Table of Contents

The *Acumen Journal* is back!

Now that the *PDF File Content and Structure 2* class is finished, the Journal will be back on its roughly-every-second-month schedule. Thanks for your patience last year.

JavaScript: All- and None-of-the-Above Checkboxes

Example on Website

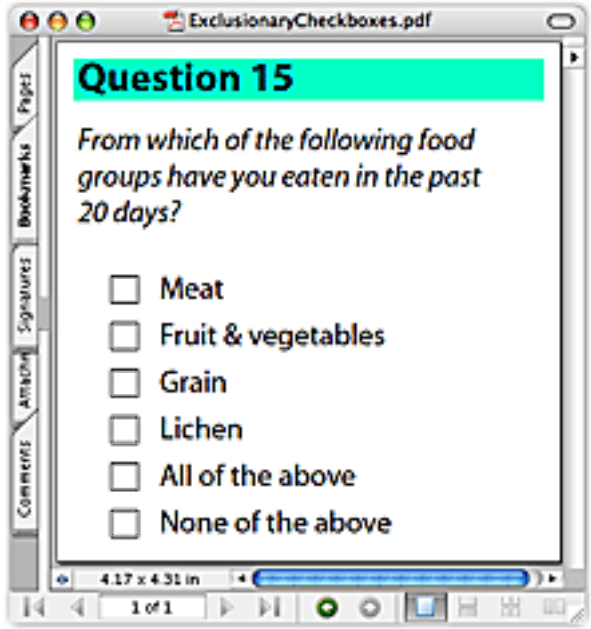
As usual, this month's sample file is available on the Acumen Training Resources page. Look among the Acrobat examples for *All of the above.zip*.

A former Acrobat class student recently asked me about a problem she wasn't sure how to solve. She was constructing a questionnaire as an Acrobat form; nothing too complex, just a series of multiple-choice questions with checkboxes associated with each of the answers. Many of the questions had "All of the above" and "None of the above" responses for which she wanted the following behavior:

- Selecting "All of the above" would cause all of the item checkboxes to become selected.
- Selecting "None of the above" would deselect all of the item checkboxes.
- Selecting any item checkbox would deselect the "None of the above" checbox.
- Deselecting any item checkbox would also deselect the "All of the above" checkbox.

Play with the check boxes at right to see this set of characteristics in action.

This is a relatively easy task, but it does require a set of JavaScripts. Since All- and None-of-the-above checkboxes are common in questionnaires, this month's article will examine how to reproduce this behavior.



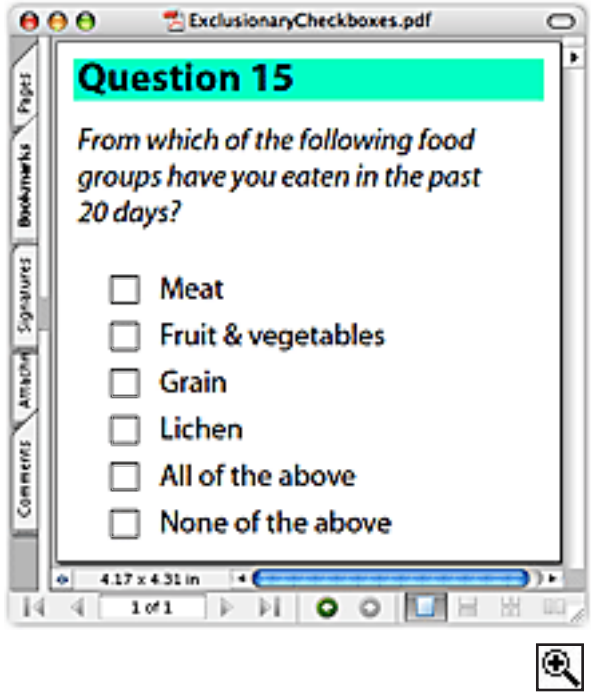
Next page ->

Acumen Journal: Acrobat User 2

The Project

In this article, we shall implement the All/None checkboxes for the questionnaire at right. This PDF form has six checkboxes, named *chkMeat*, *chkFruit*, *chkGrain*, *chkLichen*, *chkAll*, and *chkNone*.

All of these checkboxes have had their export values set to the string "Yes." Thus, setting a checkbox's value to "Yes" will select the box; setting the value to any other string will unselect the box.



Assumptions

This article assumes that you already know the basics of creating Acrobat form fields, such as checkboxes. It also assumes that your knowledge of Acrobat's JavaScript is roughly equivalent to that presented in my book *Extending Acrobat Forms With JavaScript*. In particular, you should know how to attach a JavaScript to an Acrobat form field.

