Adobe

Adobe® Acrobat®

Technical Note # 5186

Acrobat JavaScript Object Specification

Version 5.0

Revised: March 1, 2001

Table of Contents

Acrobat JavaScript Object Specification

Introduction

Welcome to Acrobat™ JavaScript

Welcome to the Acrobat 5.0 JavaScript Reference Manual. In the pages to follow, you'll find all the information you need to get started using JavaScript in your PDF documents. With the aid of Acrobat's powerful JavaScript binding, you'll be able to customize forms and other documents in ways that greatly enhance their appearance, utility, and interactivity.

This manual is not only a reference for Acrobat's JavaScript objects, properties and methods, it also includes some instruction on the basics of JavaScript programming and numerous useful examples that illustrate the properties and methods of Acrobat JavaScript as well as relevant programming techniques.

What Is JavaScript?

JavaScript is the powerful, object-oriented scripting language developed by Netscape Communications to enhance web-page interactivity. Originally designed for Netscape's browser software, JavaScript has rapidly evolved to become a widely used, general-purpose programming language. It is now accepted as a standard under ISO-16262 of the International Standards Organization. (The first industry-standard version of the language, endorsed by the European Computer Manufacturers Association, was known as ECMAScript.) The core language has undergone several revisions, the most current being version 1.5, the one used by Acrobat 5.0.

What Is Acrobat™ JavaScript?

Core JavaScript has been extended in various implementations to meet various needs. *Client-side* (browser based) JavaScript adds objects and methods to manipulate browser windows and their contents. *Server-side* JavaScript adds File and Database objects to deal with database queries and other typical server-side actions. Acrobat™ JavaScript extends the language by adding objects and methods designed to allow easy access to (and manipulation of) PDF document contents. The various extensions to the language that make this possible are described in detail in this manual.

The same characteristics that make browser-based JavaScript easy to work with—relaxed data typing, C-like syntax, built-in Math and String classes—are applicable to Acrobat™

JavaScript as well. In fact, Acrobat™ JavaScript is built *on top of* core JavaScript. All of the core-language features that you might be accustomed to working with in browser-based, JavaScript are available in Acrobat™ JavaScript, including Math, String, Date, Array, and RegExp (regular expression) classes.

Space considerations preclude a detailed discussion of core-JavaScript features here. This manual assumes that you have had some exposure to JavaScript as it is typically used in a browser environment. (For more information on the core language, you should visit http://developer.netscape.com/library/documentation/javascript.html on the Web, or consult the computer programming section of any good bookstore.)

If you already have some JavaScript programming experience, you will still find this manual helpful for making the transition from browser-based JavaScript to Adobe Acrobat JavaScript. You will learn how to access information about the PDF document, including properties like zoom factor and file size, as well as information about the runtime environment, such as the host computer platform and whether a document is being viewed in Acrobat, or Acrobat Reader. In addition, you will learn how to use built-in methods to access the field information in forms and modify form contents (including appearance attributes) at runtime.

Document Conventions

Tips

From time to time, you will see tips offered in boxes, like this:

Tip: *This is where you'll find a good idea or two.*

The comments offered in these boxes are optional—you can skip over them, if you want. Nevertheless, they are designed to give you practical "real-world" solutions to problems or situations that you are likely to encounter in your coding.

Quick Bars

There are symbols that are shown at the beginning of most every property or method in a "quick" bar for quick interpretation.

Example:

5.0	💾	🔑	⊗

#.# The first column is a number that indicates indicates in which version of the software a property or method became available. As this document pertains to Acrobat version 5.0, there exist some compatibility issues with older versions of the software. If the number is specified, then the property or method is available only in versions of the Acrobat software greater than or equal to that number. Before accessing this property or method, the script should check that the forms version is greater than or equal to that number if backwards compatibility is desired.

Example:

```
      if (typeof app.formsVersion != "undefined" && app.formsVersion >= 5.0) {
        // Perform version specific operations.
      }
```

If the first column is blank, then no compatibility checking is necessary.

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 ☹ in the version column.

The second column, if not blank, indicates that using this method or property will modify the PDF document. If the document is subsequently saved, the effects of this method are saved as well.

The second column can also contain the preferences symbol that indicates that even though this property does not change the document, it can permanently change a user's application preferences.

The third column, if not blank, indicates that this method or property may only be available during certain events for security reasons (e.g. batch event, application start, or execution within the console). See the discussion in the section on the Event Object for more details of the various Acrobat events.

⊗ The fourth column, if not blank, indicates that this method or property is not available in the Acrobat Reader.

If You Need Help

The Web offers a great many resources to help you with JavaScript in general as well as JavaScript for PDF. For example:

- http://www.adobe.com/support/forums/main.html—Adobe Systems, Inc. provides dedicated online support forums for all Adobe products, including Acrobat and Acrobat Reader.

- http://www.adobe.com/support/database.html—In addition to the forums, Adobe maintains a searchable support database with answers to commonly asked questions.

- http://www.pdfzone.com/resources/lists.html—The PDFZone website operates discussion forums on PDF forms, PDF development, and Acrobat usage, among other topics. Experts from around the world participate in these forums to help answer users' questions.

- http://forum.planetpdf.com/—This popular area of the PlanetPDF website contains discussion forums for beginners, developers, forms integrators, and more. As with the PDFZone lists, experts from around the world participate in the Planet PDF Forum (formerly known as AcroBuddies).

- PlanetPDF (http://www.planetpdf.com) has a section specifically devoted to JavaScript examples, called Planet PDF Developers.

The ultimate authority on the core JavaScript language continues to be its originator, Netscape Communications. See http://developer.netscape.com/library/documentation/javascript.html for a list of current documentation devoted to JavaScript.

Useful Documents

Core JavaScript 1.4 Documentation

Complete documentation for JavaScript 1.5, the version used by Acrobat 5.0, is available on the web at http://developer.netscape.com/docs/manuals/javascript.html.

Core JavaScript Guide, Part I. Core Language Features, Netscape Communications Corporation. Part I of this document gives a conceptual overview of the core JavaScript language. Available in both html and PDF format. http://developer.netscape.com/docs/manuals/js/core/jsguide15/contents.html. Note: The rest of the document concerns Netscape's extensions to core JavaScript and are not applicable in the Acrobat environment.

Core JavaScript Reference, Part I. Object Reference and Part II. Language Elements, Netscape Communications Corporation. Parts I and II are a reference to the core JavaScript language. Available in both html and PDF format. http://developer.netscape.com/docs/manuals/js/core/jsguide15/contents.html. Note: The rest of the document concerns Netscape's extensions to core JavaScript and are not applicable in the Acrobat environment.

Adobe Web Documentation

PDF Reference, second edition, Adobe Portable Document Format, Version 1.3, Adobe Systems Incorporated, published by Addison-Wesley, July 2000. This document is available in bookstores, and in electronic form at http://partners.adobe.com/asn/developer/acrosdk/docs/PDFRef.pdf.

Technical Note #5190, Acrobat Viewer plug-in API Overview. Gives an overview of the objects and methods provided by the plug-in API of the Acrobat viewer. This document is available with the Adobe Acrobat Plug-ins SDK CD-ROM or on the Adobe Web site http://partners.adobe.com/asn/developer/acrosdk/docs/apiovr.pdf.

Technical Note #5167, Acrobat Viewer plug-in API Development. Describes 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://partners.adobe.com/asn/developer/acrosdk/docs/apidev.pdf.

Technical Note #5191, 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://partners.adobe.com/asn/developer/acrosdk/docs/apiref.pdf.

Adobe CD Documentation

There are a variety of documents and samples on the Acrobat CD that discuss using JavaScript in a variety of ways.

Batch Sequences: an indepth discussion of batch sequences and how to use JavaScript to solve common problems.

Getting Started with ADBC: a primer for accessing database information through the Acrobat Database Connectivity object including a sample mail merge batch sequence.

Tutorial on form field authoring: discusses creating and manipulating form fields with a step-by-step introduction to authoring an invoice.

Forms System Implementation Notes: this document discusses interfacing Acrobat and web servers for form submission.

What's In A Name: this document talks about how the names for fields can make a big difference in form authoring and processing.

Getting Started with Acrobat™ JavaScript

JavaScript programming opens up exciting new vistas for authors of PDF documents. With relatively little effort, you'll find that you can use JavaScript to enhance PDF documents in countless ways. For example, you can use JavaScript to:

- *Execute menu commands (in Acrobat or Reader) programmatically.*
- *Issue system beeps and alerts.*
- *Format form input as the user is entering it.*
- *Bind field dependencies in a form.*
- *Change the appearance of form fields (including background colors and text fonts) in response to user actions.*
- *Cause e-mail to be sent automatically if the user clicks a link or button.*
- *Query a user for input via a "response" dialog, from a regular PDF document or inside a form.*
- *And much more.*

In this section, we'll explore the basics of the JavaScript language and begin to look at some of the ways in which the core language has been expanded to accommodate PDF-related features and functionality. Along the way, we'll provide a few tips for making your code more reliable and easier to maintain. If you are new to programming, don't expect to understand every single point. It takes a while to get used to programming conventions, and during the learning process you're bound to make mistakes. But with relatively little effort, you should be able to craft your own basic JavaScripts (based on the working code examples shown here) in no time.

Where Can You Use JavaScript?

JavaScripts are either part of the PDF document itself, or exterior to the document.

JavaScripts within the Document

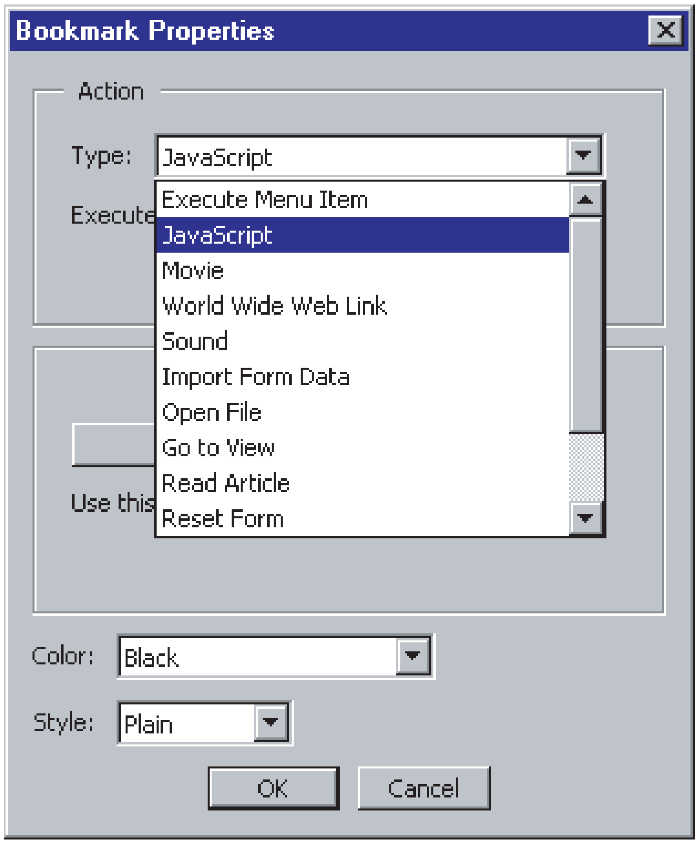
There are six ways in which you can attach JavaScript to PDF documents:

1. You can add JavaScript to **Page Open** or **Page Closed** actions. (In Acrobat, use **Document->Set Page Action...**) Scripts added here will execute each time a page opens (or closes), including the first time a document opens or immediately before a document closes.

2. You can add JavaScripts to **Document Actions**. (In Acrobat, use **Tools-> JavaScript-> Set Document Actions...**). Scripts can be added for several types of document related actions like **Document Will Close** (when a document closes), **Document Will Save** (right before a document is

saved), **Document Did Save** (right after a document is saved), **Document Will Print** (right before a document is printed), **Document Did Print** (right after a document is printed).

