Adobe Acrobat SDK

Common Tasks JavaScript Collection

A collection of JavaScript code snippets organized by function and fully indexed; for more information, click the *About* link below. The samples are separated into two general functional categories: JavaScript for forms, and JavaScript for documents. The list of samples can be accessed using the links below. This document also contains short introductory sections on creating JavaScript actions associated with events, and on testing JavaScript in PDF documents.

Documents

JavaScript samples for everything except forms, security, or collaboration.

Forms

JavaScript samples to help create and manage Acrobat forms.

July 15, 2008

Adobe

Copyright 2004—2008 Adobe Systems Incorporated. All rights reserved.

NOTICE: All information contained herein is the property of Adobe Systems Incorporated. No part of this publication (whether in hardcopy or electronic form) may be reproduced or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of the Adobe Systems Incorporated.

PostScript is a registered trademark of Adobe Systems Incorporated. All instances of the name PostScript in the text are references to the PostScript language as defined by Adobe Systems Incorporated unless otherwise stated. The name PostScript also is used as a product trademark for Adobe Systems' implementation of the PostScript language interpreter.

Except as otherwise stated, any reference to a "PostScript printing device," "PostScript display device," or similar item refers to a printing device, display device or item (respectively) that contains PostScript technology created or licensed by Adobe Systems Incorporated and not to devices or items that purport to be merely compatible with the PostScript language.

Adobe, the Adobe logo, Acrobat, the Acrobat logo, Acrobat Capture, Distiller, PostScript, the PostScript logo and Reader are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Apple, Macintosh, and Power Macintosh are trademarks of Apple Computer, Inc., registered in the United States and other countries. PowerPC is a registered trademark of IBM Corporation in the United States. ActiveX, Microsoft, Windows, and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. UNIX is a registered trademark of The Open Group. All other trademarks are the property of their respective owners.

This publication and the information herein is furnished AS IS, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies, makes no warranty of any kind (express, implied, or statutory) with respect to this publication, and expressly disclaims any and all warranties of merchantability, fitness for particular purposes, and noninfringement of third party rights.

Forms

JavaScript for Forms

Date handling

Date timestamp: first time only 8
Gets today's date as a one-time timestamp so it won't be updated each time the page is opened.

Date Parsing 8
Formats the current date using `AFParseDateEx()`.

Displaying a localized date 8
Uses the util.printd method to display a localized date, including for CJK language documents. [Acrobat 7 or higher]

Getting today's date 9
Get today's date and display it.

Calculations

Calculations: Turn calculations on/off 9
Turn off calculations to optimize performance.

Delay redrawing of appearance changes 9
Delays redrawing of all fields until all changes are completed.

Field Operations

Add/Remove a new field programmatically 10
Places a new form field on the specified page.

Add/Remove fields on multiple pages 10
Create or remove identical text fields on every page of a document.

Changing field properties 11
Change any or all of the properties of a form field.

Color: defining a custom color 12
Define a custom RGB color for use, for example, as a form field background color.

Check a check box field 12
Inspect whether a check box field has been checked.

Formatting a number in a form field 12
Shows how to format numbers in decimal, hex, and floating formats as well as formatting the value as a string.

Formatting: removing zeros so field is blank 13
Removes zeros (0.00) in auto-calculated fields.

Hiding/un-hiding an existing field based on conditions 13
Hide or expose a field based on the state of a check box field.

Highlighting a single field 13
Highlight a single field for the On Focus event, and turn off the highlighting for the On Blur event.

Highlight fields using a function 14
Create a function to highlight a field, to be called from multiple fields.

Highlighting multiple fields using a for loop 14
Defines a Document Level JavaScript function that will highlight any field that calls the function when the mouse cursor enters that field. Code can be added to all fields by running a script from the JavaScript console.

Locating fields and listing field names 15
List all form fields including their name, type, and page number.

Printing: Set field to print or not print 16
Code to control the print property of a field so the field will either print or not print.

Print selected form pages 16
Print a selected range of form pages (for example, to exclude the

Forms

instruction pages in a form document).

Rollover help

Display help text 16
Display help text for a field for the Mouse Enter event.

Search fields

Search for all fields with a specified prefix 17
Search/enumerate through all fields that begin with specified string.

Set an action for a single field 17
Attaches an action to a single field.

Set an action for multiple fields 18
Execute code from the JavaScript console to attach an action to multiple fields.

Form checking and resetting

Final field checks

Check fields before submitting 18
Check that all required fields were completed before the form is submitted.

Resetting a form 19
Resets all fields, or specified fields, to their original default value.

Validation

Validate that number is in the correct range 20
Verify that entered value is in one of two ranges.

Verify that required entry was made 20
Check required fields for a valid entry.

Convert user input to uppercase characters 21
Converts lowercase characters typed by the user into uppercase characters.

Restricting keystrokes to uppercase characters 21
Restrict keystrokes to Y (for Yes) or N (for No) .

Documents

JavaScript for Documents

Annotations

Add/remove annotations

Create an annotation 22
Illustrates how to create an annotation of type "Text".

Remove an annotation 22
Removes an annotation.

Application Version

Version checking 22
Checks the version and type of the Acrobat application, and lists all plug-ins.

Bookmarks

Create a new bookmark 23
Creates a new child bookmark at the specified location.

Add a child bookmark 23
Inserts a bookmark as a child of the bookmark in the previous sample..

Remove a bookmark 23
Removes a bookmark and all children bookmarks from the bookmark tree.

Document Operations

Create a new blank PDF document 23
Creates a new document in the Acrobat viewer.

Metadata: Get document metadata 24
Allows you to retrieve and display document metadata.

Open an existing document 24
Opens the document whose filename is specified as the string argument.

Prompt user for a document to open 24
Executes the File Open menu item in Acrobat, which prompts the user for a file to open.

Links

Add navigation links to a page 24
This sample adds navigational links at the bottom of each page in the document.

Convert specified word to a weblink 25
Search for a word and create a weblink.

Convert URL text to live links 25
Convert all content that appears to be a URL into a web link.

Remove all links on a page 25
Removes all links on the current page.

Set an action for a link 26
Sets the specified JavaScript action for the Mouse Up trigger for the link object.

Menu operations

Add a new menu item 26
Adds a new menu item and an associated action.

Execute menu item 27
Executes the menu item specified by the argument.

Hide a menu item 27
Hides a menu item so user won't have access to that item.

List menu items 27
Display a list of menu items for the Acrobat application; for use with methods that reference menu items.

Documents

Messages/Dialogs

Beep the user 28
Causes the system to play a sound to alert the user.

Prompt the user for a response 28
Prompt and get a response from the user.

Simple alert 28
Displays a message to the user.

Navigation

Add a pop-up menu for navigation 29
Creates a pop-up menu that can be triggered by any of a variety of events.

Adding page numbers to a document 29
Adds page numbers to a document in the specified location, using the specified font style.

Get the number of the current page 29
Gets the current page number.

Go to a specified page number 30
Sample code to go to a specific page number.

Go to first page 30
Sets the page being viewed to the first page (zero-based numbering) in the document.

Go to next page 30
Increments the page being viewed to the next page.

Go to previous page 30
Navigates to the previous page of the document.

Go to last page 30
Sets the page being viewed to the last page in the document.

Go to another page: create a pop-up menu 30
Creates a pop-up menu which gives the user a choice of pages to which to go. The script can be attached to a button.

Add navigational arrows on every page 31
Add navigational symbols to every page of a document for each document in a selected set.

OCG (Optional Content Group)

Beep when OCG state changes 31
Execute a JavaScript after every state change for a give OCG.

Show all OCG objects on the currrent page 31
Make visible all OCG objects in a specified name family.

Hide all OCG objects on current page 32
Hide all OCG objects in a specified name family.

Lock OCG objects 32
Locks the OCG so its state cannot be toggled through the UI.

Toggle the state of OCG objects 32
Specifies whether an OCG object is hidden or not.

Page Operations

Insert new page 32
Inserts a new page based on the selected template and then jumps to the new page.

Insert/Delete/Replace/Extract Pages 32
Executes a menu item to insert, delete, replace, or extract pages. The user is prompted to specify the appropriate pages.

Printing

Get a list of printers as a combo box 33
Shows how to get a list of printers and display result as a combo box.

Documents

Print page using printer selected in combo box 33
Code to print the current page using the printer selected by user from the combo box presented in previous example.

Print only the current page 33
Sample code to print only the current page.

Print current page (no dialog) 33
Print the current page without the usual dialog.

Searching

A simple find operation 34
A alternative to the standard Acrobat search function; it can be used to create customized searches.

Spell checking

Creating a menu item to spellcheck a document 34
Spellchecks a document and uses annotations to mark the misspelled words. [Version 6.0 or higher]

Timing

Set countdown timer 35
Prompt the user for a time interval, and alert the user when that time has elapsed.

Viewing

Viewing Mode

Go to fullscreen mode and start slide show 36
Switches the viewer to full screen mode and starts a slide show.

Setting zoom values

Change zoom value 36
Set the zoom to a percentage of full size.

Prompt for a zoom value 36
Ask the user for a zoom value and set the view to that value.

Pop-up menus for zooming 37
The following example creates a pop-up menu which might be attached to a Mouse Up or Mouse Down action for a button field.

Watermarks

Adding a watermark from an external file 37
Add a watermark from an external PDF file to a selected range of pages in the current document.

Adding a watermark from text 37
Add specified text to current document as a watermark.

Forms: Code Samples

JavaScript for Forms

Date handling

Acrobat reports and manipulates dates using the date object data structure, but dates are displayed in form fields as a string. Hence some conversion is required, as shown in the following examples.

