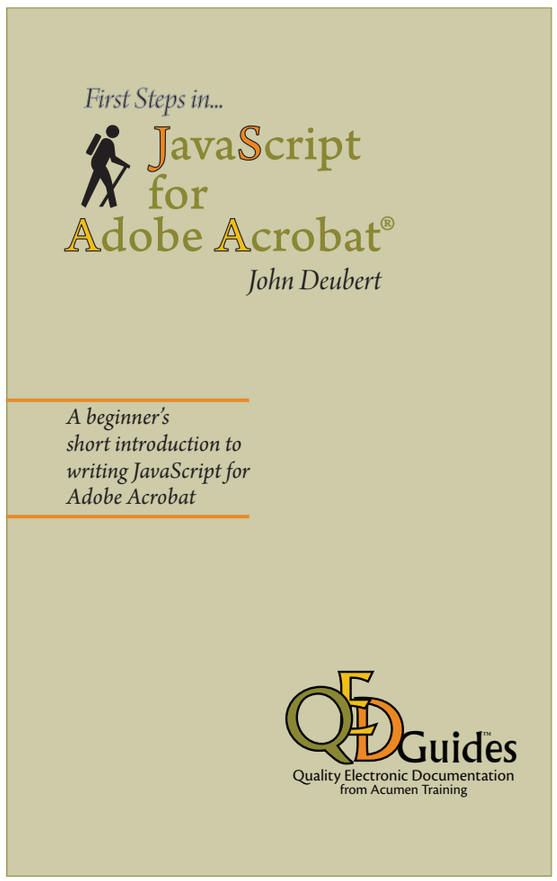
Table of Contents

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



An Object-Oriented Text Block in PostScript

The Variable Data PostScript class is one of my favorite classes to teach (though I don't teach it often; it's a small market) because I really get a chance to make PostScript show its stuff as a programming language. One characteristic of the class is that the project we work on, one of whose output pages is at right, entails printing on each page several blocks of text that vary in their purpose. Some of the blocks are uniform in font and point size, some are styled, some have embedded tags that need to be replaced with variable data.

This is a situation that invites an object oriented approach; in other languages we'd define an object class that represents a block of text and then create various subclasses to accommodate the different types of text we need to print.

PostScript isn't an object-oriented language, but it *is* a programming language, so we can do pretty much anything we want, include implement an object-oriented structure for our text blocks.

Let's see how to do this.

