Adobe

Adobe® Acrobat®

Technical Note # 5186

Acrobat Forms JavaScript Object Specification

Version 4.0

Revised: January 27, 1999

Table Of Contents

4.0 JavaScript Object Specification

Introduction

JavaScript, the scripting language developed by Netscape Communications, enables you to easily create interactive Web pages.

JavaScript v1.2 comes with six predefined classes: *Boolean, Number, Date, Math, String*, and *Array*. As part of the integration with Adobe® Acrobat® Forms, we have defined additional classes and objects to allow access to portions of the PDF file.

This document describes these classes, as well as details the load and execution of JavaScripts in the Adobe Acrobat environment. Of particular note is the Field Object section which handles all processing of Acrobat Form fields including their formatting, calculation, and validation.

The intended audience of this document is assumed to be familiar with Adobe Acrobat, the Acrobat Forms plug-in and the Adobe Acrobat plug-in API.

Other useful documents

For more information on JavaScript, please see Netscape's JavaScript Reference Manual for details on JavaScript objects and on language syntax.

Portable Document Format Reference Manual, version 1.2 or later, describes the PDF representation of a form and its fields. Appendix H, describes the Forms Data Format, which is one of the formats of data exported from an Acrobat form. This document is available with the Adobe Acrobat Plug-ins SDK CD-ROM or on the Adobe Web site http://www.adobe.com/supportservice/devrelations/PDFS/TN/PDFSPEC.PDF.

Technical Note #5166, Acrobat Viewer plug-in API Overview. Gives an overview of the objects and methods provided by the Acrobat viewers' plug-in API. This document is available with the Adobe Acrobat Plug-ins SDK CD-ROM or on the Adobe Web site http://beta1.adobe.com/ada/acrosdk/DOCS/VWRPIOVR.PDF.

Technical Note #5167, Acrobat Viewer plug-in API Development. Tells how to develop Acrobat viewer plug-ins on the various platforms available. This document is available with the Adobe Acrobat Plug-ins SDK CD-ROM or on the Adobe Web site http://beta1.adobe.com/ada/acrosdk/DOCS/VWRPIDEV.PDF

Technical Note #5168, Acrobat Viewer plug-in API On-Line Reference. Describes in detail the objects and methods provided by the Acrobat viewer's plug-in API. This document is available

with the Adobe Acrobat Plug-ins SDK CD-ROM or on the Adobe Web site http://beta1.adobe.com/ada/acrosdk/DOCS/VWRPIREF.PDF.

Document Conventions

As this document pertains to Acrobat Forms version 3.5 and greater, there exists some compatibility issues with older versions of the software. If a property, method, or object is marked with a #.# symbol, then it is available only in versions of the Acrobat Forms software greater than or equal to #.#. Before accessing this object, property, or method, the script should check that the application version is greater than or equal to #.# if backwards compatibility is desired.

Example:

```
if (typeof app.formsVersion != "undefined" && app.formsVersion >= 4.0) {
  // Perform version specific operations.
}
```

As the JavaScript extensions to Acrobat Forms have evolved, some properties or methods have been superseded by other, more flexible or appropriate properties or methods. The use of these older methods are discouraged and marked by Mr. Unhappy.

What's New for 4.0

Object	Properties	Methods
App Object	formsVersion, numPlugIns, openInPlace, toolbarHorizontal, toolbarVertical, viewerVersion	getNthPlugInName(), hideMenuItem(), hideToolbarButton(), mailMsg()
Color Object	dkGray, gray, ltGray	
Doc Object	calculate, delay, external, numFields	exportAsFDF(), getNthFieldName(), getURL(), importAnFDF(), mailDoc(), mailForm(), *spawnPageFromTemplate()*, *submitForm()*
Field Object	display	setItems(), deleteItemAt()
Util Object		scand()

Note: *Items shown in italics are not new to this release but they have been modified or added to.*

New sections of the documentation for 4.0:

New Section	Description
Working With The Date Object	Describes the proper methods for formatting and parsing dates and performing date arithmetic.
Acrobat Forms and Security	Shows how to properly define security settings in a form so that it cannot be tampered with.
Working with Signature Fields	Details the proper usage of scripts when signing documents including inverse operations upon form reset

Acrobat Forms JavaScript Object Model

Acrobat Forms defines an object model on top of JavaScript 1.2. These objects are only defined within the Adobe Acrobat realm and are specific to Acrobat Forms. They basically mirror the Acrobat Forms components and give the forms developer a way to access these components programmatically in order to query and change their properties. In addition to defining forms specific objects, there are additional generic objects that allows the developer to access the underlying document and perform certain actions on it.

App Object

The App object is a static JavaScript object that defines a number of Acrobat specific functions plus a variety of utility routines and convenience functions.

App Object Properties

calculate

☹

Type: Boolean *Access: R/W*

If this property is set to *true*, it will allow calculations to be performed. If set to *false*, this property prevents all calculations in all documents from happening. Its default value is *true*. See also the document calculate property which supersedes this property in later versions.

formsVersion

4.0

Type: Number *Access: R*

This property indicates the version number of the forms software running inside the viewer. Use this method to determine whether objects, properties, or methods in newer versions of the software are available if you wish to maintain backwards compatibility in your scripts. See Document Conventions for more details.

Example:

```
if (typeof app.formsVersion != "undefined" && app.formsVersion >= 4.5) {
  // Perform version specific operations here.
}
```

fullscreen

Type: Boolean *Access: R/W*

This property puts the Acrobat viewer in fullscreen mode vs. regular viewing mode.

Example:

```
// on mouse up, set to fullscreen mode
app.fullscreen = true;
```

In the above example, the Adobe Acrobat viewer is set to fullscreen mode when *app.fullscreen* is set to *true*. If *app.fullscreen* was *false* then default viewing mode would be set. The default viewing mode is defined as the original mode the Acrobat application was in before full screen mode was initiated.

language

Type: String *Access: R*

This property defines the language of the running Acrobat Viewer. It returns the following strings:

String	Language, Country
DEU	German, Germany
ENU	English, The United States of America
ESP	Spanish, Spain
FRA	French, France
ITA	Italian, Italy
JPN	Japanese, Japan
NLD	Dutch, The Netherlands
SVE	Swedish, Sweden

numPlugIns

4.0

Type: Number *Access: R*

This property indicates the number of plug-ins that have been loaded by Acrobat. See also the getNthPlugInName method.

openInPlace

4.0

Type: Number *Access: R/W*

This property determines whether cross-document links are opened in the same window or opened in a new window.

platform

Type: String *Access: R*

This property returns the platform that the script is currently executing on. Valid values include “WIN”, “MAC”, and “UNIX”.

toolbar

Type: Boolean *Access: R/W*

This property allows a script to show or hide both the horizontal and vertical Acrobat tool bars. It does not hide the tool bar in external windows (i.e. in an Acrobat window within a Web browser).

Example:

```
// Opened the document, now remove the toolbar.
app.toolbar = false;
```

toolbarHorizontal

4.0

Type: Boolean *Access: R/W*

This property allows a script to show or hide the Acrobat horizontal tool bar. It does not hide the tool bar in external windows (i.e. in an Acrobat window within a Web browser).

toolbarVertical

4.0

Type: Boolean *Access: R/W*

This property allows a script to show or hide the Acrobat vertical tool bar. It does not hide the tool bar in external windows (i.e. in an Acrobat window within a Web browser).

viewerType

Type: String *Access: R*

This property determines if the running Acrobat Viewer is the Reader vs. Exchange. Its value is “Reader” or “Exchange” respectively.

viewerVersion

4.0

Type: Number *Access: R*

This property indicates the version number of the current viewer (i.e. Reader or Exchange).

App Object Methods

alert

Parameters: cMessage, [nIcon], [nType]
Returns: nButton

This method displays an alert dialog on the screen. The minimum required parameter is a string containing the message to be displayed. Optionally, an icon type can be specified by using the *nIcon* parameter. The following is a list of icons and their associated values:

Icon	Value
Error (default)	0
Warning	1
Question	2
Status	3

Additionally, a button group type can be specified:

Button Group	Value
OK (default)	0
OK, Cancel	1
Yes, No	2
Yes, No, Cancel	3

This method returns the type of the button that was pressed by the user:

Button Type	Value
Error	0
OK	1
Cancel	2
No	3
Yes	4

beep

Parameters: [nType]
Returns: none

This method causes the system to play a sound. The various sounds and the values used are as follows:

Message Type	Value
Error (default)	0
Warning	1
Question	2
Status	3
Default	4

On Apple Macintosh and UNIX systems the beep type is ignored.

execMenuItem

4.0

Parameters: cMenuItem
Returns: nothing

This method executes the specified menu item. To find out the exact menu item name, see the Acrobat Viewer Plug-in API On-line Reference under § *Lists*: *Menu item names*. The menu item name is case sensitive and has to exactly match strings in the list.

