 30% faster.)

So, for reference, on the next page, is a more common, less fun, but faster implementation of *ConcatStrings.* (This is in *ConcatStrings0.ps* in this month's zip file.)

Though slower in execution, our *NullEncode* method arguably still has its uses; it is *much* easier to do *ConcatStringArray* with the NullEncode method.

Next Page ->

Acumen Journal: PostScript Tech 22

*Traditional ConcatStrings*

```
/ConcatStrings      % (s1) (s2) => (s1s2)
{
      dup length                % => (s1) (s2) len2
      2 index length            % => (s1) (s2) len2 len1
      add string                % => (s1) (s2) (s)
      dup 3 -1 roll             % => (s1) (s) (s) (s2)
      3 index length exch       % => (s1) (s) (s) len1 (s2)
      putinterval               % => (s1) (s)
      dup                       % => (s1) (s) (s)
      3 -1 roll                 % => (s) (s) (s1)
      exch copy pop             % => (s)
} bind def
```

I shall leave a detailed examination of this procedure to the readers' copious idle time.

Return to Main Menu

# Schedule of Classes, Dec 2003 – Mar 2004

Following are the dates and locations 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. The Acrobat class schedule is on the next page.

The PostScript classes are taught in Orange County, California and on corporate sites world-wide. See the Acumen Training web site for more information.

## Technical Classes

| | | | | |
|---|---|---|---|---|
| New! | PDF File Content and Structure | | Feb 17–20 | |
| | PostScript Foundations | | | Mar 12–16 |
| | Variable Data PostScript | Dec 15-19 | | |
| | Advanced PostScript | | | Mar 22–26 |
| | PostScript for Support Engineers | | | |
| | Jaws Development | | *On-site only* | |

**Not many classes**

I'm going to be teaching a *lot* of on-site classes in January and February, so I'm afraid there aren't many classes on my site until March.

**Technical Course Fees** The PostScript and PDF classes cost $2,000 per student.

Registration Info
Acrobat Classes

# Acrobat Class Schedule

These classes are taught quarterly 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. *Troubleshooting With PitStop* (full day) is $340.00. In all cases, 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:** 25142 Danalaurel, 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:** 25142 Danalaurel, 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?

## Nothing Too New, Really

The new *PDF File Content and Structure* class is doing very well.

Quiet, otherwise, as befitting the year's end.

I hope 2004 is a good year for us all. Peace, prosperity, the occasional mad fling.

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? Does it make you really appreciate the clear, concise writing style of e. e. cummings?

**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