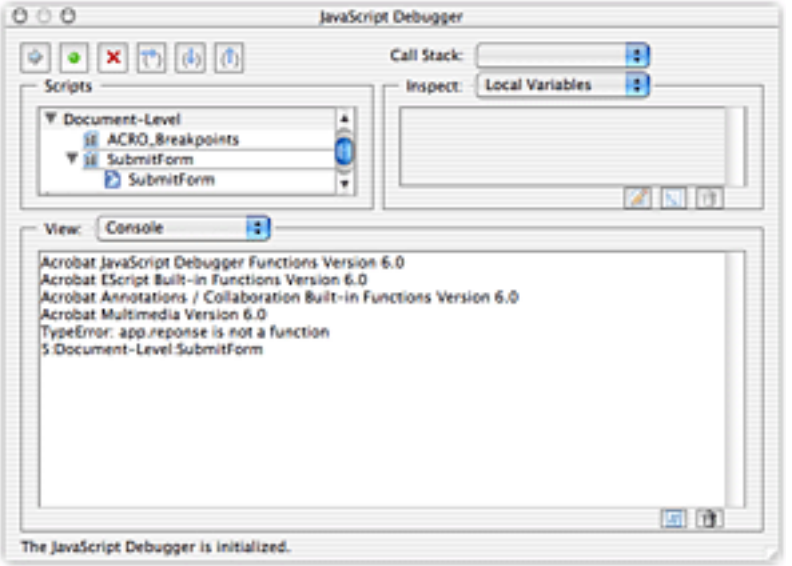
Table of Contents

Acumen Training

JavaScripter: Acrobat 6 Pro's JavaScript Debugger

The JavaScript debugger is available only in *Acrobat 6 Professional.* For brevity in this article, I shall refer to this as "Acrobat," but remember that this month's topic applies only to *Pro.*

Acrobat 6 Professional introduces a feature of great importance to JavaScript programmers: a JavaScript *debugger.* This is a vastly useful tool that will allow a JavaScript programmer to much more effectively track down bugs and weaknesses in scripts within an Acrobat document.



You will have already seen the debugger if you encountered any errors in your JavaScript code. When a JavaScript error occurs, Acrobat 6 drops you into the debugger at the point in your code where the error takes place.

This month, we're going to take a preliminary look at how to use this debugger. My assumption is that you have done at least some JavaScript programming with Acrobat, consistent with, say, having read my book *Extending Acrobat Forms With JavaScript.*

(Hint. Hint.)

Next Page ->

Background

What's a Debugger?

A debugger is a piece of software whose purpose is to execute other programming code in such a way that you can easily gather information about that other code, usually with an eye toward diagnosing and fixing errors. All professional programming environments, such as Microsoft's *.net* environment and Metrowerk's *CodeWarrior,* provide a debugger to help the programmers fix their code.

Debugger Features

Acrobat's JavaScript debugger has all the features common to debuggers in other environments. Specifically, the JavaScript debugger lets you do the following:

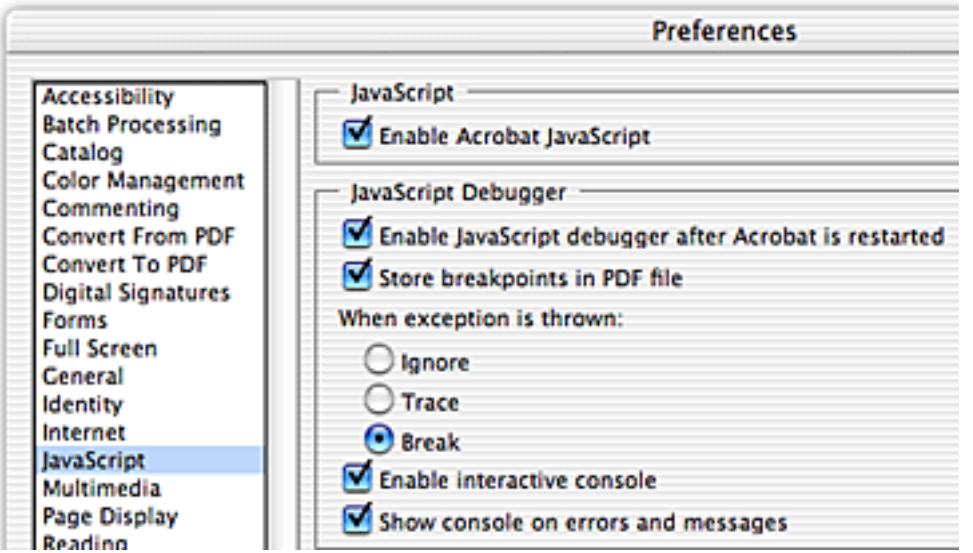
- Execute a particular script within the current Acrobat document.
- Execute that JavaScript one line at a time, pausing after each line.
- Define *breakpoints* within the JavaScript. When you run the script—either through the debugger or by clicking on a button or other trigger within your Acrobat file—Acrobat will halt execution of the script at the breakpoint's position and launch the debugger.
- Examine and change the values of any variables or properties defined within a paused script.

Next Page ->

Debugger Basics

Enabling the Debugger

To use the Acrobat debugger, you must first ensure that it is turned on. To do this, go to the Acrobat Preferences and select *JavaScript* from the list of categories on the left side.



There are two checkboxes that are of particular importance to us among these preferences:

- *Enable JavaScript Debugger…*

  Make sure this checkbox is selected. Note that you will need to restart Acrobat before the debugger becomes active.

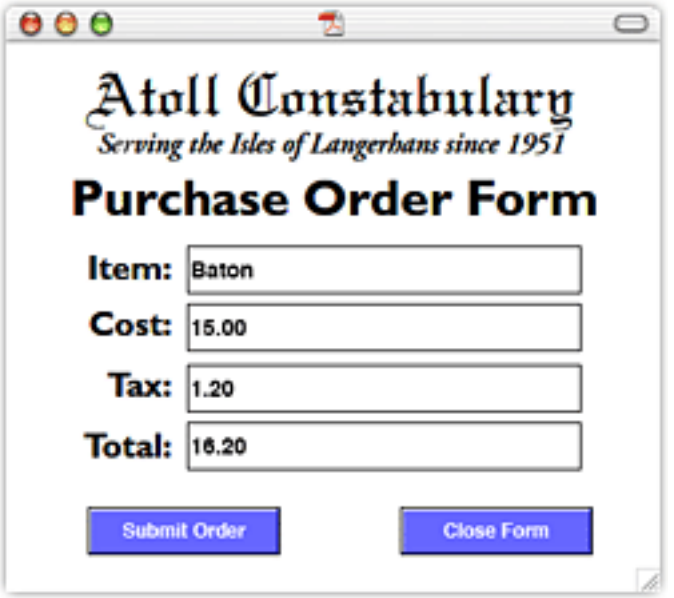
- *Store breakpoints in PDF file*

  All your breakpoints will be saved when you save an Acrobat document. This is very convenient when debugging a complex form. (Make sure remove all your breakpoints—or turn this feature off—when you save a form for distribution.)