Next page ->

Acumen Journal: Acrobat User 3

Overview

We are going to need to write three JavaScripts for this task, each attached to the *Mouse Up* event of one or more checkboxes:

chkAll JavaScript

Check to see if *chkAll* has been selected and, if so, select all of the food checkboxes and unselect *chkNone*.

chkNone JavaScript

Check to see if *chkNone* has been selected and, if so, unselect all of the other checkboxes.

chkMeat, etc. JavaScript

This JavaScript, attached to all of the food item checkboxes, will look to see if the *Mouse Up's* checkbox is selected. If so, it will uncheck *chkNone,* since if the checkbox is selected, then "None of the above" cannot be valid. If the checkbox is not selected, the JavaScript will uncheck *chkAll.*

Checking & Setting Field Values

event object

All three of these *Mouse Up* scripts will need to check the value of the checkbox to which they are attached. We shall do this with the predefined *event* object that is available for use in all form field JavaScripts.

Next page ->

Acumen Journal: Acrobat User 4

You may remember that that *event.target* is a reference to the form field that triggered the JavaScript; in our case, this will be the checkbox that the user clicked. The value of that checkbox is therefore available as *event.target.value*. If this value is the string "Yes" (the checkbox's export value), then the checkbox is selected; any other value indicates the checkbox is unselected.

We can also assign a value to *event.target.value* to set or clear the checkbox:

```
event.target.value = "Yes"
```

The above line sets the checkbox's value to "Yes", turning the checkbox on. Setting the field's value to anything other than the field's export value will unselect the field, clearing the checkbox.

Attaching a JavaScript to a Form Field

To briefly review how to attach a JavaScript to a form field:

1. Double-click on the field with either the appropriate form field tool or (more generally conveniently) the Object Selection tool (in the Advanced Editing toolbar).

   Acrobat will present you with the Check Box Properties dialog box (next page).

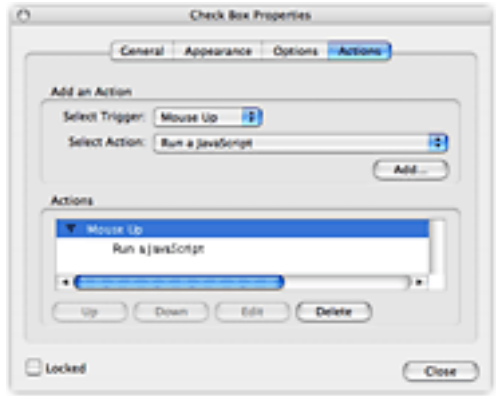
2. Click on the *Actions* tab and select the *Mouse Up* event.

3. In the Action pop-up window, select *Run a JavaScript*.

Next page ->

Acumen Journal: Acrobat User 5

4. Click the *Add* button; Acrobat will present you with a text field into which you may type your JavaScript.
5. Click the *Close* button to return to your Acrobat page.



The JavaScripts

"All of the Above"

As we discussed earlier, the *chkAll* checkbox needs a JavaScript that does the following:

- Check to see if the checkbox is selected, that is, if its value is "Yes."
- If so, do the following:
  - Select all of the food checkboxes.
  - Turn off the None-of-the-above checkbox.

Here's the JavaScript for the *chkAll* field:

```
if (event.target.value == "Yes")     {          // Checkbox selected?
     this.getField("chkMeat").value = "Yes"     // Turn on the items'
     this.getField("chkFruit").value = "Yes"    //   checkboxes
     this.getField("chkGrain").value = "Yes"
     this.getField("chkLichen").value = "Yes"
     this.getField("chkNone").value = "ABC"     // Turn off "None"
}
```

Let's look this in detail.

Next page ->

Acumen Journal: Acrobat User 6

Step-by-step

```
if (event.target.value == "Yes")    {
      ...
      ...
}
```

The first thing this script does is examine the value of the *chkAll* field to see if the value is the string "Yes." If so, then the user has just selected this checkbox and we need to turn on all of the individual food checkboxes.

We test the state of the checkbox within a JavaScript *if* statement. You may remember that the *if* keyword is followed in the script by two things:

- Parentheses containing a comparison
- One or more JavaScript statements in braces.

If the parenthetical comparison is true, then the statements in braces will be executed.

In our case, the clause in parentheses compares the value of the checkbox (obtained through *event.target.value)* with the string "Yes". If the comparison is true, we shall execute all of the statements between the braces.

Remember that the two equal signs in the parentheses are the "is equal to" comparison.

Next page ->>

```
this.getField("chkMeat").value = "Yes"
```

This is the first line between the braces, to be executed only if the *if* comparison is true. This somewhat complicated-looking line sets the value of the *chkMeat* to "Yes," selecting it in the user interface. It is a condensed version of an action we might often carry out with two lines of JavaScript:

```
var f = this.getField("chkMeat")
f.value = "Yes"
```

In the above lines, we get a reference to the form field named "chkMeat," assigning the reference to the variable *f*. We then assign the string "Yes" to the field's *value* property, which will "turn on" the checkbox.