Example:

```
/* This example executes File->Open menu item. It will display a dialog to the
user asking for the file to be opened. */
app.execMenuItem("Open");
```

Note: *For security reasons, we will not allow the script to execute the* ***SaveAs*** *and* ***Close*** *menu items. If either of those two menu items is passed as cMenuItem, this function will do nothing.*

---

getNthPlugInName

4.0

Parameters: nIndex
Returns: cName

This method returns the name of the Nth plug-in that has been loaded by the viewer. See also the numPlugIns property.

goBack

Parameters: None
Returns: nothing

Use this function to go to the previous view on the view stack. This is equivalent to pressing the go back button on the Acrobat tool bar.

goForward

Parameters: None
Returns: nothing

Use this function to go to the next view on the view stack. This is equivalent to pressing the go forward button on the Acrobat tool bar.

hideMenuItem

4.0

Parameters: cName
Returns: nothing

This method only works in the *Config.js* Plug-in folder level script. It allows a forms integrator to customize the look of the Acrobat viewer by removing the menu item specified by *cName*. Language independent names for toolbar buttons can be obtained from the *Acrobat Viewer Plug-In API On-line Reference (Technical Note #5168)*. See Other useful documents for more details.

hideToolbarButton

4.0

Parameters: cName
Returns: nothing

This method only works in the *Config.js* Plug-in folder level script. It allows a forms integrator to customize the look of the Acrobat viewer by removing the toolbar button specified by *cName*. Language independent names for toolbar buttons can be obtained from the *Acrobat Viewer Plug-In API On-line Reference (Technical Note #5168)*. See Other useful documents for more details.

mailMsg

4.0

Parameters: bUI, cTo, [cCc], [cBcc], [cSubject], [cMsgBody]
Returns: nothing

This method sends out an e-mail message with or without user interaction depending on the value of *bUI*. If it is set to true then the rest parameters are used to seed the compose new message window that is displayed to the user.

If *bUI* is set to false, the *cTo* parameter is required and others are optional. You must use a semicolon ";" to separate multiple recipients in *cTo*, *cCc*, *cBcc* parameters. The length limit for *cSubject* and *cMsgBody* is 64k bytes.

Example:

```
/* This will pop up the compose new message window */
app.mailMsg(true);
/* This will send out the mail to fun1@fun.com and fun2@fun.com */
app.mailMsg(false, "fun1@fun.com; fun2@fun.com", "", "", "This is the subject",
  "This is the body of the mail.");
```

Note: *This is a Windows-only feature. In addition, the client machine must have its default mail program configured to be MAPI enabled in order to use this method.*

---

response

Parameters: cQuestion [cTitle], [cOldValue]
Returns: cResponse or null on cancel

This method displays a dialog box containing a question and an entry field for the user to reply to the question. Optionally, the dialog may have a title or a default value for the answer to the question. The return value is a string containing the user's response. If the user presses the cancel button on the dialog the response is the null object.

Color Arrays

A color is represented in JavaScript as an array containing 1, 2, 4, or 5 elements corresponding to a transparent, gray, RGB, or CMYK color space, respectively. The first element in the array is a string denoting the color space type. The subsequent elements are numbers that range between zero and one inclusive. The following table illustrates this:

Color Space	String	# of Additional Elements
Transparent	“T”	0
Gray	“G”	1
RGB	“RGB”	3
CMYK	“CMYK”	4

For example, the color red can be represented as [“RGB” 1 0 0].

Invalid strings or insufficient elements in a color array cause the color to be interpreted as the color black.

A *transparent* color space indicates a complete absence of color and will allow those portions of the document underlying the current field to show through.

Colors in the *gray* color space are represented by a single value—the intensity of achromatic light. In this color space, 0 is black, 1 is white, and intermediate values represent shades of gray (i.e. “.5”, “.7” etc.).

Colors in the *RGB* color space are represented by three values: the intensity of the *red*, *green*, and *blue* components in the output. RGB is commonly used for video displays because they are generally based on red, green, and blue phosphors.

Colors in the *CMYK* color space are represented by four values. These values are the amounts of the *cyan*, *magenta*, *yellow*, and *black* components in the output. This color space is commonly used for color printers, where they are the colors of the inks traditionally used in four-color printing. Only cyan, magenta, and yellow are necessary, but black is generally used in printing because black ink produces a better black than a mixture of cyan, magenta, and yellow inks, and because black ink is less expensive than the other inks.

Color Object

The *color* object is a convenience static object that defines the basic colors. These colors are accessed in JavaScripts via the color object. Use this object whenever you want to set a property or call a method that require a color array. The color object is defined in *AForm.js*.

Color Properties

The color object defines the following colors and there associated keywords:

Color Object	Keyword	Version
Transparent	color.transparent	
Black	color.black	
White	color.white	
Red	color.red	
Green	color.green	
Blue	color.blue	
Cyan	color.cyan	
Magenta	color.magenta	
Yellow	color.yellow	
Dark Gray	color.dkGray	4.0
Gray	color.gray	4.0
Light Gray	color.ltGray	4.0

Example:

```
// This example sets the text color of the field to red
// if the value of the field is negative, else it sets it
// to black.
var f = event.target; /* field that the event occurs at */
if (event.value < 0)
    f.textColor = color.red;
else
    f.textColor = color.black;
```

Console Object

The Console object is a static object to access the JavaScript console for displaying debug messages. It functions only within Acrobat Exchange.

Console Methods

show

Parameters: none
Returns: none

This method shows the console window.

hide

Parameters: none
Returns: none

This method closes the console window.

println

Parameters: cMessage
Returns: none

This method prints the string value of *cMessage* to the console window with an accompanying carriage return.

Example:

```
// This example prints the value of a field to the console window
var f = event.target;
console.println("Field value = " + f.value);
```

clear

Parameters: none
Returns: nothing

This method clears the console windows buffer of any output.

Doc Object

The JavaScript Doc object provides the interfaces between a PDF document open in the viewer and the JavaScript interpreter. It provides methods and properties of the PDF document.

Doc Access from JavaScript

Accessing the Doc object from JavaScript can be done either through the this Object, which usually points to the Doc object of the underlying document. The target event property points to the field that initiated the event for all *mouse*, *calculate*, *validate*, and *format* events. For all other events, it directly points to the Doc object. The examples below illustrates the use of the Event object to access the Doc object of the underlying document.

```
// In Mouse, calculate, validate, format events
var doc = event.target.doc;

// In all other events
var doc = event.target;
```

Doc Object Properties

author

Type: String *Access: R/W*

This property defines the author of the document.

```
this.author = "Robert Frost";
```

calculate

4.0

Type: Boolean *Access: R/W*

If this property is set to *true*, it will allow calculations to be performed for this document. If set to *false*, this property prevents all calculations from happening for this document. Its default value is *true*. This property supersedes the application level calculate property whose use is now discouraged.

creator

Type: String *Access: R*

This property defines the creator of the document (e.g. *"Adobe FrameMaker"*, *"Adobe PageMaker"*, etc.).

creationDate

Type: Date *Access: R*

This property defines the documents creation date.

delay

4.0

Type: Boolean *Access: R/W*

This property delays the redrawing of any appearance changes to every field in the document. It is generally used to buffer a series of changes to fields before requesting that the fields regenerate their appearance. Setting the property to *true* forces all changes to be queued until *delay* is reset to *false*. Once set to *false* then all the fields on the page are re-drawn. See also the field level delay property.

dirty

Type: Boolean *Access: R/W*

This property identifies whether the document has been dirtied as the result of a changes to the document (and therefore needs to be saved). It is useful to reset the *dirty* flag in a document when performing changes that do not warrant saving, for example, updating a status field in the document.

```
var f = this.getField("Status");
var b = this.dirty;
f.value = "Press the reset button to clear the form.";
this.dirty = b;
```

external

4.0

Type: Boolean *Access: R*

This property indicates whether the current document is being viewed in the Acrobat application or in an external window (such as a web browser).

filesize

Type: Integer *Access: R*

This property determines the file size of the document in bytes.

keywords

Type: String *Access: R/W*

This property specifies the document's keywords in the Adobe Acrobat *File->Document Info->General* dialog box.

modDate

Type: Date *Access: R*

This property contains the date the document was last modified.

numFields

4.0

Type: Integer *Access: R*

This property returns the total number of fields in the document. See also the getNthFieldName method.

numPages

Type: Integer *Access: R*

This property contains the number of pages in the document.

numTemplates

Type: Integer *Access: R*

This property returns the number of templates in the document (see also getNthTemplate and spawnPageFromTemplate methods).

path

Type: String *Access: R*

This property defines the device independent path of the document, for example /Acrobat3/Exchange/doc.pdf

pageNum

Type: Integer *Access: R/W*

Use this property to get or set a page of the document. When setting the *pageNum* to a specific page, remember that the values are “0” based.

```
// This example will go to the first page of the document.
this.pageNum = 0 ;
```

Or *pageNum* can be used to advance “*n*” pages in the document:

```
// This example will advance the document to the next page
this.pageNum++;
```

producer

Type: String *Access: R*

This property contains producer of the document (e.g. “Acrobat Distiller”, “PDFWriter”, etc.).

subject

Type: String *Access: R/W*

This property defines the document’s subject.

title

Type: String *Access: R/W*

This property specifies the document’s title

zoom

Type: Integer *Access: R/W*

Use this property it to get or set the current page *zoom* level. The values allowed are 12% and 800% specified as an integer.