Next Page ->

Sample File

In the course of this article, we are going to be discussing how to use the JavaScript debugger with a particular sample Acrobat file, pictured at right. This file is available on the Acumen Training website's Resources page under the name *JSDebugger.zip.* You may wish to download this file before continuing with the rest of this article so that you can follow along.



We are going to be using the debugger with two scripts in this form:

- A Document JavaScript attached to the file.
- The Mouse Up JavaScript attached to the *Close Form* button.

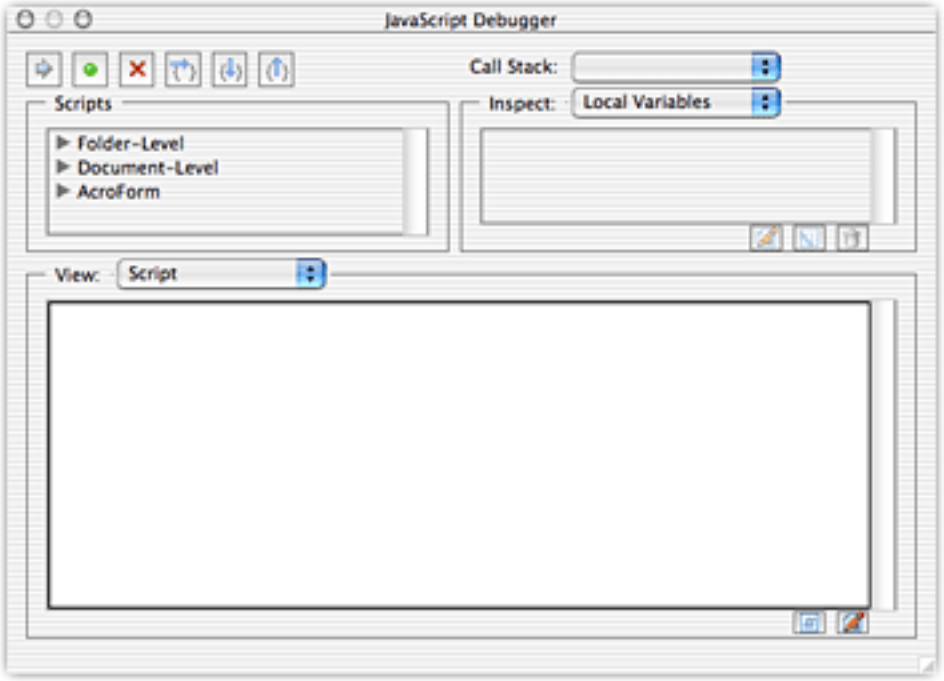
This file is taken from the JavaScript book, by the way; it's the sample file for Chapter 14.

Launching the Debugger

Acrobat automatically launches the debugger when an error takes place in a JavaScript. For debugging purposes, you will often want to explicitly launch the debugger yourself by selecting *Advanced>JavaScript>Debugger.*

Next Page ->

Acrobat will present you with the Debugger window, as at right. This window has several sets of controls of special interest to us:

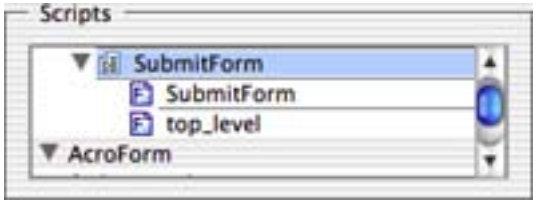


- A set of buttons that control the execution of the script we are debugging.

  These buttons let us execute one line of code, run the rest of the script, etc. We shall discuss these in more detail in a moment.

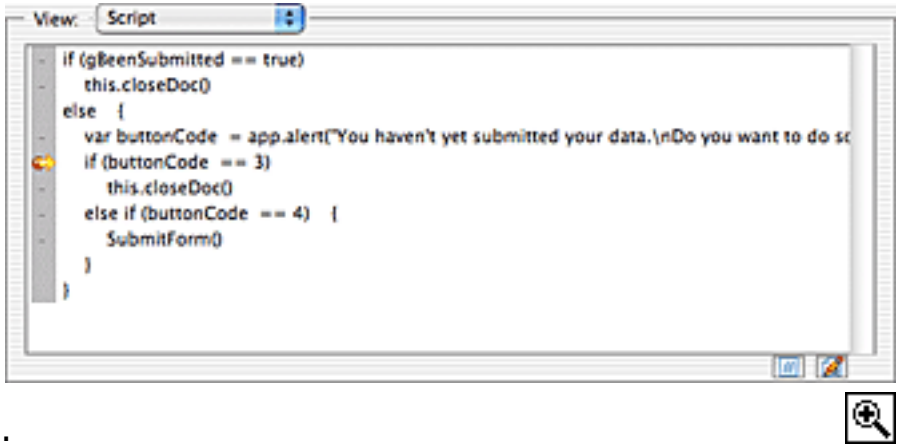
- A list of all the scripts available to this Acrobat file. These are organized according to type of script: Folder-level (executed by Acrobat at start-up), Document-level, or Form scripts.



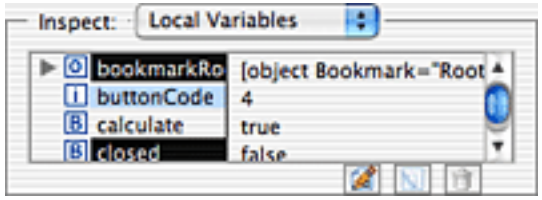
Next Page ->

- A large text box that lets us see the JavaScript that we are currently executing. The line of code that will be executed next is marked with an arrow.

  Using the pop-up menu above this text box, we can have the box alternatively display the JavaScript console.



- A list of the names and values of all the variables and object properties existing at this part of the code. In the illustration at right, we can see that the variable *buttonCode* currently has a value of *4.*



  The pop-up menu allows you to select among local variables (visible only within this script), global variables (visible throughout the document), breakpoints (where execution should pause) and something we won't discuss in this article: *watches.*

  You can also change the values of the variables visible in the list.

Next Page ->

Using the Debugger

Let us use the debugger to step through the execution of part of one of the scripts in our *Constabulary* form. The script I have in mind is the *Mouse Up* script for the form's "Close Form" button:

```
if (gBeenSubmitted == true)
     this.closeDoc()
else {
     var buttonCode  = app.alert("You haven't yet submitted your data.
                                          \nDo you want to do so now?",2,3)
     if (buttonCode  == 3)
          this.closeDoc()
     else if (buttonCode  == 4)     {
          SubmitForm()
     }
}
```

I want to examine the execution of this script starting at the line

```
     if (buttonCode == 3)
```

to make sure it is running correctly.

I'm going to do this by setting a *breakpoint* (a pausing point) at that line in the script and then single-step through the script from that point, on.

You may want to follow along using the sample file from the website.

