# Table of Contents


**The Acrobat User** — **The JavaScript *Date* Object**
The *Date* object allows your JavaScripts to determine and display the current date, as well as to perform some date-related arithmetic.

**PostScript Tech** — **Miscellany**
Some miscellany this month. I wanted to discuss three useful PostScript commands and features that are often overlooked even by relatively experienced PostScript programmers.

**Class Schedule** — May-June-July

**What's New?** — **JavaScript e-book; expanded consulting; new web design. I've been busy!**
The JavaScript e-book is finally out (and about time!). I'm expanding my PostScript consulting services. And I've redesigned the Acumen Training website. It beats being bored.

**Contacting Acumen** — Telephone number, email address, postal address



Journal feedback: suggestions for articles, questions, etc.

# Miscellany

This month, I thought I'd present some this-and-that topics that I've long wanted to talk about, but that aren't individually meaty enough to make up an article. So, three topics today:

- Using integers and other non-names as dictionary keys.
- Using radix notation to specify integer constants.
- Using packed arrays.

These are all very long-standing features of PostScript, the first two dating back to Level 1. Still, a surprising number of experienced PostScript programmers don't know about one or more of these or have forgotten and never use them.

So, let's introduce these features to some people and jog the memories of the rest.

## Non-Names as Dictionary Keys

It comes as a surprise to many people that PostScript dictionary keys don't have to be names. In nearly all instances, a name object *is* the sensible thing to use as a key in a PostScript dictionary:

```
/MyDict <<
   /x  27      % The keys /x & /y are names
   /y  18
>> def
```

However, sometimes you want to associate a value with some key other than a name; for example, I've had one client who needed to associate values with integers; he started out using names, something like this:

```
/NumDict   <<
   /1  (One)
   /2  (Two)
>> def
```

Again, the keys here are names: **/1**, **/2**, etc. He fetched values from this dictionary using code similar to this (which assumes *x* is an integer):

```
NumDict x ( ) cvs cvn get
```

He could have simplified his code by using integers as keys:

```
/NumDict    <<
   1    (One)
   2    (Two)
>> def
```

Now the keys here are the actual integers and their associated values can be looked up with simpler code:

```
NumDict x get
```

There is no restriction on what kind of PostScript object may be used as a key. My favorite illustration of this was created many years ago by a PostScript Foundations student in, of all places, Scarborough, U.K. He needed to replace several operator definitions with PostScript procedures, but only if the operator had not already been replaced (this was a real possibility in his situation). So, immediately in class after we discussed non-name keys, he created a dictionary whose keys were *operator definitions.*

Now, that was just cool!

```
/OperatorDicts  <<
   systemdict /showpage get {...showpage replacement code...}
   systemdict /show get {...show replacement code...}
   systemdict /setlinewidth get {...setlinewidth replacement code...}
>> def
```

He could then conditionally replace the operator definitions thusly:

```
OperatorDicts       % Push the dictionary on the stack
/showpage load      % Get the current definition of showpage (or whatever)
2 copy known        % Does the showpage definition exist as a key in OperatorDicts?
{ get /showpage exch def }  % Yes: redefine showpage
{ pop pop }                 % No: throw away the dictionary and key
ifelse
```

It was from this pretty easy to use **forall** to traverse **systemdict** and replace all the operators that had not already been redefined.

*Very* fun!

## Radix Notation

It happens not too infrequently that you want to refer to a hexadecimal value in your PostScript code (perhaps mapping 1-byte color values into PostScript 0-1 values). Usually, we end up converting the hex values ahead of time into decimal values and using those:

```
212 72 186 sethexrgbcolor
```

This is perfectly fine; no real complaints. Still, sometimes it'd be convenient—for readability, if nothing else—if we could use in our PostScript code the actual hex values we'd looked up in a book somewhere. Well, we can:

```
16#D4 16#48 16#BA sethexrgbcolor
```

You can infer the format here, I'm sure:

```
base#value
```

The number base can be any decimal integer value 2-36; the values are expressed using characters 0–9 and a-z (case-insensitive), as appropriate to the base.

