Table of Contents

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

May I assume you've already bought a copy of *Beginning JavaScript for Adobe Acrobat?* If not, why not? Cheap, easy, fun, and your Folks will admire you for it!

Variable-Argument Procedures in PostScript

Many programming languages allow you to specify default values for a function's arguments so that they may be omitted when you call the function. C and C++ let you specify a function protoype such as:

```
void voteNow(int candidate, bool holdNose = true);
```

which allows you to omit the **holdNose** argument; both

```
voteName(0);
```

and

```
voteName(0, true);
```

Are valid calls to the **voteName** function. If you don't specify the **holdNose** argument in the function call, it receives a value of true, according to the above declaration.

Now, some argue that such a procedure declaration is bad, in the long run, because it introduces an ambiguity in exactly how a such a procedure is going to behave; however, there's no denying that such a procedure can be very convenient at times.

This short article will show you how to implement such a procedure in PostScript. We're going to develop a "**==String**" procedure (pronounced "emit string," if you're curious) that will take either of two sets of arguments:

```
==String    % (string)  =>  ---

==String    % (string) boolCRLF  =>  ---
```

In the first case, **==String** will simple emit the string to stdout, with no newline afterwards. In the second case, if the boolean is true, **==String** will emit the string, followed by a newline; if the boolean is false, it will emit only the string.

I grant you, this procedure is only barely useful, but it will allow us to demonstrate the technique.

Implementing the Procedure

Our **==String** procedure must be able to distinguish between two sets of arguments:

- A string only: `(str) => ---`
- A string and a boolean: `(str) bool => ---`

It will do this by looking at the topmost argument it finds on the stack. If that argument is a boolean value, then it know that it has a string-and-boolean pair of arguments; otherwise, it will presume it has received only a string.

Our implementation of an optional-argument procedure will use the PostScript **type** command.

```
obj  type  =>  /typename
```

This command takes a PostScript object and returns a predefined name indicating the type of data the object represents. **Table 1** lists the set of names that **type** can return; I presume their meanings are self-explanatory.

Table 1 Predefine Data Type Names

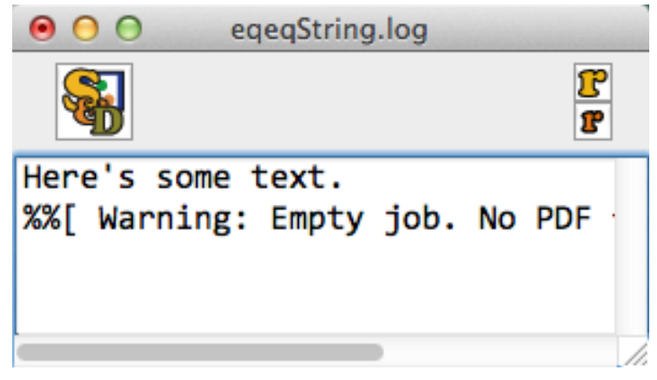
/arraytype	/gstatetype	/operatortype
/booleantype	/integertype	/packedarraytype
/dicttype	/marktype	/realtype
/filetype	/nametype	/savetype
/fonttype	/nulltype	/stringtype

The Code

Here's our definition; the example below produces the log file displayed in **Figure 1**.

```
/==String   {
    dup type /booleantype ne      % Is the topmost argument not a boolean?
    { false } if                  % Not a boolean: supply one with a value of false
    { = }{ print } ifelse         % Execute either = or print, depending
} bind def

(Here's ) ==String (some ) ==String
(text.) true ==String
```



This is easy and short; in detail, here's what it does:

Step-by-step

`dup type`

We duplicate the topmost argument (since **`type`** will consume it) and then check it's data type.

`/booleantype ne`

We place a boolean on the stack indicating whether the **`type`** operator returned a name *other* than **`/booleantype`**. (Note we're using the **`ne`** operator to do our comparison.)

`{ false } if`

If the name returned by **`type`** was *not* **`/booleantype`**, we shall assume that we were passed only a string argument; we'll supply the missing boolean ourselves, pushing a boolean *false* on the stack.

At this point, we have a boolean on top of the stack (either the original boolean arguent or one we supplied), indicating whether the string should be followed by a newline when sent to stdout.