Date timestamp: first time only

Gets today's date as a one-time timestamp so it won't be updated each time the page is opened.

The following code can be assigned as the Page Open action for the desired page. It checks to see if the value of the field has changed from its default value (usually it is blank). If it is the default value, then it gets the date and displays it as a string.

```
var today = new Date();
var td = this.getField("DateStamp");
// test if new value is the same as the
// default value, which was set to blank ("")
if (td.value == td.defaultValue) {
// if so, display formatted string in field
  this.getField("DateStamp").value =
    util.printd("dd mmmm, yyyy",today);
}

// change field to "Read-only"
this.getField("DateStamp").readonly = true;
```

Date Parsing

Formats the current date using ***AFParseDateEx()****.*

The following sample uses **AFParseDateEx()** to check the validity of the date and to display the resulting date formatted as a string. The user is alerted as to whether the parsing was successful or not. The code should be entered as a custom validation script for the field.

```
if(event.value) {
   var DateOK =
     AFParseDateEx(event.value,"mm/dd/yyyy");
   if(DateOK != null) {
      app.alert("date ok");
      event.value = util.printd("mm/dd/yyyy",
           DateOK);
   }
   else
   {
      event.value = "";
         app.alert("date not ok");
   }
}
```

Displaying a localized date

Uses the util.printd method to display a localized date, including for CJK language documents. [Acrobat 7 or higher]

You can use the **bXFAPicture** parameter (optional, Acrobat 7 or greater; see Acrobat JavaScript Scripting Reference for more information) of the **util.printd** method to display a localized version of dates and times.

Forms: Code Samples

It uses the XFA-Picture Clause format to provide extensive support for localized dates and times for Chinese, Chinese (Taiwan), Japanese, and Korean (CCJK) times and dates. It is supported for both Acro Form and XFA JavaScript.

The following is an example of **util.printd** being used to display the result in the JavaScript console, but it could also be used as an action for a page or document open event, or just to display the date in any field:

```
// execute in console
console.println(
util.printd("EEE, 'the' D 'of' MMMM, YYYY",
   new Date(), true));
```

The result displayed in the console would be:

```
Tue, the 13 of July, 2004
```

The next example shows its use for localized output. The script can be entered as a custom format script of a text field. It gives the current date formatted for a Japanese locale (the last parameter is the **bXFAPicture** boolean set to *true*).

```
event.value = util.printd("date(ja){ggYY/M/
D}", new Date(), true)
```

Getting today's date

Get today's date and display it.

The following code can be attached to the Page Open event for the page on which the date will appear:

```
var today = new Date();
var td = this.getField("DateField");
td.value = util.printd("dd mmmm, yyyy",today);
```

This code will get and display a new date every time the page is opened. For a date timestamp that doesn't change once it has been set, see "Date timestamp: first time only" on page 8.

Calculations

Calculations: Turn calculations on/off

Turn off calculations to optimize performance.

When a large number of fields are being checked or updated, each change of a field value can trigger all fields to be re-calculated, which can affect performance. Setting the **calculate** property of the doc object to *false* tells Acrobat not to execute the calculations until they are turned on again.

To turn off calculations until ready:

```
this.calculate = false;
```

and to allow calculations to resume:

```
this.calculate = true;
```

Delay redrawing of appearance changes

Delays redrawing of all fields until all changes are completed.

```
this.delay = true;  // queue changes
```

Forms: Code Samples

When set to *false*, all fields will be redrawn. See also the **`field.delay`** property in the *Acrobat JavaScript Scripting Reference*.

Field Operations

Add/Remove a new field programmatically

Places a new form field on the specified page.

The following code, which can be executed from the JavaScript console, adds a text field named "myField" to the first page (zero-based) of the document. The field will be filled with yellow, and have a medium weight border.

```
var aRect = [300, 660, 600, 700];
var f = this.addField("myField", "text", 0,
   aRect);
f.fillColor = color.yellow;
f.strokeColor = color.red;
f.lineWidth = 2; // medium line width
```

and to remove the field:

```
this.removeField("myField");
```

NOTE: You can add an action to the newly-created field as shown in "Set an action for multiple fields" on page 18.

Add/Remove fields on multiple pages

Create or remove identical text fields on every page of a document.

The following code creates a field named "myField" on every page of a document. It could be entered through the JavaScript console, or in any other JavaScript code in the document.

```
for (var p = 0;  p < this.numPages; p++) {
   var aRect = [300, 660, 600, 700];
   var f = this.addField("myField", "text",
       p, aRect);
   f.strokeColor = color.black;
   f.fillColor = color.yellow;
}
```

With the above script, all added fields will have the same name, which means they will always all have the same value. One way to give them different names is to append the page number by changing the third line in the example above to:

```
var f = this.addField("myField" + p,
     "text", p, aRect);
```

The result will be a new field on each page, named, for example, myField0, myField1, etc.

NOTE: Also see "Add navigational arrows on every page" on page 31 for another example of adding an action to all pages in a document.

Forms: Code Samples

All of the added fields can be removed using the following code (for the sequentially-named fields):

```
for (var p = 0; p < this.numPages; p++)
   var f = this.removeField("myField" + p);
```

Changing field properties

Change any or all of the properties of a form field.

The following are examples of how to change the properties for a field, for a variety of commonly used field properties. For a complete list of properties and their values, see the *JavaScript for Acrobat API Reference*.

IMPORTANT: *All of the following code samples use the variable* `f` *for the field object. So the following statement must precede any of the code samples:*

```
var f = this.getField("MyField");
```

Alignment of a text field:

```
f.alignment = "center";
```

Background color:

```
f.fillColor = color.blue;
```

Current value:

```
// Set a variable to the field's value
var v = f.value
```

Default value:

```
// Specify default value
f.defaultValue = "Enter your name here.";
```

Font:

```
f.textFont = font.Helv;
```

And for Acrobat 5.0 or higher, the PostScript® font name can be used for any font:

```
f.textFont = "Arial,BoldMT";
```

The first font name example above shows the use of an alias font name that is defined for a limited number of core fonts as defined in the Acrobat font object. The second example: Arial,BoldMT — shows the use of the font family name and style. You can also use the PostScript® font name, which you can see under the Fonts tab of the Document Properties window (press Ctrl-d when in Acrobat). For more details, see the *JavaScript for Acrobat API Reference*, under the properties for the `Field` object.

Font size:

```
f.textSize = 20;
```

Hide or display field:

```
f.display = display.noPrint;
```

Multiline field:

```
f.multiline = true;
```

Forms: Code Samples

Password hiding – using asterisks:

```
f.password = true;
```

Required field:

```
f.required = true;
```

Stroke Color:

```
f.strokeColor = color.yellow;
```

Stroke line width:

```
f.lineWidth = 3;      // 3 = wide line
```

Color: defining a custom color

Define a custom RGB color for use, for example, as a form field background color.

```
color.lightYellow = new Array
    ("RGB",1,0.99,0.8);
```

Check a check box field

Inspect whether a check box field has been checked.

JavaScript can be used to see if a check box field has been checked by the user by inspecting the export value of that field (specified in the "Field Properties" dialog box).

The following example contains code that toggles the state of the check box field. It could be attached as the Mouse Up action for a button field.

```
// Get named reference
var cB1 = this.getField("CheckBox1");

// check if export value is Yes
if (cB1.value == "Yes")
   cB1.value = "No"  // if checked, toggle it
else
   cB1.value = "Yes" // toggle back again
```

Or, the code could be written to use the event object:

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

Formatting a number in a form field

Shows how to format numbers in decimal, hex, and floating formats as well as formatting the value as a string.

You can cut-and-paste the following script into the JavaScript console, select the script, and press Enter. The resulting formatting is shown below under "Output." (The "Math.PI" expression is a JavaScript constant equal to the value of π).

```
var n = Math.PI * 100;

// Decimal format:
console.println(util.printf("Decimal format:
%d", n));

// Hex format:
console.println(util.printf("Hex format: %x",
n));
```

Forms: Code Samples

```
// Floating point format:
console.println(util.printf("Float format:
%.2f", n));

// String format:
console.println(util.printf("String format:
%s", n));
```

Output:

```
Decimal format: 314
Hex format: 13A
Float format: 314.16
String format: 314.159265358979
```

Formatting: removing zeros so field is blank

Removes zeros (0.00) in auto-calculated fields.

Many times when form fields use predefined calculations, the resulting value may be zero. It is usually not desireable to display a value of "$0.00". The solution is to use JavaScript to calculate the field values, and to include the following code in your calculations for each field:

```
if (event.value == 0)
   event.value = "";
```

Hiding/un-hiding an existing field based on conditions

Hide or expose a field based on the state of a check box field.

A field may need to be hidden or exposed based on whether or not a check box is checked. The following code could be used, for example, as a Mouse Up action for the check box field. When the box is checked the "textField1" field is displayed; when the box is unchecked, the field is hidden.

```
// get name of text field to hide or not
var tF1 = this.getField("textField1")

// if box is checked, make text field visible
if (event.target.value == "Yes")  {
    tF1.hidden = false
}
else  {
    tF1.hidden = true
}
```

Highlighting a single field

Highlight a single field for the On Focus event, and turn off the highlighting for the On Blur event.

The following code can be set as the action for a single field (for example, "textField1") that you want to highlight for the On Focus event (user tabs into or clicks on the field), and un-highlight at the On Blur event (when user leaves the field.