Example:

```
// This example will zoom in to twice the current zoom level.
this.zoom *= 2;

// This now sets the zoom to 200%
this.zoom = 200;
```

zoomType

Type: String *Access: R/W*

This property specifies the current zoom type of the document. Valid zoom types are: *none*, *fit page*, *fit width*, *fit height*, and *fit visible width*. A convenience *zoomType* object that defines all the valid zoom types is provided for use from JavaScript. It provides the following zoom types:

Zoom Type	Keyword
NoVary	zoomtype.none
FitPage	zoomtype.fitP
FitWidth	zoomtype.fitW
FitHeight	zoomtype.fitH
FitVisibleWidth	zoomtype.fitV
Preferred	zoomtype.pref

Example:

```
// This example sets the zoom type of the document to fit the width.
this.zoomType = zoomtype.fitW;
```

Doc Object Methods

calculateNow

Parameters: none
Returns: nothing

Use this function to force computation of all calculation fields in the current document.

exportAsFDF

4.0

Parameters: [bAllFields], [bNoPassword], [aFields], [bFlags]
Returns: nothing

Use this method to export an FDF file to the local hard drive. Upon invocation, a dialog will be shown to let the user select the file to export to.

The optional *bAllFields* parameter indicates, if true, that all fields are exported, including those that have no value, and if false (the default) to exclude those that currently have no value.

The optional *bNoPassword* parameter indicates, if true (the default), not to include in the exported FDF text fields that have the “password” flag set.

The optional *aFields* parameter is the array of field names to submit or a string containing a single field name. If this parameter is present then only the fields indicated are exported, except those excluded by parameter *bEmpty* or *bNoPassword*. If this parameter is omitted or is null then all fields in the form are exported (again subject to the restrictions of *bEmpty*).

You can include non-terminal fields in the array or the string passed as a parameter: this is a simple shortcut for having a whole subtree exported.

Example:

```
/* Export the entire form (including empty fields) with flags. */
this.exportAsFDF(true, true, null, true);
/* Export the name subtree with no flags. */
this.exportAsFDF(false, true, “name”);
```

The example above illustrates a shortcut to exporting a whole subtree. Passing “name” as part of the *aFields* parameter, exports *“name.title”*, *“name.first”*, *“name.middle”* and *“name.last”*, etc.

The optional *aFlags* parameter indicates, if true, that field flags should be included in the exported FDF. The default is false.

This method does not work in the Acrobat Reader.

getField

Parameters: cName
Returns: object

Use this function to map a field object in the PDF document to a JavaScript variable. The *cName* parameter is the name of the field of interest. This function returns a Field JavaScript object representing the form field in the PDF document.

getNthFieldName

4.0

Parameters: nIndex
Returns: cName

Use this function to obtain the *n*th field name in the document (see the numFields property).

Example:

```
// Enumerate through all of the fields in the document.
for (var i = 0; i < this.numFields; i++) {
  console.println("Field[" + i + "] = " + this.getNthFieldName(i));
}
```

getNthTemplate

Parameters: nWhich
Returns: cName

Use this function to retrieve the name of a template within in the document. (See also the numTemplates property and spawnPageFromTemplate method.)

getURL

4.0

Parameters: cURL, [bAppend]
Returns: nothing

This method retrieves the specified URL over the internet using a GET. If the current document is being viewed inside the browser or Acrobat Web Capture is not available, it uses the Weblink plug-in to retrieve the requested URL. If *bAppend* is true (the default) and Acrobat Web Capture is available, the resulting pages are appended to the current document.

gotoNamedDest

Parameters: cName
Returns: nothing

Use this method to go to a named destination within the PDF document. For more details on named destinations and how to create them, see the PDF Reference Manual Version 1.3

importAnFDF

4.0

Parameters: [cFile]
Returns: nothing

This method imports the specified FDF file. The *cFile* parameter specifies the device-independent pathname to the FDF file. See Section 7.4 of the PDF Reference Manual for a description of the device-independent pathname format (it should look like the value of the /F key in an FDF exported via the exportAsFDF method or via the "File->Export->Form Data" menu item). The pathname may be relative to the location of the current document. If the parameter is omitted a dialog will be shown to let the user select the file. This method does not work in the Acrobat Reader.

mailDoc

4.0

Parameters: bUI, cTo, [cCc], [cBcc], [cSubject], [cMsgBody]
Returns: nothing

This method saves the current PDF document and mails it as an attachment to all recipients with or without user interaction depending on the value of *bUI*. If it is set to *true* then the rest

of the parameters are used to seed the compose new message window that is displayed to the user.

If *bUI* is set to false, the *cTo* parameter is required and all others are optional. You must use a semicolon ";" to separate multiple recipients in *cTo*, *cCc*, *cBcc* parameters. The length limit for *cSubject* and *cMsgBody* is 64k bytes.

Example:

```
/* This will pop up the compose new message window */
this.mailDoc(true);
/* This will send out the mail with the attached PDF file to fun1@fun.com and
fun2@fun.com */
this.mailDoc(false, "fun1@fun.com", "fun2@fun.com", "", "This is the subject",
"This is the body.");
```

Note: *This is a Windows-only feature. In addition, the client machine must have its default mail program configured to be MAPI enabled in order to use this method.*

---

mailForm

4.0

Parameters: bUI, cTo, [cCc], [cBcc], [cSubject], [cMsgBody]
Returns: nothing

This method exports the form data and mails the resulting FDF file as an attachment to all recipients, with or without user interaction depending on the value of *bUI*. If it is set to *true* then the rest of the parameters are used to seed the compose new message window that is displayed to the user.

If *bUI* is set to false, the *cTo* parameter is required and all others are optional. You must use a semicolon ";" to separate multiple recipients in *cTo*, *cCc*, *cBcc* parameters. The length limit for *cSubject* and *cMsgBody* is 64k bytes.

Example:

```
/* This will pop up the compose new message window */
this.mailForm(true);
/* This will send out the mail with the attached FDF file to fun1@fun.com and
fun2@fun.com */
this.mailForm(false, "fun1@fun.com; fun2@fun.com", "", "", "This is the
subject", "This is the body of the mail.");
```

Note: *This is a Windows-only feature. In addition, the client machine must have its default mail program configured to be MAPI enabled in order to use this method.*

print

Parameters: bInteractive, [nFirstPage], [nLastPage], [bSilent]
Returns: nothing

Use this function to print all or a specific number of pages of the document. If *bInteractive* is *true*, Acrobat displays the standard print dialog and all other parameters are ignored. The optional *nFirstPage* and *nLastPage* parameters specify the range of pages to print. If using *nFirstPage* and *nLastPage* parameters *bInteractive* must be *false*. Set the optional *bSilent* flag to *true* if you don't want to display the cancel dialog box while the document is printing. The default value for *bSilent* flag is *false*.

Example:

```
// This Example will print current page the document is on
this.print(false, this.pageNum, this.pageNum);
```

resetForm

Parameters: [aFields]
Returns: nothing

Use this method to reset the field values within a document. If the *aFields* parameter is present, then only the fields indicated are reset. If not present or null then all fields in the form are reset. You can include non-terminal fields in the array. Use this as a simple shortcut for having a whole subtree reset. For example, if you pass "name" as part of the fields array then "name.first", "name.last", etc. will be reset.

```
var fields = new Array(2);
fields[0] = "P1.OrderForm.Description";
fields[1] = "P1.OrderForm.Qty";
this.resetForm(fields);
```

scroll

Parameters: xOrigin, yOrigin
Returns: nothing

Use this function to scroll the current page to the location specified by *xOrigin* and *yOrigin*. These coordinates must be defined in user space. Please refer to the PDF Reference Manual Version 1.3 for more details on the user space coordinate system.

spawnPageFromTemplate

Parameters: cTemplate, [nPage], [bRename], [bInsert]
Returns: nothing

Use this method with a template name, such as the ones returned by getNthTemplate. The optional parameter *nPage*, represents the page number (zero-based) into which the template will be spawned. If that page already exists, then the template contents are appended to that page. If *nPage* is omitted, a new page is created at the end of the document. The optional parameter *bRename*, is a boolean that indicates whether fields should be renamed. The default for *bRename* is *true*.

Example:

```
var n = this.numTemplates;
var cTempl;
for (i = 0; i < n; i++) {
    cTempl = this.getNthTemplate(i);
    this.spawnPageFromTemplate(cTempl);
}
```

4.0 Addition

If *bInsert* is specified then the template is inserted before the page specified by *nPage* as opposed to being overlaid on top of that page. The default for *bInsert* is false.

submitForm

Parameters: cURL, [bFDF], [bEmpty], [aFields], [bGet]
Returns: nothing

Use this method to submit the form to a specific URL. The first parameter, *cURL*, is the URL to submit to. This string must end in "*#FDF*" if the result from the submission is FDF and the document is being viewed inside a browser window.

The optional *bFDF* parameter is a boolean that indicates to submit as FDF or HTML. If set to *true*, the default, it submits the form data as FDF. If *false*, it submits it as HTML (URL encoded).

The optional *bEmpty* parameter is a boolean that indicates, when *true*, that all fields are submitted, including those that have no value and if *false* to exclude those that currently have no value. The default for *bEmpty* is *false*.

The optional *aFields* parameter is the array of field names to submit or a string containing a single field name. If this parameter is present then only the fields indicated are submitted, except those excluded by parameter *bEmpty*. If this parameter is omitted or is null then all fields in the Form are submitted (again subject to the restrictions of *bEmpty*).