This is purely a convenience and has absolutely no effect on the actual integers used in the execution of your code; that is, the following two lines yield exactly the same result:

```
100 200 moveto
8#144 8#310 moveto
```

Radix notation is for the visual convenience of the programmer only.

## Packed Arrays

Finally, here's a minor memory saver: packed arrays.

A packed array is just like a regular array, except:

- It's read-only.
- It takes up substantially less VM than a regular array.

Packed arrays are useful in situations where the amount of VM is chronically low; it can help reduce the incidence of VM errors and avoid provoking garbage collection quite so often.

## Creating Packed Arrays

PostScript maintains a "packing" parameter that determines whether arrays are created as packed or not; it defaults to false, of course, so the arrays you usually create will be normal, unpacked, writable arrays.

If you set the packing parameter to *true* using the **`setpacking`** operator, then all arrays will be created as packed arrays until you turn the parameter to *false* again.

```
true setpacking
/myArray [ (This is) (a packed) (array.) ] def
false setpacking
```

I've been doing a bit of nosing around with **`vmstatus`** and, by and large, packed arrays seem to be 20%-30% smaller than regular arrays. Not a huge savings in a world of cheap and plentiful RAM, but worth doing if you have memory issues in your PostScript environment.

*Why would I use this?* You might be saying to yourself, "I don't generally use much in the way of read-only arrays, so does this buy *me?*"

Well, remember that procedure bodies are actually executable arrays. You can reduce the memory footprint of your PostScript prolog by defining all of your procedures with the  packing parameter set to *true:*

```
%%BeginProlog
true setpacking
/bd { bind def } bind def
/inch { 72 mul } bd
/rshow { dup stringwidth pop neg 0 rmoveto show } bd
... lots more procedure definitions
false setpacking
%%EndProlog
```

In the above snippet, all of the procedures are created as packed executable arrays.

*A Fly in the Farina* There is one, usually minor, problem associated with packed arrays: they are slower than regular arrays. The difference in speed is miniscule if the array is accessed sequentially (first using element 0, then element 1, etc.), as is the case with procedure definitions. Random access, however, is very much slower than non-packed arrays. I haven't been able to hunt up (or measure) any reliable timings of packed vs. non-packed arrays, so we'll just need to take Adobe's word for it.

# Displaying the Current Date with JavaScript

My new Acrobat JavaScript e-book *(Beginning JavaScript for Adobe Acrobat)* is finally finished and available for purchase, so I thought I'd celebrate with a series of JavaScript articles over the next couple of *Acumen Journal* issues. (See the What's New page for more information on the e-book.)

This month we're going to use the JavaScript **Date** object to implement an automatically-updating "Current Date" field (such as the one at right) in an Acrobat form.

25 April 2012

This article is intended as supplementary reading for people who have read *Beginning JavaScript*, so I'll be assuming that you know (or, at least, once knew) the basics of JavaScript programming and how to attach a script to a form field or a page. If you don't know this, I'll be reviewing it briefly as we go, but not in any great detail at all.

**The Full Scoop**

This article presents a light introduction to the JavaScript **Date** object. Complete documentation is on the Mozilla Developer website.

Click here.

Have I mentioned you'll find the book useful?

The **Date** object, by the way, is standard JavaScript; it isn't a feature specific to Adobe Acrobat. You can use this object in any JavaScript environment, including web pages.

## The Project

Our sample project for this article is built around the PDF file pictured in **Figure 1**. This one-page document contains a single, locked text field named *txtDate* (**Figure 2**). Because the field is locked, it will appear to the user as static text, not editable by simply clicking on it.