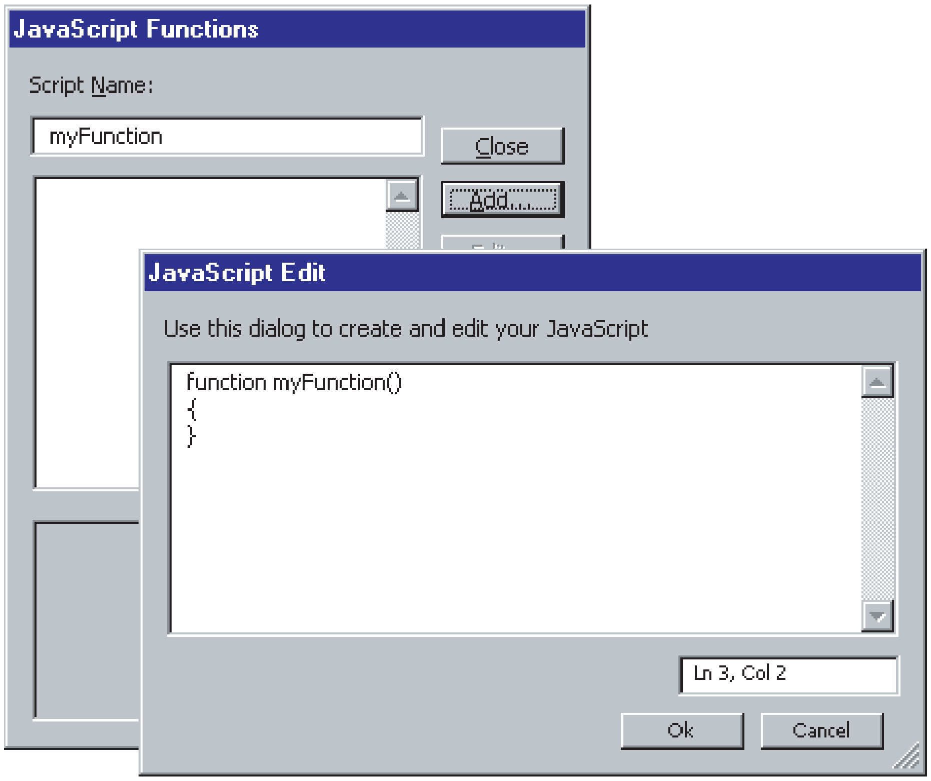
3. You can add JavaScript to Bookmarks. In Acrobat, after creating a Bookmark with Control-B (or Command-B), bring up the Bookmark Properties dialog using Control-I (or Command-I). Be sure a Bookmark is selected first. When the Bookmark Properties dialog appears, choose JavaScript from the Action popup menu, as shown below:



After choosing JavaScript, click the **Edit…** button to add a script to the Bookmark. The script you add here will execute every time the Bookmark is clicked.

4. You can also add JavaScript to a Link created by means of Acrobat's Link Tool. Either select an existing link, or enable the Link tool (by pressing the 'L' key on your keyboard) and drag out the outline of a new link somewhere on the page. When you've created a new link, the Create Link/Link Properties dialog will appear. (If you've selected a preexisting link, you can make the Link Properties dialog appear with Control-I/Command-I.) In the Action Type popup, select JavaScript and hit the **Edit…** button to add or edit your script.

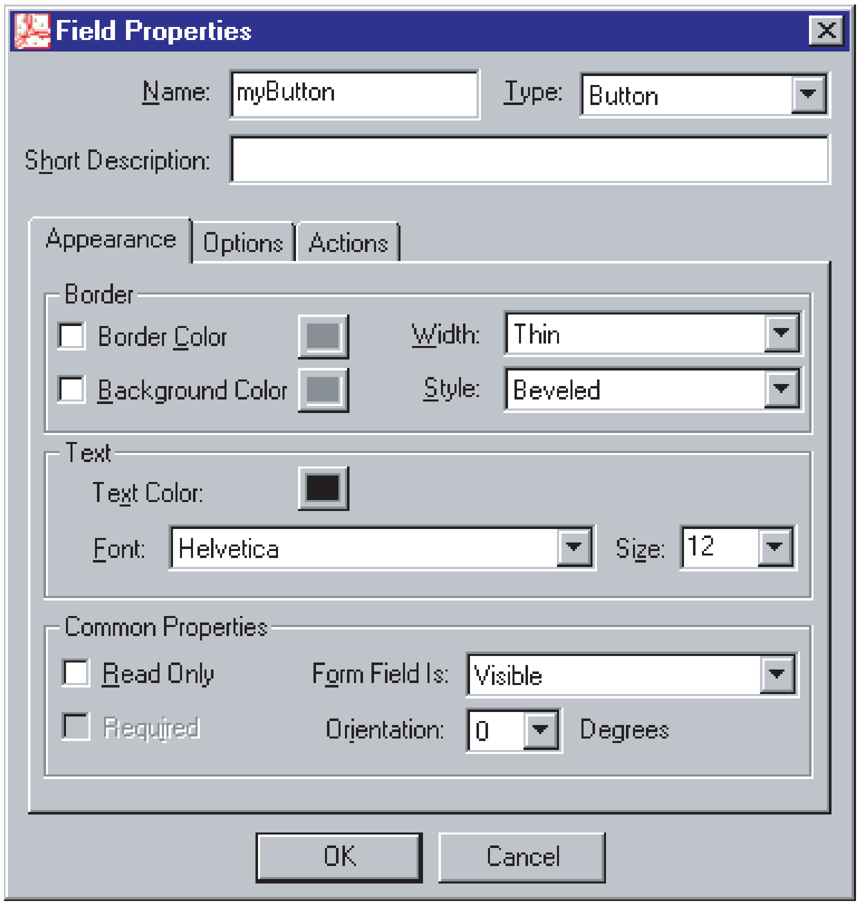
5. Document-level (or so-called "top level") JavaScripts can be added in Acrobat by using **Tools->JavaScript->Document JavaScripts...** When the dialog appears, you can add your own custom JavaScript functions here, as shown below:



```
function myFunction()
{
}
```

Any code you add at this level can be "seen" (that is, called or used) from any other JavaS-cripts in the document, which is why this is often called "top-level" code.

6. Finally, you can add JavaScript to individual form fields created using the Form Tool. Either doubleclick an existing field, or use the Form Tool (type 'F' on the keyboard to enable it) and drag out the outline of a new field, to bring up the Field Properties dialog:


Field Properties

Name: myButton
Type: Button
Short Description:

Border
Border Color ☐
Background Color ☐
Width: Thin
Style: Beveled

Text
Text Color: ■
Font: Helvetica
Size: 12

Common Properties
Read Only ☐
Required ☐
Form Field Is: Visible
Orientation: 0 Degrees

Within this dialog, there are several ways to add JavaScripts to a field. The Actions tab exposes a pane (as shown above) in which you can add JavaScripts via the **Add…** button. Note that when adding JavaScripts to Button fields (the case shown here), you can attach separate scripts to **Mouse Up, Mouse Down, Mouse Enter,** and **Mouse Exit** button events, as well as **On Focus** and **On Blur.** You can also add *more than one script to one event* (they are executed sequentially in the order you add them.) Other field types expose other areas where JavaScripts can be added. For example, Text fields not only have mouse-event actions but can accept JavaScripts as Format, Validate, and/or Calculation scripts. (Appropriate panes exist for these options when "Text" is selected in the Type popup in the Field Properties dialog.)

JavaScripts external to the Document

In addition to JavaScripts that are saved with the document, JavaScripts external to the document can be executed as well. There are three other ways external JavaScripts can be executed.

1. Folder Level JavaScripts: JavaScripts can be saved in a file, with file extension of *.js*, and placed in one of the two JavaScript folders (the application and user folders). When the Acrobat application starts up, the application reads these folders and executes any JavaScript files found.

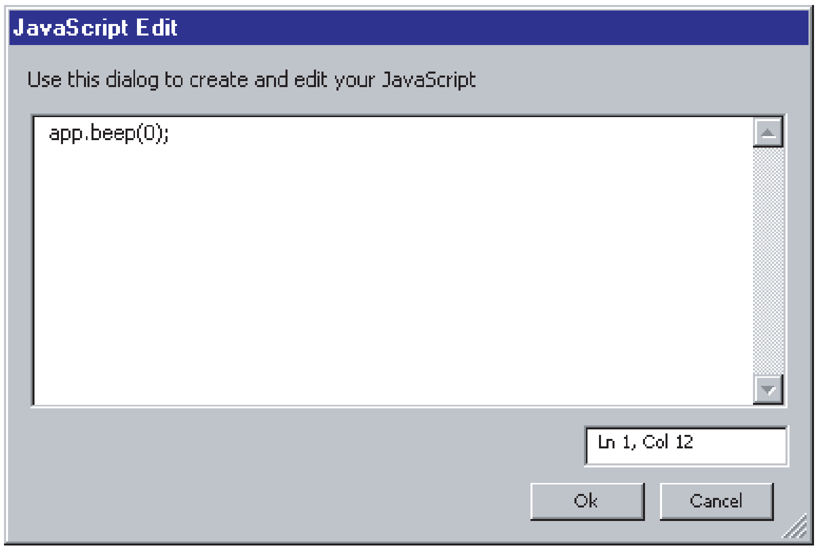
2. Console JavaScripts: JavaScripts can be typed into the Acrobat console and executed. See the Console Object for a discussion of this feature. This is mostly a testing feature of Acrobat.

3. Batch JavaScripts, also called batch sequences. Beginning with Acrobat 5.0, a powerful batch system for the execution of JavaScript for each file that has been selected to be processed. See the document "Batch Sequences: Tips, Tricks and Examples" available one the Acrobat CD for more details.

A Quick Example

One of the most common uses for JavaScript is to enable a calculation or other action in conjunction with a mouse or keyboard event associated with a form field. For example, you may want a script to execute whenever a user presses a button on a form. To set this up, you would click the Actions tab inside the Form Properties dialog (as shown above), then Add a JavaScript to a mouse event—most likely a mouse-up event.

When adding JavaScript to a form field, you will see the following kind of dialog box for doing script editing:



```
app.beep(0);
```

Whatever you type in this box will be executed when the appropriate event occurs in conjunction with the field that the script is attached to. In this example, we have attached a small script (with one line of JavaScript code) to a Button field's mouse-up event. The code issues a standard system beep, using the beep method of the App Object class. The net result is that whenever the user presses the button in the form, the mouse-up event occurring inside the button will generate a system beep. (This is obviously a rather trivial example, but it shows how to attach custom JavaScript code to mouse events in form fields.)

Tip: *It is considered a good user interface design practice to associate script actions with mouse-up (rather than mouse-down) events. The reason most scripts should execute only after the user lets his finger off the mouse button is that until then, the user may wish to reconsider what he is doing and move the mouse outside the button (cancelling the action before it happens). This is the default button behavior in most commercial software applications.*

Core Language Features

Before we discuss features of Acrobat™ JavaScript that apply specifically to PDF documents, let's take a moment to review some of the fundamental concepts on which the JavaScript language is based.

Data Types

JavaScript supports primitive (core) data types—such as *Booleans*, *numbers*, and *strings*—as well as composite data types, like *objects* and *functions*. You'll be working with primitive data types a lot. Consequently, it's important that you understand what they mean and how JavaScript interprets them.

Booleans are the simplest data type, since they can have just two values: *true* and *false*.

Tip: *Internally, JavaScript represents* true *and* false *values as 1 and 0, respectively. But anything that has a non-zero value will be evaluated by the JavaScript interpreter as true in a Boolean context.*

Numbers are represented using the standard scientific notation, for example, 3.14, 2000, or 6.023e+23. Unlike some other languages, JavaScript does not distinguish between integer or floating-point values: In JavaScript, all numbers (regardless of how you write them) are represented internally as 64-bit IEEE floating-point values. In base-ten terms, you have about 20 digits of precision in which to work, which ought to be fine for most applications. (If you can't achieve the precision you're looking for in JavaScript, chances are you won't easily find it in other languages, either.)