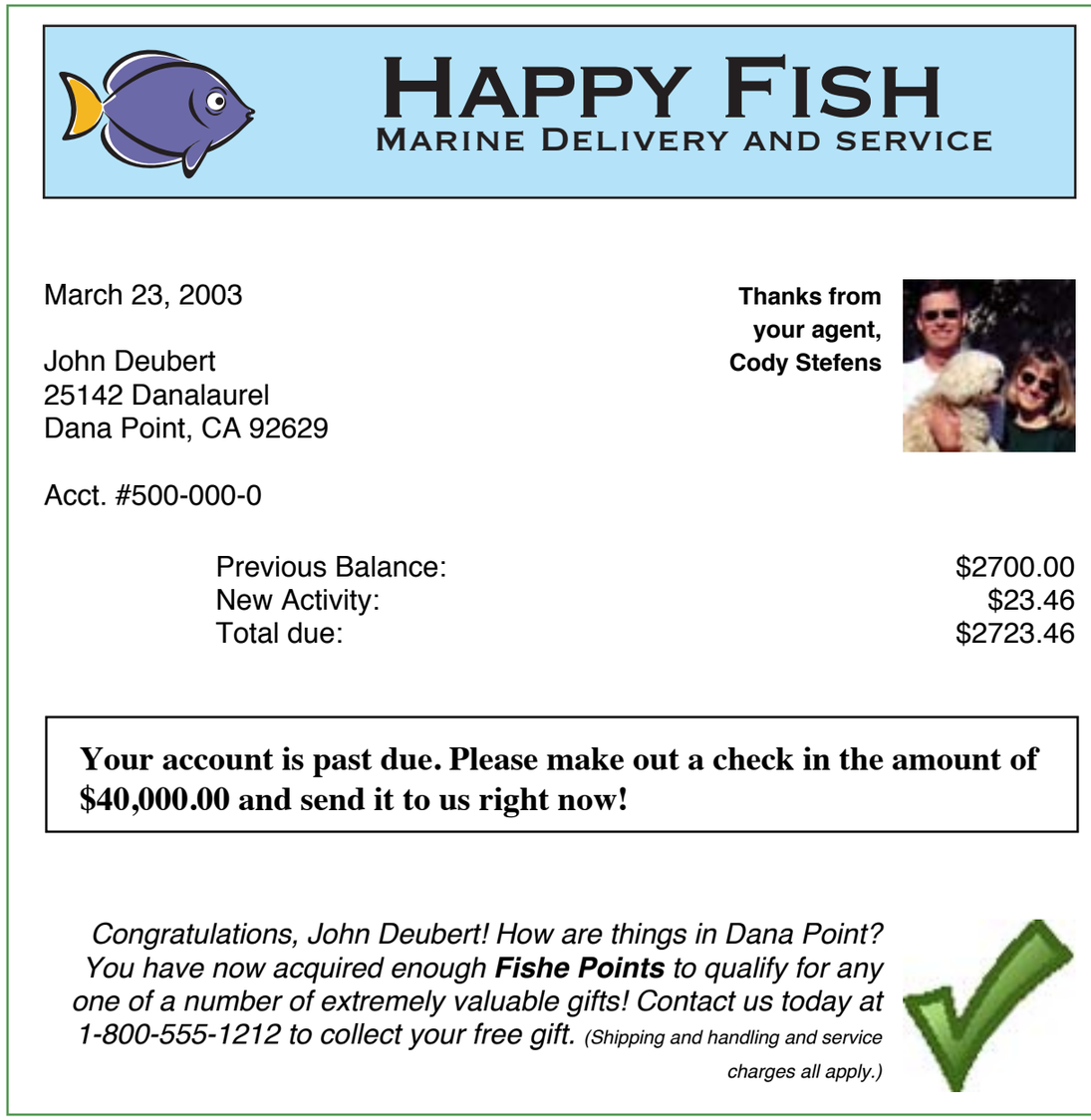


Figure 1. The output pages in the Variable Data PostScript class have a collection of blocks of text, each with difference characteristics.

A TextBlock Object

This article's code implements a *TextBlock* dictionary that is a roughly self-contained representation of a piece of text that needs to be printed at some arbitrary place on the page. In **Figure 1,** the boxed text and the "Congratulations" paragraph are both represented by TextBlock dictionaries.

A TextBlock object contains everything needed to print that particular piece of text: the text itself, of course, but also the font and point size, the line height, and, significantly, the PostScript procedure that actually draws the text. To draw a TextBlock, you push its dictionary on the dict stack and execute the Show procedure, stored in the TextBlock dictionary itself.

Show draws the text in whatever manner is appropriate for that text block. The basic TextBlock, like Figure 1's boxed text, just draws itself within a specified field width, breaking lines as needed. Figure 1's "Congratulations" text, on the other hand, is a much-less-basic TextBlock; its Show procedure supports styled text and replaces pre-defined tags with variable data.

The Sample Code

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

Both TextBlocks were printed the same way: the TextBlock dictionary was pushed on the dictionary stack and the block's Show procedure was executed. In the Variable Data class we did this with a utility procedure named ShowTextBlock:

```
/ShowTextBlock      % << TextBlock >>  x y →  ---
{
   3 -1 roll begin
   Show
   end
} bind def
```

Note that the blocks' Show procedures expect to find an *x,y* pair on the operand stack.

Dependencies

Speaking of utility routines, the TextBlock implementation I present here makes use of two procedures, PrintWord and PrintParagraph, that together take a string and print it within a specified field width. These are defined in a dictionary named *TextBlockCommon*, as follows:

```
/TextBlockCommon       <<
   /newline          % --- => ---
   { 0 currentpoint exch pop lineHeight sub moveto } bind
```

```
/PrintWord      % (word) => ---
{
    dup stringwidth pop
    currentpoint pop
    add maxWidth gt
    { newline } if
    show ( ) show
} bind

/PrintParagraph      % (paragraph) => ---
{
    ( )
    {
        search exch PrintWord
        not { exit } if
        } loop
        newline
    } bind

>> def
```

These procedures are described in full detail in the January 2006 (issue 41) of the *Acumen Journal;* you should probably go read that article now, since this article talks no further about how they work.

The TextBlock Object

The TextBlock implementation is built from a generic TextBlock dictionary that forms the basis for all the specific TextBlocks we need. The Generic TextBlock supplies default values for the text parameters; these are placeholders, really, since they will nearly always need to be overridden for a specific TextBlock instance.