We are going to attach a Page JavaScript to this document's single page that will set the value of the text field to the current date so that whenever a user opens the document

**The Sample File**

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

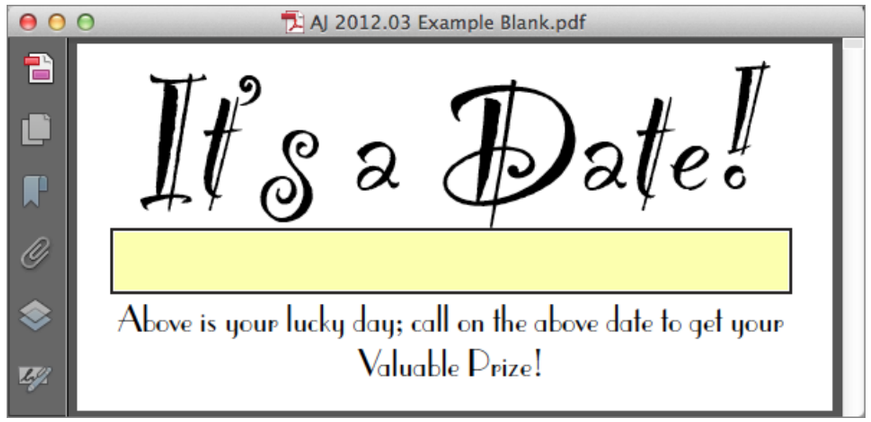



**Figure 1.** We are going to attach a JavaScript to this page that sets the yellow text field's value to the current date.


**Figure 2.** Our page has a single text box named *txtDate*.

(and therefore opens the page), he or she will see the current date in the box. It's a very simple JavaScript that will give us a chance to do some initial exploration of the **Date** object.

## Page JavaScripts: a Review

As I said, we are going to attach a Page JavaScript to the document. You no doubt remember (*yes* you do!) that a Page JavaScript is executed whenever the user goes to a particular page in the PDF document. You attach a script to a document's page through the Page Thumbnails navigation panel.

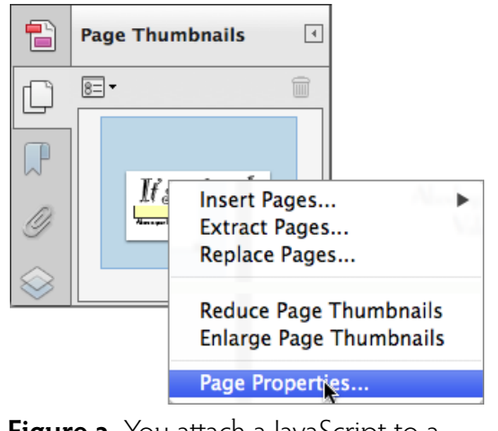



**Figure 3.** You attach a JavaScript to a page through the page's Properties.

Here's a reminder of the procedure:

1. In the Page Thumbnails navigation pane, right-click on the page to which you want to attach the JavaScript and select *Page Properties* from the resulting contextual menu (**Figure 3**).

   Acrobat will display the Page Properties dialog box (**Figure 4**).

2. In the Actions panel (shown in Figure 3), select *Page Open* in the Select Trigger pop-up menu and select *Run a JavaScript* in the Select Action pop-up menu.

3. Click the Add button.

   Acrobat will present you with the JavaScript Editor window (**Figure 5**).

4. Type your JavaScript into the text field.

5. Close all windows and dialog boxes until you are back at the PDF document window.

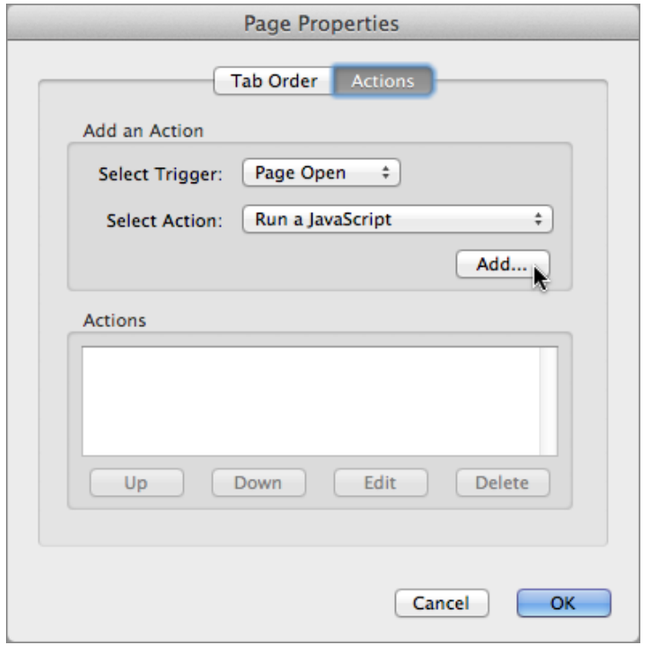



**Figure 4.** We'll attach a JavaScript to the Page Open event.

That's it; the JavaScript you typed in will execute every time Acrobat opens that page.