In our actual code, we eliminated the temporary variable *f;* since *this.getField* returns a Field object, we can immediately reference that returned field's *value* property with:

```
this.getField("chkMeat").value
```

In our case, we addigned to this value the string "Yes."

```
this.getField("chkFruit").value = "Yes"
this.getField("chkGrain").value = "Yes"
this.getField("chkLichen").value = "Yes"
```

In the same manner, we turn on the other three food group checkboxes…

```
this.getField("chkNone").value = "ABC"
```

…and then turn off the "None of the above" checkbox.

Next page ->

Acumen Journal: Acrobat User 8

Again, note that to turn off a checkbox, we set its value to any string other than the field's export value. In our program, any string other than "Yes" would have served to uncheck the field.

"None of the Above"

The JavaScript for *chkNone* is similar in flavor to that for *chkAll*. It must:

- Check to see if the checkbox has been turned on.
- If so, turn off all the other checkboxes.

Here's the code:

```
if (event.target.value == "Yes")     {
     this.getField("chkMeat").value = "No"
     this.getField("chkFruit").value = "No"
     this.getField("chkGrain").value = "No"
     this.getField("chkLichen").value = "No"
     this.getField("chkAll").value = "No"
}
```

This is similar enough to the previous script that I shall not step through it in detail. However, note that the comparison clause in the *if* operator's parentheses is identical to our that in our previous script. The phrase "event.target.value" will return the value of whatever form field triggered the script. In our previous script, this clause evaluated to the value of *chkAll;* in this script, the same clause will return the value of *chkNone*.

Next page ->

Acumen Journal: Acrobat User 9

"Food Group" Scripts

Finally, each of the individual food group checkboxes needs to have a JavaScript that does the following:

- Check to see if the checkbox is selected.
- If so, turn off the None-of-the-above checkbox.
- If not, turn off the All-of-the-above checkbox.

The script can be exactly the same for all of these, since we want them to behave in exactly the same manner. Here's our JavaScript:

```
if (event.target.value == "Yes")     {
     this.getField("chkNone").value = "No"
}
else {
     this.getField("chkAll").value = "No"
}
```

This is the same *if* comparison we had before, except that now we have an else clause. If the parenthetic comparison is true, then the JavaScript line following the *if* will be executed; otherwise, the line following *else* will be executed.

Next page ->

Acumen Journal: Acrobat User 10

Note that I could have left out the braces after the *if* and *else,* since there's only a single line of conditionally-executed code in each case. Thus, the above script could have been:

```
if (event.target.value == "Yes")
     this.getField("chkNone").value = "No"
else
     this.getField("chkAll").value = "No"
```

This is a bit more streamlined, but some people prefer to retain the braces for clarity.

Done

That's all there is to it. The method we are using is a bit tedious if you have a lot of form fields that need to be turned on or off. One technique for reducing the work in that case would be to put the food field names into an array and then run through the array, turning each field on or off as appropriate.

However, we'll do that another time.

Return to Main Menu

Setting Text With a *PrintParagraph* procedure

At the base of PostScript's unique usefulness as a graphics and printer language is the fact that it is, in fact, a complete programming language. That being the case, there's nothing you can't do, one way or another, in PostScript.

Many people, particularly in the variable data printing field, have taken advantage of this characteristic to do very elaborate document layout entirely in PostScript. They write PostScript programs that read data from the input stream and then construct and print a set of documents based on that data.

In the next few months, we are going to look at some PostScript programming techniques that are common to this kind of document layout. We shall start with a common task: setting a paragraph of text between specified left and right margins. We shall define a *PrintParagraph* procedure that takes as its argument a string containing a paragraph of text; starting at the current point, *PrintParagraph* will print the contents of that string, breaking lines between words as appropriate to the margins.

If you have taken the *Variable Data PostScript* class, this will look familiar; we use a very similar procedure in that class.

Next page ->>

But First: *PrintWord*

Let's start with a simpler case: let's define a *PrintWord* procedure. This procedure takes as its argument a string containing a single word, and does the following:

- Decide whether that word, if printed at the current point, would extend past the right margin.
- If so, move the current point to the beginning of the next line; if not, leave the current point where it is.
- Print the word.

Called repeatedly with a series of words,

```
(This) PrintWord (text) PrintWord (is) PrintWord
(printed) PrintWord (between) PrintWord
(the) PrintWord (margins) PrintWord
```

This text is printed between the margins

PrintWord will print the complete text between specified left and right margins, as in the box at right, above.

If you have taken either the *PostScript Foundations* or the *PostScript for Support Engineers* class, you will have seen this procedure before, since we step through a similar example on the second day of both those classes.