Next Page ->

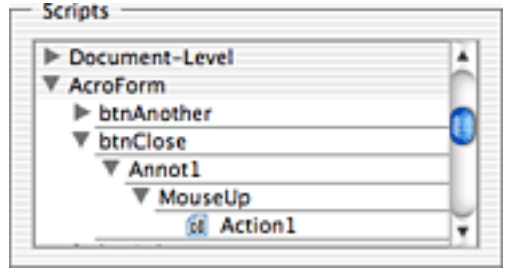
Setting the Breakpoint

Let's start by setting the breakpoint where we want our script to pause in its execution.

- Start with the form open in Acrobat and then launch the debugger (select *Advanced>JavaScript>Debugger).*

  You will be looking at the debugger window, as we saw earlier.

- In the list of scripts, open the *AcroForm* sublist and select *btnClose>Annot1>MouseUp>Action1,* as at right.

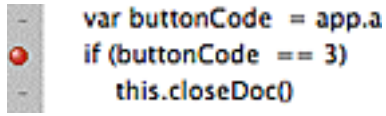
  

  The large text field in the lower half of the window will now display the JavaScript code for the *Close Form* button's *MouseUp* event, as below right.

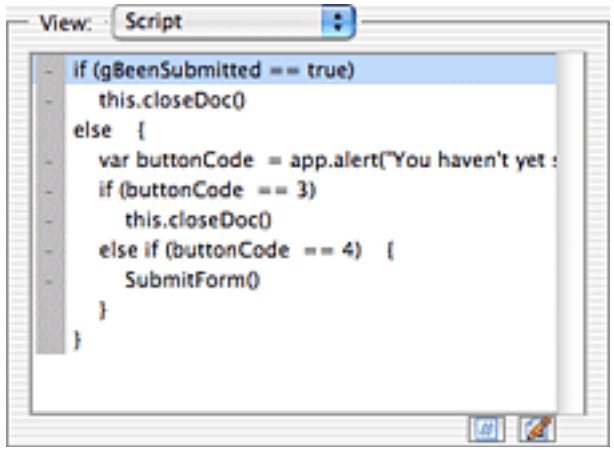
- Click on the minus sign to the left of the line

  ```
  if (buttonCode == 3)
  ```

  The minus sign turns into a small, red "bullet," indicating that you have placed a break-point at that line.

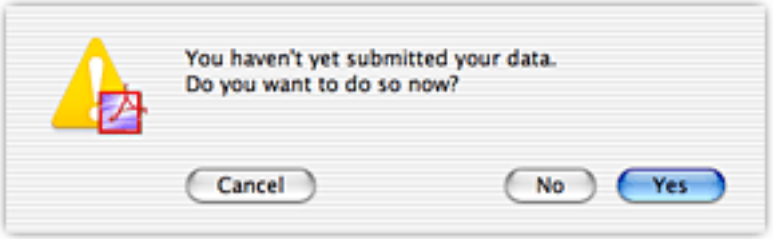
  Whenever the this script executes, Acrobat will pause the script and launch the debugger when it reaches this line.



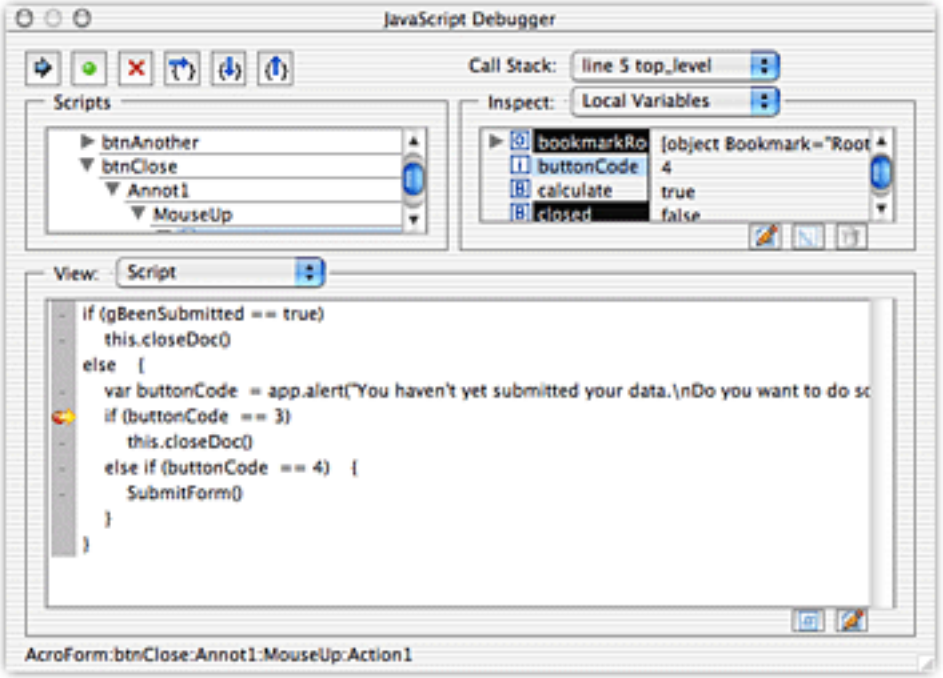
Next Page ->

Running the Script

Now close the debugger window and click on the form's *Close Form* button. The button's JavaScript will put up a dialog box asking if you want to Submit the form before closing, as at right. Click the *Yes* button.



Acrobat will immediately open the debugger and pause execution of the JavaScript. The debugger window will look like the illustration at right.



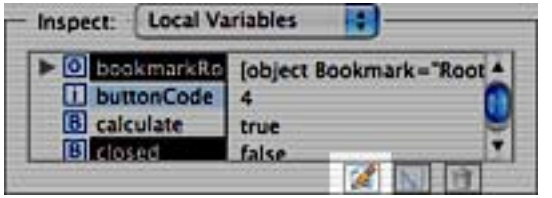
There is an arrow superimposed on the breakpoint bullet, indicating this is where we stopped.

Note that the variable list (above and to the right of the code listing) is now populated with the variables available to this script. Scroll down this list and you will eventually find the variable *buttonCode,* whose value will be *4* if you clicked the *Yes* button in the earlier alert.

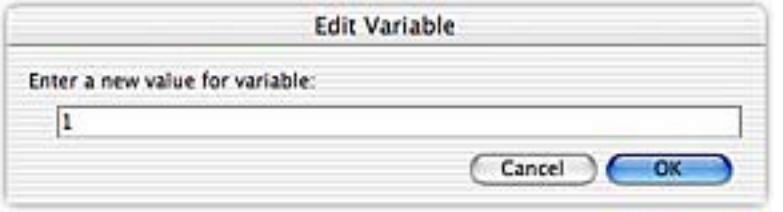
Next Page ->

Acumen Journal: Acrobat User 10

Changing Variable Values



While you are paused in a script you can examine the values of any of the variables in the Variables List. You can also change those values (perhaps to see how your script behaves with rarely-encountered values) by selecting a variable and clicking on the "Change Value" button, just below the list (highlighted at right).



Acrobat will present you with a dialog box that lets you type in a new value for the selected variable.