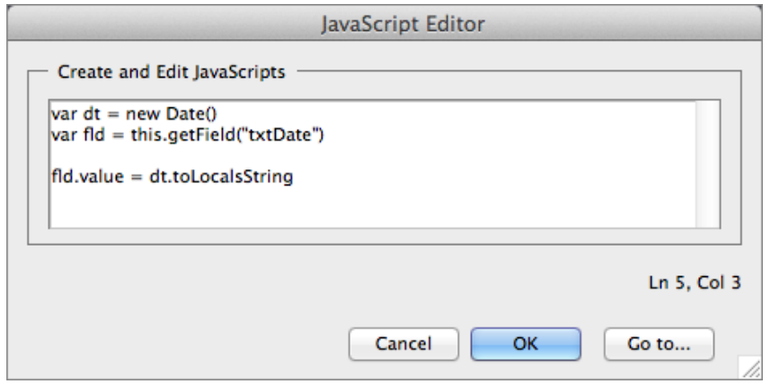



**Figure 5.** Type the JavaScript code into the JavaScript Editor window.

## The Code

Our page JavaScript is a simple little thing that gets the current date and then sets the value of the text field:

```
var dt = new Date()
var fld = this.getField("txtDate")

fld.value = dt.toDateString()
```

Let's look at it in detail.

### *Step by Step*

**`var dt = new Date()`**

Here we are getting a **`Date`** object and assigning it to the variable **`dt`**.

The JavaScript **`new`** command, you may recall, creates an instance of a class, in this case the **`Date`** class. Note that we are passing no arguments when we create the **`Date`** object; the parentheses are empty. When we do this, the newly created **`Date`** object will be initialized with the current date and time. There are other possibilities; we could have initialized the **`Date`** object with a particular date by passing that date as a string:

```
var dt = new Date("February 11, 1997")
```

There are a few other options you could exercise, as well, but I'll let you look those up in the on-line documentation.

**`var fld = this.getField("txtDate")`**

You saw this line a *lot* in the JavaScript book; we are getting a reference to the field **`txtDate`** and assigning it to the variable **`fld`**. Remember that **`this`** here refers to the current document.

```
fld.value = dt.toDateString()
```

Here is the line that does the visible work. We set the value of the text field to a string representation of the date using the **Date** object's **toDateString** method. The result is that our PDF page looks like Figure 6.

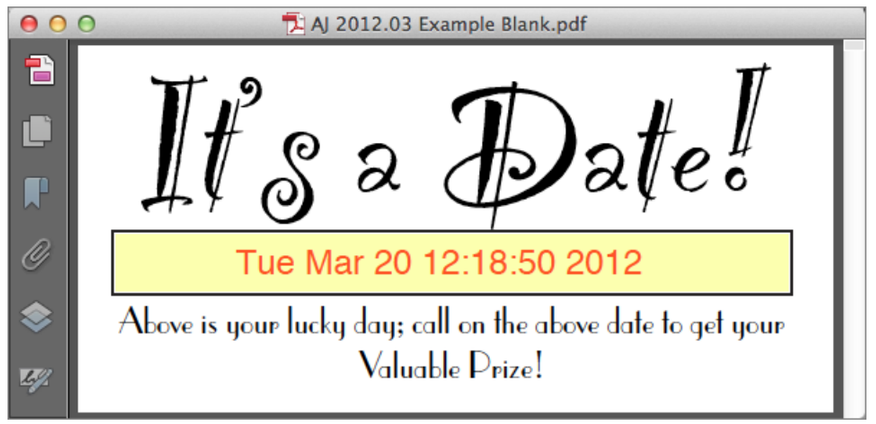



**Figure 6.** The **toDateString** method returns a text representation of the complete date and time.

The **toDateString** method returns a string for the **Date** object's entire value, including the time of day. Given the other text on the page, we probably want only the date. We can fix this most easily by using the **toLocaleDateString** method:

```
fld.value = dt.toLocaleDateString()
```

This method returns a string version of the **Date** object's date, excluding the time of day; the result is that our text box looks like Figure 7. Note that you have no control over exactly how the date is represented; **toLocaleDateString** takes its format from your system's settings and is immutable beyond that.