`{ = }{ print } ifelse`

We place a pair of procedure bodies on the stack containing calls to the **`=`** and **`print`** operators. You may recall that the **`=`** operator prints the topmost item on the stack to stdout, followed by a newline; **`print`** takes a string as its argument, sending the string to stdout, *not* following it with a newline.

We execute one or the other of the procedure bodies, depending on the value of the boolean.

Why do this?

Although variable-argument procedures are a powerful tool in other languages, its application in the PostScript world is fairly limited. I've used it mostly when modifying existing code (not written by myself), usually replacing an existing procedure definition with a multi-argument version that retains its original

behavior when called by the existing Script, but lets me add new behavior in the Script code I'm modifying.

That said, I don't think the situation has come up more than a small handful of times in the decades I've been working with PostScript. If you have encountered a situation where you have used this technique, drop me a line; I'd be curious to know about it.

Simple PDF Signatures using *flattenPages*

Adobe Acrobat has a very impressive facility for electronically signing a document. An electronically-signed PDF contract is considered to be a legally binding document in many states and for many government institutions. However, the Acrobat signature mechanism requires a certain amount of set-up, the details depending on the version of Acrobat you and your co-signer have; you must have an electronic identity established on your computer; you must send a certificate containing this identity to the person who sent you the contract; that person must import the certificate into a local copy of Acrobat, etc., etc.

Sometimes you want something not quite that robust, something quick-and-dirty that makes it clear that you and a client have agreed to something but that isn't likely to become the basis of a lawsuit.

For example, in my consulting agreements, I do the following: I have them type their name, title, and the date into a trio of Text form fields and then click on a button that converts the form fields and their contents into artwork on the PDF page, a process known as "flattening" the PDF pages. Once the PDF file is flattened, their name, title, and the date are fixed onto the page as normal text.

Perhaps not legally binding, but it does give me (and my client) something that makes it clear we were in agreement on the terms.

Try it out; fill in the form fields at right with your name and then click the button; the fields will immediately become welded to the page, completely uneditable (well, except with the use of touch up and other editing tools).

Cool, huh? Note that I also make the button disappear, since it's no longer needed.

We carry out this little trick by attaching to the button a JavaScript that executes the Doc object's `flattenPages` method. It's very easy to use; let's look at it.

Time to sign yer life away, pal!

Type your name here:

And today's date:

Now click here: *Commit to it!*

This example will work only with Adobe Acrobat.

The *flattenPages* Method

The Doc object's **flattenPages** method is remarkably simple to use. To flatten the annotations on one or more pages in a document, you would make a call like the following:

```
this.flattenPages({
    nStart: 0,
    nEnd: 3,
    nNonPrint: 2
})
```

The above call will flatten all the annotations on pages 0 through 3. (Remember that JavaScript numbers document pages starting with zero.) The named arguments here are:

nStart — The zero-based page number of the first page to be flattened.

nEnd — (Optional) The final page to be flattened.

Annotations on all the pages from **nStart** to **nEnd** will be flattened. If you omit **nEnd**, then annotations on only page **nStart** will be affected.

nNonPrint — (Optional) A code, 0-2, indicating what should happen to non-printing annotations. Possible values are given in **Table 1**.

The reason we could need this parameter is that once the annotations are flattened, they are artwork like all the rest of the text, graphics, and images on the page. If you print the document, the formerly-non-printing annotations will print along with all the other page contents. Specifying an **nNonPrint** argument lets something else happen.

Table 1 *nNonPrint* Code Values

Code	*Meaning*
0	Flatten non-printing annotations
1	Don't flatten non-printing annotations
2	Remove non-printing annotations

As usual, if you wish you can omit the argument names, passing them in the above order:

```
this.flattenPages(0,3,2)
```

This takes up less room, but is also less readable.

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.

Flattening Form Fields

If `flattenPages` is intended to flatten a document's annotations, why does it also work on form fields?

It turns out that, under the hood, the visual component of a form field is just an annotation (of type Widget); each form field is represented within the PDF file by two objects:

- A form field dictionary that defines all of the active characteristics of the field (that is, the parameters that determine how it should behave when collecting data from the user).
- A Widget annotation dictionary that defines what the form field should look like.

Since form fields are built around an annotation, they are affected by `flattenPages`. This means you can have some unintended consequences when you flatten a page. For example, you may have noticed that when you clicked on the Do It button at the start of this article, all of that page's navigation arrows, implemented as form field buttons, became flattened, as well, and therefore stopped working.