Forms: Code Samples

NOTE: If you want highlighting for multiple fields, see "Highlighting multiple fields using a for loop," below.

```
var myField = this.getField("textField1");
if(event.name == "Focus")
    myField.fillColor = color.blue;
else if(event.name == "Blur")
    myField.fillColor = color.transparent;
```

Highlight fields using a function

Create a function to highlight a field, to be called from multiple fields.

The following example uses a Document JavaScript function **ChangeColor()** to handle all highlighting events. Then, you would add a **ChangeColor()** call to each field you would like highlighting applied, for both the On Focus and On Blur events. This function toggles the color of the field, so it can be called for both events.

```
function ChangeColor() {
  var myField = event.target;
  if(event.name == "Focus") {
    myField.fillColor = color.blue;
    myField.borderColor = color.black;
  }
  else if (event.name == "Blur")
    myField.fillColor = color.transparent;
}
```

Highlighting multiple fields using a for loop

Defines a Document Level JavaScript function that will highlight any field that calls the function when the mouse cursor enters that field. Code can be added to all fields by running a script from the JavaScript console.

The above example, "Highlighting a single field," defines the **ChangeColor()** function which will change the color of a field as the user's cursor enters or leaves a field (the On Focus and On Blur events, respectively).

Select *Tools* > *JavaScript* > *Document JavaScripts*, and in the resulting dialogue box enter a name and the **ChangeColor()** function shown in the above section, "Highlight fields using a function".

Then, the following script can be run from the console to assign the **ChangeColor()** function (defined in the sample "Highlight fields using a functionon" on page 14) as the On Focus action for each field in the form. Run the following code from the JavaScript console:

```
// loop through all fields in document
for (var i = 0; i < this.numFields; i++) {
  // set the ChangeColor function as the
  // action for the field
  var f =
     this.getField(this.getNthFieldName(i));
  f.setAction("OnFocus", "ChangeColor();");
  f.setAction("OnBlur", "ChangeColor();");
}
```

The code will loop through all fields in the form and add the highlighting code to each field. For fields to which

Forms: Code Samples

you don't want the actions to apply (such as for a button field), double-click them with the object selection tool to get the Field Properties window; click the Action tab; and then delete the attached script.

Locating fields and listing field names

List all form fields including their name, type, and page number.

Sometimes a complicated form may have many fields that are hard to find, or some fields may obscure other fields, or the field names may be too long to display clearly. The following script will output the names of all fields, the type, and the page number, to the console:

```
console.clear();
console.show();
console.println("Page, Name, Type");
for (var i=0; i<this.numFields; i++) {
  fieldName = this.getNthFieldName(i);
  f = this.getField(fieldName);
  console.println(f.page+ ", " + f.name + ",
    " + f.type)
}
```

Another variation is the following script, which will place the name of each field below the field, and allow you to print the resulting form. The added field label is added in a new field created by the script, so it is intended only as a development tool to help identify fields.

This following script labels all fields in a form document; the print statement arguments should be adjusted for the pages yoou want to be printed.

```
// Define an array for field names
nameArray  = new Array (this.numFields);

// Put all field names into an array
for (var i = 0; i<this.numFields; i++)
   nameArray[i] = this.getNthFieldName(i);
// loop through name array
for (i=0; i<nameArray.length; i++) {
    // get object for existing field
    var f = this.getField(nameArray[i]);
    // make a copy of rectangle coordinates
    var myRect = f.rect;
    var name = f.name;
    // Get length of field name
    var flength = f.name.length;
    // myRect[1] = f.rect[3];
    myRect[3] = f.rect[3] - 10;
    myRect[2] = f.rect[0] + (flength * 3);
    // make a new field just under old one
    f = this.addField("abyz" + i, "text",
        this.pageNum, myRect);
    f.value = name;
}
this.print(true, this.pageNum, this.pageNum);
```

In the above example, the field bounding box coordinates are returned for rotated user space, and are in this order: ulx (upper-left x), uly (upper-left y), lrx (lower-right x), and lry (lower-right y), which is different from bounding boxes for annotation rectangles.

Forms: Code Samples

Printing: Set field to print or not print

Code to control the print property of a field so the field will either print or not print.

```
var f = this.getField("myField");
f.print = false;
```

NOTE: Most printing examples are in the JavaScript for Documents section under "Printing" on page 33.

Print selected form pages

Print a selected range of form pages (for example, to exclude the instruction pages in a form document).

Some forms consist of one or more pages of instructions which don't need to be printed. The following example might be for a form where the first page contains only instructions, and the second page contains the form and a print button. Hence the page range parameters both point to the second page:

```
this.print(false,this.pageNum,this.pageNum);
```

Rollover help

Two methods for providing rollover help include: 1) descriptive help text can be displayed; or 2) a default text or value can be displayed. In both cases, the rollover help text should be displayed for the Mouse Enter event, and it can be removed for the Mouse Exit event.

The first example below shows how to display help text, but it could also be used to display the default value by using the **defaultValue** property of the field.

NOTE: Basic "rollover" highlighting of a form field is shown in "Highlighting a single field" on page 13.

Display help text

Display help text for a field for the Mouse Enter event.

For a Mouse Enter event:

```
var f = this.getField("Text1");
f.value = "Enter full e-mail address"
```

And for the Mouse Exit event:

```
// Remove help text only if present
var f = this.getField("Text1")
f.value = "";
```

The problem with the above code is that it works fine when the mouse enters and leaves the field, but when the user clicks in the field to enter data, the help text is not removed. Also, if the user did remove the help text manually and then entered data, the next time the mouse passed over the field, the user data would be removed.

The following improved script checks field contents before changes are made. It adds an On Focus event (when the user tabs to, or clicks in, the field) to distinguish when the user begins to enter data.

Forms: Code Samples

For the Mouse Enter event:

```
//Display help text only if field is blank
var f = this.getField("Text1")
if (f.value == "")
    f.value = "Enter full e-mail address";
```

For the Mouse Exit event:

```
// Remove field text but only if it is the
// text from the Mouse Enter event, otherwise
//   user input could be deleted
var f = this.getField("Text1")
if (f.value == "Enter full e-mail address")
       f.value = "";
```

For the On Focus event:

```
// Reset field to blank only if
//     help text is present
var f = this.getField("Text1")
if (f.value == "Enter full e-mail address")
    f.value = ""
```

Search fields

Search for all fields with a specified prefix

Search/enumerate through all fields that begin with specified string.

The following code loops through all fields in the form and displays field names that begin with (for example) "txt".

```
for (var i = 0; i < this.numFields; i++) {
 if (this.getNthFieldName(i).substring(0,3)
      == "txt")
  app.alert(this.getNthFieldName(i));

  // or to list in console window:
  // console.println(this.getNthFieldName(i))
}
```

The above code uses the JavaScript substring method. It specifies the beginning index of the string (0) and the length of the string it is trying to match (3) . The code can be entered in the JavaScript console as means of locating and potentially altering all fields with a given name (using additional code), or it could be an action associated with a button, or be used in any other script associated with the form.

Set an action for a single field

Attaches an action to a single field.

The following code sets the function **ChangeColor()** (defined on page 13) as the action for the field named **myField**, for both the On Focus and On Blur events.

```
var f = this.getField("myField");
f.setAction("OnFocus", "ChangeColor();");
f.setAction("OnBlur", "ChangeColor();");
```

Forms: Code Samples

Set an action for multiple fields

Execute code from the JavaScript console to attach an action to multiple fields.

A script to set an action for multiple fields is shown in "Highlighting multiple fields using a for loop" on page 14. The function call and the actions can be modified to suit whatever is needed. The script adds the specified actions to all fields in the document. Depending on the form, it might be easier to add the actions to all fields using a for loop, and then deleting by hand the actions for the fields that do not need those actions.

Form checking and resetting

Final field checks

Check fields before submitting

Check that all required fields were completed before the form is submitted.

The following code loops through all fields, allowing you to verify, check, or validate each field before the form is submitted. The code could be used, for example, as part of the action for a "Submit" button.

```
for (var i=0; i<this.numFields; i++) {
   var strFieldName =
      this.getNthFieldName(i);
   var field = this.getField(strFieldName);
   if (field.value == "") {
      // Add your tests here
      // Build a list of names and pages
      // to alert user
   }
}
```

The following script checks all fields for a prefix of "required." (for example, **required.username**). It then checks if an entry has been made in that field, and alerts the user if no data has been entered.

```
for (var i=0; i<this.numFields; i++)  {
 if (this.getNthFieldName(i).substring(0,8)
     == "required"){
   var f1 =
   this.getField(this.getNthFieldName(i));
   if (f1.value == "") {
     app.alert(this.getNthFieldName(i) + "
     is a required field; please enter
     data");
   f1.setFocus();}
  }
}
```

Another option would be to put the names of all required fields in an array; in this example they are hard-coded in the script. The user clicks a button and is alerted as to which fields are blank. They then enter the data, and click the button again until no error messages are displayed.

```
// Specify a list of required field names:
var requiredField = ["Text10", "Text11",
"Text12"];
// Loop through the array of field names:
for (var i=0; i<requiredField.length; i++){
```

Forms: Code Samples

```
   var f = this.getField(requiredField[i]);
   if (f.value == ""){
     app.alert("The " + requiredField[i] + "
field is required; please click OK on this
message; enter data; and then click this
button again until there are no error
messages")
   break;
   }
   else if(i == requiredField.length - 1) {
       app.alert("Thank you, all required
fields have been completed");
   }
}
```

Resetting a form

Resets all fields, or specified fields, to their original default value.

The doc object method **resetForm()**, with no arguments, will reset all fields in the form to their default value:

```
this.resetForm();
```

It could be set as the action for a "Reset Form" button.