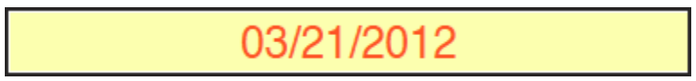



**Figure 7.** The **toLocaleDateString** method returns a string version of just the date part of the Date object.

But, what if we want to display our date with a format of our own?

Ah, well, then we need to do a little more work.

## Constructing Our Own Date String

Here is a page JavaScript that produces the results in **Figure 8**.


**Figure 8.** We can also create our own format for the `Date` object's string value.

```
var dt = new Date()
var fld = this.getField("txtDate")

var monthNames = ["January","February","March","April","May","June",
                  "July","August","September","October","November","December"]

fld.value = dt.getDate() + " " + monthNames[dt.getMonth()] + " " + dt.getFullYear()
```

Here we are getting the numeric values of the current date, month, and year and assembling them into a string representation of our own choosing.

*Step by Step*

```
var dt = new Date()
var fld = this.getField("txtDate")
```

We start, as before, by creating a `Date` object and getting a reference to the text field.

```
var monthNames = ["January","February","March","April","May","June",
                  "July","August","September","October","November","December"]
```

Here we are creating an array of month names. The `Date` object gives us a way of getting the current month as a numeric value between *0* and *11;* the `monthNames` array lets us convert a month number into a month name by using the former as an index into the array. Thus, `monthNames[3]` would return the string "April". (Remember array indices start at *0*.)

```
fld.value = dt.getDate() + " " + monthNames[dt.getMonth()] + " " + dt.getFullYear()
```

Finally, we set the value of the text field.

This line of code uses three "get" methods of the **Date** object: **getDate()** gets the **Date** object's day of the month (1-31); **getMonth()** returns the month as an integer 0-11; **getFullYear()** returns the full year (2012, as of right now). The **Date** object provides a good collection of *get* methods, including **getHour()**, **getMinutes()**, and **getSeconds()**.

Our final JavaScript line

```
fld.value = dt.getDate() + " " + monthNames[dt.getMonth()] + " " + dt.getFullYear()
```

uses these methods to create a string that will be the value the text field; the string consists of:

- *dt.getDate()* The day of the month.
- *" "* A space.
- *monthNames[dt.getMonth()]*
  The name of the month. Note that we are using the month code, returned by **dt.getMonth()** as an index into the **monthNames** array. JavaScript
- *" "* Another space.
- *dt.getFullYear()* The current year.

Put them all together and you have the text string we show in Figure 8.

**Worth Knowing, Yes?** The **Date** object is a useful little beggar; I think you'll find that perusing the documentation on the Mozilla site (here) will be time very well spent.

# Schedule of Classes, May 2012– July 2012

At right are the dates of Acumen Training's upcoming classes. Clicking on a class name will take you to 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 | | Jun 11–14 | |
| PDF 2: Advanced File Content | | | |
| Support Engineers' PDF | May 10–11 | | Jul 19–20 |

### *PostScript Classes*

| | | | |
|---|---|---|---|
| PostScript Foundations | | Jun 4–8 | |
| Advanced PostScript | | Jun 25–28 | |
| Variable Data PostScript | | | Jul 30 – Aug 3 |
| Troubleshooting PostScript | May 7–9 | | Jul 16–18 |