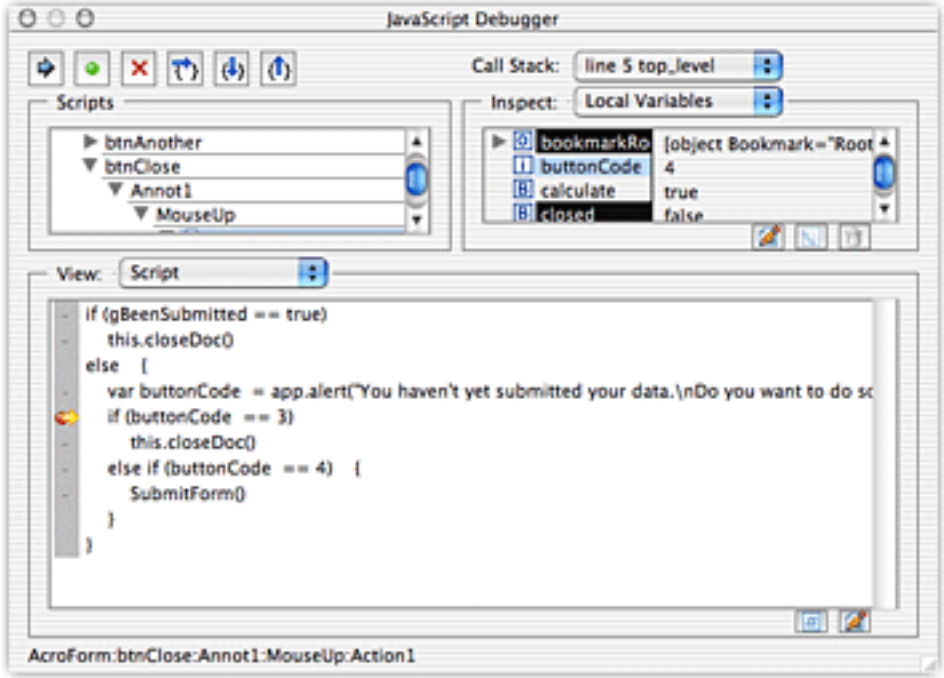
Be careful with this; in particular, don't change the values of any variables you did not create in your own scripts. The Variable List contains *all* the variables that exist at the time the break occurred in your script, including variables created and used by Acrobat, itself. If you change Acrobat's variables, you may change its behavior in unpleasant ways (such as provoking a crash, perhaps).

Next Page ->

Stepping Through Code

The yellow arrow lying on top of the breakpoint bullet points to the "current line" in our script, the next line to be executed.

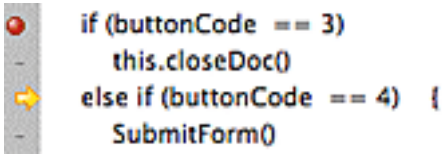
The six navigation buttons at the upper left of the debugger window allows us to execute our script in a precise, controlled way.



The "Step Over" Button

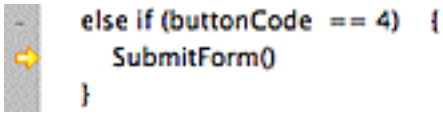
Click the "Step Over" button, pictured at left. This button tells the debugger to execute the current line of JavaScript code (the line marked with the yellow arrow) and pause at the next executable line in this script.

In our case, the debugger executes the *if* command, which looks at the boolean expression "`buttonCode == 3`". This expression evaluates to *false,* since you clicked the *Yes* button in the *Do you want to submit…* dialog box (which sets *buttonCode* to *4).* The debugger moves the "current line" arrow to the next executable line, the *else* statement. (It skips over the call to *this.closeDoc,* since the *if* boolean expression is false.)



Next Page ->>

Click the Step Over button again and the debugger will execute the *else if* line. Since this boolean expression (`buttonCode == 4`) is true, execution will move to the invocation of *SubmitForm.*



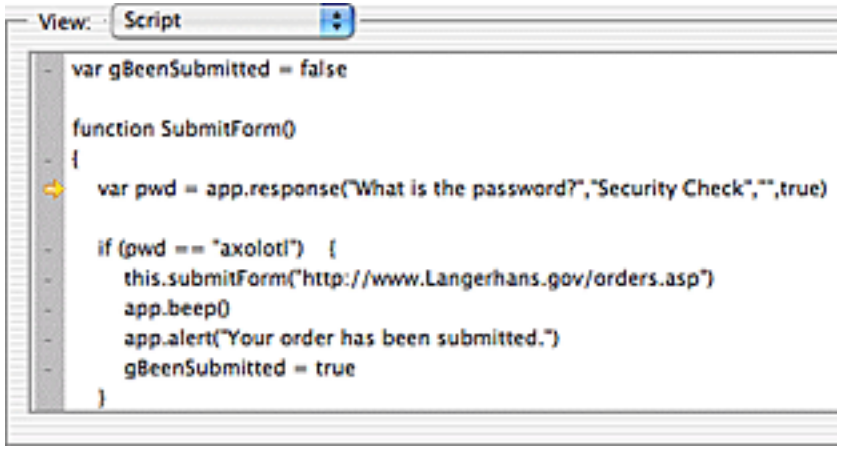
The "Step Into" Button

Now we have an interesting situation. If we click the Step Over button again, the debugger will execute *SubmitForm* and move the Current Line arrow to the next line in the script (which, in our case, is the end of the script).

However, *SubmitForm* is, itself, a JavaScript procedure. If we want to follow the execution of our JavaScript *into* the execution of *SubmitForm,* executing that procedure's statements one at a time, we need to click on the *Step Into* button, shown at left.

When we do so, the Current Line arrow will move to the first line in the definition of *SubmitForm.* We can now use the Step Over button to step through the remainder of *SubmitForm.*

When execution reaches the end of *SubmitForm,* another click of the *Step Over* button returns us to the *Close Form* script, whence we came.



Next Page ->

The "Step Out" Button

If, while stepping through the *SubmitForm* procedure, we want to just execute the rest of that procedure and return to the *Close Form* script (or whatever script executed *SubmitForm),* we can do so easily by clicking the *Step Out* button.

This button tells the debugger to execute the remainder of the current procedure and then pause upon returning to the script that executed the procedure.

Finishing Up

When you are finished stepping through your script, you can click the *Resume Execution* button. This tells Acrobat to continue executing your script without pausing at each line. THe script will execute all the way to the end (as it would if you weren't running the debugger) or until it encounters another breakpoint.

Halting Execution

If you need to completely halt the execution of your JavaScript, you can click on the *Quit* button. This tells Acrobat to immediately cease execution of your JavaScript and exit the debugger.

Setting Interrupts

The *Set Interrupt* button tells Acrobat to launch the debugger the next time the user does something that triggers a JavaScript.

Thus, rather than opening a script in the debugger and explicitly setting a breakpoint, you can launch the debugger and click the *Set Interrupt,* then click on a button or take some other action in your form that runs a JavaScript.