If the default is not set to a specfied value, (which means the value is Null), the above method has the effect of clearing the fields so they are left blank.

If you want to reset a subset of all fields, you need to specify an array that contains the names of all fields to be reset. For example, the following code builds an array of two field names and passes the array name as the parameter to the **resetForm()** method:

```
var f = new Array(2);
f[0] = "textField";
f[1] = "totalField";
this.resetForm(f);
```

You could use the following alternate syntax for setting up the array:

```
var f = new Array("textField", "totalField");
```

Or you could use the Acrobat naming convention; for example, if you want to reset all name fields, and they are named **name.first**, **name.last**, etc., then the following will reset all fields that begin with "**name**":

```
var a = new Array(1);
a[0] = "name"
this.resetForm(a);
```

If you want to set a field to be blank (Null) even though its default value is something else, you can set it to Null by using the null string, which consists of two double-quote characters:

```
var f = this.getField("myField");
f.value = "";
```

Forms: Code Samples

Validation

Validate that number is in the correct range

Verify that entered value is in one of two ranges.

Standard Acrobat validation can be used for simple validations such as whether a number is in a given range, but a custom script can be used for other types of validations. The following script would be defined as a custom validation script to be run after a value is entered.

```
if ((event.value >= 40 && event.value <=200)
  || (event.value
  >= 1000 && event.value <= 2000))
     event.rc = true ;
else {
  app.alert("value out of range");
  event.rc = false ;
}
```

Verify that required entry was made

Check required fields for a valid entry.

A field's properties can be set through the Acrobat UI to indicate that it is a required field, but that field property is not checked for a data entry value until the form is submitted (this example is similar to the example "Check fields before submitting" on page 18).

There can be situations where you might want to check if specific required entries have been made, before the user can proceed to other pages. Or you might want to give them a choice in a dialog to leave some fields blank and return later.

The following code will check a required field to verify that required information has been entered (it checks to see if the field's value is not Null).

```
f1 = this.getField("employeeNumber")
if (f1.value == "") {
   app.alert('Please enter your employee
   number');
}
```

Another way to check required fields before they are submitted would be by using a field name prefix of "required". You could then use the following code which uses the **substring()** method to identify fields whose names begin with "required":

```
// Enumerate through all fields
for (var i = 0; i < this.numFields; i++)
{
  if (this.getNthFieldName(i).substring(0,8)
     == "required"){
   var f1 =
     this.getField(this.getNthFieldName(i));
   if (f1.value == "") {
      app.alert(this.getNthFieldName(i) + "
      is a required field!");
      f1.setFocus();
   }
  }
}
```

When a required field has a null value, the Focus is set to that field (**f1.setFocus()**) so the user can enter

Forms: Code Samples

required information. You might want to also change the border style or fill color of that field to draw attention to it, and then reset those properties when data entry has been completed.

Another method would be to loop through all fields, as above, and check for the *required* property for each field.

```
for (var i = 0; i < this.numFields; i++) {
   var f = this.getNthFieldName(i);
   var f1 = getField(f);
   if (f1.required == true){
      if (f1.value == "") {
         app.alert(this.getNthFieldName(i) +
           "is a required field!")
      }
   }
}
```

Convert user input to uppercase characters

Converts lowercase characters typed by the user into uppercase characters.

The following code could be used as a custom keystroke script which will change all lowercase characters to uppercase (non-lowercase characters typed by the user will not be affected).:

```
// Custom keystroke for text field
if (!event.willCommit)
   event.change =
     event.change.toUpperCase();
```

Restricting keystrokes to uppercase characters

Restrict keystrokes to Y (for Yes) or N (for No) .

The following custom keystroke script will only allow the user to type *Y* or *N* (or *y* or *n*).

```
if (!event.willCommit) {
  if (event.change.match(/[YN]/))  {
    event.rc = true;
  }
  else {
    app.beep();
    event.rc = false;
  }
}
```

JavaScript for Documents

This section contains sample JavaScripts formany PDF document operations.

Annotations

Add/remove annotations

Create an annotation

Illustrates how to create an annotation of type "Text".

```
try {
//  Create a document
var myDoc = app.newDoc();

//  Create an annot
var  myAnnot = myDoc.addAnnot
({
   page: 0,
   type: "Text",
   point: [300,400],
   name: "myAnnot",
});
} catch (e) {
app.alert(e);
}
```

Remove an annotation

Removes an annotation.

The following code will remove all annotations of the type "Text", on the first page of the document (page 0 in a zero-based numbering system):

```
var annots = this.getAnnots({nPage:0});
for (var i = 0; i < annots.length; i++)
   if (annots[i].type == "Text")
         annots[i].destroy();
```

Application Version

Version checking

Checks the version and type of the Acrobat application, and lists all plug-ins.

This script could be run from the console or attached to a button, or included with other JavaScript code in a document.

```
// Query App version and output results

var echo = this.getField("Text_output");
echo.value = "Viewer type is " +
   app.viewerType;
echo.value += "\n" + "Viewer version is " +
   app.viewerVersion;

// Now get array of plug-in names
var aPlugins = app.plugIns;
//Get number of plug-ins
var nPlugins = aPlugins.length;
//Enumerate names of all plug-ins
```

```
for (var i = 0; i < nPlugins; i++)
     console.println("Plug-in \#"+i+" is " +
aPlugins[i].name);
```

NOTE: This code works for Acrobat 5.0 or newer; for earlier versions, see *JavaScript for Acrobat API Reference*.

Bookmarks

To create a new bookmark, invoke the **`Bookmark`** object **`createChild`** method, to which you may submit the following parameters: **`cName`** (the name to appear in the navigational panel), **`cExpr`** (an optional JavaScript to be executed when the bookmark is clicked), and **`nIndex`** (an optional zero-based index into the **`children`** array).

NOTE: Bookmarks cannot be created or removed in Acrobat Reader.

Create a new bookmark

Creates a new child bookmark at the specified location.

Begin by accessing the bookmarkRoot, which is a property of the current document representing the top node in the bookmark tree, and then creating a child bookmark that displays a greeting when clicked:

```
var myRoot = this.bookmarkRoot;
myRoot.createChild("myBookmark", "app.alert('
   Hello!');");
```

Add a child bookmark

Inserts a bookmark as a child of the bookmark in the previous sample..

The bookmark generated in this example will be named "myBookmarkChild," and the associated action is to display the message "Hello child."

```
var current = myRoot.children[0];
current.createChild("myBookmarkChild",
  "app.alert('Hello child!');");
```

Remove a bookmark

Removes a bookmark and all children bookmarks from the bookmark tree.

```
bookmarkRoot.remove();
```

You can remove a specific bookmark, or the above command will remove all bookmarks in the tree.

Document Operations

Create a new blank PDF document

Creates a new document in the Acrobat viewer.

Because of security restrictions, this code can only be executed during batch, console, or menu events. Running it in the console is an easy way to generate a blank PDF document for testing purposes.

```
app.newDoc();
```

Documents: Code Samples

Metadata: Get document metadata

Allows you to retrieve and display document metadata.

Document metadata is stored in a PDF file as XML data. This sample gets the path and filename of the document, and also the XML text for the document metadata. The following code can be set as an action attached to a button:

```
// Display metadata in the
// Text_meta.filename, Text_meta.fpath,
// and Text_meta.data fields

this.getField("Text_meta.fpath").value =
this.path;
this.getField("Text_meta.filename").value=
this.documentFileName;
this.getField("Text_meta.data").value=this.m
etadata;
```

and the following script could be set as the action for a button labelled "Clear":

```
//Clear the fields:
var fields = this.getField("Text_meta");
this.resetForm(fields);
```

Open an existing document

Opens the document whose filename is specified as the string argument.

```
app.openDoc("MyFile.pdf");
```

Prompt user for a document to open

Executes the File Open menu item in Acrobat, which prompts the user for a file to open.

```
app.execMenuItem("Open");
```

IMPORTANT: *As of Acrobat 8, this method can only be executed in Privileged mode, which means during batch or console events. See Note under* *"Execute menu item" on page 27*.

Links

Add navigation links to a page

This sample adds navigational links at the bottom of each page in the document.

This sample creates two navigtaion links for each page. For this example, the links direct the user to either page 0 or 2, such as might be used to go to the cover page or the Table of Contents page of a document.

The links could also be placed over custom navigation images in the document. This script can also be run from the JavaScript console.

```
// Define two link rectangles
var linkRect1 = [70, 70, 225, 50];
var linkRect2 = [260, 70, 340, 50];

// For pages 0 to n (or range of your choice)
for ( var i=0; i < this.numPages;  i++ ) {
    // Add link for current page, "i"
```

```
    var link1 = this.addLink(i, linkRect1);
    link1.borderWidth = 3;
    link1.borderColor = color.blue;
    var link2 = this.addLink(i, linkRect2);
    link2.borderWidth = 3;
    link2.borderColor = color.blue;

    // For this example, link to pages 0 & 2;
    // (For example, the Cover page
    // and Table of Contents pages
    link1.setAction("this.pageNum = 0");
    link2.setAction("this.pageNum = 2");
}
```

Convert specified word to a weblink

Search for a word and create a weblink.

```
var p = this.pageNum;
var numWords = this.getPageNumWords(p);
for (var i=0; i<numWords; i++)
{
  var ckWord=this.getPageNthWord(p, i, true);
  if ( ckWord == "Acrobat") {
   var q = this.getPageNthWordQuads(p, i);
   // convert quads in Default User Space to
   //  Rotated User Space used by Links.
   m = (new Matrix2D).fromRotated(this,p);
   mInv = m.invert()
   r = mInv.transform(q);
   r = r.toString();
   r = r.split(",");
   l = addLink(p, [r[4], r[5], r[2], r[3]]);
   l.borderColor = color.red;
   l.borderWidth = 1;
   l.setAction("this.getURL('http://
   www.adobe.com/products/acrobat', false);");
    }
}
```

Convert URL text to live links

Convert all content that appears to be a URL into a web link.