Note that you should not write integer values with preceding zeros in JavaScript, since JavaScript interprets '021' as the octal representation of the base-ten number 17. (In addition to octal triplets, you can write numbers in hexadecimal using the '0x' prefix: e.g., 0xFF represents a value of 255.)

A *string* is a sequence of letters, digits, punctuation, or other characters enclosed in quotation marks. Single quotations or double quotations can be used; it doesn't matter. (If your string happens to contain double-quotes as part of the desired string sequence, you should enclose the entire string in single quotes. Likewise, if your string happens to contain single-quotes as part of the string sequence, you should enclose the entire string in *double* quotes.) The following are all legal string values:

```
'This will work.'
"3.14"
'The password is "zxcv"'
```

JavaScript recognizes a number of *escape sequences* for representing characters inside strings that would otherwise be impossible to represent. The following table summarizes those escape sequences.

Table 1 *JavaScript Escape Sequences*

Sequence	Character
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Tab
\’	Single quote
\”	Double quote
\\	Backslash
\uXXXX	Unicode character specified by four hexadecimal digits

Arrays are collections of data values. In JavaScript, the members of an array can be any primitive data type, and the same array can (if you want) hold *different* data types. As in the C language, members of an array are specified or accessed by numeric indexes enclosed in square brackets. Thus, to talk about the value stored in the first element of an array called *names*, you would refer to *names[0]*. The second element of the array would be *names[1]*; the third element, *names[2]*, and so on.

Tip: *Bear in mind that core JavaScript has built-in classes, with many helpful (and powerful) utility methods, for Strings and Arrays, not to mention additional core-language classes such as Math, Date, and RegExp. All of these classes (and their associated methods) are available to you in Acrobat™ JavaScript. (For information on these classes and methods, consult any good book on JavaScript.)*

Variables

JavaScript allows you to declare your own variables using the 'var' keyword. Some things you should know about variables in JavaScript:

- *All identifiers in JavaScript are case-sensitive, which means that a variable named 'MyVariable' is not the same as one named 'myVariable'.*
- *Variables created in JavaScript are permanent, within their scope. Once you declare a variable, there is no way to "undeclare" or destroy it. (JavaScript's garbage-collection mechanism will automatically deallocate variables when you're through with them.)*
- *All variables in JavaScript have a scope, which determines the variable's lifetime and accessibility (i.e., whether it is usable at a given time). Variables declared inside functions are said to have local scope, which means they can be used only inside the function in which they were declared.*

To declare a variable, simply type:

```
var circumference;
```

Or, you can declare a variable and initialize it in one statement:

```
var circumference = 6.28;
```

After declaring a variable, you no longer need to preface it with the 'var' keyword. You can simply use it in expressions as you would use any primitive data type.

Tip: *It is legal, in JavaScript, to declare a variable without using the 'var' keyword. But since (as we mentioned) all JavaScript variables must have a scope, this leaves the interpreter in a bit of a quandary as to how to "scope" a non-'var' variable. The interpreter resolves this problem by attaching the unscoped variable to the global object (the "mother of all runtime objects"), which has the effect of making the variable in question "usable" from all points in a program. This may or may not be what you want. Usually, it is not.*

Unlike most other “high level languages”, JavaScript is rather relaxed when it comes to variable typing. This means that you, as a programmer, don’t have to worry about telling JavaScript whether a given variable should behave like a number or a string. JavaScript will take its “best guess” and make a variable behave as a number in numeric contexts and as a string in string contexts.

JavaScript is an object-oriented language, yet unlike Java or C++ requires no prototyping of objects. To create a new object, you can simply type:

```
var hue = new Object();
```

This statement calls JavaScript's object constructor function which creates a generic, propertyless object. Once an object has been created, you can then assign fields (which can be *properties* or *methods*, depending on whether they refer to primitive data types or functions) to the object as follows:

```
hue.favorite = "taupe";
hue.worst = "teal";
hue.maximum = 15;
hue.sum = function (a,b) { return a + b; }
```

All four statements are legal. The first three statements create properties and assign values to them. (In one case, the value is numeric; in the other two cases, the values are strings.) The fourth statement creates a new *method* for the object by assigning a function to it. The method can be called and used just like any other function:

```
var x = hue.sum( 2, 2 );  // 'x' now has the value '4'
```

In some ways, arrays and objects are alike, because both are collections of arbitrary data types. With arrays, the component pieces are accessed using numeric indexes and square brackets. With objects, the components are accessed through identifiers, using a period between the object name and the component name.

Undefined Variables

In JavaScript, as in other languages, if you attempt to use a variable in an expression without declaring it first, you will generate a runtime error. Odds are, the console window (in Acrobat Business Tools or full Acrobat) or a message box (in Acrobat Reader) will pop up, telling you (or your user!) that “such-and-so variable is not defined”. Consider the following code:

```
function memberSum() {
  var member = new Object();              // create a new, blank object
  member.john = 1;                        // create a new property, 'john'
  return (totalmembers + member.john);    // error!
}
```

In this scenario, *totalmembers* is a new variable that was not declared anywhere. Using it inside the *return* expression generates a runtime error. The line of code that creates *member.john and* assigns a value of '1' to it is not an error, since in JavaScript, you can create new properties for your objects on-the-fly like this. (See discussion under "Objects," above.)

Earlier, we said that JavaScript has relaxed typing rules. This is not the same as saying that it is an untyped language. JavaScript does have data types; in fact, it is this very property that allows you to check a variable at runtime to see if it is defined. To test a variable to see if it is defined use the *typeof* operator:

```
if (typeof aVariable == "undefined") // undefined variable?
  app.alert("Undefined variable.");
else {
  // the variable is safe to use
}
```

Here, we use the App Object's predefined alert method to put an alert dialog onscreen if the variable being tested is not defined. Notice that JavaScript uses the C-like `==` operator to test for equality. Also note that strings are compared *by value*; hence two strings can be compared directly using the `==` operator.

Comments

You can insert comments in your JavaScript code using either the C++ or C-style commenting protocol. That is to say, any text between a double-slash (//) and the end of a line will be ignored by JavaScript. Also, any text between `/*` and `*/` will be treated as a non-executable item (or comment). The following are valid comment styles:

```
// This is a single-line comment.
/* This is also a comment. */    // as is this
/*
 *
 * You can, if you want, also do this.
 *
 */
```

Comments are extremely important tools because they make programs much more readable and easier to maintain. JavaScript uses a succinct, C-like syntax, which means that you can accomplish a lot in just a few lines of code. Unfortunately, that also means you can very easily wind up with programs that are compact, but nearly indecipherable and hard to debug. As a result, you should take extra care to “comment your code” liberally as you write it. The net result will be code that's more reliable and more likely to make sense when you sit down six weeks from now to look at it again.

Punctuation

Strictly speaking, you do not have to put semicolons at the end of JavaScript statements (as you do with C or Java statements). Statements without semicolons will execute normally. However, it is good practice to insert semicolons wherever appropriate. Likewise, you can omit parentheses from certain constructor functions, such as:

```
var n = new Object;
```

But this, too, is not good form. Most functions require parentheses; using them in every case promotes consistency and readability.

Parameter Specification for Methods

5.0			

The parameter (or argument) list of a Acrobat JavaScript method can be specified in two ways:

1. Use the standard core JavaScript technique of listing the parameters separated by commas. Optional parameters may be specified, skipped over using an empty string, or truncated from the list. The parameters must be listed in the correct order. For example,

```
app.alert("Hello World!", 1, 1);
app.alert("Hello World!","",1); // Take default for second parameter
app.response("How are you today?");
app.response("How are you today?","", "Fine"); // Specify cDefault only
```

2. Use an *object literal* to pass argument information to the method. An object literal is a "list of zero or more pairs of property names and associated values of an object, enclosed in curly braces ({})"[1]. The following code lines repeat the examples above:

```
app.alert( {cMsg: "Hello World!", nIcon: 1, nType: 1} );
app.alert( {cMsg: "Hello World!", nType: 1} );
app.response( {cQuestion: "How are you today?"} );
app.response( {cQuestion: "How are you today?", cDefault: "Fine"} );
```

The property names that appear in the object literal are the parameter names given in the specification of the method; for example, the app.response method begins this way:

response

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

There is one required parameter, and three optional ones. The names of the parameters are the ones used in the creating the object literal as the argument of the App.response, see the examples above. (There is no significance to the name of the return value, *cResponse*, other than to suggest that the method returns a string.)

Note: *The object literal method of specifying parameters may not available be for all Acrobat JavaScript functions.*

---

[1]."Core JavaScript Guide", Version 1.4, Netscape Communications Corporation, October 30, 1998, page 31.

Quick Help for Methods

5.0			

As an added convenience, if you give any Acrobat method an argument of *acrohelp* and execute that method (for example, in the console editor), the method will return a list of its arguments names and types.

Example: In the console, type

```
app.response(acrohelp);
```

While the cursor is still on the line you entered, press either Ctrl-Enter or the Enter key on the numeric key pad. The output to the console is seen to be:

```
uncaught exception: Console:Exec:1: Help for App.response
====> [cQuestion: string]
====> [cTitle: string]
====> [cDefault: string]
====> [bPassword: boolean]
```

Optional arguments are listed in square brackets; required parameters have no brackets.

Dealing With Exceptions

5.0			

Acrobat JavaScript methods will throw an exception on error. The script writer can better control these exceptions by using the *try/catch/finally* (JavaScript 1.4), possibly in conjunction with the *throw* statement. For example, try to connect to a database using the ADBC plugin.

```
function getConnected()
{
    try  {
          ConnectADBCdemo = ADBC.newConnection("ADBCdemo");
          if (ConnectADBCdemo == null) throw "Could not connect";
          statement = ConnectADBCdemo.newStatement();
          if (statement == null) throw "Could not execute newStatement";
          if (statement.execute("Select * from ClientData"))
              throw "Could not execute the requested SQL";
          statement.nextRow() // Statement.nextRow() throws an exception...
          return true;        // if there is no next row
      } catch(e) {
          app.alert(e);
          return false;
    }
```

```
        }
```

Developers and script writer can gain information about the structure of the exception object by executing the following script:

```
        try {
            app.alert();    // force an exception, one argument required for alert
        } catch(e) {
            for ( var i in e )
                console.println( i + ": " + e[i])
            // now display the error message, which is e.toString()
            console.println("e: " + e );
        }
```

Error handlers can be written to deal with exceptions thrown by one of the Acrobat plugins and by custom written JavaScript code, such as the first (very simple) example above.

A very common case for throwing an exception is if the underlying object no longer exists (i.e. it is dead but a reference to the object still exists). Consider the following piece of code:

```
        var myDoc = app.newDoc();
        myDoc.close();
        myDoc.pageNum = 3;
```

This will throw an exception when the third line is executed. The document has been closed and no longer exists. A reference to this document is still being held in *myDoc* and any attempt to use it will throw an exception:

```
        uncaught exception:Console :Exec:1: Doc.pageNum object is dead.
```

Editing JavaScripts in Acrobat 5.0

5.0			⊗

Editing all JavaScripts in a Document

The user can now view and edit all JavaScripts in a PDF document via the Acrobat menu item *Edit All JavaScripts* (**Tools-> JavaScript -> Edit All JavaScripts...**). All the scripts will be displayed and organized by XML tags. You'll notice that you can easily identify one script among many via the XML tags content. It's very important not to edit the XML tags added for script identification purposes by Acrobat. Your changes may not be ccepted in case Acrobat finds they have been edited. The same feature can be accessed via buttons strategically located in some dialogs (e.g.: the Actions tab in the Form Fields Properties dialog).

External Editor

The user can now specify an External Editor program to edit JavaScripts in Acrobat. (**Edit-> Preferences-> General-> JavaScript**). You can choose and specify an external editor program which will be used any time a JavaScript has to be edited from inside Acrobat. Acrobat will generate a temporary file and open it in the external editor program. The user has to save the file in order for his changes to be accepted by Acrobat. Acrobat will be inaccessible to the user while the file is being edit in the external editor program. Close the editor to regain access to Acrobat.