Next Page ->

Acumen Journal: Acrobat User 14

A Welcome Addition

Acrobat 6 Pro's new JavaScript debugger is a very powerful tool for examining how your scripts are working (or not). If you have never used such a debugger before, you will find this new feature makes for a whole new scripting experience.

You'll wonder how you did without it!

The Official Documentation

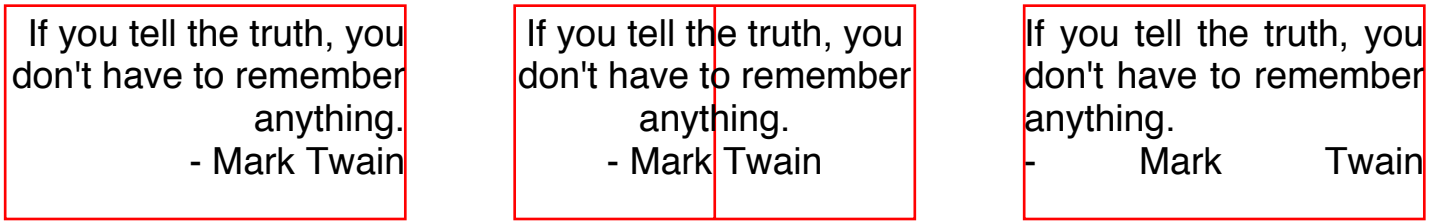
For complete information on using the JavaScript debugger, you should look at Adobe's *Acrobat JavaScript Scripting Guide* (Tech note 5430). This is available from Adobe's website; a link to this PDF file is also on the Acumen Training Resources page.

Return to Main Menu

Fully Justified Text

The *show* operator is the primary PostScript tool for printing text. It takes a string from the operand stack and prints that string in the current font starting at the current point.

In Western typography, there is a printing task that continually comes up and must be addressed: printing text that is right-aligned, centered, or fully-justified. The first two of these are very easy tasks; the third is less obvious in its implementation.



This month, we shall see how to print a string fully justified—both flush right and left—within the left and right margins.

Next Page ->

Easy Ones First

Print text right aligned or centered is relatively easy. To print text right aligned against the current point you simply do the following:

If you tell the truth, you don't have to remember anything.
- Mark Twain

Right-aligned Text

- Calculate the width of the string.
- Move the current point (the red dot at right) back by that distance.
- Print the string with *show.*

This month's examples are in the file *alignedText.zip* on the Acumen Training *resources* page.

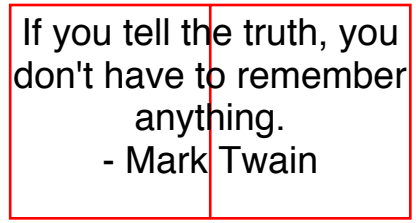
Here is a PostScript procedure, *rshow,* that does the job:

```
/rshow          %  (str)  =>  ---
{ dup stringwidth pop    % Find the string's width on the page
  neg 0 rmoveto          % Move the currentpoint back by the width
  show                   % Print the string
} bind def

/Helvetica 14 selectfont
100 200 moveto
(If you tell the truth, you) rshow
100 180 moveto
(don't have to remember) rshow
100 160 moveto
(anything) rshow
```

Next Page ->

Centered Text

Centered text follows almost exactly the same procedure, except that we move the current point back by only half the string's width, as in the following definition of *cshow:*



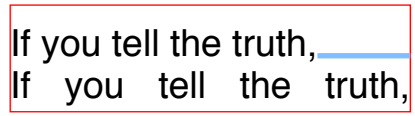

```
/cshow          %  (str)  =>  ---
{ dup stringwidth pop    % Find the string's width on the page
  -2 div 0 rmoveto       % Move the currentpoint back by half the width
  show                   % Print the string
} bind def

/Helvetica 14 selectfont
100 200 moveto
(If you tell the truth, you) cshow
100 180 moveto
(don't have to remember) cshow
100 160 moveto
(anything) cshow
```

Next Page ->

Full Justification

Printing a string with full justification is trickier than simple right or center alignment. The simplest method for preforming this task is as follows:

- Calculate how much whitespace would be left between the end of the string and the right margin if the string were printed normally (the blue line in the diagram at right).



- Print the string, distributing the end-of-line whitespace between the words, as in the second line of text in the diagram.

There are other ways, arguably better, of carrying out full justification; we'll discuss these shortly. For now let's stay with the simple method.

The *widthshow* operator

Key to printing fully justified text is the *widthshow* operator:

```
Δx Δy cc (str) widthshow => ---
```

This operator takes an *x* and *y* offset, a character code, and a string. It prints the string, offsetting the current point by *Δx* and *Δy* after each instance of the character code *cc.* In printing fully justified text, the character code will be that of the space character, typically *32.*

Next Page ->

The *jshow* procedure

The following program defines a procedure, *jshow,* that takes a string and the position of the right margin and prints the string fully justified between the current point and the specified margin. The output from this program is at right.

If you tell the truth, you don't have to remember anything.
- Mark Twain

Terminology: *measure*

In typesetting, the "measure" is the distance between the left and right margins.

```
/jshow          %  (str)  rm  =>  ---
{    currentpoint pop sub            %  Calculate the "measure" (rm - current x)
     1 index stringwidth pop sub     %  Calc. measure - string's width
     /spcCount 0 def                 %  spcCount will count spaces
     1 index ( )                     %  stk: (str) wtspc (str) ( )
     {    search                     %  Begin loop with call to search
          { /spcCount spcCount 1 add def  %  Space found: incr. spcCount...
            pop                      %  ...and discard the word before the space
          }
          { pop exit }               %  Space not found: we're done; leave loop
          ifelse
     } loop
     spcCount 0 eq          %  Did we find any spaces?
     { pop show }           %  No: discard measure and show string
     { spcCount div         %  Yes: divide the whitespace amt by spcCount
       0 32                 %  stk:   (str) Δx 0 32
       4 -1 roll            %  stk:   Δx 0 32 (str)
       widthshow
     }
     ifelse
} bind def
```

Next Page ->

Acumen Journal: PostScript Tech 20

```
/nl       %  This is a standard "newline" procedure. See your PS student notes.
{ 0 currentpoint exch pop 16 sub moveto } bind def

72 600 translate          %  I like to print at the origin

1 0 0 setrgbcolor         %  Print a red rectangle
0 0 150 80 rectstroke

/Helvetica 14 selectfont %  Now print our text
0 setgray
0 65 moveto

(If you tell the truth, you) 150 jshow nl
(don't have to remember) 150 jshow nl
(anything.) 150 jshow nl
(- Mark Twain) 150 jshow nl
```

Step by Step

```
/jshow          %   (str)   rm   =>   ---
```

Our *jshow* procedure will take a string and the *x* value of our right margin from the operand stack and print the string fully justified between the current point and the specified margin.

```
{     currentpoint pop sub          % stk: (str) rm-x
```

Our procedure starts by calculating the difference between the right margin and our current *x* position. This is the distance within which we shall justify our text.