The following sample will convert all URLs in the document into a weblink and display the number of links created. The code can be entered and executed in the console.

```
var numWeblinks = this.addWeblinks();
console.println("There were " + numWeblinks +
" instances of text that looked like a web
address,");
```

Remove all links on a page

Removes all links on the current page.

The following code can be entered in the JavaScript console, either for the current page, or it could be used in a loop to remove links for multiple pages. It first gets the coordinates of the current page, and then tells the **doc** object to remove all links within that rectangle.

```
var b = this.getPageBox("Crop",
   this.pageNum);
this.removeLinks(this.pageNum, b);
```

Set an action for a link

Sets the specified JavaScript action for the Mouse Up trigger for the link object.

`Link.setAction` specifies the action for the Mouse Up trigger for the **`link`** object, which is obtained from **`doc.addLink`** or **`doc.getLinks`**.

For this example, the action (**`this.pageNum++`**) causes the viewing application to go to the next page:

```
link.setAction("this.pageNum++");
```

Menu operations

NOTE: This section contains tasks related to application menu items. For an example of how to add your own pop-up menus, see: "Pop-up menus for zooming" on page 37.

Add a new menu item

Adds a new menu item and an associated action.

This first example adds a menu item to the top of the file submenu and it puts up an alert dialog displaying the active document title. This menu item is only enabled if a document has been opened.

NOTE: Because of security constraints, code to add or hide menu items can only be used in the JavaScript console or in folder level JavaScript files which execute when the application is opened.

The menu changes will then apply to all PDF files that are opened by the application.

Example 1

Adds a "Hello" entry to the File menu; when selected, the menu item title is displayed on the user's screen.

```
app.addMenuItem({
   cName: "Hello",
   cParent: "File",
   cExec: "app.alert(event.target.info.title,
      3);",
   cEnable: "event.rc = (event.target !=
      null);",
   nPos: 0
});
```

Example 2 [Acrobat version 6.0 or newer]

Places two menu items in the "File" menu, one before the "Close" item, and the other after the "Close" item. The names "_myProc1" and "_myProc2" are placeholders for the JavaScript procedures you would write to handle the intended task for the menu items you are adding.

```
// insert after the "Close" item
//    (the default behavior)
app.addMenuItem( {
   cName: "myItem1",
   cUser: "My Item 1",
   cParent:"File",
   cExec: "_myProc1()",
   nPos: "Close"});
// insert before the "Close" item,
//   set bPrepend to true.
```

```
app.addMenuItem( {
   cName: "myItem2",
   cUser: "My Item 2",
   cParent: "File", cExec: "_myProc2()",
   nPos: "Close", bPrepend: true
});
```

Execute menu item

Executes the menu item specified by the argument.

For example:

```
app.execMenuItem("Open");
```

will execute the menu item File > Open, which will prompt the user for the file to be opened.

NOTE: As of Acrobat 8, the **execMenuItem** method is restricted to a short list of safe menu entries. It will silently fail if a named menu item is executed that is not on the safe menu list. The JavaScript developer is notified that menu items may be removed in future releases, or their behavior may change. The **app.execMenuIte**m method may be executed, without restriction, in a privileged context, such as in the console or in a batch sequence. For folder JavaScript, **app.execMenuItem** can be executed, again, without restriction, through a trusted function with raised privilege. See *JavaScript for Acrobat API Reference* for more information.

Hide a menu item

Hides a menu item so user won't have access to that item.

NOTE: Because of security constraints, code to add or hide menu items can only be used in the JavaScript console or in folder level JavaScript files which execute when the application is opened. The menu changes then apply to all PDF files that are opened by the PDF viewer.

```
app.hideMenuItem("Edit");
```

NOTE: See *List menu items* below to get exact menu names.

List menu items

Display a list of menu items for the Acrobat application; for use with methods that reference menu items.

In order to hide or execute a menu item, you need to know the exact name of the menu item.

Example 1

List all menu item names to the JavaScript console.

```
var menuItems = app.listMenuItems()
for (var i in menuItems)
   console.println(menuItems[i] + "\n");
```

Example 2

List all menu items to console, but more nicely formatted.

```
function FancyMenuList(m, nLevel) {
```

```
var s = "";
for (var i = 0; i < nLevel; i++) s += " ";
console.println(s + "+-" + m.cName);
if (m.oChildren != null)
for (var i = 0; i < m.oChildren.length; i++)
   FancyMenuList(m.oChildren[i], nLevel + 1);
}
var m = app.listMenuItems();
for (var i=0; i < m.length; i++)
   FancyMenuList(m[i], 0);
```

Messages/Dialogs

Beep the user

Causes the system to play a sound to alert the user.

```
app.beep();
```

Prompt the user for a response

Prompt and get a response from the user.

The user's response is returned as the value of the variable ***reply***. That string can then be tested using an **`if`** statement.

```
var dialogTitle = "Please confirm";
var defaultAnswer = "No.";
var reply = app.response("Did you really mean
to type that?", dialogTitle, defaultAnswer);
```

Simple alert

Displays a message to the user.

The following code posts a simple alert message:

```
app.alert("Error! Incorrect entry");
```

The **`app.alert`** object also allows you to customize by: 1) using your own title instead of the standard Acrobat title; 2) displaying your choice of icons; 3) choosing the type of buttons shown to the user. The following example uses a few of those options (the optional **`cTitle`** parameter is only supported for Acrobat 6 or newer):

```
app.alert({
   cMsg: "Error! Incorrect entry!\n\tYou may
want to try again",
   cTitle: "Acme Corporation"
});
```

In this example, the default title "Adobe Acrobat" was replaced by "Acme Corporation", and a second line of the message is begun on a new line (**`\n`**), and a tab is set (**`\t`**) to indent it.

For more information, see the *JavaScript for Acrobat API Reference*.

Documents: Code Samples

Navigation

Add a pop-up menu for navigation

Creates a pop-up menu that can be triggered by any of a variety of events.

This script could be attached as the action for a "Navigation Menu" button (for example), which could be placed at strategic locations in a document.

```
// pops up a menu for going to other pages
var cChoice = app.popUpMenuEx
(
   {cName: "Cover page", bEnabled:false},
   {cName: "-"},
   {cName: "Page 2", cReturn: "2"},
   {cName: "More",
      oSubMenu: [
         {cName: "Page 3", cReturn: "3"},
         {cName: "More",
            oSubMenu: [
            {cName:"Page 4", cReturn: "4"},
            {cName:"Page 5", cReturn: "5"}
             ]
     }
      ]
   },
   {cName: "Last page" },
   {cName: "-"},
   {cName: "Beep On", bMarked:global.bFlag}
)
//app.alert("You chose the \"" + cChoice +
"\" menu item");
if(!isNaN(parseInt(cChoice)))
    this.pageNum =parseInt(cChoice) - 1;
else if(cChoice=="Last page")
    this.pageNum = this.numPages - 1;
else if(cChoice=="Beep On")
     global.bFlag = ! (global.bFlag);
if(global.bFlag) app.beep(0);
```

Adding page numbers to a document

Adds page numbers to a document in the specified location, using the specified font style.

```
var r =[612 -72, 792 -72, 612, 792];
// 1 inch from top right
for( var i = 0; i < numPages; i++) {
  // Get page number string
  var fn = String("page"+i+1);
  var f = this.addField(fn,"text",i,r);
  f.textSize = 14;
  f.textColor = color.blue;
  f.fillColor = color.transparent;
  f.textfont = font.HelvB;
  f.borderStyle = border.s;
  f.strokeColor = color.transparent;
  f.value = String(i+ 1);
}
```

Get the number of the current page

Gets the current page number.

```
var currentPage = this.pageNum;
```

Documents: Code Samples

Go to a specified page number

Sample code to go to a specific page number.

```
// Go to page 20 (zero-based page numbers)
 this.pageNum = 20;
```

Go to first page

Sets the page being viewed to the first page (zero-based numbering) in the document.

```
this.pageNum = 0;
```

Go to next page

Increments the page being viewed to the next page.

```
this.pageNum++;
```

Go to previous page

Navigates to the previous page of the document.

```
 this.pageNum--;
```

Go to last page

Sets the page being viewed to the last page in the document.

```
this.pageNum = this.numPages - 1;
```

NOTE: Because the JavaScript page numbering is zero based, if the number of pages in the document is, for example, 45, then the zero-based number of that page will be 44.

Go to another page: create a pop-up menu

Creates a pop-up menu which gives the user a choice of pages to which to go. The script can be attached to a button.

```
// pops up a menu for going to other pages
var cChoice = app.popUpMenuEx
(
   {cName: "Cover page", bEnabled:false},
   {cName: "-"},
   {cName: "Page 2", cReturn: "2"},
   {cName: "More",
      oSubMenu: [
         {cName: "Page 3", cReturn: "3"},
         {cName: "More",
            oSubMenu: [
            {cName:"Page 4", cReturn: "4"},
            {cName:"Page 5", cReturn: "5"}
             ]
     }
      ]
   },
   {cName: "Last page" },
   {cName: "-"},
   {cName: "Beep On", bMarked:global.bFlag}
)

//app.alert("You chose the \"" + cChoice +
"\" menu item");
if(!isNaN(parseInt(cChoice)))
    this.pageNum =parseInt(cChoice) - 1;
```

```
else if(cChoice=="Last page")
    this.pageNum = this.numPages - 1;
else if(cChoice=="Beep On")
     global.bFlag = ! (global.bFlag);
if(global.bFlag) app.beep(0);
```

Add navigational arrows on every page

Add navigational symbols to every page of a document for each document in a selected set.