# 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**

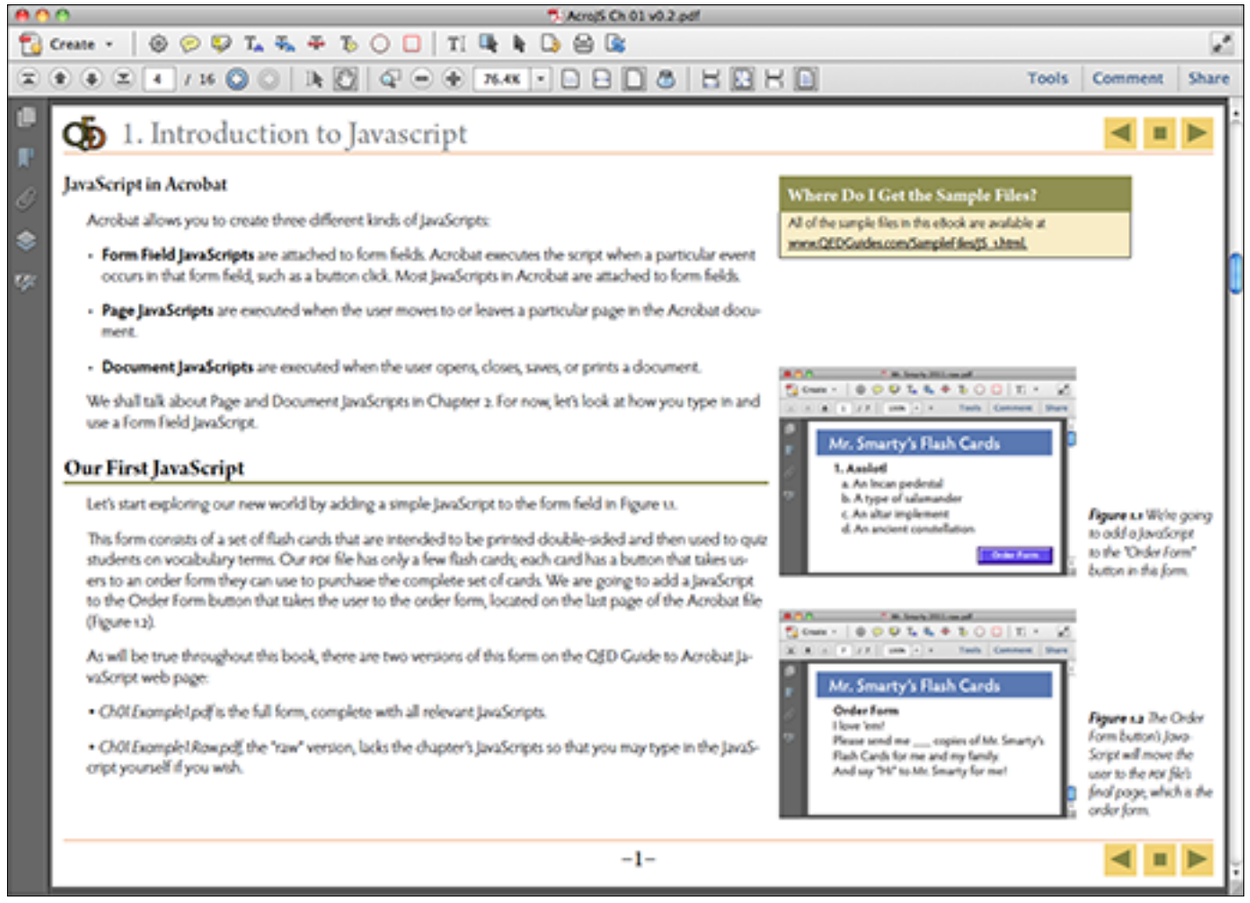
All issues of the *Acumen Journal* are available at the Acumen Training website: www.acumenjournal.com/AcumenJournal.html

# What's New at Acumen Training?

## Busy Times!

### New e-Book: *Beginning JavaScript for Adobe Acrobat*

*Beginning JavaScript for Adobe Acrobat* is finished; its release is awaiting some dickering with Peachpit Press over some legal issues. In the meantime, you can get more information, the table of contents, and a free, 2-chapter sampler from the Acumen Training website; click here.




### Expanded PostScript Consulting

I'm going to be expanding my PostScript and PDF consulting services. 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 Website Design

Speaking of which, the Acumen Training website has undergone a face lift. The new design is clean, attractive, and still easy to navigate. I like it a lot and hope everyone who visits will, too.