Next page ->

Acumen Journal: PostScript Tech 13

PrintWord's Tasks

Here is what *PrintWord* must do, given a one-word string:

- Calculate the width that string would have on the page if it were printed.
- Add the string's width to the current point's *x* position.
- Compare that sum to the right margin.
- If the sum is greater than the right margin, move the current point to the start of the next line.
- Print the string.

Our *PrintWord* definition will use a set of variables to record the margins and the distance between lines; it will also define a utility procedure that moves the current point to the beginning of the next line of text.

Our sample program will draw vertical lines marking the positions of the margins and then print a block of text between those margins using *PrintWord*.

Let's look at the code.

Next page ->

Acumen Journal: PostScript Tech 14

The Code

Define our variables

```
/font /Helvetica def          % The font we'll use...
/ptSize 15 def                % ...and the point size
/leading 18 def               % The distance from one line to the next
/lm 80 def                    % Left margin
/rm 300 def                   % Right margin
```

Two procedure defs

```
% The following procedure moves the current point to the start
% of the next line; see the text for a detailed run-through
/newline          % --- => ---
{ lm currentpoint exch pop leading sub moveto } bind def


% Here's our PrintWord
/PrintWord        % (word) => ---
{     dup stringwidth pop     % Get the string's width
      currentpoint pop        % Get our current x position
      add                     % Add them together
      rm gt                   % Compare the sum to the right margin
      { newline } if          % Do a newline if width + x > rm
      show ( ) show           % Print the word and a trailing space
} bind def
```

Code on Website

As usual, the code for this month's example is on the Acumen Training Resources page. Look among the PostScript examples for *PrintParagraph.zip*. That file contains two PostScript files, *PrintWord.ps* and *PrintParagraph.ps*, the two sample programs from this issue.

Next page ->

Acumen Journal: PostScript Tech 15

Now let's draw the page

```
% Draw our margins
lm 0 moveto 0 792 rlineto
rm 0 moveto 0 792 rlineto
1 0 0 setrgbcolor
stroke

% Set the font and initialize the current point
0 setgray
font ptSize selectfont
lm 700 moveto

% Draw the text
(Twas) PrintWord (brillig) PrintWord (and) PrintWord (the) PrintWord
(slithy) PrintWord (toves) PrintWord (did) PrintWord (gyre) PrintWord
(and) PrintWord (gymbol) PrintWord (in) PrintWord (the) PrintWord
(wabe.) PrintWord

showpage
```

Let's look at this in detail.

Next page ->

Acumen Journal: PostScript Tech 16

The Code, Step-by-Step

Variable Definitions

```
/font /Helvetica def
/ptSize 15 def
/leading 18 def
/lm 80 def
/rm 300 def
```

We start by defining some variables we shall use over the course of the program. The *font* and *ptSize* variables are, of course, the font and size of the text we shall print. These are for convenience only, since they are not directly used by *PrintWord*.

The last three variables are used by either *PrintWord* or the *newline* utility routine:

leading	The distance from one baseline to the next in our block of text.
lm	The *x* coordinate of the left margin.
rm	The *x* coordinate of the right margin.

The newline Procedure

```
/newline          % --- => ---
{
```

The *newline* procedure moves the current point from its current position to the beginning of the next line. Specifically, it moves the current point to the left margin at a *y* position *leading* points lower than the current *y* position.

Next page ->

Acumen Journal: PostScript Tech 17

```
lm                              % stack: 80
```

The procedure starts by pushing the value of *lm* on the stack; eventually, this will be our new *x* value.

```
currentpoint exch pop           % stack: 80  y
```

We then execute the *currentpoint* operator, which pushes the current *x* and *y* values on the stack, in that order. The *y* value we want to decrement by *leading;* we have no use for the *x* at all, so we discard it with an `exch` (which brings it to the top of the stack) and a `pop` (which throws it away).

We now have the left margin value on the bottom of the stack and our current *y* value on the top.

```
leading sub                     % stack: 80  y-18
```

We subtract *leading* from our *y* value. We now have on the stack the *x* and *y* values for the beginning of the next line: an *x* value equal to the left margin and a *y* value 18 less than the current *y* value.

```
    moveto
} def
```

Our *newline* procedure finishes by doing a *moveto,* moving the current point to the beginning of the next line.

Next page ->

Acumen Journal: PostScript Tech 18

PrintWord

```
/PrintWord        % (word) => ---
{
     dup                  % stack:(wrd) (wrd)
```

PrintWord takes a single argument: a string containing a word of text to be printed somewhere between the margins. We need to do two things with that string: we need to determine its width, so we can decide whether it fits on the current line, and then we need to print it.