The following sample code can be used in a batch sequence or as folder level JavaScript, or you can execute it from the JavaScript console.

```
var inch = 72;
for (var p = 0; p < this.numPages; p++) {
  // position rectangle (.5 inch, .5 inch)
  var aRect = this.getPageBox( {nPage: p} );
  aRect[0] += .5*inch; // from upper left
  aRect[2] = aRect[0]+.5*inch; // .5" wide
  aRect[1] -= .5*inch;
  aRect[3] = aRect[1] - 24; // 24 points high
  // now construct button field with a right
  //   arrow from ZapfDingbats
  var f = this.addField("NextPage", "button",
     p, aRect )
  f.setAction("MouseUp", "this.pageNum++");
  f.delay = true;
  f.borderStyle = border.s;
  f.highlight = "push";
  f.textSize = 0; // auto sized
  f.textColor = color.blue;
  f.fillColor = color.ltGray;
  f.textFont = font.ZapfD
  f.buttonSetCaption("\341") // a right arrow
  f.delay = false;
}
```

NOTE: For another method for adding navigational links to document pages, see "Add navigation links to a page" on page 24.

OCG (Optional Content Group)

OCG objects (images or text) can be selectively viewed or hidden content in a PDF document using JavaScript.

Beep when OCG state changes

Execute a JavaScript after every state change for a give OCG.

```
/* Beep when the given OCG is changed */
function BeepOnChange(ocg) {
   ocg.setAction("app.beep()");
}
```

Show all OCG objects on the currrent page

Make visible all OCG objects in a specified name family.

The following code gets a list of all OCG objects on the current page, loads them into an array, and then sets their state to *true*, which makes them visible. It could be set as the action for a button to display a given set of OCG objects.

```
var ocgArray = this.getOCGs(this.pageNum);
for (var i=0; i < ocgArray.length; i++)
    ocgArray[i].state = true;
```

Hide all OCG objects on current page

Hide all OCG objects in a specified name family.

The following code gets a list of all OCG objects on the current page, loads them into an array, and then sets their state to *false*, which hides them. It could be set as the action for a button to hide a given set of OCG objects.

```
var ocgArray = this.getOCGs(this.pageNum);
for (var i=0; i < ocgArray.length; i++) {
    ocgArray[i].state = false;
}
```

Lock OCG objects

Locks the OCG so its state cannot be toggled through the UI.

This sample locks all OCG objects on the current page:

```
var ocgArray = this.getOCGs(this.pageNum);
for (var i=0; i<ocgArray.length; i++)
    ocgArray[i].locked = true;
```

Toggle the state of OCG objects

Specifies whether an OCG object is hidden or not.

This code could be associated with a button that toggles whether a watermark graphic or text is hidden or not.

```
// Get list of OCG objects for current page
var ocgArray = this.getOCGs(this.pageNum);

// If name begins with "Watermark",
//     toggle its display state
for (var i=0; i < ocgArray.length; i++) {
    if (ocgArray[i].name == "Watermark")
      ocgArray[i].state =!ocgArray[i].state;
}
```

Page Operations

Insert new page

Inserts a new page based on the selected template and then jumps to the new page.

The following code selects the first template in the document, and adds a new page at the end.

```
// get the template
var t = this.getNthTemplate(0);

// spawn a new page after last page
this.spawnPageFromTemplate(t);

// go to new page:
this.pageNum = this.numPages-1;
```

Insert/Delete/Replace/Extract Pages

Executes a menu item to insert, delete, replace, or extract pages. The user is prompted to specify the appropriate pages.

For example, to delete pages:

```
app.execMenuItem("DeletePages");
```

IMPORTANT: *See the note about security restrictions under* *"Execute menu item" on page 27*.

The other page operation menu options include: **InsertPages**, **ExtractPages**, and **ReplacePages**.

Printing

Get a list of printers as a combo box

Shows how to get a list of printers and display result as a combo box.

The following code could be set as the action for a button to get a list of available printers and populate the combo box with that list:

```
this.getField("ComboBox_printers").setItems(
app.printerNames);
```

Print page using printer selected in combo box

Code to print the current page using the printer selected by user from the combo box presented in previous example.

This example uses the printer selected from the combo box created in the example above. The current page of the document is printed to that printer.

```
// get printParams object of default printer
var pp = this.getPrintParams();

// print this page
pp.firstPage = this.pageNum;
pp.lastPage = this.pageNum;
// set some properties
pp.interactive =
  pp.constants.interactionLevel.automatic;
pp.colorOverride =
  pp.constants.colorOverrides.auto;

// use the printer selected
var f = this.getField("ComboBox_printers");
var aPrinter = f.valueAsString;
if(aPrinter!=null) pp.printerName = aPrinter;

// print
this.print(pp);
```

Print only the current page

Sample code to print only the current page.

The following code causes the user to be prompted with the printing dialog box:

```
this.print(true,this.pageNum,this.pageNum);
```

The first parameter, true, specifies that the print UI wil be presented to the user to obtain printing information and confirm the action.

Print current page (no dialog)

Print the current page without the usual dialog.

```
// first param = false = no dialog
this.print(false,this.pageNum,this.pageNum);
```

Documents: Code Samples

Searching

A simple find operation

A alternative to the standard Acrobat search function; it can be used to create customized searches.

The following sample shows a folder level JavaScript to add a new menu item "Find" to the Tools menu.

```
app.addMenuItem({
     cName: "Find",
     cParent: "Tools",
     cExec: "easyFind()"
});

function easyFind() {
    var wordToFind = app.response({
       cQuestion:"Enter the word you wish to
search for",
       cTitle:"Quick Search"
     });
     if(wordToFind != null)
       search.query(wordToFind, "ActiveDoc")
};
```

Also, it is easy to specify search parameters to improve the quality of the search. The following code from the **easyFind()** function shown above illustrates more possibilities:

```
if(wordToFind != null) {
   // what to search:
   search.bookmarks = true;        // 6.0
   search.attachments = true;      // 7.0
   search.docXMP = true;           // 6.0

   // how to match:
   search.matchWholeWord = true;   // 6.0
   search.matchCase = true;        // 6.0
   search.matchWholeWord = true;   // 6.0
   search.ignoreAccents = true;    // 7.0

   // do search:
   search.query(wordToFind, "ActiveDoc")
}
```

The above example specifies the scope of what will be searched, and the rules for how to match.

NOTE: The comment next to each search property indicates the version number of Acrobat that supports that feature. See the *JavaScript for Acrobat API Reference* for more information.

Spell checking

Creating a menu item to spellcheck a document

Spellchecks a document and uses annotations to mark the misspelled words. [Version 6.0 or higher]

The spell checking can be done by creating a folder level JavaScript that creates a custom menu item to do the spellchecking. This sample is similar to the spell checker

in Acrobat, except that with JavaScript the search parameters can be customized.

```
app.addMenuItem({
     cName: "SpellCheck",
     cParent: "Tools",
     cExec: "SpellCheck();"
});

function SpellCheck(){
  var ckWord, numWords;
  for (var i = 0; i < this.numPages; i++) {
    numWords = this.getPageNumWords(i);
    for (var j = 0; j < numWords; j++) {
      ckWord =
     spell.checkWord(this.getPageNthWord(i, j))
      if ( ckWord != null ) {
        this.addAnnot({
        page: i,
        strokeColor: color.red,
        type: "Squiggly",
     quads: this.getPageNthWordQuads(i, j),
     author: "JavaScript SpellChecker",
     contents: ckWord.toString()
     });
   }
}
}
app.alert ("Acrobat is done spell checking the
current document",3)
}
```

Timing

Set countdown timer

Prompt the user for a time interval, and alert the user when that time has elapsed.

Setting a timer can be used for many purposes, but as a simple example, the following code prompts the user to enter the number of seconds, and when the time has elapsed, a message is displayed.

```
function TimedOut() {
    app.alert ("The time has elapsed");
    // cancel the timer:
    app.clearInterval(timeout);
}
// Prompt for number of seconds
var secondCount = app.response("Enter number
of seconds", "Number of Seconds", "", "")
// Start counter
if (secondCount >= 0)
// Timer set for thousandths of a second
var timeout = app.setInterval("TimedOut()",
    1000*secondCount);
```

Documents: Code Samples

Viewing

Viewing Mode

Go to fullscreen mode and start slide show

Switches the viewer to full screen mode and starts a slide show.

The following code could be attached as the action for a "Click Me" or "Start Slideshow" button action.

```
this.pageNum = 0;
// full screen
app.fs.cursor = cursor.visible;
app.fs.defaultTransition = "WipeDown";  //
BoxIn, BoxOut
app.fs.timeDelay = 2; // delay 2 seconds
app.fs.useTimer = true; // activate
automatic page turning
app.fs.usePageTiming = true; // allow page
override
app.fs.clickAdvances = true;
app.fs.escapeExits = true;
app.fs.loop = true;
app.fs.backgroundColor = color.ltGray;

// go
app.fs.isFullScreen = true;
```

Setting zoom values

Change zoom value

Set the zoom to a percentage of full size.

```
this.zoom = 40;    // Set to 40% zoom
```

Prompt for a zoom value

Ask the user for a zoom value and set the view to that value.