Next Page ->

```
1 index stringwidth pop sub   %  stk:  (str) whiteSpcAmt
```

We then find the width our our string, using the *stringwidth* operator, and subtract this from the distance between our margins. The result, left on the stack, is the amount of "whitespace" between the string's native length (without justification) and the right margin. This is the space we shall need to distribute between the words in our string.

Remember that the *stringwidth* operator takes a string from the stack and returns the distance in *x* and *y* that the current point would move if we were to print that string.

```
(str) stringwidth => Δx Δy
```

In our case, Δ*y* will be zero, since our text prints horizontally; Δ*x* is the width of our string, the number we want for our calculation. In our PostScript code, above, we discard the *y* and subtract the *x* from the size of the measure.

```
/spcCount 0 def                  %  stk:  (str) whiteSpcAmt
```

We now need to determine how many spaces there are in the string. We start by creating a variable, *spcCount,* that will hold the number of space characters.

```
1 index ( )                      %  stk:  (str) whiteSpcAmt (str) ( )
```

We load the top of the stack with a copy of our string and a new string containing a single space character. We are going to use these as arguments in a call to the *search* operator.

Next Page ->

Acumen Journal: PostScript Tech 22

```
{ search
```

We now initiate an indefinite loop. Each time through this loop, we shall search the string for another space. If we find one, we shall increment *spcCount;* otherwise, we shall leave the loop.

The loop starts with a call to the *search* operator.

```
(str) (tgt) search   =>   (post) (tgt) (pre) true
                     or   (str) false
```

The *search* operator searches for the first instance of *tgt* in *str.* If the search is successful, *search* returns the following on the stack: the text that follows the instance of *tgt* in *str;* the target string again; the text the preceded *tgt* in the original string; the boolean value *true.* If the search fails, *search* returns the string, unchanged, and the boolean *false.*

In our case, *search* will find our string and a space character on the stack. The operator searches for the first space in our text, returning arguments appropriate to whether it finds one or not.

```
{ /spcCount spcCount 1 add def  % stk: (str) whiteSpcAmt (post) ( ) (pre)
  pop                           % stk: (str) whiteSpcAmt (post) ( )
}
```

If we successfully find a space character, we increment *spcCount* and discard the word preceding the space. having done this, the two topmost items on the stack are exactly what we need to find the *next* space in our text: a string holding the text that followed the space character in our original string and a string containing a space character.

Next Page ->

```
{ pop exit }          % stk: (str) whiteSpcAmt
ifelse
```

If *search* fails to find a space character, we discard the copy of the string left on the stack and exit from the loop.

```
} loop                % stk: (str) whiteSpcAmt
```

This is the end of our loop. We continue circling through this loop, each time looking for a space and incrementing *spcCount* until we run out of spaces, at which point we exit from the loop.

When we exit the loop, *spcCount* will contain the number of spaces in our original string. The operand stack will contain our original string, unchanged, and the length of the whitespace that we need to distribute among the words. Both of these were on the stack when we entered the loop; the loop had no effect on them.

```
spcCount 0 eq
```

We check to see if *spcCount* is zero, that is, if our original string had no spaces in it. If this is true, we are going to just print our string left-justified, using *show.*

```
{ pop show }
```

This is the *true* clause of our *ifelse,* executed if *spcCount* is zero. In this case, we pop the whitespace distance off the stack and then *show* the string. That is, if there are no spaces in our string (it consists of only one word), then we print our string left justified.

Next Page ->

```
{ spcCount div       % stk: (str) Δx
  0 32               % stk: (str) Δx 0 32
  4 -1 roll          % stk:  Δx 0 32 (str)
  widthshow
}
ifelse
```

If *spcCount* is not zero, meaning our string has more than one word in it, then we print the string using *stringwidth:*

- We divide the whitespace amount (still on the stack after exiting our loop) by *spcCount,* yielding the amount of space (called *Δx* in the comments above) we need to add between each pair of words in our string.
- We push a zero and 32 on the stack (the *y* offset and the character code for space).
- We roll the stack contents into the correct order for a call to *widthshow.*
- We execute *widthshow.*

```
} bind def
```

This ends our definition of *jshow;* now it's time to use the new procedure.

```
/nl
{ 0 currentpoint exch pop 16 sub moveto } bind def
```

We define a fairly standard "newline" procedure. This procedure moves the current point down by 16 points and back to a left margin of zero. I'm not going to step through this in detail; you have examined a very similar procedure if you have taken

Next Page ->

either the *PostScript Foundations* or *PostScript for Support Engineers* class. (You do still have your student notes, don't you?)

```
72 600 translate
```

We move the origin to some convenient place on the page. (This is just because I have a bias toward drawing things at the origin.)

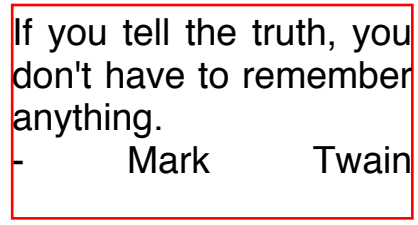
```
1 0 0 setrgbcolor
0 0 150 40 rectstroke
```

We draw a red rectangle into which we shall be drawing our justified text.

```
/Helvetica 14 selectfont
0 setgray
0 65 moveto
```

We set our current font to Helvetica, set the current color to black, and move to an initial position on the page.

```
(If you tell the truth, you) 150 jshow nl
(don't have to remember) 150 jshow nl
(anything.) 150 jshow nl
(- Mark Twain) 150 jshow nl
```



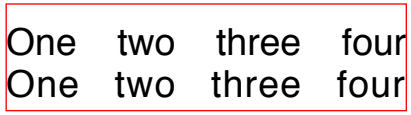
Now we print our text, fully justified.

Next Page ->

Other Justification Methods

The method of justifying text we use here is the simplest and, to many typesetters' eyes, remarkably ugly. There are many other methods that yield subtly different results that many typesetters prefer.

For example, one alternative method takes the end-of-line whitespace and distributes one quarter of it between all the characters in the string and the other three quarters between the words. This makes the amount of extra space between words less jarring to the eye. (In the illustration at right, the bottom illustration uses this method; the top illustration uses the simple method of our *jshow* procedure.)



awidthshow

You do such justification with the *awidthshow* operator:

Δx_1 Δy_1 cc Δx_2 Δy_2 (str) awidthshow

The *awidthshow* operator prints the string, adding Δx_2 and Δy_2 after each character and an additional Δx_1 and Δy_1 after each instance of character code *cc.*

The file *alignedText.zip,* on the Acumen Training Resources page, contains a file *jshow2.ps,* that defines a procedure that implements this ¼–¾ method of justification.

I'll let you examine it on your own.

Return to Main Menu

Schedule of Classes, July – Sept 2003

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**			Sep 29–Oct 2
PostScript Foundations	Jul 14–18		Sep 15–19
Variable Data PostScript		Aug 18–22	
Advanced PostScript		Aug 25–29	
PostScript for Support Engineers	Jul 28–Aug 1		
Jaws Development		*On-site only*	

On-site Classes

These classes may also be taught on your organization's site. Go to Acumen Training's On-site Classes page for more information.