Since both of these activities will consume the string, our *PrintWord* definition begins with a pop, giving us two instances of the string on the stack.

```
stringwidth pop          % stack: (wrd) wid
```

The *stringwidth* operator consumes the top instance of our string argument, returning the *x* and *y* offset that would be applied to the current point if we were to print that string.

The *x* value is the number we want, since it will represent the width of the printed string. We have no use for the *y* offset, which will be zero in any case, since our text prints horizontally. (That's an assumption, of course, but I'll leave generalizing the procedure as an Exercise for the Student.) We discard the *y* value with a pop, leaving the text's width on top of the stack.

Next page ->

Acumen Journal: PostScript Tech 19

```
currentpoint pop          % stack: (wrd) wid x
```

We want to add the text's width to our current *x* position, which we obtain with *currentpoint*. This operator leaves both the *x* and *y* values on the stack; we have no use for the latter, so we discard it with a pop.

```
add rm gt                 % stack: (wrd) bool
```

We add together the text width and the current *x* and compare the sum to our current right margin, leaving a Boolean *true* on the stack if the sum is greater than *rm*.

If this boolean is true, it means the word does not fit on the current line, so we need to move the current point to the beginning of the next line.

```
{ newline } if            % stack: (wrd)
```

To do this, we push a procedure containing a call to *newline* on the stack and then execute the *if* operator. If the Boolean returned by *gt* is true, then *if* executes *newline,* moving the current point to the start of the next line.

```
  show ( ) show
} bind def
```

One way or another, the current point is now at the position on the page where our word should be printed; it is either at the start of the next line or has been left where it was. All we need to do now is print the string, which is all that is left on the stack.

We finish our *PrintWord* procedure by printing the string with *show*. We then print a trailing space character to separate this word from the next word in our block of text.

Next page ->

Acumen Journal: PostScript Tech 20

Draw the page We finally put some marks on the page.

```
lm 0 moveto 0 792 rlineto
rm 0 moveto 0 792 rlineto
1 0 0 setrgbcolor
stroke
```

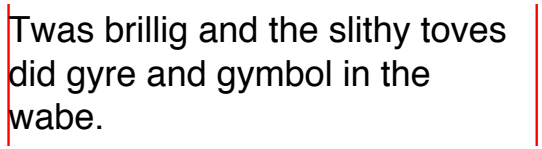
First we draw two vertical, red lines, indicating where the left and right margins are.

```
0 setgray
font ptSize selectfont
lm 700 moveto
```

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

```
('Twas) PrintWord (brillig) PrintWord (and) PrintWord
(the) PrintWord (slithy) PrintWord (toves) PrintWord
(did) PrintWord (gyre) PrintWord (and) PrintWord (gymbol) PrintWord
(in) PrintWord (the) PrintWord (wabe.) PrintWord
```

Finally, we call *PrintWord* once for each word in our block of text. The result is to print the text with line breaks as needed to remain between the specified margins.



Next page ->

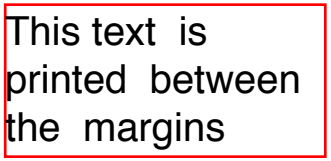
Acumen Journal: PostScript Tech 21

PrintParagraph

Now let's extend our work to print an entire paragraph at a time. Our *PrintParagraph* procedure will take from the stack a string containing an entire paragraph's text and print the text between the left and right margins. Thus, the following line:

```
(This text is printed between the margins) PrintParagraph
```

will print as at right.



This will be simpler than it sounds, since we can use our earlier *PrintWord* procedure to do the line breaks. *PrintParagraph* needs only parse individual words out of the string and hand them to *PrintWord*.

Next page ->>

The *search* operator

We shall use the PostScript *search* operator to extract the individual words from the string.

```
(src) (tgt) search   => (post) (tgt) (pre) true
                     => (src) false
```

The *search* operator looks for the first instance of a target string within a source string. If it fails to find such an instance, the operator returns the source string again and a Boolean *false,* indicating the failure.

If it finds an instance, it returns the following (from the bottom of the stack to the top):

(post) All of the source string that comes after the found instance of the target.

(tgt) The target string again.

(pre) All of the source string that comes before the found instance of the target.

true A Boolean indicating an instance of the target was found.

This is a very useful order for the return values. It makes it quite easy to print the *pre* string (which will be a single word in our case) and then immediately do another search on the remainder of the string.