It may be useful to have a button which, when clicked, prompts the user to change the zoom value. For example, some PDF files may specify that the tool bars be hidden when the document is viewed, so the zoom tools are not readily available. Also, this might be useful for the situation where some OCG (Optional Content Group) objects are only visible at certain zoom values.

```
// Prompt for a value:
var zoomValue = app.response({
   cQuestion:"Enter zoom value:",
   cTitle:"Zoom Percentage",
})

// Check if Null; if so, don't change value
if(zoomValue == "")
    app.alert("No value entered");
else
    this.zoom = zoomValue;
```

Pop-up menus for zooming

The following example creates a pop-up menu which might be attached to a Mouse Up or Mouse Down action for a button field.

```
// pops up a menu for various zoom choices
var cChoice = app.popUpMenuEx(
    {cName: "Zoom options:"},
     {cName: "-"},
     {cName: "Zoom In", cReturn: "zIn"},
    {cName: "View @ 100%", cReturn: "z100"},
     {cName: "Zoom Out", cReturn: "zOut" },
     {cName: "-"}
)

//
if(cChoice == "zIn")
    this.zoom = 200;
else if (cChoice == "z100")
    this.zoom = 100;
else if(cChoice=="zOut")
    this.zoom = 40;
```

Watermarks

Following are two methods for adding watermarks to a PDF document—using either the watermark from an external file or from text specified in the JavaScript.

Adding a watermark from an external file

Add a watermark from an external PDF file to a selected range of pages in the current document.

The following code(for Acrobat 7 or higher) adds a watermark image, from an external file, to the first two pages of the document.

```
this.addWatermarkFromFile({
  cDIPath: "/C/watermark.pdf",
  nSourcePage: 0, // select source doc page 0
  nEnd: 1,
  nHorizAlign: 0,
  nVertAlign: 0,
  nHorizValue: 288,
  nVertValue: -144,
});
```

NOTE: Because of security restrictions, the above script can only be executed in the JavaScript Console or in batch mode.

Adding a watermark from text

Add specified text to current document as a watermark.

The following code places the word “Confidential” on all pages of the document. This code could be run from the JavaScript console.

```
this.addWatermarkFromText(
   "Confidential", 0, font.HelvB, 24,
    color.red );
```

About this Document

Who should read this document

This document is a tutorial-level guide for users who need to add scripting intelligence to PDF documents and forms.

Document contents

This collection of JavaScript snippets provides code samples that address a variety of basic tasks commonly faced when trying to add JavaScript to enhance PDF forms and documents.

The samples are organized into two main categories:

Forms: For adding intelligence to Acrobat forms.

Documents: For all document operations other than those most suited to use in forms.

Within these categories, the samples are organized by their functional use rather than by the JavaScript object name, which makes it easier to find samples for a given task. For more information, see the *Developing Acrobat Applications using JavaScript*.

Prerequisites

This document is intended for users with little or no experience with JavaScript. However, its use will be greatly enhanced if you have a basic knowledge of the following:

- The types of events which can trigger JavaScripts actions
- How to use the Acrobat UI to create those actions
- The hierarchy of scripting objects, properties, and methods (see *JavaScript for Acrobat API Reference*).
- The PDF coordinate system used in PDF documents
- How to use the JavaScript Console and Debugger to enter, execute, and test scripts you are developing.

For information on the above topics, see *Developing Acrobat Applications Using JavaScript* as well as other documents listed in the Resources chapter.

Setting JavaScript actions

Adding JavaScript to PDF documents often requires setting actions for a variety of field types and events. The chapter Creating actions for events gives a brief summary of how to set those actions.

Multiple scripting models

A PDF file can either Acrobat forms and documents, or Adobe LiveCycle XML forms. The following table shows the relationship

between the various scripting models, the data formats, and what tools can be used to create and edit the document.

Form type	Data type	Scripting model	File format	Development / Editing tool
Acrobat	FDF	• JavaScript for Acrobat	PDF	Acrobat (with built-in JavaScript editor & debugger
LiveCycle	XML	• JavaScript (ECMA script) for LiveCycle • JavaScript for Acrobat[a] • FormCalc	PDF XDP[b]	LiveCycle Designer

a. Only a subset of the Acrobat scripting objects can be used in LiveCycle forms; see JavaScript for Acrobat for use in LiveCycle XML forms , below.

b. XDP format is used with LiveCycle forms

NOTE: You can tell if a form you open in a PDF file is an XML form because the form tools in the Tools Menu will be grayed out.

JavaScript for Acrobat for use in LiveCycle XML forms

JavaScript for Acrobat can also be used in XML forms for LiveCycle, but its use is restricted to those objects that do not affect form fields or the structure of the form. For example, it can be used for tasks such as to open or attach external files, manipulate annotations, check the version of the viewing application, etc. It cannot be used to change field values, get field names, add or remove fields, delete or extract pages, add an action to a link, create a template, or change tabbing order. Those types of operations must be done using JavaScript for LiveCycle.

For more information see *Converting Acrobat JavaScript for Use in LiveCycle Designer Forms*, available at:

http://partners.adobe.com/public/developer/en/acrobat/sdk/AcroJS_DesignerJS.pdf

Product Compatibility

Most samples shown in this document are intended for use with Acrobat 6.0 or greater, except where noted.

Setting JavaScript Actions

Creating actions for events

This section explains how to set the various types of JavaScript actions mentioned in this document.

NOTE: This section covers only the most basic action types. For complete information, see *Developing Acrobat Applications Using JavaScript*.

Setting up for JavaScript

Before you can use JavaScript, you must enable it as described in *Developing Acrobat Applications Using JavaScript*.

Executing JavaScript in the JavaScript console

The JavaScript console provides an interface for testing and debugging your JavaScript code., or you can use the console as a tool to interact with the document or form. You can either add intelligence to a PDF document, or you can execute JavaScript from the console as a processing tool. For example, you could add a new field to selected pages of an existing forms document, or add page numbers to a document without them.

From within the Adobe Acrobat application, open the Acrobat JavaScript console window by:

- Select Advanced > Document Processing > JavaScript Debugger... or...
- Type Ctrl-j (Windows), or...
- Type Control-j (Macintosh)

To evaluate or execute JavaScript code from the console:

- Type in the code, highlight it, and press the Enter key on the numeric keypad, or Ctrl+Enter on the regular keyboard.

Field actions

The following briefly describes how to enter custom Field JavaScripts for triggers such as:

- Mouse Up/Down
- Mouse Enter/Exit
- On Focus/Blur

NOTE: The *JavaScript for Acrobat API Reference* recommends using the Mouse Up action rather than the Mouse Down action, to avoid potential problems.

To create a custom action:

1. Using the object tool, double-click on a field to bring up the Field Properties dialog.
2. Select the Actions tab
3. In the Add an Action pane, under Select Action: select a trigger type such as Mouse Up, etc.
4. Under Select Action, select Run a JavaScript from the pop-up menu
5. Click the Add button and enter the script in the script editor window that pops up.

Setting JavaScript Actions

Custom Validation scripts

The following steps will set a custom validation script for a field. Custom validation scripts allow you, for example, to examine and validate what the user has entered before it is stored.

1. Double click on the field to bring up the Field Properties window.
2. Select the Validate tab.
3. Click the "Run custom validation script" radio button.
4. Click the Edit button, and enter the script.
5. Click OK when done.

Custom keystroke scripts

To create a script which can inspect the user's keystrokes and either respond with a warning message or change the keystroke, do the following:

1. Bring up the Text Field Properties dialog box.
2. Select the Format tab.
3. Under Select Format Category, select Custom from the pop-up menu.
4. Under Custom Keystroke Script, click on the Edit button.
5. A JavaScript Editor window will pop up, into which you can enter your custom keystroke script.

Page Actions

Page action scripts allow you to perform actions when a specified page is opened or closed. To create a Page action script:

1. Click on the Pages tab on the left side of the document viewing window in Acrobat. Page thumbnails will appear.
2. Click on the icon of the page to which you want to add the action.
3. Click on the Options pop-up menu at the top of that pane, and from the pop-up list select Page Properties.
4. Select the Actions tab.
5. From the Select Trigger pop-up menu, select either Page Open or Page Close, as appropriate.
6. In the Select Action pop-up menu, select Run a JavaScript.
7. Click the Add button and enter the script.
8. When done, click OK.

To define a function or a document open action

A document JavaScript executes when the document is opened, and can be used to either define a function which can be used globally in the document, or to execute an action you want performed before the user sees the document displayed.

To define a document JavaScript function, choose:

Advanced > JavaScript > Document JavaScripts

You can then enter a name for the script and the code you want executed, into the resulting dialog box, and then click Close.

Setting JavaScript Actions

To set document actions

To set a specific document action, choose:

Advanced > Document Processing > Set Document Actions...

You will get the Document Actions dialog, wherein you can select events such as Document Will Close, Document Will Print, Document Did Print, etc. Enter the JavaScript action you would like to associate with that event and click OK.

To create folder level JavaScript

Folder level JavaScripts are executed by Acrobat when the application is first opened. For example, if JavaScript is executed to add a menu item to the application, the resulting menu item will remain in effect for all documents, as long as the application is open.

Folder level JavaScripts are placed in separate files that have a .js file extension, and which are stored in either the Acrobat Application *JavaScripts* folder or in the user's JavaScripts folder. These scripts are loaded when the Acrobat application starts execution, and are associated with the Event object's Application Initialization (App/Init) Event.

To create folder level JavaScripts, you may use the JavaScript Editor, or any text editor and save the resulting file with the `.js` file extension.

Batch level JavaScripts

A batch level JavaScript is a script that can be applied to a collection of documents, and operates at the application level.