An Object-Oriented Text Block

Here's my definition of the generic TextBlock:

```
/TextBlockGeneric  <<
    /font /Helvetica findfont
    /textSize 16
    /lineHeight 19
    /maxWidth 300
    /text ()

    /Show       % x y => ---
    {
        TextBlockCommon begin
        gsave
        translate
        font textSize scalefont
        setfont
        0 0 moveto
        text PrintParagraph
        grestore
        end
    } bind
>> def
```

This is pretty lucid, I think. We supply:

- The font, point size, and leading we want for the printed text;
- The distance (`maxWidth`) between the left and right margins (for simplicity, the text block moves the origin to the starting x,y coordinate, so the left margin is always *0*).
- The text (defaulting here to an empty string).

Finally, the dictionary contains a Show procedure that prints the text at a specified *x,y* location on the page.

Note that this procedure calls the PrintParagraph procedure defined in "TextBlockCommon". Note also that Show presumes the TextBlock dictionary has already been pushed onto the dictionary stack.

Creating a TextBlock Instance

To create a TextBlock that represents a specific piece of text on our page, we need to do the following:

- Make a copy of the generic TextBlock dictionary.
- Redefine, in the new dictionary, any entries that need to be overridden; this will often be all or most of the generic entries.

Here's a procedure that does this:

```
/CreateTextBlock        % <<Generic>>  <<overrides>>  →  << New Block >>
{  exch dup length dict copy  % Make a copy of the generic dictionary...
   begin                      % ...and push it onto the dict stack
    { def } forall            % Copy dict override contents
    currentdict               % Push current dict onto operand stack
    end                       % and clean up the dict stack.
} bind def
```

This procedure takes two dictionaries: a generic TextBlock (or any other dictionary, for that matter) and a dictionary of overrides, key-value pairs that should be added to the newly-made TextBlock dictionary.

A typical call to this procedure would look like this:

```
/TextBlock1
   TextBlockGeneric
   << /font /Times-Roman findfont
       /TextSize 20
       /text (We're all adults here and can be trusted not to keep a secret.)
   >> CreateTextBlock
def
```

You could then print the Block's text with a call to ShowTextBlock:

```
TextBlock1 100 600 ShowTextBlock
```

Subclassing, Sort of...

So what if we want to create another type of TextBlock, perhaps one that prints colored text? We'd like this new "class" to have all the contents of the previous class, plus an array containing an RGB value; it will also need a new version of Show that will set the color before printing the text.

We need to create a new species of generic TextBlock that contains all the old TextBlock contents and then adds the color value and a new version of Show. In addition, if we can preserve the old Show procedure, perhaps renaming it, our new Show could call the old one, letting us reuse the earlier code.

Here's a MakeSubclass procedure that does all this; it's generic, itself, and could be used with any dictionary that has a Show procedure in it:

```
/MakeSubclass   % << Base >> << overrides >> => <<subclass>>
{
    exch dup length dict copy  % Create a copy of the base class...
    begin                      % ...and put it on the dict stack
    /Show0 /Show load def      % Preserve the old Show proc
    { def } forall             % Copy the override contents
    currentdict                % Push the new Generic dict on the op stack
    end                        % and remove it from the dict stack
} bind
```

An Object-Oriented Text Block

So, to make the generic ColorTextBlock, you'd do this:

```
/TextBlockColorGeneric
    TextBlockGeneric            % The base class
    <<                          % The additional key-value pairs
        /rgb [ .3 .7 .5 ]
        /Show                   % x y => ---
        {
            gsave
            rgb aload pop setrgbcolor  % Set the color…
            Show0                      % …and call the old Show
        } bind
    >>
    MakeSubclass            % Leaves the new generic ColorTextBlock dict on the stack
def
```

Now we can create and use a colored text block as follows:

```
/RedText
    TextBlockColorGeneric
    <<  /Font /Times-Italic findfont
        /text (Aristotle was famous for knowing everything. He taught that the
brain exists merely to cool the blood and is not involved in the process of
thinking. This is true only of certain persons.)
        /rgb [ 1 .3 .5 ]
    >> CreateTextBlock
def

RedText 100 450 ShowTextBlock
```

Although basic and red TextBlock objects are called with identical invocations of ShowTextBlock, each prints text according to its nature (**Figure 2**).