Next page ->

Acumen Journal: PostScript Tech 23

The Code

Here is our new program, abbreviating the parts we saw earlier:

```
/font /Helvetica def          /ptSize 15 def     % Font and point size
/leading 18 def                                  % Dist. between lines
/lm 80 def                    /rm 300 def        % Margins

%*** newline and PrintWord definitions go here ***

/PrintParagraph      % (paragraph) => ---
{    ( )                              % This is our word delimiter
     {    search exch PrintWord       % Loop: extract and print a word
          not { exit } if             % Exit from the loop if done
     } loop                           % Otherwise, do it again.
     newline                          % Do a newline afterward
} bind def

0 setgray    font ptSize selectfont    lm 700 moveto

% Print the entire paragraph with a single call to PrintParagraph
(Twas brillig and the slithy toves did gyre and gymbol in the wabe.)
PrintParagraph
```

Next page ->

Acumen Journal: PostScript Tech 24

Step by Step...

```
/font /Helvetica def
/ptSize 15 def
...
/newline
{
     ...
} bind def

/PrintWord
{
     ...
} bind def
```

The first part of our program is identical to our earlier *PrintWord* example; it defines a handful of variables and the *newline* and *PrintWord* procedures.

```
/PrintParagraph      % (paragraph) => ---
{    ( )
```

PrintParagraph will be called with the stack holding a string containing a paragraph of text. The definition begins by pushing onto the stack a string containing a single space. This will be the target string for our eventual call to *search;* the string passed as an argument will, of course, be the source string for *search*.

Next page ->

Acumen Journal: PostScript Tech 25

`{ ... } loop`

PrintParagraph is built around a *loop* loop. Each time through this loop, we shall:

- Search the remaining source string for a space character
- Print everything that came before the space (a single word)
- Check to see if we are finished and exit the loop if so.

Remember the loop is initially called with our original string argument on the stack and a single-space string above that.

`search`

The first thing we do in our loop is call *search,* to find the next word in our source string. As we said eariler, *search* returns on the stack one of two sets of data:

- If the target string is found: `(post) (tgt) (pre) true`
- If the target is not found: `(src) false`

Note that in either case, the next word to be printed is the second item on the stack, immediately under the boolean. (The word is either the text that came before the space character or the original source string itself, if there was no space.)

`exch PrintWord`

This brings the word to the top of the stack and hands it to *PrintWord,* which prints the string, wrapping to the next line, if necessary.

Next page ->

Acumen Journal: PostScript Tech 26

The next thing to do is examine the Boolean to see if we are done:

```
not { exit } if
```

Since the Boolean returned by *search* is false if no space character was found, we reverse the Boolean with a `not` and then exit the loop if the reversed Boolean is true. Since a failed *search* leaves only the source string (consumed by *show)* and the Boolean (consumed by *if)* on the stack, the stack will be empty when we leave the loop; we have no clean-up to do.

```
} loop
```

If the Boolean returned by *search* is true, meaning it found a space character, then we do the loop again. Note that we will have consumed the Boolean, leaving on the stack the remainder of our source string and our single-space target string, exactly what we need for the *search* operation the next time through the loop.

The *loop* will proceed through each space character in the source string, each time printing the word that preceded that space, until the end of the string, at which point we exit the loop.

```
 newline
```

Having exited the loop, *PrintParagraph* executes a final *newline,* because I wanted the procedure to leave the current point at the beginning of the next line.

Next page ->

Acumen Journal: PostScript Tech 27

Final Thoughts

Setting text and doing (non-wysiwyg) page layout in PostScript has a long history. Indeed, in the Long Ago Past, when I worked at Adobe, there were no applications or drivers that generated PostScript, so the only way we could produce documents was with hand-written code, using procedures similar to *PrintParagraph*.

In a future issue, we shall look at how to embed formatting commands (such as explicit line breaks) within the paragraph.

Return to Main Menu

Schedule of Classes, January-April 2006

Following are the dates 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.

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

Technical Classes

	PDF File Content and Structure 1		Feb 27–Mar 2	
NEW!	PDF File Content and Structure 2	Jan 16–19		Apr 3–6
	PostScript Foundations	Jan 30–Feb 3		
	Variable Data PostScript			
	Advanced PostScript			Mar 13–17
	PostScript for Support Engineers		Feb 13–17	

Course Fee The PostScript and PDF classes cost $2,000 per student.

Registration Info

Acrobat Class Schedule

These classes are taught occasionally in Costa Mesa, California, and on corporate sites. Clicking on a course name below will take you to the class description on the Acumen Training web site.

Acrobat Essentials — *No Acrobat classes scheduled for this quarter.* See the Acumen Training website regarding setting up an on-site class.

Interactive Acrobat

Creating Acrobat Forms

Acrobat Class Fees — *Acrobat Essentials* and *Creating Acrobat Forms* (½-day each) cost $180.00 or $340.00 for both classes. 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: 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: http://www.acumentraining.com/registration.html

email: registration@acumentraining.com

telephone: 949-248-1241

mail: 24996 Danamaple, Dana Point, CA 92629

Back issues

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

Return to First Page

What's New at Acumen Training?

PDF File Content & Structure 2 Is Here!