You can include non-terminal fields in the array or the string passed as a parameter: this is a simple shortcut for having a whole subtree submitted.

Example:

```
/* Submit the form to the server */
this.submitForm("http://myserver/cgi-bin/myscript.cgi#FDF");
/* Or */
this.submitForm("http://myserver/cgi-bin/myscript.cgi#FDF",
          true, "name");
```

The example above illustrates a shortcut to submitting a whole subtree. Passing “name” as part of the *field* parameter, submits *“name.title”, “name.first”, “name.middle”* and *“name.last”*.

4.0 Addition

The optional *bGet* parameter indicates, if true, that the submission uses the GET HTTP method and if false (the default) a POST. GET is only allowed if using Acrobat Web Capture to submit (the browser interface only supports POST) and only if the data is sent as HTML (i.e. *bFDF* should be false).

Note: *You need to be running inside a web browser or have the Acrobat Web Capture plug-in installed, in order to call the* ***submitForm*** *method (unless the URL uses the “mailto” scheme, in which case it will be honored even if not running inside a web browser, as long as the SendMail plug-in is present).*

Note: *Usage of the* ***https*** *protocol is supported for secure connections.*

Event Object

All JavaScripts are executed as the result of a particular event (also referred to as a trigger). Acrobat Forms accepts the following events and executes any scripts that are specified for these events: *mouse enter, mouse down, mouse up, mouse exit, keystroke, format, validate,* and *calculate*. It is important to JavaScript writers to know what these events are and when and what order they are processed.

Event Processing

Mouse Enter

The *mouse enter* event is triggered when a user moves the mouse pointer inside the rectangle of a field. This is the typical place to open a text field to display help text, etc.

Mouse Down

The *mouse down* event is triggered when a user starts to click on a form field and the mouse button is still down. It is advised that you perform very little processing (i.e. play a short sound) during this event. A mouse down event will not occur unless a *mouse enter* event has already occurred.

Mouse Up

The *mouse up* event is triggered when the user clicks on a form field and releases the mouse button. This is the typical place to attach routines such as the submit action if a form. A *mouse up* event will not occur unless a *mouse down* event has already occurred.

Mouse Exit

The *mouse exit* event is the opposite of the *mouse enter* event and occurs when a user moves the mouse pointer outside of the rectangle of a field. A *mouse exit* event will not occur unless a *mouse enter* event has already occurred.

Keystroke

The *keystroke* event occurs whenever a user types a keystroke into a *text box* or *combo box* (this includes cut and paste operations), or selects an item in a *combo box* drop down or *listbox* field. A keystroke script may want to limit the type of keys allowed. For example, a numeric field might only allow numeric characters.

The user interface for Acrobat Forms allows the author to specify a *Selection Change* script for list boxes. The script is triggered every time an item is selected. This is implemented as the keystroke event where the keystroke value is equivalent to the user selection. This behavior is also implemented for the combo box—the "keystroke" could be thought to be a paste into the text field of the value selected from the drop down list.

There is a final call to the keystroke script before the validate event is triggered. This call sets the *willCommit* property to *true* for the event. With keystroke processing, it is sometimes useful to make a final check on the field value before it is committed (pre-commit). This allows the script writer to gracefully handle particularly complex formats that can only be partially checked on a keystroke by keystroke basis.

Validate

Regardless of the field type, user interaction with the field may produce a new value for that field. After the user has either clicked outside a field, tabbed to another field, or pressed the enter key, the user is said to have *committed* the new data value.

The *validate* event is the first event generated for a field after the value has been committed so that a JavaScript can verify that the value entered was correct. If the validate event is successful, the next event triggered is the *calculate* event.

Calculate

The *calculate* event causes all fields that have a calculation script attached to them to be executed. All fields that depend on the value of the validated field will now be re-calculated. These fields may in turn generate additional *validate, calculate,* and *format* events.

Calculated fields may have dependencies on other calculated fields whose values must be determined beforehand. The *calculation order array* contains an ordered list of all the fields in a document that have a calculation script attached. When a full calculation is needed, each of the fields in the array is calculated in turn starting with the zeroth index of the array and continuing in sequence to the end of the array.

To change the calculation order of fields, use the *Tools->Forms->JavaScript->Set Calculation Order...* menu item in Adobe Acrobat.

Format

Once all dependent calculations have been performed the *format* event is triggered. This event allows the attached JavaScript to change the way that the data value appears to a user (also known as its presentation or appearance). For example, if a data value is a number and the context in which it should be displayed is currency, the formatting script can add a dollar sign ($) to the front of the value and limit it to two decimal places past the decimal point.

Event Object Properties

change

Type: String *Event: Keystroke, Selection ChangeAccess: R/W*

This property specifies the *change* in value that the user has just typed. The *change* is replaceable such that if the JavaScript wishes to substitute certain characters, it may.

commitKey

4.0

Type: Number *Event: Keystroke, Format* *Access: R*

Use this property to determine how a form field lost focus. Valid values are:

0 - value was not committed (e.g. escape key was pressed).

1 - value was committed because of a click outside the field using the mouse.

2 - value was committed because of hitting the enter key.

3 - value was committed by tabbing to a new field.

For example, to automatically display an alert dialog after a field has been committed add the following to the field's format script:

```
if (event.commitKey != 0)
  app.alert("Thank you for your new field value.");
```

modifier

Type: Boolean *Event: Keystroke, Mouse events Access: R*

This property is a boolean that specifies whether the modifier key is down during a particular event. The modifier key on the Microsoft Windows platform is Control and on the Macintosh platform is Option or Command. The *modifier* property is not supported on UNIX.

rc

Type: Boolean *Event: Keystroke, Validate* *Access: R/W*

This property is used for validation. It indicates whether a particular event in the event chain should succeed. Set *rc* to *false* to prevent a change from occurring or a value from committing. By default *rc* is *true*.

For each event, except the mouse related events, the static event object is populated with the following data. In most events, it is important for JavaScript to set the *rc* (return code) property to indicate that the event can proceed.

selEnd

Type: Integer *Event: Keystroke* *Access: R/W*

This property specifies the ending position of the current text selection during a keystroke event.

selStart

Type: Integer *Event: Keystroke* *Access: R/W*

This property specifies the starting position of the current text selection during a keystroke event.

shift

Type: Boolean *Event: Keystroke, Mouse events Access: R*

This property is a boolean that specifies whether the shift key is down during a particular event.

target

Type: Object *Event: All events* *Access: R*

This property contains the target object that triggered the event. In all mouse events it is the field object that triggered the event. In other events like page open and close it is the document or this Object.

value

Type: Various *Event: Validate, Calculate, Format, SelChangeAccess: R/W*

For the *validate* event, *value* is the value that the field contains when it is committed. The current field value is the *value* property for the field. For example, the following JavaScript verifies that the field value is between zero and 100.

Example:

```
if (event.value < 0 || event.value > 100) {
      app.beep(0);
```

```
        app.alert("Invalid value for field " + event.target.name);
        event.rc = false;
    }
```

For a *calculate* event, JavaScript should set this property. This is the value that the field should take upon completion of the event. For example, the following JavaScript sets the calculated value of the field to the value of the SubTotal field plus tax.

```
var f = this.getField("SubTotal");
event.value = f.value * 1.0725;
```

For a *format* event, JavaScript should set this property. This is the value used when generating the appearance for the field. By default, it contains the value that the user has committed. For example, the following JavaScript formats the field as a currency type of field.

```
event.value = util.printf("$%.2f", event.value);
```

willCommit

Type: Boolean *Event: Keystroke* *Access: R*

Use this property to verify the current keystroke event before the data is committed. This is useful to check the target form field values and for example verify if character data instead of numeric data was entered. JavaScript sets this property to *true* after the last *keystroke* event and before the field is validated.

Example:

```
var value = event.value
if (event.willCommit)
  // Final value checking.
else
  // Keystroke level checking.
```

For more examples of using *willCommit*, refer to the Acrobat Forms JavaScript application (Aform.js) in your Acrobat plug-ins directory.

Field Object

The Field object represents an Acrobat form field (that is, a field created using the Acrobat form tool). In the same manner that an author might want to modify an existing field's properties like the border color or font, the Field object gives the JavaScript user the ability to perform the same modifications.

Field Access from JavaScript

Before a field can be accessed, it must be "bound" to a JavaScript variable through a method provided by the this Object methods interface. More than one variable may be bound to a field by modifying the field's object properties or accessing its methods. This affects all variables bound to that field.

```
var f = doc.getField("Total");
```

This example allows the script to now manipulate the form field *Total* via the variable "*f*".