Limitation to this Implementation

The code in this article supports only one level of subclassing, since all I do to preserve the base class's Show is rename it to a fixed name. A more generic approach would be to have each TextBlock maintain an array of text-showing procedures and have the Show procedure execute all the procedures in this array:

```
/ShowPipeline [  {show0} {show1} {show2}... ]
/Show   {
    ...
    ShowPipeline { exec } forall
    ...
} bind
```

The MakeSubclass procedure would simply add the Show procedure in the overrides dictionary to the head of the array. We didn't bother with this in the VDPS class; it never became necessary.

I will leave this as an Exercise for the Student.

So, What's It Good For?

Consider this food for thought, if nothing else. This object-oriented approach entails a bit of effort, but it helps organize your work when your task has a large number of variations on a theme.

Also, it was fun to develop.

Aristotle was famous for knowing everything. He taught that the brain exists merely to cool the blood and is not involved in the process of thinking. This is true only of certain persons.

Aristotle was famous for knowing everything. He taught that the brain exists merely to cool the blood and is not involved in the process of thinking. This is true only of certain persons.

Figure 2. The output pages in the Variable Data PostScript class have a collection of blocks of text, each with difference characteristics.

The JavaScript *Math* Object

Someone complained to me recently that, as far as he could tell, JavaScript is missing some very basic mathematical functions, including all of the trigonometric functions.

How could that possibly be?

Well, not surprisingly, it isn't so; JavaScript has all the math functions you could want. However, they are unexpectedly hidden from immediate view. In fact, all of the math functions except the very most basic reside as class methods in the JavaScript *Math* class. This class is not the least bit hard to use, but the fact of its existence isn't obvious when you first get started with JavaScript.

What you should know...

This article assumes you have some knowledge of Acrobat JavaScript, equivalent to having read my e-book, *Beginning JavaScript for Adobe Acrobat.*

Information about the book is available here.

The Math Object

The Math object contains a fair collection of class methods that implement math functions, listed in **Table 1.** Examining the table, you'll see that these methods are all pretty straightforward both in concept and in use. Since these are class methods, you don't need to create an actual Math object to use them; you can simply invoke them using the class name, as follows:

```
var root6 = Math.sqrt(6)
```

The above line will set the variable `root6` to the value 2.45 (or thereabouts).

Table 1 Math Object Properties & Methods

Properties		
E	LOG10E	SQRT1_2
LN10	LOG2E	SQRT2
LN2	PI	
Methods		
abs	cos	pow
acos	exp	random
asin	floor	round
atan	log	sin
atan2	max	sqrt
ceil	min	tan

An Example

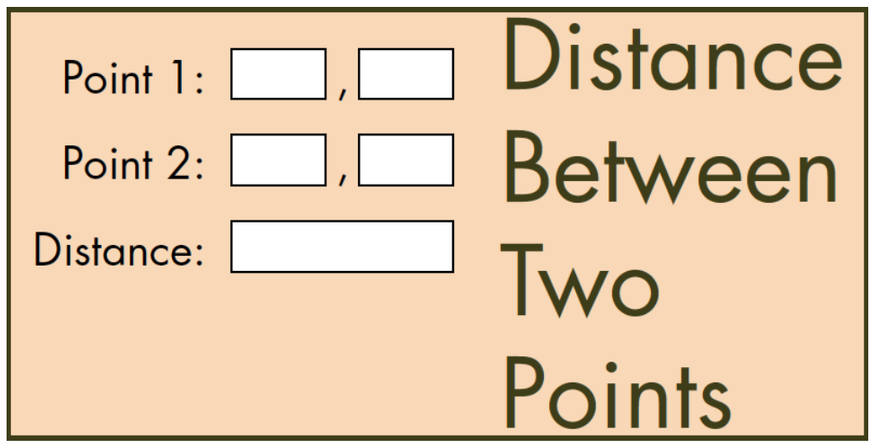


Figure 1. This JavaScript uses methods from the Math object.

To see the Math object in action, consider the form fields at right. These collect the x and y coordinates of two points and calculates and displays the distance between them. (The fields are active, by the way; try them out.)

Remembering the distance formula (*yes* you do!), the distance between two points is

$$d = \sqrt{(x_2-x_1)^2 + (y_2-y_1)^2)}$$

This is the value we want to put into our Distance field.

The form field names are named Pt1_x, Pt1_y, Pt2_x, Pt2_y, and Distance.