The pilot *PDF File Content and Structure 2* class ran at Adobe Systems last December. The first publically-available class is scheduled for January 16–19 and, of course, the class is available for on-site. The list of topics are:

Overprinting	File Specification	Multibyte fonts
Masked Images	Halftones	Linearized PDF
Marked Content	AcroForms	Rendering Intents
Transfer Functions	Functions dictionaries	Smooth shading
Shape dictionaries	Xref streams	Object streams
Name Dictionaries	More on data structures	Print-important annotations

The prerequisite for this class is the *PDF File Content and Structure 1* class.

If you are curious about the flavor of *PDF File Content & Structure 2* class, I have posted a sample chapter from the student notes on the Acumen Training Resources page. Look among the PDF resources for "PDF FC&S Sample.pdf." The chapter included is that on PDF file specification.

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? Did it make you stare off into space and remember the '60s?

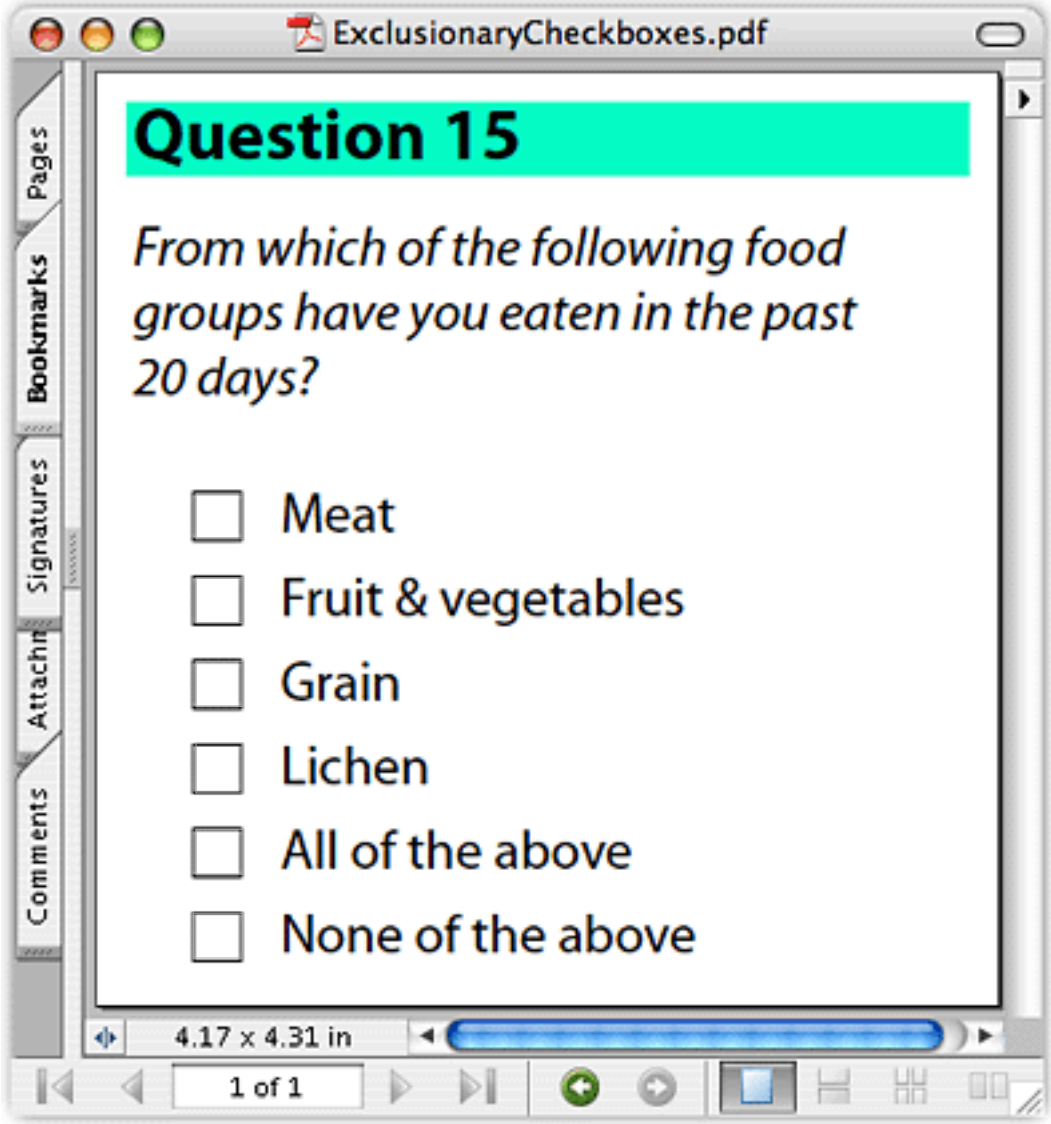
Suggestions for articles. Each Journal issue contains one article each on PostScript and Acrobat. What topics would you like me to write about?