Field Properties

The following is a chart of field property names used by Acrobat with Javascript and the field properties that are available:

Field Property Name	Type	Text Field	Combo Box	List Box	Push Button	Check Box	Radio Button	Signature	Read Access	Write Access
alignment	String	✓							✓	✓
borderStyle	String	✓	✓	✓	✓	✓	✓	✓	✓	✓
calcOrderIndex	Integer	✓	✓						✓	✓
charLimit	Integer	✓							✓	✓
defaultValue	String	✓	✓	✓		✓	✓		✓	✓
delay	boolean	✓	✓	✓	✓	✓	✓	✓	✓	✓
display	Integer	✓	✓	✓	✓	✓	✓	✓	✓	✓
doc	Object	✓	✓	✓	✓	✓	✓	✓	✓	
editable	Boolean		✓						✓	✓
fillColor	Array	✓	✓	✓	✓	✓	✓	✓	✓	✓
hidden	Boolean	✓	✓	✓	✓	✓	✓	✓	✓	✓

Field Property Name	Type	Text Field	Combo Box	List Box	Push Button	Check Box	Radio Button	Signature	Read Access	Write Access
highlight	String				✓				✓	✓
lineWidth	Integer	✓	✓	✓	✓	✓	✓	✓	✓	✓
lineWidth	Boolean	✓							✓	
name	String	✓	✓	✓	✓	✓	✓	✓	✓	
numItems	Integer		✓	✓					✓	
password	Boolean	✓							✓	
print	Boolean	✓	✓	✓	✓	✓	✓	✓	✓	✓
readonly	Boolean	✓	✓	✓	✓	✓	✓	✓	✓	✓
required	Boolean	✓	✓	✓		✓	✓	✓	✓	✓
strokeColor	Array	✓	✓	✓	✓	✓	✓	✓	✓	✓
style	String					✓	✓		✓	✓
textColor	Array	✓	✓	✓	✓	✓	✓	✓	✓	✓
textColor	String	✓	✓	✓	✓				✓	✓
textSize	Integer	✓	✓	✓	✓	✓	✓		✓	✓
type	String	✓	✓	✓	✓	✓	✓	✓	✓	
userName	String	✓	✓	✓	✓	✓	✓	✓	✓	✓
value	Various	✓	✓	✓		✓	✓	✓	✓	✓

alignment

Type: String *Fields: Text* *Access: R/W*

This property determines how the text is laid out within the text field. Valid alignments include *left*, *center*, and *right*.

```
var f = this.getField("MyText");
f.alignment = "center";
```

borderStyle

Type: String *Fields: All* *Access: R/W*

This property specifies the border style for a field. Valid border styles include *solid*, *dashed*, *beveled*, *inset*, and *underline*. The border style determines how the border for the rectangle is drawn.

- *The **solid** style strokes the entire perimeter of the rectangle with a solid line.*
- *The **dashed** style strokes the perimeter with a dashed line.*
- *The **beveled** style is equivalent to the **solid** style with an additional beveled (pushed-out appearance) border applied inside the solid border.*
- *The **inset** style is equivalent to the **solid** style with an additional inset (pushed-in appearance) border applied inside the solid border.*
- *The **underline** style strokes the bottom portion of the rectangle's perimeter.*

The *border* object is a static convenience constant that defines all the border styles of a field. The following example illustrates how to set the border style of a field to *solid*:

```
var f = this.getField("MyField");
f.borderStyle = border.s; /* border.s evaluates to "solid" */
```

The following chart defines the *borderStyle* property and its associated keywords:

Type	Keyword
solid	border.s
beveled	border.b
dashed	border.d
inset	border.i
underline	border.u

calcOrderIndex

Type: Integer *Fields: Text, Combo Box* *Access: R/W*

Use this property to change the calculation order of fields in the document. When a computable *Text* or *Combo Box* field is added to a document, it is added to the end of the document and the field's name is placed in the *calculation order array*. The *calculation order array* determines the order fields are calculated in the document (see Event Processing for more information about *calculation order arrays*). The *calcOrderIndex* property works similarly to the *Calculate* tab used by the Acrobat Exchange Form tool. Note the following example:

```
var a = this.getField("newItem");
var b = this.getField("oldItem");
a.calcOrderIndex = b.calcOrderIndex + 1;
```

In this example, the this Object method *getField*, gets the *newItem* field that was added after '*oldItem*' field. It then changes the *calcOrderIndex* of the *oldItem* field so that it is calculated before '*newItem*' field.

charLimit

Type: Integer *Fields: Text* *Access: R/W*

This property limits the number of characters that a user can type into a text field.

defaultValue

Type: String *Fields: All but Button* *Access: R/W*

This property exposes the default value of a field. This is the value that the field is set to when the form is reset.

delay

Type: Boolean *Fields: All* *Access: R/W*

This property delays the redrawing of a field's appearance. It is generally used to buffer a series of changes to the properties of the field before requesting that the field regenerate its appearance. Setting the property to *true* forces the field to wait until *delay* is set to *false*. The update of its appearance then takes place, redrawing the field with its latest settings.

```
// Get the MyCheckBox field
var f = this.getField("MyCheckBox");
// set the delay and change the fields properties
// to beveled edge and medium thickness line.
f.delay = true;
f.borderStyle = border.b;
f.strokeWidth = 2;
// force the changes now
f.delay = false;
```

There is a corresponding document level delay flag if changes are being made to many fields at once.

display

4.0

Type: Integer *Fields: All* *Access: R/W*

This property controls whether the field is hidden or visible on screen and in print:

Effect	Keyword
Field is visible on screen and in print	display.visible
Field is hidden on screen and in print	display.hidden
Field is visible on screen but doesn't print	display.noPrint
Field is hidden on screen but prints	display.noView

This property supersedes the older hidden and print properties.

doc

Type: Object *Fields: All* *Access: R*

This property defines the document the field belongs to. Its value is the this Object or the this Object . Please refer to the this Object section for more details.

editable

Type: Boolean *Fields: Combobox* *Access: R/W*

Combo boxes can be editable, that is, the user can type in a selection. This property determines whether the user can type in a selection or must choose one of the provided selections.

```
var f = this.getField("MyComboBox");
f.editable = true;
```

fillColor

Type: Array *Fields: All* *Access: R/W*

This property specifies the background color for a field. The background color is used to fill the field's rectangle. Values are defined by using *transparent*, *gray*, *RGB* or *CMYK* color. Refer to the Global Object section for information on defining color arrays and how values are used with this property.

```
var f = this.getField("MyField");
if (color.equal(f.fillColor, color.red))
```

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

In older versions of this specification, this property was named *bgColor*. The use of *bgColor* is now discouraged but for backwards compatibility is still valid.

hidden

Type: Boolean *Fields: All* *Access: R/W*

This property controls whether the field is hidden or visible to the user. If the value is *false* the field is visible, *true* the field is invisible. The default value for *hidden* is *false*.

```
// Set the field to hidden
var f = this.getField("MyField");
f.hidden = true;
```

See also the display property which supersedes this property in later versions.

highlight

Type: String *Fields: Button* *Access: R/W*

This property defines how a button reacts when a user clicks it. The four highlight modes supported are *none*, *invert*, *push* and *outline*.

- The *none* highlight does not indicate visually that the button has been clicked.
- The *invert* highlight causes the region encompassing the button's rectangle to invert momentarily.
- The *push* highlight displays the down face for the button (if any) momentarily.
- The *outline* highlight causes the border of the rectangle to invert momentarily.

The '*highlight*' object defines all the characteristics that a button can have. Use it in your scripts to access the highlight of choice. The following chart shows the *highlight* object and its associated keywords:

Type	Keyword
none	highlight.n
invert	highlight.i
push	highlight.p
outline	highlight.o

The following example sets the *highlight* property of a button to "*invert*".

```
// set the highlight mode on button to invert
var f = this.getField("MyButton");
f.highlight = highlight.i;
```

lineWidth

Type: Integer *Fields: All* *Access: R/W*

This property specifies the thickness of the border when stroking the perimeter of a field's rectangle. If the stroke color is transparent, this parameter has no effect except in the case of a beveled border. You can set the *lineWidth* property in JavaScript by using the integer values below:

Line Width	Key Value
none	0
thin	1
medium	2
thick	3

For example:

```
// Change the border width of the Text Box to medium thickness
f.lineWidth = 2
```

The default value for *lineWidth* is 1 (*thin)*. Any integer value can be used. However, values beyond 5 may distort the field's appearance.

In older versions of this specification, this property was *borderWidth*. The use of *borderWidth* is now discouraged but for backwards compatibility is still valid.

multiline

Type: Boolean *Fields: Text* *Access: R*

This read-only property determines how the text is wrapped within the field. Text fields can be single line (clip to field boundaries) or multi-line (wrap to field boundaries).

name

Type: String *Fields: All* *Access: R*

This property allows you to access the fully qualified field name of the field as a string object.

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

numItems

Type: Integer *Fields: Combo & Listbox* *Access: R*

The number of items in the combo or list box.

password

Type: Boolean *Fields: Text* *Access: R*

This property causes the field to display asterisks for the data entered into the field. Upon submission, the actual data entered is sent. Fields that have the password attribute set will not have the data in the field saved when the document is saved to disk.

print

L

Type: Boolean *Fields: All* *Access: R/W*

This property determines whether a given field prints or not. Set the *print* property to *true* to allow the field to appear when the user prints the document, set it to *false* to prevent printing. This property can be used to hide control buttons and other fields that are not useful on the printed page.

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

This property has been superseded by the display property and its use is discouraged.

readonly

Type: Boolean *Fields: All* *Access: R/W*

This property sets or gets the read-only characteristic of a field. If a field is *read-only*, the user can see the field but cannot change it.

required

Type: Boolean *Fields: All but Button* *Access: R/W*

This property sets or gets the *required* characteristic of a field. If a field is *required* its value must be non-null when the user clicks a submit button that causes the value of the field to be posted. If the field value is null, the user receives a warning message and the submit does not occur.