Each of the four x or y form fields has an "On Blur" script attached to it that simply calls a function named CalculateDistance, defined in a document JavaScript. So, the On Blur scripts are, in total:

```
CalculateDistance
```

The CalculateDistance function is defined in a document script, as follows:

```
function CalculateDistance()
{
   var x1, y1, x2, y2
   var dist

   x1 = this.getField("Pt1_x").value
   y1 = this.getField("Pt1_y").value
   x2 = this.getField("Pt2_x").value
   y2 = this.getField("Pt2_y").value

   dist = Math.sqrt(Math.pow(x2 - x1,2) + Math.pow(y2 - y1,2))
   dist = Math.round(100 * dist) / 100
```

```
    this.getField("Distance").value = dist
}
```

This script uses three of the Math object methods:

Math.sqrt(n) Returns the square root of *n*.

Math.pow(m,n) Returns m^n.

In our code, pow is used in the visually-confusing

```
Math.pow(x2 - x1, 2)
```

which returns $(x_2 - x_1)^2$

Math.round(n) Returns *n* rounded to the nearest integer.

The Math object is, as I say, pretty easy to use. A full description of all of its methods can be found in the ClientSide JavaScript Reference.

Schedule of Classes, July – October 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		Aug 5–8	Sept 23–24
PDF 2: Advanced File Content			
Support Engineers' PDF		Aug 15–16	Oct 17-18

PostScript Classes

PostScript Foundations	July 15-19		Sept 9-13
Advanced PostScript			
Variable Data PostScript	July 29–Aug 2		
Troubleshooting PostScript		Aug 12–14	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

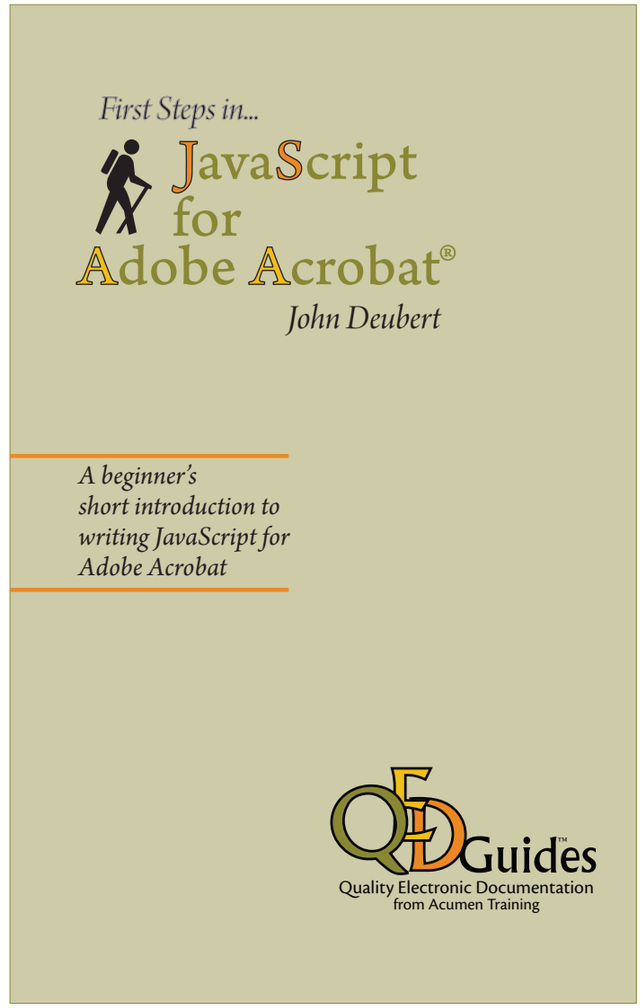
All issues of the *Acumen Journal* are available at the Acumen Training website: www.acumenjournal.com/AcumenJournal.html

What's New at Acumen Training?

First Steps in JavaScript for Adobe Acrobat is out

The first volume of the *First Steps Guides*, a series of cheap, short, but very useful e-books, is out and available on Amazon.com. This initial offering provides a solid start to your study of JavaScript for Adobe Acrobat.

It's an excellent deal at $1.99!



Acumen Training has a new classroom site

Acumen Training has a new location for its classes in Orange County:

Acumen Training
31726 Rancho Viejo Road, Suite 219
San Juan Capistrano, CA 92675

Our classes are now conducted in the attractive Ortega Business Center, a three block walk from the famous Mission San Juan Capistrano. It's surrounded by many places to eat and a five minute drive from the California coast, including Dana Point Harbor and some of the best Southern California surfing beaches.