Note: *This feature is not available on the Macintosh.*

Tabbing within the Internal Editor

Pressing the Tab key will insert four spaces at the insertion point. Pressing Shift+Tab will move the cursor four (or less) spaces to the left.

Highlighting a line, or a portion of a line, and then pressing Tab (or, Shift+Tab) will move the whole line to the right (respectively, to the left) four spaces (or less, in the case of Shift+Tab). Blocks of lines can be tabbed in the same way by first highlighting the lines then pressing either Tab or Shift+Tab.

Interactive JavaScript Console

5.0			⊗

The Acrobat JavaScript console is now editable and interactive. Any JavaScript code that you write to the console can now be immediately evaluated. The alternate <Enter> key on the Numeric pad, or, for Win and Mac users, the Ctrl+Return key combination, is used for that purpose. The descriptions for executing JavaScript and for tabbing within the console given below are valid for any of the internal JavaScript editors: the console, the document level JavaScript editor, the form action JavaScript editor, and so on.

Executing JavaScript

There are two basic ways of evaluating JavaScript code on the console. To evaluate a block of code, just type in the code, highlight it and hit the <Enter> key (Ctrl+Return). To evaluate a line of code, just locate the cursor on the line you want evaluated and hit the <Enter> key (Ctrl+Return) to obtain the results.

Any immediate results of the code evaluated will be printed to the console. Be aware that when evaluating a block of code, only the result from the last JavaScript expression in the block will be printed to the console. Also, the result of an expression is not the same as the value of a variable set in the expression. For example, when evaluating the following expression where the variable "x" is set to a value of 0:

```
var x = 0;
```

The result below will be printed to the console:

```
undefined
```

This does not mean that the value of "x" is "undefined". It just means that the expression as a whole returns "undefined". To get the value of "x", highlight only the letter "x" in the expression, and hit the <Enter> key to get the value 0 as result. Sometimes it's also useful to use the console.println() method when using the console to get quick results.

The native Acrobat JavaScript edit dialog can also be used to evaluate JavaScript code. It behaves the same way as the console except for the fact that the results will still be output to the console. This is so that the code you are entering on the edit dialog doesn't get messed up by the printing of evaluation results.

Parameter Help

If you give an Acrobat method an argument of "?" (including the double quotes) and execute that method in console editor (or any internal JavaScript editor), the method will return a list of its own arguments:

Example: In the console, type

```
app.response("?")
```

While the cursor is still on the line you entered, press either Ctrl-Enter or the Enter key on the numeric pad. The output to the console is seen to be:

```
uncaught exception: Console:Exec:1: Help for App.response
====> [cQuestion: string]
====> [cTitle: string]
====> [cDefault: string]
====> [bPassword: boolean]
```

Parameters listed in square brackets indicate optional parameters.

Tips for Writing Reliable Code

Bugs are a fact of life in programming. Most of us would just as soon not have to deal with any, however. You shouldn't pass up any opportunity to make your code more reliable through good coding practices. Here are some tips for getting more gain, and less pain, from your code:

1. If you are new to programming, the easiest way to learn is to start by modifying somebody else's already-debugged example code. Make one small change at a time until you understand the full effect of your changes.

2. Choose meaningful names for your variables. A descriptive name like numberOfAvailableColors, while long, is usually preferable to a short but cryptic name, such as num or n.

3. Be consistent about variable-naming. Some programmers like to use descriptive prefixes on variable names, such as 'n' at the beginning of numeric variable names (like nTotal, nMaximum, etc.), 'b' at the beginning of Booleans, 's' on strings, 'g' on the front of variables with global scope, etc. Anything you can do to introduce consistency into your code will pay big dividends later.

4. Watch out for uninitialized variables. Declaring a variable and initializing it (assigning an initial value to it) are two different things. If you try to use a variable that has no value (in a comparison statement, for example), you will generate an error. Assigning a safe "dummy value" to every variable at the time it's created will prevent this kind of error.

5. Go for legibility, not elegance. If an operation can be more clearly expressed in two JavaScript statements than in one, use two statements. You're not going to win any awards for conciseness, and in many cases, execution speed is the same (or nearly so) with two statements as it is with one.

6. Use Cut and Paste operations when editing code, to save on keystrokes. The fewer keystrokes you use, the less chance you have of introducing typos into working code.

7. Balance those brackets and parentheses. Check nested statements carefully (or write them on more than one line) to be sure every left-paren is balanced by a right-paren and every left-bracket is balanced by a right-bracket.

8. Make liberal use of comments. Code that is sparsely commented is hard to maintain.

9. "Factor out" complex operations into short functions. Generally speaking, it's easier to understand and debug six two-line functions than it is to understand and debug one 12-line function. Also, factoring out your code will pay dividends in terms of code reuse.

10. Always have working code. Make frequent backup copies as you work. Before editing an existing piece of "good" code, make a copy of it so that you can revert to working code if you end up with seemingly unfixable bugs in your new code.

11. Study other programmers' code every chance you get. Don't reinvent the wheel. Learn from what others have done. Incorporate the best ideas into your own scripts. Soon, you'll be coding like a pro yourself!

Using JavaScript in PDF Forms

JavaScript can be particularly useful in conjunction with AcroForms. By attaching scripts to buttons, text fields, combo lists, and other "form widgets", you can greatly enhance the interactivity, appearance, and reliability of your forms. For example, you can use JavaScript to format and validate user input; submit form data to CGI scripts on a server, or to e-mail addresses; or create custom actions for buttons. In this section, we'll take a look at some common script actions and how to set up typical form interactions using JavaScript.

Organizing Your Code

As mentioned in Where Can You Use JavaScript?, it is possible to attach JavaScript to PDF documents in several ways: at the Field level (attached to Actions, or as custom Format, Validation, or Calculation scripts); at the level of Page Open or Page Closed actions; in links or bookmarks; and/or at the Document level. Deciding where to place your code is usually a simple matter: Code should usually be linked as closely as possible to the action or form field it is designed to control. In large projects, however, you will often find that the same kinds of operations need to be performed in many different places. For example, you may have a recurring need to calculate the average of a group of numbers. Rather than write the same number-averaging code over and over, it makes more sense to write one utility function that does this task and store that function at the document level, where it can be accessed by any other script, anywhere in your document. For example, in **Tools->JavaScript->Document JavaScripts...** , you could type:

```
function average() {
  if (arguments.length == 0) // nothing to do
    return 0;
  var sum = 0.0;
  for (var i = 0; i < arguments.length; i++)
    sum += arguments[i] - 0;
  return sum/arguments.length;
}
```

Then, from any other code at any place in the document, you could call on this function to perform numeric averaging.

Tip: *Note that the above function makes use of the core-JavaScript* arguments *array, which is available inside any JavaScript function. You can pass any number of arguments to the* average() *function, and it will calculate the average of the passed-in arguments—even if some or all of those arguments are strings! Subtracting zero from* arguments[i] *is a standard JavaScript trick for forcing the interpreter to convert* arguments[i] *to a number, if it's a string. Thus,* average(1,5,9) *will return the same value as* average("1","5","9").

Storing often-used functions at the document level is beneficial in several ways. It not only reduces clutter and streamlines your field-level code but promotes code reuse and contributes to readability (hence maintainability). "Factoring out" utility functions into document-level scripts will make your projects more manageable and easier to debug.

Working with Fields

One very common use for JavaScripts is to perform calculations on user input. To do this, you have to be able to fetch the value associated with a given form field. This is really a two-step process. The JavaScript is quite straightforward:

```
var theField = this.getField("City");
var theValue = theField.value;
```

Let's take a careful look at what's going on in the first line of code. First, we are declaring a new variable, *theField*, and assigning a value to it. On the right side of the assignment operator (the equal sign) is *this.getField("City").* The reference to 'this' is a JavaScript shortcut. In a document-level script or field-level script, 'this' means the *current document object*. Among the many predefined methods associated with the document object (see Doc Object) is the *getField()* method. This method takes a string argument—a string containing the name of the field whose properties you wish to manipulate or examine. For this example, we've assumed that the form contains a field somewhere with the name "City".

The second line of code declares a second variable, *theValue*, which will actually hold the value of the field (that is, whatever the user typed in). Field objects have their own properties and methods (see Field Object). The 'value' property contains the field's user-assigned value. For a text field, this value will be a string.

Field values can be altered at runtime by means of JavaScript. To do this, you have to set the 'value' property of the appropriate Field object to whatever value you want it to have. Note the correct and incorrect ways of doing this:

```
var theValue = "San Jose";          // Wrong. Does not alter field.
var theField.value = "San Jose";    // Correct. Will alter field.
```

The first line sets our local variable, *theValue*, to a new value, but this is not the same as making a permanent change to the "City" field of our form. All we have done in this case is change the value of a local variable, not a field. To act on the field requires that we have access to the Field Object itself. Since we obtained a reference to this object in *theField*, we can use the second method shown above to alter the value shown in the form. (The form will update its appearance immediately to show the new value.)

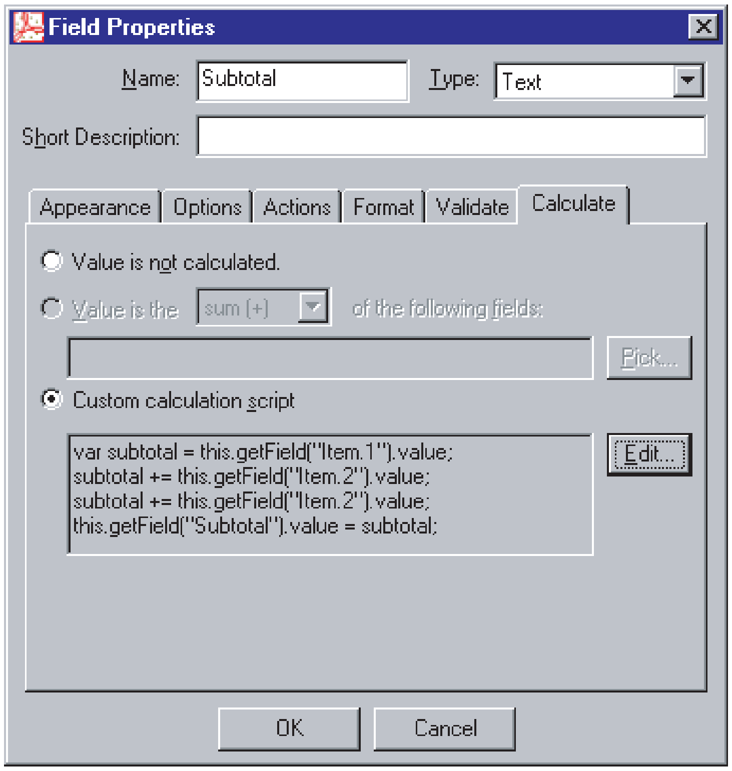
Another way to change the value in this field would simply be to write:

```
this.getField("City").value = "San Jose";
```

In one line of code, we have successfully overwritten the old value (if any) of our form's "City" field with a new value.

Binding Field Dependencies

Another common use for JavaScript is to bind field dependencies in forms. For example, shipping charges might depend on the "Zip Code" field of a form; or you might want to keep a running subtotal of user purchases in a "Subtotal" field. Usually, the best way to set up these sorts of dependencies is to use Calculation scripts. Create a new form field (or doubleclick on an existing one) to bring up the Field Properties dialog. Make sure the field type is either Text or Combo Box. Then select the Calculation tab within the Field Properties dialog:



```
var subtotal = this.getField("Item.1").value;
subtotal += this.getField("Item.2").value;
subtotal += this.getField("Item.2").value;
this.getField("Subtotal").value = subtotal;
```

To enter a Calculation script, which will be executed every time the form is updated, toggle the **Custom calculation script** radio button and hit the **Edit...** pushbutton on the right.