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

strokeColor

Type: Array *Fields: All* *Access: R/W*

This property specifies the stroke color for a field which is used to stroke the field's rectangle with a line as large as the line width. Values are defined by using *transparent*, *gray*, *RGB* or *CMYK* color. Refer to the Color Arrays section for information on defining color arrays and how values are used with this property.

In older versions of this specification, this property was *borderColor*. The use of *borderColor* is now discouraged but for backwards compatibility is still valid.

style

Type: String *Fields: Checkbox, Radio Button* *Access: R/W*

This property allows the user to set the *style* of a check box or a radio button, that is, sets the glyph used to indicate that the check box or radio button has been selected. Valid styles include check, cross, diamond, circle, star, and square. The following defines the *style* properties and the associated keywords:

Style	Keyword
check	style.ch
cross	style.cr
diamond	style.di
circle	style.ci
star	style.st
square	style.sq

The following example illustrates the use if this property and the style object:

```
var f = this.getField("MyCheckbox");
f.style = style.ci;
```

textColor

Type: Array *Fields: All* *Access: R/W*

This property determines the foreground color of a field. It represents the text color for *text*, *button*, or *list box* fields and the check color for *check box* or *radio button* fields. Values are defined the same as the fillColor property. Refer to the Color Arrays section for information on defining color arrays and how values are set and used with this property.

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

In older versions of this specification, this property was *fgColor*. The use of *fgColor* is now discouraged but for backwards compatibility is still valid.

textFont

Type: String *Fields: Text, Combo, List & Button* *Access: R/W*

The *textFont* property determines the font that is used when laying out text in a text field, combo box, list box or button. Valid fonts are defined as properties of the "font" object as follows:

Text Font	Keyword
Times-Roman	font.Times
Times-Bold	font.TimesB
Times-Italic	font.TimesI
Times-BoldItalic	font.TimesBI
Helvetica	font.Helv
Helvetica-Bold	font.HelvB
Helvetica-Oblique	font.HelvI
Helvetica-BoldOblique	font.HelvBI
Courier	font.Cour
Courier-Bold	font.CourB
Courier-Oblique	font.CourI
Courier-BoldOblique	font.CourBI
Symbol	font.Symbol
ZapfDingbats	font.ZapfD

The following example illustrates the use of this property and the font object.

```
// set the font of MyField to Helvetica
var f = this.getField("MyField");
f.textFont = font.Helv;
```

textSize

Type: Integer *Fields: All* *Access: R/W*

This property determines the text size in points that is used in all controls. In combo box and radio button fields, the text size determines the size of the check. Valid text sizes include zero and the range from 4 to 144 inclusive. A text size of zero means that the largest point size that can still fit in the field's rectangle should be used (in multi-line text fields and buttons this is always 12 point).

```
// set the text size of MyField to 28 point
var f = this.getField("MyField");
f.textSize = 28;
```

type

Type: String *Fields: All* *Access: R*

This read-only property returns the *type* of the field as a string. Valid *types* that are returned include button, checkbox, combobox, listbox, radiobutton, and signature.

userName

Type: String *Fields: All* *Access: R/W*

This property returns/sets the user name of the field (short description) as a string. The user name is intended to be used as tool tip text whenever the mouse cursor enters a field. It can also be used as a user friendly name when generating error messages instead of the field name (which can sometimes not be suitable for human consumption).

value

Type: Various *Fields: All But Button* *Access: R/W*

This property gets the value of the field data that the user has entered. Depending on the type of the field, the *value* may be a *string*, *date*, or *number*. Typically, the *value* is used to create calculated fields.

```
var oil = this.getField("Oil");
```

```
var filter = this.getField("Filter");
event.value = (oil.value + filter.value) * 1.0725;
```

In this example, the *value* of the field being calculated is set to the sum of the *oil* and *filter* fields and multiplied by the state sales tax. *Value* is perhaps the most important of all the field properties.

Field Methods

buttonImportIcon

Parameter: none
Returns: nothing

This method imports the appearance of a button from another PDF file. It prompts the user to select a PDF file available on the system.

clearItems

Parameters: none
Returns: nothing

This method clears all the values in a *list box* or *combo box*.

```
// Clear the field MyListbox
var f = this.getField("MyListBox");
f.clearItems();
```

deleteItemAt

4.0

Parameters: [nIndex]
Returns: nothing

This function deletes an item in a *combo box* or a *list box*. The parameter *nIndex* is the index of the item in the list to delete (zero-based). If *nIndex* is not specified then the currently selected item is deleted.

```
var a = this.getField("MyListBox");
a.deleteItemAt();
```

getArray

Parameters: None
Returns: an array of fields.

This function returns an array of terminal children fields (i.e. fields that can have a value) for a parent field. This method can be particularly useful for doing field calculations in tables where a parent field value is the sum of all of its children. To understand more about field name and hierarchies, please see the section on Field Properties.

```
// f has 3 children: f.v1, f.v2, f.v3
var f = this.getField("f");
var a = f.getArray();
var v = 0.0;

for (j =0; j < a.length; j++)
    v += a[j].value;
// v contains the sum of all the children of field "f"
```

getItemAt

Parameters: nIndex
Returns: internal value in an item in a *list* or *combo box*

This function gets the internal value of an item in a *combo box* or a *list box*. The parameter *nIndex* is the index of the item in the list to obtain. If *nIndex* is set to -1, it returns the value of the last item in the list. The *getItemAt* function returns the exported or internal value of the item at *nIndex* in the *combo box* or *list box*.

```
// returns value of first item in list l
var a = this.getField("MyListBox");
var v = a.getItemAt(0);
```

insertItemAt

Parameters: cName, cExportValue, [nIndex]
Returns: nothing

This function inserts a new item into a combo box or a list box. *cName* is the index at which to insert the item in a *list box* or *combo box*. *cExportValue* is the string export value of the item i.e. internal value of the item being inserted. *nIndex* is the index in the list to insert the item at. If *nIndex* is 0, *cName* is inserted at the top of the list. If *nIndex* is –1, *cName* is inserted at the end of the list. The default value for *nIndex* is 0.

```
var l = this.getField("l"); /* l is a list box */
var v =l.insertItemAt("sam", "s", 0); /* inserts sam to top of list l */
```

setItems

4.0

Parameters: array

Returns: nothing

This method sets the list of items for a combo box or a list box. The single parameter necessary to call this method must be an array. Each element in the array must either be an object convertible to a string or another array. If the element can be converted to a string, the user and export values for the list item are equal to the string. If the element is an array it must consist of two sub-elements. The first sub-element should be convertible to a string which will be used as the user value, the second element will be used as the export value.

```
var l = this.getField("ListBox");
l.setItems(["One", "Two", "Three"]);

var c = this.getField("StateBox");
c.setItems([["California", "CA"],["Massachusetts", "MA"],["Arizona", "AZ"]]);

var c = this.getField("NumberBox");
c.setItems(["1", 2, 3, ["PI", Math.PI]]);
```

See also the clearItems, getItemAt, and insertItemAt field methods.

Global Object

The Global object is a static JavaScript object that allows you to share data between documents and have data be persistent across sessions. This is referred to as *persistent global data*. Global data-sharing and notification across documents is done through a *publish* and *subscribe* mechanism. This mechanism gives you the ability to monitor global data variables and report their value changes across documents.

Global data can be specified by adding properties to the global object. The property type can be a string, a boolean, or a number. For example, to add a variable called "volume" and to allow all document scripts to have access to this variable, a script would simply define it as:

```
global.volume = 80;
```

To delete a variable or a property from the global object, use the *delete* operator to remove the defined property. For more information on the reserved JavaScript keyword *delete*, please see Netscape's JavaScript Reference Manual. For example, to remove the property "volume" from the global object, call the following script:

```
delete global.volume
```

Global Object Methods

setPersistent

parameters: cVariable, bPersist
Returns: none

This method sets *cVariable* to be persistent. It requires that *bPersist* is set *true*. This means the *cVariable* will exist across invocations of Acrobat Exchange or Reader. If *bPersist* is *false* (the default for any global property) then the property will be accessible across documents but not across the Acrobat Viewer sessions. For example, to make the "volume" property persistent and accessible for other documents you could use:

```
global.setPersistent("volume", true);
```

Note: *Persistent global data only applies to variables of type boolean, number or string. For all persistent data there is a 32k limit for the maximum size of the global persistent variables. Any data added to the string after the 32k limit will be dropped.*

It is recommended that JavaScript developers building scripts for Acrobat Forms, utilize some type of naming convention when specifying persistent global variables. One suggestion is to start all variables with your company name. For example, if your company name is *Xyz,* start

all variables with "*xyz_*". This will prevent collisions with other persistent global variable names throughout the documents.