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

Troubleshooting with Enfocus' PitStop

Acrobat Class Fees — *Acrobat Essentials* and *Creating Acrobat Forms* ($\frac{1}{2}$-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: www.acumenjournal.com/AcumenJournal.html

Return to First Page

What's New at Acumen Training?

PDF File Content and Structure

September will see the introduction of a new, much-requested engineering class.

PDF File Content and Structure is a four day, hands-on class on PDF files, intended primarily for printer engineers and support personnel. The course describes how a PDF file is organized and constructed, how it draws on the page, how fonts are handled, everything, in fact, that has an impact on the interpreting of a PDF file by a printer. Although the course will be extremely useful for people who must construct PDF files, it is intended primarily for people working with printing devices that must consume PDF files.

Topics

The following is a partial list of topics in the course:

- PDF File Format Overview
- PDF Data Types
- Objects and Streams
- The Page Tree
- Drawing Commands
- Text Commands
- Images
- Transparency
- Linearized PDF
- Font embedding
- Color in PDF
- PDF from PostScript *(pdfmark)*

Availability

The first public presentation of *PDF File Content and Structure* will be **Sept 29 – Oct 2.**

The course will be available for on-site classes beginning in October.

For More Information

For more information about this class, go to the Acumen Training *PDF FC&S* web page.

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 think that perhaps Frodo Baggins was right?

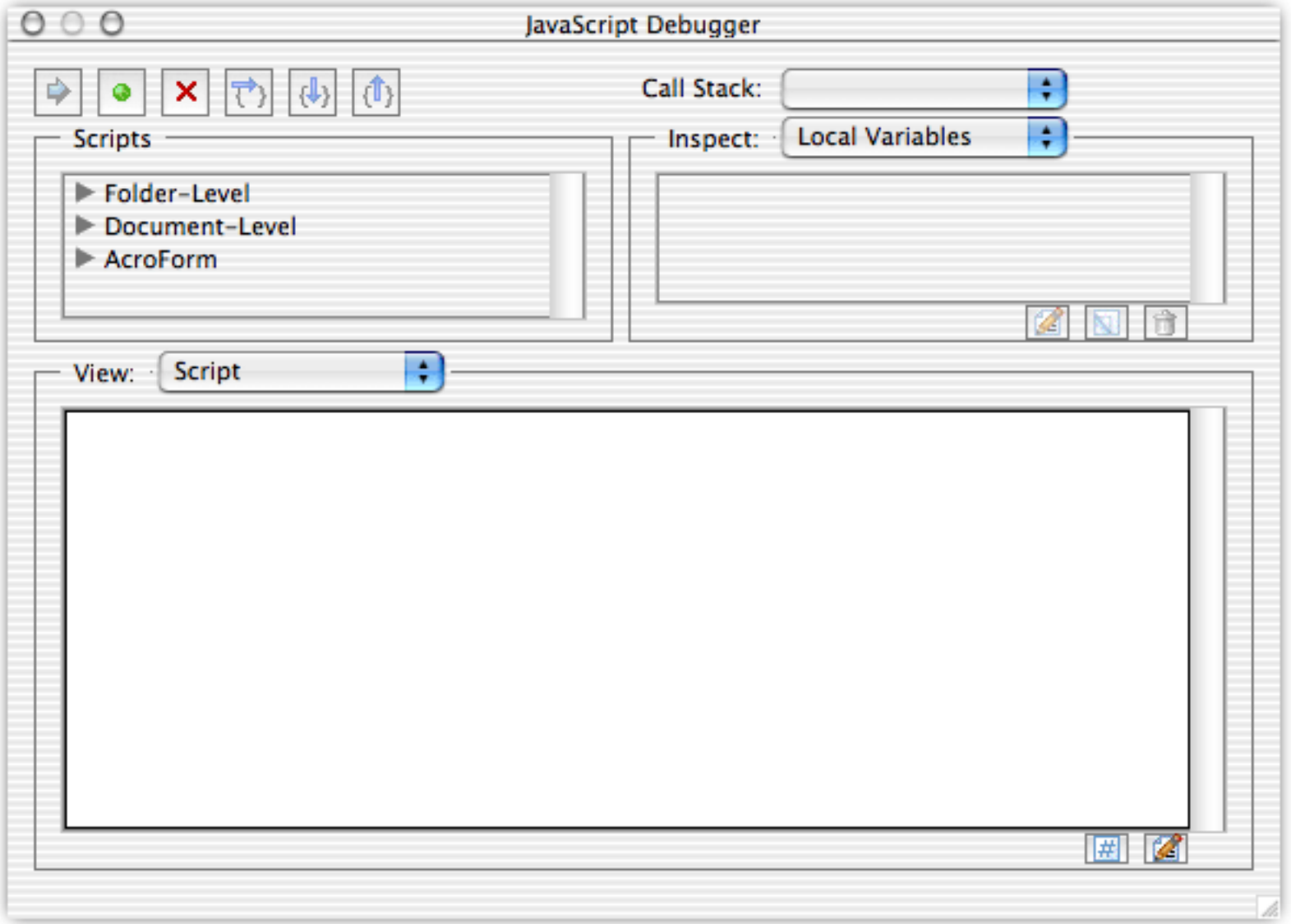
Suggestions for articles. Each Journal issue contains one article each on PostScript and Acrobat. What topics would you like me to write about?