Tip: *Note that you can change the calculation order of fields manually, in Acrobat, using Tools -> Forms -> Set Field Calculation Order...; or you can set the calculation order programmatically, using the calcOrderIndex property of the Field Object. Acrobat's default ordering of fields may not always be what you want.*

Formatting and Validation Scripts

You can use JavaScript to automatically format and/or validate user input as it is happening. Inside the Field Properties dialog (for Text and Combo Box fields only), you will find Format and Validate tabs. When selected, these tabs expose dialog panes that allow you to attach custom scripts to the field for formatting or validation purposes. What's the difference? Think of format scripts as having to do with the *appearance* of user-entered data (for example, whether phone number area codes have parentheses), whereas validation scripts have to do with the *underlying value* of the data (the phone number itself).

If you want to filter the user's keystrokes as they happen, select the Format tab in Field Properties and enter a custom keystroke script. In this kind of script, you will typically examine the value of the change property of the Event Object, and either accept or reject the value that the user has entered. The following short script checks to see that the user is entering only numeric values, rejecting all other types of input:

```
if (event.change.match(/\D+/g)) {
    app.alert("Enter numeric characters only.");
    event.rc = false;
}
```

The condition check (the top line of this code fragment) uses the built-in *match()* method of JavaScript's String class to check the user's input against a *regular expression* representing *non-numeric characters*. (That’s what \D means.)

Tip: *Regular expressions use a special symbol notation to achieve pattern-matching based on well-defined rules. Consult any good JavaScript text for more information on how to use regular expressions to perform powerful string-search, replace, and match operations.*

If the user enters a value that contains a non-numeric character, two things happen: First, an alert dialog appears onscreen asking the user to enter numeric characters only; then the 'rc' property of the Event Object is set to *false*, which has the effect of preventing the user’s input from appearing onscreen.

Note that *event.change* is a string. It could very well refer to a lengthy string of just-pasted text, rather than a single typed character. For that reason, we treat it like a string and check every possible character in it.

Advanced Formatting

Sometimes, it is necessary to apply strict formatting criteria to user-entered field data. For example, it may be necessary to limit numeric values to two decimal places of precision, or apply a particular set of formatting criteria to user-entered dates or currency values, etc. Accomplishing this can be trickier than it first appears. Many formatting options can be handled using Acrobat's built-in formatting options for dates, times, currency values, etc. (See the Format tab in Field Properties.) But occasionally you may need to supplement Acrobat's built-in filters with custom scripts of your own. There are two sources of help to consider here. First, be aware of the fact that the Util Object class offers several built-in methods that can be helpful in performing formatting. For example, the printx method can be used to apply formatting to strings of numbers:

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

In this instance, the string "aaa14159697489zzz" ends up formatted as "1 (415) 969–7489". (See printx for more information.) Likewise, the printd and scand methods are useful for formatting dates according to various criteria. The C-language standby printf is also available.

A second source of help for carrying out low-level formatting consists of the utility functions in the *AForm.js* file, found in the JavaScripts directory **(Acrobat->JavaScripts).** This Adobe-supplied file contains numerous convenience functions and utilities for carrying out manipulation of date strings and numeric values. For example, the function *AFMakeNumber()* attempts to make a number out of a string that may or may not use a period as the separator. (In many parts of the world, a comma is used instead.) The source code for this and many other utility functions is given in *AForm.js.*

Tip: *Note that* AForm.js *is loaded automatically whenever Acrobat (or Reader) runs. All of the global variables and functions contained in* AForm.js *are available to your scripts at runtime.*

PDF Is Not HTML

If you are used to writing JavaScript code for HTML web pages, you may be tempted (from time to time) to call on methods like *window.open()* or *document.write()*. You'll find, however, that many of the objects, methods, and properties you're accustomed to working with in a browser environment either don't work or don't exist at all in PDF JavaScript. That's because JavaScript for PDF runs inside Acrobat or Acrobat Reader. The runtime interpreter, in this case, *isn't in the browser*. The objects and methods you use in PDF JavaScript are scoped to the PDF document itself, not to an HTML page.

Whether you realized it before or not, many of the objects and methods you are used to seeing in HTML-based JavaScript are *not part of the core language*. Rather, they are part of the *client-side extensions* to the core language. Server-based JavaScript, likewise, makes use of other “added in” methods and objects, relevant to the server environment. These are *server-side* extensions. PDF, in turn, has its own objects and methods. Always remember that the code you write for an HTML page executes in the browser; the code you write for server-side JavaScript executes in the server; and JavaScript for PDF documents executes in Acrobat (or Acrobat Reader). The methods available in each case differ.

This also means, of course, that any JavaScript you write for an HTML page cannot “see inside” a PDF document. Nor can the code you write for a PDF document see inside an HTML page. (However, both types of code can communicate back to the server.)

What's New For 5.0

Object	Properties	Methods
ADBC Object		getDataSourceList(), newConnection()
Annot Object	alignment, AP, arrowBegin, arrowEnd, attachIcon, author, contents,doc, fillColor, hidden, modDate, name, noteIcon, noView, page, point, popupRect, print, points, print, rect, readOnly, rotate, strokeColor, textFont, type, soundIcon, width	destroy(), setProps(), setProps()
App Object	activeDocs, fs, plugIns, viewerVariation	addMenuItem(), addSubMenu(), clearInterval(), clearTimeOut(), listMenuItems(), listToolbarButtons(), newDoc(), openDoc(), popUpMenu(), setInterval(), setTimeOut()
Bookmark Object	children, doc, name, open, parent, style	createChild(),execute(), insertChild(), remove()
Color Object		convert(), equal()
Connection Object		newStatement(), getTableList(), getColumnList()
Data Object	creationDate	creationDate, modDate, MIMEType, name, path, size

Object	Properties	Methods
Doc Object	bookmarkRoot, icons, info, layout, securityHandler, selectedAnnots, sounds, templates, URL	addAnnot(), addField(), addIcon(), addThumbnails(), addWeblinks(), bringToFront(), closeDoc(), createDataObject(), createTemplate(), deletePages(), deleteSound(), exportAsXFDF(), exportDataObject(), extractPages(), flattenPages(), getAnnot(), getAnnots(), getDataObject(), getIcon(), getPageBox(), getPageLabel(), getPageNthWord(), B: This method will throw an exception if the document security is set to prevent content extraction.(), getPageRotation(), getPageTransition(), getSound(), importAnXFDF(), importDataObject(), importIcon(), importSound(), importTextData(), insertPages(), movePage(), print(), removeDataObject(), removeField(), removeIcon(), removeTemplate(), removeThumbnails(), removeWeblinks(), replacePages(), saveAs(), selectPageNthWord(), setPageBoxes(), setPageLabels(), setPageRotations(), setPageTransitions(), submitForm(), syncAnnotScan()
Event Object	changeEx, keyDown, targetName	
Field Object	buttonAlignX, buttonAlignY, buttonPosition, buttonScaleHow, buttonScaleWhen, currentValueIndices, doNotScroll, doNotSpellCheck, exportValues, fileSelect, multipleSelection, rect, strokeColor, submitName, valueAsString	browseForFileToSubmit(), buttonGetCaption(), buttonGetIcon(), buttonSetCaption(), buttonSetIcon(), checkThisBox(), defaultIsChecked(), isBoxChecked(), isDefaultChecked(), setAction(), signatureInfo(), signatureSign(), signatureValidate()

Object	Properties	Methods
FullScreen Object	backgroundColor, clickAdvances, cursor, defaultTransition, escapeExits, isFullScreen, loop, timeDelay, transitions, usePageTiming, useTimer	
Global Object		subscribe()
Identity Object	corporation, email, loginName, name	
Index Object	available, name, path, selected	
PlugIn Object	certified, loaded, name, path, version	
PPKLite Signature Handler Object	appearances, isLoggedIn, loginName, loginPath, name, signInvisible, signVisible, uiName	login(),logout(), newUser(), setPasswordTimeout(),
Report Object	size, absIndent, color	Report(), writeText(), breakPage(), divide(), indent (), outdent(), open(), save(),mail()
Search Object	available, indexes, matchCase, maxDocs, maxDocs, proximity, proximity, refine, soundex, stem	addIndex(), getIndexForPath(), query(), removeIndex()
Security Object	handlers, validateSignaturesOnOpen	getHandler()

Object	Properties	Methods
Spell Object	available, dictionaryNames, dictionaryOrder, domainNames	addDictionary(), addWord(), check(), checkText(), checkWord(), removeDictionary(), removeWord(), userWords()
Statement Object	columnCount, rowCount	execute(), getColumn(), getColumnArray(), getRow(), nextRow()
Template Object	hidden, Although reading this property is valid, setting this property in Acrobat Reader will generate an exception.	spawn()

Other 5.0 Changes

This manual contains an extensive appendix entitled A Short Acrobat JavaScript FAQ. If particular importance are the sections How can I create a form field programmatically?, Quick Reference: Forms and How can I create an Annotation programmatically? which summarize how the numerous properties and methods (both old and new) can can be use to create form fields and annotations (also called comments). Many interesting examples and programming tips are contained in these sections.

In addition to the new objects, properties and methods outlined in the table above, there has been a number of other changes and enhancements that should be noted.

Added: The console can now act as an editor and can execute JavaScript code, see the section entitled Interactive JavaScript Console.

Changed/Enhanced: The following properties and methods have been enhanced: App.language, App.execMenuItem; Event.type; Doc.exportAsFDF, Doc.print, Doc.submitForm; Field.buttonImportIcon, Field.textFont, Field.getItemAt, Field.value; Util.printd. The section related to Event Object has been greatly enhanced to facilitate better understanding of the Acrobat JavaScript Event model.

Deprecated: The following properties and methods have been deprecated: App.fullscreen, App.numPlugIns, App.getNthPlugInName; Doc.author, Doc.creationDate, Doc.creator, Doc.getNthTemplate, Doc.keywords, Doc.modDate, Doc.numTemplates, Doc.producer, Doc.spawnPageFromTemplate, Doc.title; Field.hidden; Tts.soundCues, Tts.speechCues.

ADBC Object

5.0			⊗

The Acrobat Database Connectivity (ADBC) plug-in allows JavaScripts inside of PDF documents to access databases through a consistent object model that is identical across platforms. The object model is based on general principles used in the object models for the ODBC and JDBC APIs. Like ODBC and JDBC, ADBC is a means of communicating with a database though SQL or Structured Query Language.

ADBC is a Windows-only feature and requires ODBC (Open Database Connectivity from Microsoft Corporation) to be installed on the client machine.

Security *: It is important to note that ADBC provides no security for any of the databases it is programmed to access. It is the responsibility of the database administrator to keep all data secure.*

The *ADBC Object* is a global object whose methods allow a JavaScript to create database connection contexts or connections. In addition to the *ADBC Object*, described below, there are several related objects as well:

Object	Brief Description
ADBC Object	An object through which a list of accessible databases can be obtained and a connection can be made to one of them.
Connection Object	An object through which a list of tables in the connected database can be obtained.
Statement Object	An object through which SQL statements can be executed and rows retrieved based on the query.

ADBC properties

SQL Type

5.0			⊗

The ADBC object has several constant properties representing various SQL Types:

Name	value	Name	value
SQLT_BIGINT	0	SQLT_LONGVARCHAR	10
SQLT_BINARY	1	SQLT_NUMERIC	11
SQLT_BIT	2	SQLT_REAL	12
SQLT_CHAR	3	SQLT_SMALLINT	13
SQLT_DATE	4	SQLT_TIME	14
SQLT_DECIMAL	5	SQLT_TIMESTAMP	15
SQLT_DOUBLE	6	SQLT_TINYINT	16
SQLT_FLOAT	7	SQLT_VARBINARY	17
SQLT_INTEGER	8	SQLT_VARCHAR	18
SQLT_LONGVARBINARY	9		

The *type* property of the Column Object and *type* property of the ColumnInfo Object both return this SQL Type property.

