



Adobe

Adobe® Acrobat®



Technical Note # 5186

Acrobat Forms JavaScript Object Specification

Revised: November 3, 1997

© 1997 Adobe Systems Incorporated. All rights reserved.

The information in this document is furnished for informational use only, 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 that may appear in this document. The software described in this document is furnished under license and may only be used or copied in accordance with the terms of such license.

Adobe, the Adobe logo, Acrobat, Acrobat Capture, Acrobat Exchange, and Distiller are trademarks of Adobe Systems Incorporated. Microsoft and Windows are registered trademarks and ActiveX is a trademark of Microsoft in the U.S. and other countries. Macintosh is a trademark of Apple Computer, Inc. registered in the U.S. and other countries. PowerPC is a trademark of International Business Machines Corporation. UNIX is a registered trademark in the U.S. and other countries, licensed exclusively through X/Open Co. Ltd. All other products or name brands are trademarks of their respective holders.


Version History		
03-Nov-1997	Frank Flonnoy	Version 1

Table of Contents

Introduction	7
Other useful documents	7
JavaScript Locations and Loading	8
Plug-in folder level	8
The document level	8
The field level	8
Acrobat Forms JavaScript Object Model	9
Field Object	10
Form Field Hierarchies	10
Form Field Hierarchies with JavaScript	10
Field Access from JavaScript	11
Field Properties	12
alignment	13
bgColor	13
borderColor	14
borderStyle	14
borderWidth	15
calcOrderIndex	15
charLimit	16
defaultValue	16
delay	16
doc	16
editable	17
fgColor	17
hidden	17
highlight	17
multiline	18
name	18
numItems	18
password	19
print	19
readonly	19
required	19
style	19
textFont	20
textSize	21
type	21
userName	21
value	22
Field Methods	22

buttonImportIcon	22
clear	22
getArray	22
getItemAt	23
insertItemAt	23
Event Object	24
Event Processing	24
The “mouse enter” Event	24
The “mouse down” Event	24
The “mouse up” Event	24
The “mouse exit” Event	24
The “keystroke” Event	24
The “selection change” event	25
The “committed” Event	25
The “validate” Event	25
The “calculate” Event	25
The “calculation order array”	25
The “format” Event	25
Event Object Properties	26
target	26
shift	26
modifier	26
change	26
selStart	26
selEnd	26
rc	27
value	27
willCommit	28
Color Arrays	29
Color Objects	29
App Object	31
App Object Properties	31
calculate	31
fullscreen	31
toolbar	31
language	31
viewerType	32
App Object Methods	32
alert	32
beep	33
goBack	34
goForward	34
response	34

Console Object	35
Console Methods	35
show	35
close	35
println	35
clear	35
Global Object	36
Global Object Methods	36
setPersistent	36
Persistent Global Data - Naming Conventions	36
Doc Object	38
Doc Access from JavaScript	38
Doc Object Properties	38
author	38
creator	38
creationDate	38
dirty	39
filesize	39
keywords	39
modDate	39
numPages	39
numTemplates	39
path	40
pageNum	40
producer	40
subject	40
title	40
zoom	41
zoomType	41
Doc Object Methods	41
calculateNow	41
getField	42
getNthTemplate	42
gotoNamedDest	42
print	42
resetForm	43
scroll	43
spawnPageFromTemplate	43
submitForm	44
“this” Object	45
Util Object	46
Util Object Methods	46



printf	46
printx	46
printf	47

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 section titled [Field Object](#) 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>.

JavaScript Locations and Loading

JavaScripts can work with Acrobat Forms on a variety of levels such as; 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. We will discuss how these all interact.

Plug-in folder level

JavaScripts can work as individual files with the “.js” extension. These files are executed, in no particular order, when the JavaScript engine is initialized. The Acrobat Forms plug-in looks for these files on Microsoft Windows and UNIX platforms in the Acrobat plug-ins folder. On the Apple Macintosh the files are in a subfolder of the plug-ins folder called *JavaScript*. Variables and functions that might be generally useful to the application should be kept at the plug-in level. 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 document level

By using the Adobe Acrobat Exchange menu item *Tools->JavaScripts-> 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.