Note that global variables that are persistent are recorded in the plug-in level *glob.js* file upon application exit and re-loaded at application start. See the section on Plug-in folder level scripts for more details.

this Object

In JavaScript the special keyword “this” refers to the current object. In Acrobat Forms the current object is defined as follows:

- *In an object method, it is the object to which the method belongs.*
- *In a constructor function, it is the object being constructed.*
- *In a function defined at the Plug-in folder level it is undefined. It is recommended that calling functions pass the document object to any function at this level that needs it.*
- *In a Document level script or Field level script it is the document object and therefore can be used to set or get document properties and functions.*

For example, assume that the following function was defined at the Plug-in folder level:

```
function PrintPageNum(doc)
{ /* Print the current page number to the console. */
  console.println(“Page=” + doc.page);
}
```

The following script outputs the current page number to the console (twice) and then prints the page:

```
PrintPageNum(this);                        /* Must pass the document object. */
console.println(“Page=” + this.pageNum);  /* Same as the previous call. */
this.print(false, this.pageNum, this.pageNum); /* Prints the current page. */
```

Variable and Function Name Conflicts

Variables and functions that are defined in scripts are parented off of the *this* object. For example:

```
var f = this.getField(“Hello”);
```

is equivalent to

```
this.f = this.getField(“Hello”);
```

with the exception that the variable “f” can be garbage collected at any time after the script is run.

If a script uses a name that conflicts with a document level property or method and the *this* object is the current document then a run-time error will result: *Invalid or unknown property, set not possible*.

The following script fragment will generate such an error:

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

To avoid such an error, rename your variables and functions to avoid conflict:

```
var pNum = this.pageNum;
```

Util Object

The Util Object is a static JavaScript object that defines a number of utility methods and convenience functions for string and date formatting.

Util Object Methods

printf

Parameters: cFormat
Returns: cResult

This method will format one or more values as a string according to a format string. This is similar to the C function of the same name.

printx

Parameters: cMask, ...
Returns: cResult

This method formats a source string according to a masking string. Valid masking values are as follows:

Value	Effect
?	Copy next character.
X	Copy next alphanumeric character, skipping any others.
A	Copy next alpha character, skipping any others.
9	Copy next numeric character, skipping any others.
*	Copy the rest of the source string from this point on.
\	Escape character.
>	Uppercase translation until further notice.
<	Lowercase translation until further notice.
=	Preserve case until further notice (default).

To format a string as a U.S. telephone number, for example, use the following script:

```
var v = "aaa14159697489zzz";
v = util.printx("9 (999) 999-9999", v);
console.println(v);
```

printd

Parameters: cFormat, date
Returns: cResult

Use this method to format a date according to a format string. Valid string format values are as follows:

String	Effect	Example
mmmm	Long month	September
mmm	Abbreviated month	Sept
mm	Numeric month with leading zero	09
m	Numeric month without leading zero	9
dddd	Long day	Wednesday
ddd	Abbreviated day	Wed
dd	Numeric date with leading zero	03
d	Numeric date without leading zero	3
yyyy	Long year	1997
yy	Abbreviate Year	97
HH	24 hour time with leading zero	09
H	24 hour time without leading zero	9
hh	12 hour time with leading zero	09
h	12 hour time without leading zero	9
MM	minutes with leading zero	08
M	minutes without leading zero	8
ss	seconds with leading zero	05
s	seconds without leading zero	5
tt	am/pm indication	am
t	single digit am/pm indication	a
\	use as an escape character	

To format the current date in long format, for example, you would use the following script:

```
var d = new Date();
console.println(util.printd("mmmm dd, yyyy", d));
```

scand

4.0

Parameters: cFormat, cDate
Returns: date object

This method converts the supplied date, *cDate*, into a JavaScript date object according to rules of the supplied format string, *cFormat*. *cFormat* uses the same syntax as found in printd. This routine is much more flexible than using the date constructor directly.

```
/* Turn the current date into a string. */
var cDate = util.printd("mm/dd/yyyy", new Date());
console.println("Today's date: " + cDate);
/* Parse it back into a date. */
var d = util.scand("mm/dd/yyyy", cDate);
/* Output it in reverse order. */
console.println("Yet again: " + util.printd("yyyy mmm dd", d));
```

Note: *Given a two digit year for input, scand resolves the ambiguity as follows: if the year is less than 50 then it is assumed to be in the 21st century (i.e. add 2000), if it is greater than or equal to 50 then it is in the 20th century (add 1900). This heuristic is often known as the* Date Horizon.

Implementation Considerations

JavaScript Locations and Loading

JavaScripts work with Acrobat Forms on a variety of levels: the *plug-in* level, *document* level, and *field* level. These levels restrict the type of processing that can occur and are loaded at different times.

Plug-in folder level

JavaScripts can work as individual files with the "*.js*" extension. The Acrobat Forms plug-in looks for these files on the UNIX platform in the Acrobat plug-ins folder. On the Apple Macintosh and Windows platforms, the files are in a subfolder of the *plug-ins/Acroform* folder called *JavaScripts*. Variables and functions that might be generally useful to the application should be kept at the plug-in level.

There are some restrictions when writing plug-in level scripts particularly with respect to the this Object.

The standard Acrobat Forms implementation comes with two plug-in level files: *Aform.js*, which contains built-in pre-canned functions and *AFString.js,* which contains the language dependent strings needed by *Aform.js*.

The file *glob.js* is programmatically generated and contains cross-session application preferences set using the global object's setPersistent method.

4.0 If the file *Config.js* is present this file can be used to customize the look of the viewer by removing toolbar buttons and menu items (see the application methods hideMenuItem and hideToolbarButton).

The plug-in level scripts are loaded in the following order: *Config.js* (if it exists), *AFString.js, AForm.js, glob.js* (if it exists), and then all other .js files in no particular order.

Document level

By using the Adobe Acrobat menu item *Tools->Forms->Document Scripts…*, the user can add, modify, or delete document level scripts. These scripts should be function definitions that are generally useful to the document but are not applicable outside the document. Document level scripts are executed after the document has opened and after the plug-in level scripts are loaded. Document level scripts are stored within the PDF document. Therefore, the forms programmer should make every effort to package scripts as effectively as possible.

Field level

The user can add scripts to a form field definition using a dialog box in the form editing tool. (see Event Processing section below). These scripts are executed as the events occur (e.g. mouse up or calculate). Scripts that are field specific should be created at this level. Usually these scripts validate, format, or calculate field values.

Unlike plug-in folder scripts, document level and field level scripts are stored within the PDF document and therefore the forms programmer should make every effort to package his scripts as effectively as possible (e.g. code reuse) at the various levels for performance and file size reasons.

Form Field Hierarchies

Form fields typically have names like FirstName, LastName, etc. This naming convention is referred to as flat names. For many form applications, this flat hierarchy of names is sufficient and works well. The problem with using flat names is that there is no association between the fields.

Form field names can be more useful by creating a hierarchal structure. For example, if we change **FirstName** to **Name.First** and **LastName** to **Name.Last** we form a tree of fields. The period ('.') separator used in Acrobat Forms and denotes a hierarchy shift. The *Name* portion of these fields is the parent, and *First* and *Last* become the children. While there is no limit to the depth a hierarchical name can be constructed it is important that the hierarchy remain manageable.

This hierarchy can be useful in a number of ways. It can speed up authoring and ease manipulation of groups of fields in JavaScript. In addition, a form field hierarchy can improve the performance of forms applications when there are many fields in the document.

Form Field Hierarchies with JavaScript

Using our original flat names *FirstName*, *MiddleName* and *LastName*, imagine that we want to change the background color of these fields to yellow (to indicate missing data, or emphasize an important point). We would need two lines of JavaScript code for each field:

```
var name = this.getField("FirstName");
name.fillColor = color.yellow;
name = this.getField("MiddleName");
name.fillColor = color.yellow;
name = this.getField("LastName");
name.fillColor = color.yellow;
```

With our hierarchy of *Name.First*, *Name.Middle* and *Name.Last* above (and perhaps, *Name.Title* if used), we can change the background color of these fields with just two lines of code instead of six:

```
var name = this.getfield("Name");
name.fillColor = color.yellow
```

When using this hierarchy within a JavaScript, remember you can only change appearance related properties of the parent field and that appearance changes propagate to all children. You cannot change the field's value. For example if you try the following script:

```
var name = this.getField("Name");
name.value = "Lincoln";
```

the script will fail. **Name** is considered a parent field. You can only change the value of terminal fields (i.e. a field that does not have children like **Last** or **First**). The following script executes correctly:

```
var first = this.getField("Name.First");
var last = this.getField("Name.Last");
first.value = "Abraham";
last.value = "Lincoln";
```

Working With The Date Object

This section discusses the use of Date objects within Acrobat. The reader should be familiar with the JavaScript Date object and the Util Object functions that process dates. JavaScript Date objects are actually an object containing both a date and time. All references to date in this section refer to the date-time combination.

Note: *All date manipulations in JavaScript, including those methods that have been documented in this specification are Year 2000 (Y2K) compliant.*

Converting a Date to a String

Acrobat Forms provides several date related methods in addition to the ones provided by the JavaScript Date object. These methods are the preferred method of converting between Date objects and strings. Because of Acrobat Forms' ability to handle dates in many formats, the Date object does not always handle these conversions correctly.