JavaScript Type

5.0			⊗

The ADBC object has several constant properties representing various JavaScript data types.

Name	value	Name	value
Numeric	0	Time	4
String	1	Date	5
Binary	2	TimeStamp	6
Boolean	3		

The methods getColumn and getColumnArray of the Statement Object both use these types.

ADBC methods

getDataSourceList

5.0			⊗

Parameters: None
Returns: Array

The *getDataSourceList* method is used to obtain information about the databases accessible from a given system. It takes no parameters and returns a (possibly empty) array containing a DataSourceInfo Object for each accessible database on the system. This method never fails but may return a zero-length array.

The properties of the DataSourceInfo Object are listed in the table below.

DataSourceInfo Object

The DataSourceInfo object is an object that contains very basic information about a particular database.

Property	Type	Access	Description
name	string	R	A string that represents the identifying name of a database. This string could be passed to newConnection to establish a connection to the database that the DataSourceInfo object is associated with.
description	string	R	A string that contains database dependent information about the database.

See newConnection for an example.

newConnection

5.0			⊗

Parameters: cDSN, [cUID], [cPWD]
Returns: connection object | null

The *newConnection* method is used to create a Connection Object associated with the database identified by the *cDSN* (Data Source Name) parameter. Optionally, the *cUID* parameter can supply a user ID and possibly a password, *cPWD*. On success, this method returns a Connection object. On failure, it returns *null*.

```
Example:
        // First, get the array of DataSourceInfo Objects available on the system
        var aList = ADBC.getDataSourceList();
```

```
console.show(); console.clear();
try {
      // now display them, while searching for the one named "q32000data".
      var DB = "", msg = "";
      if (aList != null) {
          for (var i=0; i < aList.length; i++) {
            console.println("Name: "+aList[i].name);
            console.println("Description: "+aList[i].description);
            // and choose one of interest
            if (aList[i].name=="q32000data")
                DB = aList[i].name;
          }
      }
      // did we find the database?
      if (DB != "") {
          // yes, establish a connection.
          console.println("The requested database has been found!");
          var Connection = ADBC.newConnection(DB);
          if (Connection == null)
              throw "Not Connected!"
      } else
          // no, display message to console.
          throw "Could not find the requested database.";
} catch (e) {
      console.println(e);
}

// alternatively, we could simple connect directly.
var Connection = ADBC.newConnection("q32000data");
```

Annot Object

The functionality of the Acrobat Annotation Plug-in is exposed to JavaScript methods through the Annot Object.

The Annot object represents any particular Acrobat annotation (that is, an annotation created using the Acrobat annotation tool, or by using addAnnot method from the Doc Object). The types of annotations available are *Circle*, *FileAttachment, FreeText*, *Highlight*, *Ink*, *Line*, *Sound*, *Square*, *Squiggly*, *Stamp*, *StrikeOut*, *Text* and *Underline*. (There is no user interface to the *Squiggly* annot type; *Squiggly* can be accessed only through JavaScript.)

In addition to addAnnot, used for the creation of annotations, there are two other Doc Object methods for gathering annotations from documents, getAnnot and getAnnots

Note also that the section How can I create an Annotation programmatically? provides detailed examples of most all annotations and JavaScript techniques.

Note: *The user interface in Acrobat refers to annotations as comments.*

Annotation Access from JavaScript

Before an Annotation can be accessed, it must be "bound" to a JavaScript variable through a method provided by Doc Object methods interface:

```
var a = this.getAnnot(0, "Important");
```

This example allows the script to now manipulate the annotation named "Important" on page 1 (zero-based page numbering system) via the variable *a*. For example, the following code

```
var thetype = a.type;        // read property
a.author = "John Q. Public"; // write property
```

first stores the type of annotation in the variable *thetype*, then changes the author to "John Q. Public".

Annot Object: Quick Reference

Annotation Types	Properties
All types	type, rect, page, author, name, contents, modDate
Circle	point, popupRect, fillColor, strokeColor, width
FileAttachment	print, attachIcon

Annot Object: Quick Reference	
Annotation Types	**Properties**
FreeText	alignment, fillColor, rotate, strokeColor, textFont, textSize, width
Highlight	quads, strokeColor, point, popupRect
Ink	gestures, strokeColor, point, popupRect, width
Line	points, arrowBegin, arrowEnd, point, popupRect, fillColor, strokeColor, width
Sound	print, soundIcon
Square	point, popupRect, fillColor, strokeColor, width
Squiggly	quads, strokeColor, point, popupRect
Stamp	point, popupRect, AP
StrikeOut	quads, strokeColor, point, popupRect
Text	print, noteIcon, point, popupRect
Underline	quads, strokeColor, point, popupRect

Annot Properties

alignment

5.0	💾		⊗

Type: Number *Annots: FreeText* *Access: R/W*

The property controls the alignment of the text for a *FreeText* annotation.

Alignment	Value
Left aligned	0
Centered	1
Right aligned	2

See the section on FreeText Annotations for an example of the use of the alignment property.

AP

5.0	💾		⊗

Type: String *Annots: Stamp* *Access: R/W*

The value of this property is the named appearance of the stamp to be used in displaying a stamp annotation. The names of the standard stamp annotations are "Approved", "AsIs", "Confidential", "Departmental", "Draft", "Experimental", "Expired", "Final", "ForComment", "ForPublicRelease", "NotApproved", "NotForPublicRelease", "Sold" and "TopSecret".

Example:

```
var annot = this.addAnnot({
    page: 0,
    type: "Stamp",
    author: "A. C. Robat",
    name: "myStamp",
    rect: [400, 400, 550, 500],
    contents: "Try it again, this time with order and method!",
    AP: "NotApproved"
});
```

Note: *The name of a particular stamp can be found by opening the PDF file in the* Stamps *folder that contains the stamp in question. Now open the menu* File > Form > Page Templates, *a listing of all appearances and their names can then be seen. For a list of stamp names currently in use in the document, see the* icons *property of the* Doc Object.

arrowBegin

5.0	💾		⊗

Type: String *Annots: Line* *Access: R/W*

The value of *arrowBegin* determines the line cap style which specifies the shape to be used at the beginning of a *Line annotation.* Permissible values are "Circle", "ClosedArrow", "Diamond", "None" (the default), "OpenArrow" and "Square". See arrowEnd and setProps (for an example), as well as the section on Line Annotations.

arrowEnd

5.0	💾		⊗

Type: String *Annots: Line* *Access: R/W*

The value of *arrowEnd* determines the line cap style which specifies the shape to be used at the end of a *Line annotation* Permissible values are "Circle", "ClosedArrow", "Diamond", "None" (the default), "OpenArrow" and "Square". See arrowBegin and setProps (for an example), as well as the section on Line Annotations.

attachIcon

5.0	💾		⊗

Type: String *Annots: FileAttachment* *Access: R/W*

The value of this property is the name of an icon to be used in displaying the annotation. Recognized values are "Paperclip", "PushPin" (the default), "Graph", and "Tag".

author

5.0	💾		⊗

Type: String *Annots: all* *Access: R/W*

The author of the annotation.

See contents for an example of usage.

contents

5.0	💾		⊗

Type: String *Annots: all* *Access: R/W*

The contents of any annotation having a popup can be accessed through this property. In the case of *Sound* and *FileAttachment* annotations, the contents property specifies the text to be displayed as the description of the sound or file attachment.

Example:
```
var annot = this.addAnnot({
    page: 0,
    type: "Text",
    point: [400,500],
    author: "A. C. Robat",
    contents: "Call Smith to get help on this paragraph.",
    noteIcon: "Help"
});
```

See also the addAnnot method.

doc

5.0			⊗

Type: doc object *Annots: all* *Access: R*

This property returns the Doc Object of the document in which the annotation resides.

Example:
```
var inch = 72;
var annot = this.addAnnot({
    type: "Square",
    rect: [1*inch, 3*inch, 2*inch, 3.5*inch]
});
/* displays, e.g., "file:///C|/Adobe/Annots/myDoc.pdf" */
console.println(annot.doc.URL);
```

fillColor

5.0	💾		⊗

Type: Color *Annots: Circle, Square, Line, FreeText* *Access: R/W*

This property sets the background color for the *Circle*, *Square, Line* and *FreeText* annotations. 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.

For an example, see Circle and Square Annotations.

gestures

5.0	💾		⊗

Type: Array *Annots: Ink* *Access: R/W*

An array of arrays, each representing a stroked path. Each array is a series of alternating *x* and *y* coordinates in Default User Space, specifying points along the path.When drawn, the points are connected by straight lines or curves in an implementation-dependent way. See “Ink Annotations” in the PDF Reference, page 415, for more details.

See Ink Annotations for an extensive example.

hidden

5.0	💾		⊗

Type: Boolean *Annots: all* *Access: R/W*

If *hidden* is set to *true*, then the annotation is not shown and there is no user interaction, display or printing of the annotation.

modDate

5.0			⊗

Type: Date *Annots: all* *Access: R*

This property returns the last modification date for the annotation.

Example:
```
    // This example prints the modification date to the console
    console.println(util.printd("mmmm dd, yyyy", annot.modDate));
```

name

5.0	💾		⊗

Type: String *Annots: all* *Access: R/W*

The name of an annotation can be used by getAnnot to find and access the properties and methods of the annotation.

Example:

```
// This code locates the annotation named "myNote" and appends a comment.
var gannot = this.getAnnot(0, "myNote");
gannot.contents += "\r\rDon't forget to check with Smith";
```

noteIcon

5.0	💾		⊗

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

The value of this property is the name of an icon to be used in displaying the annotation. Recognized values of *noteIcon* are "Comment", "Help", "Insert", "Key", "Note" (the default), "NewParagraph", and "Paragaph".

See the contents property for an example and the section entitled Text Annotations.

noView

5.0	💾		⊗

Type: Boolean *Annots: all* *Access: R/W*

If *noView* is set to *true*, then the annotation is hidden, but if the annotation has an appearance, that appearance should be used for printing only.

page

5.0	💾		⊗

Type: Integer *Annots: all* *Access: R/W*

This property is the page on which the annotation resides. The code, for example,

```
annot.page = 2;
```

moves the Annot object, *annot*, from its current page to page 3 (zero-based page numbering system).

point

5.0	💾		⊗

Type: Array *Annots: Text, Sound, FileAttachment* *Access: R/W*

An *Array* of two numbers, [x_{ul}, y_{ul}] which specifies the upper left-hand corner in default, user's space, of an annotation's *Text, Sound,* or *FileAttachment* icon.

Example:
```
var annot = this.addAnnot({
    page: 0,
    type: "Text",
    point: [400,500],
    contents: "Call Smith to get help on this paragraph.",
    popupRect: [400,400,550,500],
    popupOpen: true,
    noteIcon: "Help"
});
```

See also addAnnot and noteIcon.

points

5.0	💾		⊗

Type: Array *Annots: Line* *Access: R/W*

An *Array* of two points, [[x_1, y_1], [x_2, y_2]], specifying the starting and ending coordinates of the line in Default User Space.

Example:
```
var annot = this.addAnnot({
  type: "Line",
  page: 0,
  author: "A. C. Robat",
  contents: "Look at this again!",
  points: [[10,40],[200,200]],
})
```

See addAnnot, arrowBegin, arrowEnd and setProps and Line Annotations.

popupOpen

5.0	💾		⊗

Type: Boolean *Annots: all but Sound, FreeText, FileAttachment* *Access: R/W*

If *popupOpen* is *true* then the popup text note will appear open when the page is displayed.

See the print property for an example.

popupRect

5.0	💾		⊗

Type: Array *Annots: all but FreeText, FileAttachment, Sound* *Access: R/W*

The *Array* consists of four numbers [x_{ll}, y_{ll}, x_{ur}, y_{ur}] specifying the lower-left *x*, lower-left *y*, upper-right *x* and upper-right *y* coordinates—in Default User Space—of the rectangle of the *popup annotation* associated with a parent annotation and defines the location of the popup annotation on the page.