The 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.

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.

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 [Doc 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*”.

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.bgColor = color.yellow;
name = this.getField("MiddleName");
name.bgColor = color.yellow;
```

```
name = this.getField("LastName");
name.bgColor = 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.bgColor = 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";
```

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	Read Access	Write Access
alignment	String	✓						✓	✓
bgColor	Array	✓	✓	✓	✓	✓	✓	✓	✓
borderColor	Array	✓	✓	✓	✓	✓	✓	✓	✓
borderStyle	String	✓	✓	✓	✓	✓	✓	✓	✓
borderWidth	Integer	✓	✓	✓	✓	✓	✓	✓	✓
calcOrderIndex	Integer	✓	✓					✓	✓

Field Property Name	Type	Text Field	Combo Box	List Box	Push Button	Check Box	Radio Button	Read Access	Write Access
charLimit	Integer	✓						✓	✓
defaultValue	String	✓	✓	✓		✓	✓	✓	✓
delay	Boolean	✓	✓	✓	✓	✓	✓	✓	✓
doc	Object	✓	✓	✓	✓	✓	✓	✓	
editable	Boolean		✓					✓	✓
fgColor	Array	✓	✓	✓	✓	✓	✓	✓	✓
hidden	Boolean	✓	✓	✓	✓	✓	✓	✓	✓
highlight	String				✓			✓	✓
multiline	Boolean	✓						✓	
name	String	✓	✓	✓	✓	✓	✓	✓	
numItems	Integer		✓	✓				✓	
password	Boolean	✓						✓	
print	Boolean	✓	✓	✓	✓	✓	✓	✓	✓
readonly	Boolean	✓	✓	✓	✓	✓	✓	✓	✓
required	Boolean	✓	✓	✓		✓	✓	✓	✓
style	String					✓	✓	✓	✓
textFont	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";
```

bgColor

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 [Color Arrays](#) section for information on defining color arrays and how values are used with this property.

```
var f = this.getField("MyField");  
if (f.bgColor == color.red)  
    f.bgColor = color.blue;  
else  
    f.bgColor = color.yellow;
```

borderColor

Type: Array

Fields: All

Access: R/W

This property specifies the border color for a field. The border color is used to stroke the field's rectangle with a line as large as the border 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.

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

borderWidth

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 border color is transparent, this parameter has no effect except in the case of a beveled border. You can set the *borderWidth* property in JavaScript by using the integer values below:

Border Type	Key Value
none	0
thin	1
medium	2
thick	3

For example:

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

The default value for *borderWidth* is “1” (*thin*). Any integer value can be used. However, values beyond “5” distort the field’s appearance. Using this property is the same as using the Acrobat Forms *Field Properties* -> *Appearance* dialog box.

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 [The "calculation order array"](#) section 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 [Doc 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 the update of its appearance 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.borderWidth = 2
// force the changes now
f.delay = false;
```

doc

Type: Object Fields: All Access: R

This property defines the document the field belongs to. Its value is the [Doc Object](#) or the ["this" Object](#). Please refer to the [Doc 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;
```

fgColor

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 [bgColor](#) 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.fgColor = color.red;
```

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 *true* the field is visible, *false* the field is invisible. The default value for *hidden* is *false*.

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

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;
```

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

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;
```

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;
```

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 of this property and the style object:

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

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 *text*, *button*, *combo box*, *list box*, *check box*, and *radio button*.

userName

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

This property returns the user name of the field as a string.

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.

clear

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.clear();
```

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 [Form Field Hierarchies](#).