> ***Note***
> Links are also affected by `flattenPages`, for the same reason: their visual components are implemented as annotations.

Our Sample Code

So, let's return to the example from page 1. The three form fields consist of two text fields and a button, named txtName, txtDate, and btnCommit (**Figure 1**). We are going to attach a JavaScript to the button's Mouse Up event that does two things:

- Hides the button
- Flattens the current page

I'm assuming you know how to attach a JavaScript to the Mouse Up event of a button. If not, check out the references listed in the "Fuller Scoop" sidebar on the previous page.

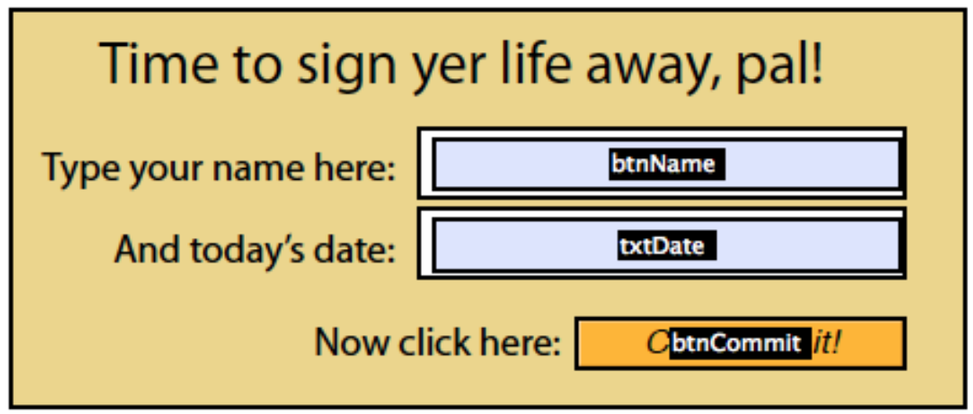


Figure 1. Our

Here's the (very short) script:

```
var btn = this.getField("btnCommit")

btn.display = display.hidden
this.flattenPages({
    nStart: btn.page
})
```

Let's follow it through in detail.

Step by step

`var btn = this.getField("btnCommit")`

We start by getting a reference to the btnCommit field and assigning it to the variable **`btn`**.

`btn.display = display.hidden`

We then set the **`display`** property of the button to the predefined constant **`display.hidden`**.

At this point, the button disappears from view.

```
this.flattenPages({
      nStart: btn.page
  })
```

Now we flatten the annotations (and form fields and navigation controls and links) on the page. Note that for the page number I used **`btn.page`**, which will be the page number on which btnCommit resides; this way I didn't need to hard-wire a page number into the code.

I could have written the method call without the argument name: **`this.flattenPages(btn.page)`**. In real life, I probably would have done so for brevity, but for teaching and writing purposes, I prefer the longer, but clearer, version.

Some Limitations

Fun as this technique is (*yes,* it is!), it has some limitations of which you should be aware.

Acrobat Only

This works only with the full Adobe Acrobat; in particular, it doesn't work with Adobe Reader, Apple's Preview app, or any other application of which I am aware. None of those implement the `Doc.flattenPages` method; the JavaScript will simply fail to execute the method, usually with no sign visible to the user that anything is wrong.

Not Legally Binding

This is not a replacement for an actual, legally-binding electronic signature. I use it simply to verify that everyone is aware of my terms, conditions, hourly rate, candy preference, etc. If you are looking for something that could conceivably form the basis of a lawsuit, then use the real electronic signature mechanism.

Whether this article's technique is enough for you depends on your clientele. The people for whom I work are pretty much always dependable; over almost three decades of experience, I've been "stood up" perhaps once, though I can't recall the incident just now. So, for me, this is acceptable fo ruse in my consulting agreements.

Schedule of Classes, December 2012– March 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	Dec 10–13	Jan 28-31	Mar 11-14
PDF 2: Advanced File Content			
Support Engineers' PDF		Feb 7-8	Mar 21-22

PostScript Classes

PostScript Foundations		Jan 7-11	Mar 4-8
Advanced PostScript			
Variable Data PostScript			
Troubleshooting PostScript		Feb 4-6	Mar 17-20

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?

Upcoming inexpensive e-books

I'm planning on releasing a series of relatively short (30-40 pages) e-books in January, including a four-book series on Acrobat forms. I'll tell you more about them next time.