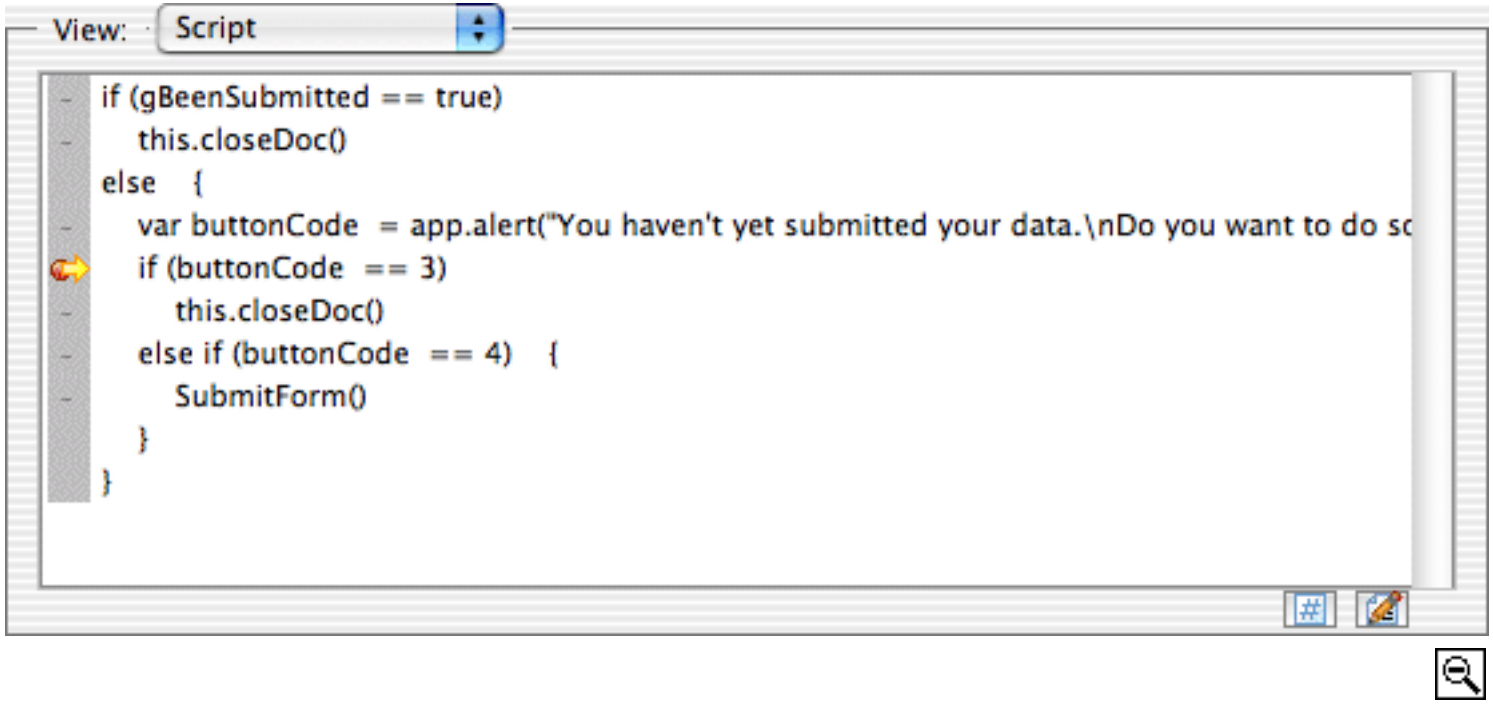
Questions and Answers. Do you have any questions about Acrobat, PDF or PostScript? Feel free to email me about. I'll answer your question if I can. (If enough people ask the same question, I can turn it into a Journal article.)

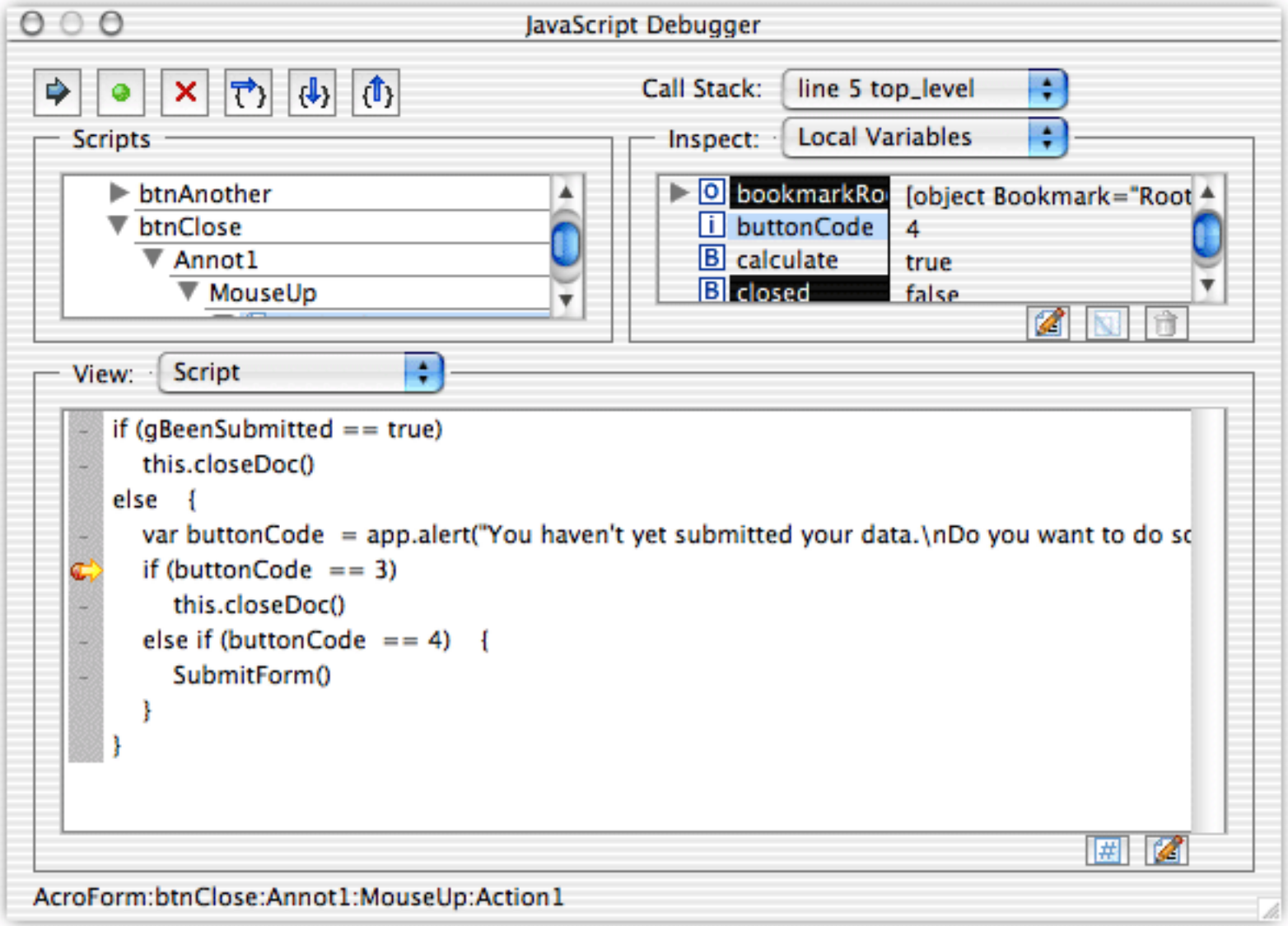
Please send any comments, questions, or problems to:

journal@acumentraining.com

Return to Menu


JavaScript Debugger
Call Stack:
Scripts
Inspect:
Local Variables
Folder-Level
Document-Level
AcroForm
View:
Script


View:
Script
if (gBeenSubmitted == true)
this.closeDoc()
else {
var buttonCode = app.alert("You haven't yet submitted your data.\nDo you want to do sc
if (buttonCode == 3)
this.closeDoc()
else if (buttonCode == 4) {
SubmitForm()
}
}


JavaScript Debugger
Call Stack:
line 5 top_level
Scripts
Inspect:
Local Variables
btnAnother
btnClose
Annot1
MouseUp
bookmarkRo
[object Bookmark="Root
buttonCode
4
calculate
true
closed
false
View:
Script
if (gBeenSubmitted == true)
this.closeDoc()
else {
var buttonCode = app.alert("You haven't yet submitted your data.\nDo you want to do so
if (buttonCode == 3)
this.closeDoc()
else if (buttonCode == 4) {
SubmitForm()
}
}
AcroForm:btnClose:Annot1:MouseUp:Action1