To create or edit a batch level JavaScript, click on the Advanced menu and choose the *Batch Processing...* menu item.

Testing JavaScript

Testing JavaScript code consists of either executing the code in the JavaScript console, or using the code in a PDF document. The section below briefly explains how to enable the built-in Acrobat debugger for JavaScrip; it is followed by some miscellaneous tips for using and debugging JavaScript. For more information on debugging, see *Developing Acrobat Applications Using JavaScript*.

Enabling the debugger

In order to make the debugger available for use, you must enable both JavaScript and the debugger. Use the Preferences dialog box (Ctrl+K) to control the behavior of the JavaScript development environment. To enable the debugger, select JavaScript from the list on the left in the Preferences dialog and make sure the Enable JavaScript debugger option is checked. Note that you must restart Acrobat for this option to take effect.

Simple debugging

Though Acrobat Professional has a full-featured JavaScript debugger, you can do simple debugging without learning the entire debugging process. For simple scripts, you can sometimes locate problems by (each described in following sections):

- Testing and/or developing script snippets in the console.
- Inserting **`console.println()`** commands to write information to the console.

Testing using the console

The JavaScript console allows you to evaluate single or multiple lines of code. There are three ways to evaluate JavaScript code while using the interactive console:

- To evaluate a portion of a line of code, highlight the portion and press either Enter on the numeric keypad or Ctrl+Enter.
- To evaluate a single line of code, make sure the cursor is positioned on that line and press either the Enter key on the numeric keypad or press Ctrl+Enter.
- To evaluate multiple lines of code, highlight the lines and press either Enter on the numeric keypad or Ctrl+Enter.

In all cases, the result is displayed in the console.

Using the `console.println()` command

Another way to debug a script is to insert code to print information to the JavaScript console. For example, if you want to find out how many times execution got through a **`for`** loop before failing, you could insert the **`console.println`** statement in the loop like this:

```
for (var i=0; i<numPages; i++)
  console.println("i =" + i)
```

or you can display the value of a variable (in this example, a fictitious variable named **`pageTotal`**) as follows:

```
  console.println("Pages processed so far: ",
pageTotal);
```

Testing JavaScript

Miscellaneous debugging tips

- **When you see "Undefined" displayed after your script executes in the console.** When testing a script in the JavaScript console, if the script executes correctly, the message "Undefined" will often appear. The debug mechanism is designed to output the return value of a function. But if your code is not a function call, there will be no returned value, and hence the message "Undefined" is displayed after execution.
- **Using breakpoints**. When you set breakpoints in the JavaScript Debugger window, you must close that window before you can test the file. It is not necessary to explicitly save the script or the file; Acrobat saves it automatically, and it will be there when you next open the window, or when execution stops at a breakpoint.

Common development problems and tips

- **Security restrictions**. If you're having problems with an action attached, for example, to a field or for a Document Open event, check the JavaScript for Acrobat API Reference to make sure that security restrictions do not limit that method to use only in batch mode or from the JavaScript console.
- **Which scripting model?** When you receive a form document from someone else, if you open it and the forms tools in the menu bars are grayed-out, it means that it is an XML-based form, and form field editing cannot be done using Acrobat. You must open the document using Adobe Designer. Also, only some JavaScript for Acrobat will work in a LiveCycle XML form, see *Converting Acrobat JavaScript for Use in LiveCycle Designer Forms*, available at:

http://partners.adobe.com/public/developer/en/acrobat/sdk/AcroJS_DesignerJS.pdf

For all other JavaScript functions, you must use JavaScript for LiveCycle Designer (which is XML-based).

- **Curly quote characters cause errors**. If you have cut-and-pasted code from another document, it is possible that the code text has "curly" quote characters (for example, ” instead of `"`), which will cause an error in execution.
- **Useful object properties and methods.** Familiarize yourself with `doc` object properties and methods such as:

  `numFields`
  `numPages`
  `pageNum`

  as well as the `doc` methods such as:

  `getNthFieldName`
  `getPageNthWord`
  `getPageNumWords`

  Those properties and methods are useful for looping through multiple objects, pages, fields, or words.

Resources

From Adobe

Adobe Acrobat technical notes are available at:

http://partners.adobe.com/public/developer/acrobat/sdk/index_doc.html#js

Adobe Documents:

JavaScript For Acrobat API Reference

Developing Acrobat Applications Using JavaScript

Acrobat Online Collaboration: Setup and Administration

Acrobat 6.0 Online Training (for ASN members)

http://partners.adobe.com/asn/tech/pdf/training.jsp

Other Resources

General JavaScript:

Core JavaScript 1.5 specification

http://partners.adobe.com/NSjscript/

Flanagan, David, *JavaScript, the Definitive Guide (Sebastopol, California:* O'Reilly, 2002).

JavaScript for Acrobat:

Deubert, John, *Extending Acrobat Forms with Javascript*. Peach Pit Press, 2003 (Written for Acrobat 5.0, but still useful for newer versions.).

Padova, Ted, *Creating Adobe Acrobat Forms*, New York: Wiley, 2002. ISBN: 0-7645-3690-7.

Index

A

addWatermarkFromFile() 37
addWatermarkFromText() 37
annotations
    create an annotation 22
    remove an annotation 22
application version number 22

B

batch level JavaScript 42
bookmarks 23
    create a new bookmark 23
    insert child bookmark 23
    remove bookmark 23

C

calculations 9
    delay redrawing 9
    Turn calculations on/off 9
color
    defining a custom color 12
console, JavaScript 40
console.println() 12
custom validation scripts 41

D

date parsing 8
date timestamp 8
dates
    Date() 8
display descriptive help text 16
document actions 42
documents
    create a new blank document 23
    open an existing document 24
    prompt user for document to open 24

E

event.target.value 12, 13
event.value 13

F

field properties
    alignment 11
    background color 11
    default value 11
    display 11
    font 11
    font size 11
    multiline 11
    password 12
    required 12
    stroke color 12
    stroke line width 12
fields
    changing field properties 11
    check a check box field 12
    check before submitting 18
    creating new fields programmatically 10
    formatting a number 12
    hiding existing field based on conditions 13
    highlighting a single field 13
    highlighting using a function 14
    removing zeros from a field 13
    set action for multiple fields 18
    set an action for a single field 17
folder level JavaScript 42
form submission, checking, and resetting 18
format, custom keystroke script 41
forms
    calculations 9
    date handling 8
    field operations 10
    final field checks 18
    validation 20
function 41

G

getNthFieldName(i) 17
getting today's date 9

J

JavaScript console 40
JavaScript function 41

K

keystroke, custom script 41

L

links
    add a link to a page 24
    link.setAction() 26
    remove all links on a page 25
    set an action for a link 26

M

Menu operations 26
menus
    add a new menu item 26
    addMenuItem() 26
    app.execMenuItem() 27
    app.hideMenuItem() 27
    app.listMenuItems() 27
    execute menu item 27
    hide a menu item 27
    list menu items 27
messages/dialogs

app.alert() 28
app.beep 28
app.response() 28
beep the user 28
prompt the user for a response 28
simple alert 28
metadata
get document metadata 24
methods
addAnnot() 22
addField() 31
addLink() 25
addMenuItem() 35
addWatermarkFromFile() 37
addWatermarkFromText() 37
AFParseDateEx() 8
app.addMenuItem() 26
app.alert() 28
app.beep 28
app.execMenuItem() 24, 27
app.fs.isFullScreen 36
app.listMenuItems() 27
app.newDoc() 23
app.openDoc() 24
app.popUpMenuEx 37
app.popUpMenuEx() 29, 30
app.response() 28
console.println() 23
Date() 8
getAnnots() 22
getNthTemplate() 32
getOCGs() 31
getPageBox() 25
removeLinks() 25
resetForm() 19
search.query() 34
setAction() 14, 25
setItems() 33
spawnPageFromTemplate() 32
spell.checkWord() 35
util.printd 8
mouse down event 17
mouse enter event 16, 17
mouse exit event 16, 17

N

navigation
add a pop-up menu 29
add navigational arrows to every page 31
adding page numbers 29
ap.popUpMenuEx() 29
app.popUpMenuEx() 30
get page number of current page 29
go to first page 30
go to last page 30
go to next page 30
go to page number 30
go to previous page 30
pop-up menu for page selection 30

O

OCG
getOCGs() 31
hide all OCG objects on current page 32
show all OCG objects on currrent page 31
toggle state of OCG objects 32
OCG (Optional Content Group) 31

P

page actions, creating 41
pages
insert new page 32
insert/delete/replace/extract pages 32
PostScript font name 11
printing
get list of printers as a combo box 33
print current page (no dialog) 33
print list of available printers 33
print selected form pages 16
set field to not print 16
properties
app.plugIns 22
app.viewerVersion 22
ocgArray.state 32
setItems(app.printerNames) 33

R

reset form 19
resetForm() 19
restricting keystrokes 21
rollover help 16

S

search
attachments 34
bookmarks 34
ignoreAccents 34
matchCase 34
matchWholeWord 34
metadata (XMP) 34
searching
a simple find operation 34
search all fields with specified prefix 17
search fields 17
search.query() 34
setAction() 14
spell checking
spell.checkWord() 35
spellcheck a document 34

T

timing
    set countdown timer 35

V

validation 20
validation, custom scripts 41
verify that required entry was made 20
version checking 22
viewing
    app.fs.isFullScreen 36
    app.popUpMenuEx 37
    change zoom value 36
    go to fullscreen and start slide show 36
    pop-up menus for zooming 37
    prompting for a zoom value 36

W

watermarks
    adding from an external file 37
    adding from text 37