To convert a Date object into a string, the printd method of the Util Object is used. Unlike the built-in conversion of the Date object to a string, printd allows an exact specification of how the date should be formatted.

```
/* Example of util.printd */
var d = new Date(); /* Create a Date object containing the current date. */
/* Create some strings from the Date object with various formats with
** util.printd */
var s1 = util.printd("mm/dd/yy", d);
```

```
var s2 = util.printd("yy/m/d", d);
var s3 = util.printd("mmmm dd, yyyy", d);
var s4 = util.printd("dd-mmm-yyyy", d);
/* print these strings to the console */
console.println("Format mm/dd/yy looks like: " + s1);
console.println("Format yy/m/d looks like: " + s2);
console.println("Format mmmm dd, yyyy looks like: " + s3);
console.println("Format dd-mmm-yyyy looks like: " + s4);
```

The output of this script would look like:

```
Format mm/dd/yy looks like: 01/15/99
Format yy/mm/dd looks like: 99/1/15
Format mmmm dd, yyyy looks like: January 15, 1999
Format dd-mmm-yyyy looks like: 15-Jan-1999
```

Note: *Given the relative closeness of the new millenium and the ever increasing length of the human lifespan, it is advised to output dates with a four digit year to avoid ambiguity.*

Converting a String to a Date

To convert a string into a Date object the scand method of the Util Object, is used. It accepts a format string that it uses as a hint as to the order of the year, month, and day in the input string.

```
/* Example of util.scand */
/* Create some strings containing the same date in differing formats. */
var s1 = "03/12/97";
var s2 = "80/06/01";
var s3 = "December 6, 1948";
var s4 = "Saturday 04/11/76";
var s5 = "Tue. 02/01/30";
var s6 = "Friday, Jan. the 15th, 1999";
/* Convert the strings into Date objects using util.scand */
var d1 = util.scand("mm/dd/yy", s1);
var d2 = util.scand("yy/mm/dd", s2);
var d3 = util.scand("mmmm dd, yyyy", s3);
var d4 = util.scand("mm/dd/yy", s4);
var d5 = util.scand("yy/mm/dd", s5);
var d6 = util.scand("mmmm dd, yyyy", s6);
/* Print the dates to the console using util.printd */
console.println(util.printd("mm/dd/yyyy", d1));
console.println(util.printd("mm/dd/yyyy", d2));
console.println(util.printd("mm/dd/yyyy", d3));
```

```
console.println(util.printd("mm/dd/yyyy", d4));
console.println(util.printd("mm/dd/yyyy", d5));
console.println(util.printd("mm/dd/yyyy", d6));
```

The output of this script would look like:

```
03/12/1997
06/01/1980
12/06/1948
04/11/1976
01/30/2002
01/15/1999
```

Unlike the date constructor (new Date(...)), scand is rather forgiving in terms of the string passed to it.

Note: *Given a two digit year for input,* scand *resolves the ambiguity as follows: if the year is less than 50 then it is assumed to be in the 21st century (i.e. add 2000), if it is greater than or equal to 50 then it is in the 20th century (add 1900). This heuristic is often known as the* Date Horizon.

Date Arithmetic

It is often useful to do arithmetic on dates to determine things like the time interval between two dates or what the date will be several days or weeks in the future. The JavaScript Date object provides several ways to do this. The simplest and possibly most easily understood method is by manipulating dates in terms of their numeric representation. Internally, JavaScript dates are stored as the number of milliseconds (one thousand milliseconds is one whole second) since a fixed date and time. This number can be retrieved through the valueOf method of the Date object. The Date constructor allows the construction of a new date from this number.

```
/* Example of date arithmetic. */
/* Create a Date object with a definite date. */
var d1 = util.scand("mm/dd/yy", "4/11/76");
/* Create a date object containing the current date. */
var d2 = new Date();
/* Number of seconds difference. */
var diff = (d2.valueOf() - d1.valueOf()) / 1000;
/* Print some interesting stuff to the console. */
console.println("It has been " + diff + " seconds since 4/11/1976");
console.println("It has been " + diff / 60 + " minutes since 4/11/1976");
console.println("It has been " + (diff / 60) / 60 + " hours since 4/11/1976");
console.println("It has been " +
```

```
  ((diff / 60) / 60) / 24 + " days since 4/11/1976");
console.println("It has been " +
  (((diff / 60) / 60) / 24) / 365 + " years since 4/11/1976");
```

The output of this script would look something like:

```
It has been 718329600 seconds since 4/11/1976
It has been 11972160 minutes since 4/11/1976
It has been 199536 hours since 4/11/1976
It has been 8314 days since 4/11/1976
It has been 22.7780821917808 years since 4/11/1976
```

The following example shows the addition of dates.

```
/* Example of date arithmetic. */
/* Create a date object containing the current date. */
var d1 = new Date();
/* num contains the numeric representation of the current date. */
var num = d1.valueOf();
/* Add thirteen days to today's date. */
/* 1000 ms / sec; 60 sec / min; 60 min / hour; 24 hours / day; 13 days */
num += 1000 * 60 * 60 * 24 * 13;
/* Create our new date that is 13 days ahead of the current date. */
var d2 = new Date(num);
/* Print out the current date and our new date using util.printd */
console.println("It is currently: " + util.printd("mm/dd/yyyy", d1));
console.println("In 13 days, it will be: " + util.printd("mm/dd/yyyy", d2));
```

The output of this script would look something like:

```
It is currently: 01/15/1999
In 13 days, it will be: 01/28/1999
```

Acrobat Forms and Security

Security in Acrobat takes on the form of restricting access to a document, restricting permissions for a form once it has been opened, and digital signatures.

Restricting Access

If the author desires to restrict access to the form in its entirety then the standard security model in Acrobat can be selected and an open password defined that requires a user to type in a password before opening the form. Other security handlers exist and are provided by third party developers as plug-ins and may also be useful. E.g. using a public/private key

infrastructure to lock a form for a particular set of people or allowing a form to expire after a certain time period.

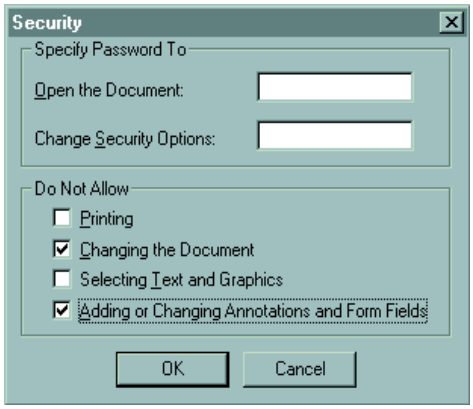
The ability to set a user password is accessed using the Save As function by choosing the appropriate security handler and configuring its settings.

Restricting Permissions

The standard security model in Acrobat is accessible at document save time and allows you to set the following restrictions on the document: printing, changing the document, selecting text and graphics, and adding and changing annotations and form fields.

Once a form has been *authored* it is often useful to lock the form so that it may be filled in but cannot be tampered with using the forms tool. For example, after authoring a form may be posted on a Web site. In order to preserve the form integrity it needs to be shielded from any changes to its formulae or internal data routines.

If the *changing the document* restriction is selected, the user can fill-in form fields and add annotations but cannot author or modify form fields or change the background text using the TouchUp plug-in.



In addition, once a form has been *filled in*, it is often desirable to lock the entire document so that it cannot be changed whatsoever. In filling out a tax or other sensitive form, the user may wish to save the document so that no further changes to the document are allowed. In order to disallow both fill-in and authoring, the *changing the document* **and** *the adding and changing annotations and form fields* restrictions must be selected.

Digital Signatures

Although these form fields do not restrict access or permissions, they do allow an author or user to verify that a document has not been changed after a signature has been applied.

An author may digitally sign a form thus signifying that it has been released for fill-in. A user can verify the signature to make sure that the form has not been tampered with and is thus

"official". A blind signature (signed without any appearance) is often useful in this situation and can be created via the pull right menu in the signatures pane.

After fill-in a user can also sign the document either by using the signing tool or filling in a pre-authored signature field, thus ensuring that the form undergoes no further changes without detection.

Working with Signature Fields

4.0

Signature fields allow the user to digitally sign a document. Once a signature is applied to the document any subsequent changes to the document will cause the signature to indicate that the "Document has been changed after signing".

A signature field's value is read-only. An unsigned signature has a null value. Once the field has been signed its value is non-null.

When crafting a custom script for when the signature field is signed remember to allow for resetting the form (i.e. the field's value is set to null). For example, imagine a form where upon signing a signature field that all fields whose names starts with "A" are locked and all fields whose names start with "B" are locked. We might start with a script fragment, to be executed at signature time, looking something like this:

```
/* This example is incorrect. */
var f = this.getField("A");
/* Lock all A fields. */
f.readOnly = true;
f = this.getField("B");
/* Unlock all B fields. */
f.readOnly = false;
```

This is a typical operation for many forms. This script is **incorrect** and when the form is reset it will lock and unlock the wrong fields. Instead, it should check the value of the signing event and react accordingly:

```
var bLock = (event.value != "");
var f = this.getField("A");
/* Lock A on sign, unlock on reset. */
f.readOnly = bLock;
f = this.getField("B");
/* Unlock B on sign, lock on reset. */
f.readOnly = !bLock;
```

There is a convenience routine available for your use called AFSignatureLock() in AForm.js (see JavaScript Locations and Loading) that performs the programmatic equivalent of the simple locking user interface in the signature properties dialog. This allows you to easily lock and unlock all fields, a particular list of fields, or all fields but those specified. The example is re-coded using this convenience routine below:

```
var bLock = (event.value != "");
AFSignatureLock(this, "THESE", "A", bLock);
```

```
AFSignatureLock(this, "THESE", "B", !bLock);
```

See the comments in AForm.js for more details.