```
// 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 1
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 */
```

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*, *selection change*, *committed*, *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

The "mouse enter" Event

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.

The "mouse down" Event

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.

The "mouse up" Event

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.

The "mouse exit" Event

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.

The "keystroke" Event

The *keystroke* event occurs whenever a user types a keystroke into a *text box* or *combo box* field. JavaScript may want to limit the type of keys allowed (i.e. a numeric field might only allow numeric characters).

The "selection change" event

The *selection change* event is similar to the *keystroke* event. This is used for *list* and *combo boxes* fields. The value is changed automatically as the item is selected from the *list* or *combo box*.

The "committed" Event

Regardless of field type, the 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

A *validate* event is generated for a field 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.

The "calculate" Event

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.

The "calculation order array"

Another important part of event processing is understanding how fields are calculated in the document. This is accomplished by the *calculation order array*. The *calculation order array* contains a list of all the calculated fields in a document. This array determines the order calculations are performed on individual fields in a particular document. To change the calculation order of a calculated field, use the *Tools->JavaScript->Set Calculation Order...* menu item in Adobe Acrobat. The dialog box will prompt you for the calculation order.

The "format" Event

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

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](#).

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.

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.

change

Type: String *Event: Keystroke, Selection Change* *Access: 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.

selStart

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

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

selEnd

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

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

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.

value

Type: *Various* Event: *Validate, Calculate, Format, SelChange* Access: *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.

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 Objects

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 defines the following colors and there associated keywords:

Color Object	Keyword
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

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.fgColor = color.red;
else
    f.fgColor = color.black;
```

The color object and all its properties are defined in *AForm.js*.

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 from happening. Its default value is *true*.

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.

toolbar

Type: Boolean *Access: R/W*

This property allows a script to show or hide the Acrobat tool bar. It does not hide the tool bar in External windows (i.e. in an Acrobat window within a Web browser). By default *toolbar* is set to *true*. To remove the toolbar set the *toolbar* property to *false*.

Example:

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

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

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.

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.

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.

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.

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.

close

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.

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 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: string *cVariable*, boolean *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.*

Persistent Global Data - Naming Conventions

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.

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 always points to the Doc object of the underlying document or through the [target](#) property of the event object. 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";
```

creator

Type: *String* Access: *R*

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

creationDate

Type: *Date* Access: *R*

This property defines the documents creation date.

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.";
if (!b)
    this.dirty = false;
```

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.

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 get 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 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*. Again 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

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.

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.

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.)

gotoNamedDest

Parameters: cName

Returns: nothing

Use this function 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 1.2](#)

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.page, this.page);
```

resetForm

Parameters: [cFields]

Returns: nothing

Use this method to reset the field values within a document. If the *cFields* parameter is present, then only the fields indicated are reset, otherwise 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] = "Pl.OrderForm.Description";
fields[1] = "Pl.OrderForm.Qty";
this.resetForm(fields);
```

scroll

Parameters: xOrigin: the x-coordinate to scroll to. yOrigin: the y-coordinate to scroll to.

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 1.2](#) for more details on the user space coordinate system.

spawnPageFromTemplate

Parameters: cTemplate, [iPage], [bRename]

Returns: nothing

Use this function with a template name, such as the ones returned by [getNthTemplate](#). The optional parameter *iPage*, 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 *ipage* is omitted, a new page is created at the end of the document. The optional parameter *bRename*, is 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);
}
```

submitForm

Parameters: *cURL*, [*bFDF*], [*bEmpty*], [*fields*]

Returns: *nothing*

Use this parameter 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.

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

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 *fields* parameter is the array of field names to submit. If this parameter is present, then only the fields indicated are submitted, except those excluded by parameter *bEmpty*. If this parameter is omitted, then all fields in the Form are submitted. This is used as a simple shortcut for having a whole subtree submitted. You can include non-terminal fields in the array.

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”.

"this" Object

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

- *In a method, it is the object to which the method belongs.*
- *In a constructor function, it is the object being constructed.*
- *In a top-level script, such as in format, validate, calculate, and perform scripts it is the document object and therefore can be used to set or get doc properties and functions.*

Example:

```
this.page /* returns the documents current page number. */  
this.print(true); /* prints the document interactively. */
```

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));
```