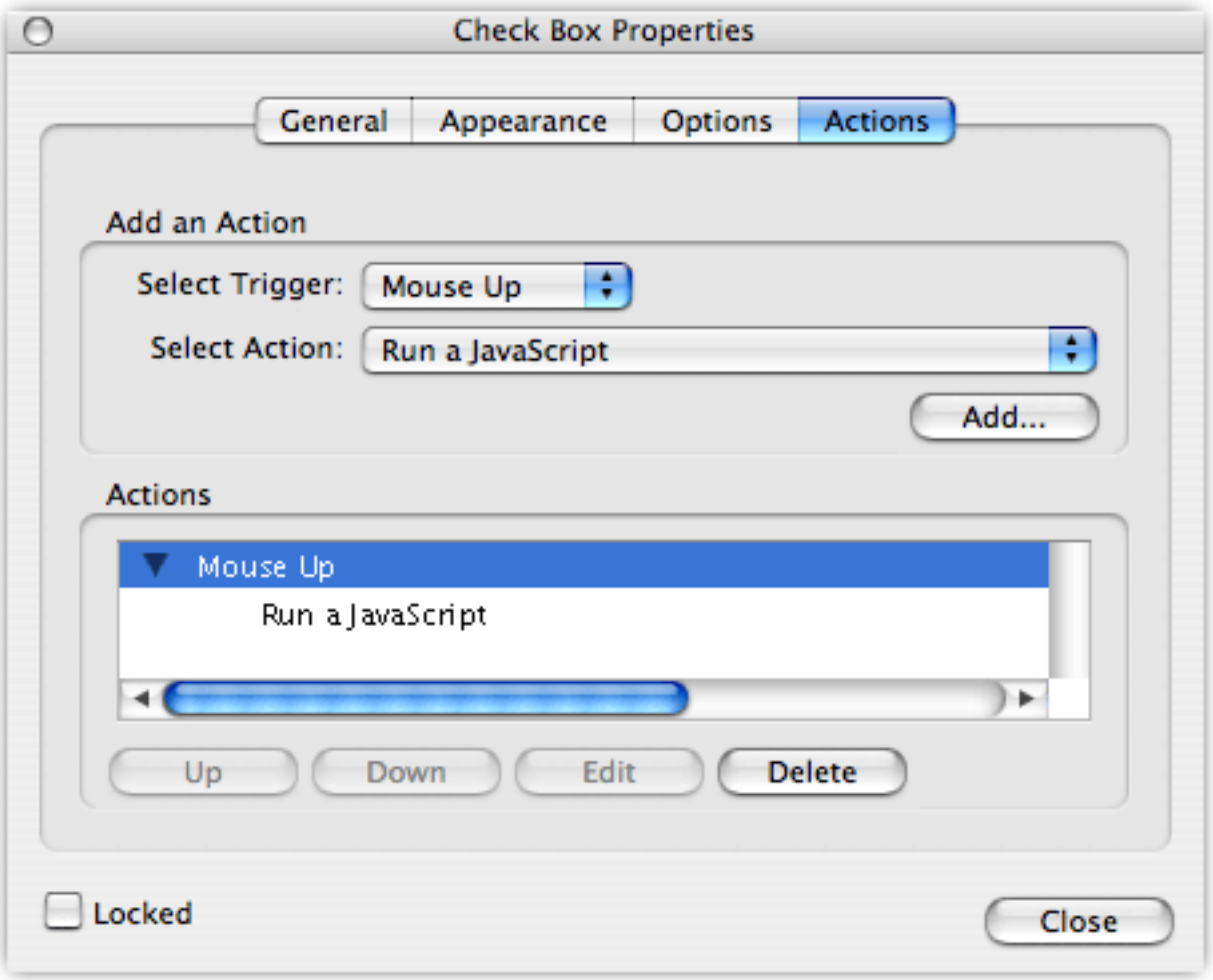
Questions and Answers. Do you have any questions about Acrobat, PDF, or PostScript? Feel free to email me about. I'll answer your question if I can. (If enough people ask the same question, I can turn it into a Journal article.)

Please send any comments, questions, or problems to:

journal@acumentraining.com

Return to Menu


ExclusionaryCheckboxes.pdf
Pages
Bookmarks
Signatures
Attachm
Comments
Question 15
From which of the following food groups have you eaten in the past 20 days?
Meat
Fruit & vegetables
Grain
Lichen
All of the above
None of the above
4.17 x 4.31 in
1 of 1


Check Box Properties
General
Appearance
Options
Actions
Add an Action
Select Trigger:
Mouse Up
Select Action:
Run a JavaScript
Add...
Actions
Mouse Up
Run a JavaScript
Up
Down
Edit
Delete
Locked
Close