See the print property for an example.

print

5.0	💾		⊗

Type: Boolean *Annots: all* *Access: R/W*

If *print* property indicates whether the annotation should be printed. When set to *true*, the annotation will be printed.

quads

5.0	💾		⊗

Type: Array *Annots: Highlight, StrikeOut,Underline, Squiggly* *Access: R/W*

An array of 8 x *n* numbers specifying the coordinates of *n* quadrilaterals in Default User Space. Each quadrilateral encompasses a word or group of contiguous words in the text underlying the annotation. See Table 7.19, page 414 of the PDF Reference for more details. The *quads* for a word can be obtained through calls to the getNthTemplate.

See getNthTemplate for an example.

rect

5.0	💾		⊗

Type: Array *Annots: all* *Access: R/W*

The *Array* consists of four numbers [x_{ll}, y_{ll}, x_{ur}, y_{ur}] specifying the lower-left *x*, lower-left *y*, upper-right *x* and upper-right *y* coordinates—in Default User Space—of the rectangle defining the location of the annotation on the page. See also popupRect.

readOnly

5.0	💾		⊗

Type: Boolean *Annots: all* *Access: R/W*

When *readOnly* is set to *true*, this indicates that the annotation should display, but not interact with the user.

rotate

5.0	💾		⊗

Type: Integer *Annots: FreeText* *Access: R/W*

The property *rotate* is the number of degrees (0, 90, 180, 270) the annotation is rotated counter-clockwise relative to the page. The Icon based annotations do not rotate, this property is only significant for *FreeText* annotations.

strokeColor

5.0	💾		⊗

Type: color *Annots: all* *Access: R/W*

This property sets the appearance color of the annotation. Values are defined by using *transparent, gray, RGB* or *CMYK* color. In the case of a *FreeText* annotation, *strokeColor* sets the border and text colors. Refer to the Color Arrays section for information on defining color arrays and how values are used with this property.

Example:

```
        // Make a text note red
        var annot = this.addAnnot({type: "Text"});
        annot.strokeColor = color.red;
```

textFont

5.0	💾		⊗

Type: String *Annots: FreeText* *Access: R/W*

The *textFont* property determines the font that is used when laying out text in a *FreeText* annotation. Valid fonts are defined as properties of the "font" object, as listed in the textFont property of the Field object:

An arbitrary font can be used when laying out a *FreeText* annotation by setting the value of *textFont* equal to a string that represents the PostScript name of the font.

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

```
// Create FreeText annotation with Helvetica
var annot = this.addAnnot({
  page: 0,
  type: "FreeText",
  textFont: font.Helv, // or, textFont: "Viva-Regular",
  textSize: 10
  rect: [200, 300, 200+150, 300+3*12], // height for three lines
  width: 1,
  alignment: 1
});
```

textSize

5.0	💾		⊗

Type: Integer *Annots: FreeText* *Access: R/W*

This property determines the text size (in points) that is used in a *FreeText* annotation. 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 will allow all the text data to still fit in the annotations's rectangle should be used. See textFont for an example.

type

5.0	💾		⊗

Type: String *Annots: all* *Access: R*

This property can be used to determine the type of annotation. The valid values of type are "Circle", "FileAttachment", "FreeText", "Highlight", "Ink", "Line", "Sound", "Square", "Squiggly", "Stamp", "StrikeOut", "Text" and "Underline".

Note: *The "type" of the annotation can only be set within the object literal argument of the* addAnnot *method .*

soundIcon

Type: String *Annots: Sound* *Access: R/W*

5.0	💾		⊗

The value of this property is the name of an icon to be used in displaying the annotation. A value of "Speaker" is recognized.

width

5.0	💾		⊗

Type: Number *Annots: Square, Circle, Line, Ink, FreeText* *Access: R/W*

The border width in points. If this value is 0, no border is drawn. The default value is 1.

Annot Methods

destroy

5.0	💾		⊗

Parameters: None
Returns: Nothing

This method destroys the annot, removing it from the page. The object becomes invalid.

Example:
```
// remove all "FreeText" annotations on page 0
var annots = this.getAnnots({ nPage:0 });
for (var i = 0; i < annots.length; i++)
    if (annots[i].type == "FreeText")
        annots[i].destroy();
```

getProps

5.0			⊗

Parameters: None
Returns: object literal

This methods returns an *object literal* of the properties of the annotation. The *object literal* is just like the one passed to addAnnot. This method can be used to copy an annotation.

Example:
```
var annot = this.addAnnot({
  type: "Text",
  rect: [40, 40, 140, 140]
});

// Make a copy of the properties of annot
var copy_props = annot.getProps();

// Now create a new annot with the same properties on every page
var numpages = this.numPages;
for (var i=0; i < numpages; i++) {
    var copy_annot = this.addAnnot(copy_props);
    // but move it to page i
    copy_annot.page=i;
}
```

setProps

5.0	💾		⊗

Parameters: object literal
Returns: annotation object

Sets many properties of the annotation simultaneously. The *object literal* is just like the one passed to addAnnot.

Example:

```
var annot = this.addAnnot({type: "Line"})
annot.setProps({
    page: 0,
    points: [[10,40],[200,200]],
    strokeColor: color.red,
    author: "A. C. Robat",
    contents: "Check with Jones on this point.",
    popupOpen: true,
    popupRect: [200, 100, 400, 200], // place rect at tip of the arrow
    arrowBegin: "Diamond",
    arrowEnd: "OpenArrow"
})
```

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

activeDocs

5.0			

Type: Array *Access: R*

This property returns an array containing the Doc Object for each active document open in the viewer. If no documents are active, *activeDocs* returns nothing, or has the same behavior as d=new Array(0) in core JavaScript.

With this property, it is possible to perform cross-document operations.

Example:

```
/* This example searches among the open documents for the document with a title
of "C", then it inserts a button in that document using addField */
var d = app.activeDocs;
for (var i=0; i < d.length; i++)
if (d[i].info.Title == "myDoc") {
    var f = d[i].addField("myButton", "button", 0 , [20, 100, 100, 20]);
    f.setAction("MouseUp","app.beep(0)");
    f.fillColor=color.gray;
}
```

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 occurring. Its default value is *true*. See also the document calculate property which supersedes this property in later versions.

focusRect

4.05	👥		

Type: Boolean *Access: R/W*

This property turns on and off the focus rectangle. The focus rectangle is the faint dotted line around buttons, check boxes, radio buttons, and signatures to indicate that the form field has the keyboard focus.

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 >= 4.0) {
  // Perform version specific operations here.
}
```

fs

5.0	👥		

Type: object *Access: R*

Returns the FullScreen Object which can be used to access the fullscreen properties.

Example:

```
// This code puts the viewer into fullscreen (presentation) mode.
app.fs.isFullScreen = true;
```

See also isFullScreen in the FullScreen Object.

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 the 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.

Note: *A PDF document being viewed from within a web browser* cannot *be put into fullscreen mode. Fullscreen mode can, however, be initiated from within the browser, but* will not occur *unless there is a document open in the Acrobat viewer application; in this case, the document open in the viewer will appear in fullscreen, not the PDF document open in the web browser.*

See the isFullScreen property of the FullScreen Object; this property supersedes App.fullscreen in later versions. See also App.fs, which returns a FullScreen Object which can be used to access the fullscreen properties.

language

Type: String *Access: R*

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

String	Language	String	Language
CHS	Chinese Simplified	KOR	Korean
CHT	Chinese Traditional	JPN	Japanese
DAN	Danish	NLD	Dutch
DEU	German	NOR	Norwegian
ENU	English	PTB	Brazilian Portuguese
ESP	Spanish	SUO	Finnish
FRA	French	SVE	Swedish
ITA	Italian		

numPlugIns

☹			

Type: Number *Access: R*

This property indicates the number of plug-ins that have been loaded by Acrobat. See the plugIns property which supersedes this property in later versions.

openInPlace

5.0	👥		

Type: Boolean *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".

plugIns

5.0			

Type: Array *Access: R*

Used to determine which plug-ins are currently installed in the viewer. Returns an array of PlugIn Objects.

Example:
```
// Get array of PlugIn Objects
var aPlugins = app.plugIns;
// Get number of plugins
var nPlugins = aPlugins.length;
// Enumerate names of all plugins
for ( var i = 0; i < nPlugins; i++)
   console.println("Plugin \#"+i+" is " + aPlugins[i].name);
```

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

☹	👥		

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).

Note: *Acrobat 5.0 has drastically changed the notion of what a toolbar is and where it can live within the frame of the application. This property has been deprecated as a result. If accessed, it will act like the toolbar property.*

toolbarVertical

☹	👥		

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).

Note: *Acrobat 5.0 has drastically changed the notion of what a toolbar is and where it can live within the frame of the application. This property has been deprecated as a result. If accessed, it will act like the toolbar property.*

viewerType

Type: String *Access: R*

This property determines if the running Acrobat Viewer is Reader vs. the full product. Its value is "Reader" or "Exchange" respectively.

viewerVariation

5.0			

Type: String *Access: R*

This property indicates the packaging of the running Acrobat Viewer. Its value is "Reader", "Fill-In", "Business Tools", or "Full".

viewerVersion

4.0			

Type: Number *Access: R*

This property indicates the version number of the current viewer.

App Object Methods

addMenuItem

5.0		🔑	

Parameters: cName, [cUser], cParent, [nPos], cExec, [cEnable], [cMarked]
Returns: Nothing

Adds a menu item to the application.

cName is the language independent name of the menu item. This language independent name is used to access the menu item for other methods (e.g. hideMenuItem).

cUser is the user string (language dependent name) to display as the menu item name. If *cUser* is not specified then *cName* is used.

cParent is the name of the parent menu item. Its submenu will have the new menu item added to it. If *cParent* has no submenu then an exception is thrown.

Menu item names can be discovered via the listMenuItems method. Language independent names for menu items can also be obtained from the *Acrobat Viewer plug-in API On-Line Reference* (See Useful Documents).

nPos is the position within the submenu to locate the new menu item. The default behavior is to append to the end of the submenu. Specifying *nPos* as 0 will add to the top of the submenu.

cExec is an expression string to evaluate when the menu item is selected by the user.

cEnable is an expression string that determines whether or not to enable the menu item. The default is that the menu item is always enabled. This expression should set *event.rc* to false to disable the menu item.

cMarked is an expression string that determines whether or not the menu item has a check mark next to it. Default is that the menu item is not marked. This expression should set *event.rc* to false to uncheck the menu item and true to check it.

Example:

```
// This example adds a menu item to the top of the file submenu that puts up an
// alert dialog displaying the active document title. This menu is only
// enabled if a document is opened.
app.addMenuItem({ cName: "Hello", cParent: "File",
    cExec: "app.alert(app.activeDocs[0].info.title, 3);",
    cEnable: "event.rc = app.activeDocs.length > 0",
    nPos: 0});
```

See also the addSubMenu, execMenuItem, hideMenuItem, and listMenuItems methods.

Security 🔑*: This method can only be executed during application initialization or console events. See the* Event Object *for a discussion of Acrobat JavaScript events.*

---

addSubMenu

5.0		🔑	

Parameters: cName, [cUser], cParent, [nPos]
Returns: Nothing

Adds a menu item with a submenu to the application.

cName is the language independent name of the menu item. This language independent name is used to access the menu item for hideMenuItem, for example.

cUser is the user string (language dependent name) to display as the menu item name. If *cUser* is not specified then *cName* is used.

cParent is the name of the parent menu item to receive the new submenu.

Menu item names can be discovered via the listMenuItems method. Language independent names for menu items can also be obtained from the *Acrobat Viewer plug-in API On-Line Reference* (See Useful Documents).

nPos is the position within the parent's submenu to locate the new submenu. Default is to append to the end of the parent's submenu. Specifying *nPos* as 0 will add to the top of the parent's submenu.

Example:

```
// This example adds a submenu "One" to the top of the File submenu.
// It has two additional menu items that display an alert message.
app.addSubMenu({ cName: "One", cParent: "File" });
app.addMenuItem({ cName: "Two", cParent: "One",
    cExec: "app.alert('Two', 3);"});
app.addMenuItem({ cName: "Three", cParent: "One",
    cExec: "app.alert('Three', 3);"});
```

See also the addMenuItem, execMenuItem, hideMenuItem, and listMenuItems methods.

Security *: This method can only be executed during application initialization or console events. See the* Event Object *for a discussion of Acrobat JavaScript events.*

alert

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

This method displays an alert dialog on the screen. The minimum required parameter is a string, *cMsg*, 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

Note: *The Macintosh OS does not distinguish between warnings and questions, so it only has three different types of icons.*

Additionally, a button group type can be specified by using the *nType* parameter:

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
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	0
Warning	1
Question	2
Status	3
Default (the default)	4

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

clearInterval

Parameters: oInterval

5.0			

Returns: Nothing

This method cancels a previously registered interval, *oInterval*, such an interval is initially set by the setInterval method.

See also the setTimeOut and clearTimeOut methods. An example of use follows the description of the setTimeOut method.

clearTimeOut

5.0			

Parameters: oTime
Returns: Nothing

This method cancels a previously registered time-out interval, *oTime*; such an interval is initially set by the setTimeOut method.

See also the setInterval, and clearInterval methods. An example of use follows the description of the setTimeOut method.

execMenuItem

4.0			

Parameters: cMenuItem
Returns: Nothing

This method executes the specified menu item.

Menu item names can be discovered via the listMenuItems method. Language independent names for menu items can also be obtained from the *Acrobat Viewer plug-in API On-Line Reference* (See Useful Documents).

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");
```

See also the addMenuItem, addSubMenu, hideMenuItem methods. The listMenuItems method conveniently lists the names of all menu items to the console.

5.0	Additions

App.execMenuItem("SaveAs") saves the current file to the user’s hard drive; a “SaveAs” dialog opens to ask the user to select a folder and file name. Executing the “SaveAs” menu item will save the current file as a linearized file, provided “Save As creates Fast View Adobe PDF files” is checked in the *Edit > Preferences > General > Options* dialog.

Security *:* App.execMenuItem("SaveAs") *can only be executed during batch, console or menu events. See the* Event Object *for a discussion of Acrobat JavaScript events.*

Note: *If the user preferences are set to “Save As creates Fast View Adobe PDF files”, do not expect a form object to survive a "SaveAs"; field objects are no longer valid, and an exception may be thrown when trying to access a field object immediately after a "SaveAs". See examples that follow.*

Example:
```
var f = getField("myField");
app.execMenuItem("SaveAs");  // Assume preferences set to save linearized
f.value = 3;                 // exception thrown, field not updated
```

Example:
```
var f = getField("myField");
app.execMenuItem("SaveAs");  // Assume preferences set to save linearized
var f = getField("myField"); // re-get the field after the linear save
f.value = 3;                 // field updated to a value of 3
```

Note: *For security reasons, scripts are not allowed to execute the* "Quit" *menu item.*

---

getNthPlugInName

☹			

Parameters: nIndex
Returns: cName

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

See the plugIns property which supersedes this property in later versions.

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 allows an integrator to customize the look of the Acrobat viewer by removing the menu item specified by *cName*.

Menu item names can be discovered via the listMenuItems method. Language independent names for menu items can also be obtained from the *Acrobat Viewer Plug-In API On-line Reference (Technical Note #5191)*. See Useful Documents.

See also the addMenuItem, addSubMenu, execMenuItem, and listMenuItems methods.

Security 🔑*: This method can only be executed during application initialization or console events. See the* Event Object *for a discussion of Acrobat JavaScript events.*

hideToolbarButton

4.0		🔑	

Parameters: cName
Returns: Nothing

This method allows a forms integrator to customize the look of the Acrobat viewer by removing the toolbar button specified by *cName*.

Menu item names can be discovered via the listToolbarButtons method. Language independent names for toolbar buttons can be obtained from the *Acrobat Viewer Plug-In API On-line Reference (Technical Note #5191)*. See Useful Documents.

Example: A file named, *myConfig.js*, containing the script

```
app.hideToolbarButton("Hand");
```

is placed in one of the Folder Level JavaScripts folders. When the Acrobat viewer is started, the "Hand" icon does not appear.

Security *: This method can only be executed during application initialization or console events. See the* Event Object *for a discussion of Acrobat JavaScript events.*

listMenuItems

5.0			

Parameters: None
Returns: Nothing

Lists all menu item names in the application to the console. Useful for writing scripts and debugging.

Language independent names for menu items can also be obtained from the *Acrobat Viewer plug-in API On-Line Reference (Technical Note #5191)*. See Useful Documents.

Example: List all menu item names to the console.

```
console.show();
app.listMenuItems();
```

See also the addMenuItem, addSubMenu, execMenuItem, and hideMenuItem methods.

listToolbarButtons

5.0			

Parameters: None
Returns: Nothing

Lists all toolbar button names in the application to the console. Useful for writing scripts and debugging.

Language independent names for menu items can also be obtained from the *Acrobat Viewer plug-in API On-Line Reference(Technical Note #5191)*. See Useful Documents.

See also the hideToolbarButton method.

mailMsg

4.0			⊗

> *Parameters: bUI, cTo, [cCc], [cBcc], [cSubject], [cMsg]*
> *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 *cMsg* 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.");
/* Or the same message can be sent as follows:
app.mailMsg( {bUI: false,
              cTo: "fun1@fun.com; fun2@fun.com",
              cSubject: "This is the subject",
              cMsg: "This is the body of the mail."} );
```

Note: *On Windows: The client machine must have its default mail program configured to be MAPI enabled in order to use this method.*

newDoc

5.0		🔑	⊗

> *Parameters: [nWidth], [nHeight]*
> *Returns: Doc object*

This method creates a new document in the Acrobat Viewer and returns the Doc Object of the newly created document. The optional parameters, *nWidth and nHeight*, are used to specify the media box dimensions in points of the document. The default values are *nWidth = 612* and *nHeight = 792*.

Example: Add a "New" item to the Acrobat File menu. Within "New", there are three menu items: "Letter", "A4" and "Custom". This script should go in a Folder Level JavaScripts folder.

```
app.addSubMenu({ cName: "New", cParent: "File", nPos: 0 })
app.addMenuItem({ cName: "Letter", cParent: "New", cExec:
    "var d = app.newDoc();"});
app.addMenuItem({ cName: "A4", cParent: "New", cExec:
    "app.newDoc(420,595)"});
app.addMenuItem({ cName: "Custom...", cParent: "New", cExec:
    "var nWidth = app.response({ cQuestion: 'Enter Width in Points',\
        cTitle: 'Custom Page Size'});"
    +"if (nWidth == null) nWidth = 612;"
    +"var nHeight = app.response({ cQuestion: 'Enter Height in Points',\
        cTitle: 'Custom Page Size'});"
    +"if (nHeight == null) nHeight = 792;"
    +"app.newDoc({ nWidth: nWidth, nHeight: nHeight })"});
```

The code is a little incomplete. In the case of the "Custom" menu item, additional lines can be inserted to prevent the user from entering the empty string, or a value too small or too large. See the “General Implementation Limits", page 546, of the PDF Reference for the current limitations.

Security *: This method can only be executed during batch, console or menu events. See the Event Object for a discussion of Acrobat JavaScript events.*

openDoc

5.0	🖫		

Parameters: cPath, [oDoc]
Returns: Doc object

This method opens the PDF document specified by *cPath*; the return value is the Doc Object of the document opened by this method, which can, in turn, be used by the script to call methods, or to get or set properties in the newly opened document.

cPath is a device independent path to the document to be opened. The path can be a relative path if the second parameter, *oDoc*, gets passed. The parameter *oDoc* and target document, must both live in the default file system.

oDoc is a Doc Object to use as a base to resolve a relative *cPath*.

Example:

```
/* This example opens another document, inserts a prompting message
   into a text field, sets the focus in the field, then closes the
   current document. */
var otherDoc = app.openDoc("/c/temp/myDoc.pdf");
otherDoc.getField("name").value="Enter your name here: "
otherDoc.getField("name").setFocus();
this.closeDoc();
```

Same example as above, but a relative path is given.

```
var otherDoc = app.openDoc("myDoc.pdf", this);
otherDoc.getField("name").value="Enter your name here: "
otherDoc.getField("name").setFocus();
this.closeDoc();
```

Note: *The current document as well as the target document must be in the default file system.*

See also the closeDoc and setFocus methods.

popUpMenu

5.0			

Parameters: [*cItem* | *Array*] ...
Returns: cItem

This method creates a pop-up menu at the current mouse position. The menu will contain one or more items as specified by the supplied arguments. The method returns the name of the menu item that was selected. The menu item name "-" is reserved to draw a separator line in the menu.

If the argument is a string then it is listed in the menu as a menu item.

If the argument is an array then it appears as a submenu where the first element in the array is the parent menu item. This array can contain further submenus if desired.

```
var cItem = app.popUpMenu("Introduction", "-", "Chapter 1", [ "Chapter 2",
    "Chapter 2 Start", "Chapter 2 Middle", [ "Chapter 2 End", "The End"]]);
app.alert("You chose the \"" + cItem + "\" menu item");
```

Note: *It is important, given the platform dependent interaction between the mouse and the popup menu, to only invoke this method on a* Field/Mouse Down *event. If it is issued at any other time, it may work on some platforms and not on others.*

response

Parameters: cQuestion, [cTitle], [cDefault], [bPassword]
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.

cQuestion is the question to be posed to the user.

cTitle is an optional title to appear in the dialog's window title.

cDefault is a default value for the answer to the question. If not specified, no default value is presented.

bPassword, if *true*, indicates that the user's response should show as asterisks (*) or bullets (•) to mask the response, which might be sensitive information.

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.

Example:

```
var cResponse = app.response({ cQuestion: "How are you today?", cTitle:
    "Your Health Status", cDefault: "Fine" });
if ( cResponse == null)
    app.alert("Thanks for trying anyway.");
else
    app.alert("You responded, \""+cResponse+"\", to the health question.",3);
```

setInterval

5.0			

Parameters: cExpr, nMilliseconds
Returns: timeout object

This method registers an expression to be evaluated each time the specified period elapses (specified in milliseconds). For example, to create a simple color animation on a field called "Color" that changes every second:

```
function DoIt() {
    var f = this.getField("Color");
    var nColor = (timeout.count++ % 10 / 10);
    // Various shades of red.
    var aColor = new Array("RGB", nColor, 0, 0);
    f.fillColor = aColor;
}
timeout = app.setInterval("DoIt()", 1000);
// Add a property to our timeout object so that DoIt() can keep a count going.
timeout.count = 0;
```

See also the clearInterval, setTimeOut and clearTimeOut methods. See the setTimeOut method for an additional example.

setTimeOut

5.0			

Parameters: cExpr, nMilliseconds
Returns: timeout object

This method registers an expression to be evaluated after a specific period elapses (specified in milliseconds).

See also the clearTimeOut, setInterval and clearInterval methods.

Example: This example creates a simple running marquee. Assume there is a text field named "marquee". The default value of this field is "Adobe Acrobat version 5.0 will soon be here!".

```
// Document level JavaScript function
function runMarquee() {
    var f = this.getField("marquee");
    var cStr = f.value;                // get field value
    var aStr = cStr.split("");         // convert to an array
    aStr.push(aStr.shift());           // move first char to last
    cStr = aStr.join("");              // back to string again
    f.value = cStr;                    // put new value in field
}

// Insert a mouse up action into a "Go" button
run = app.setInterval("runMarquee()", 100);
// stop after a minute
stoprun=app.setTimeOut("app.clearInterval(run)",6000);

// Insert a mouse up action into a "Stop" button
try {
      app.clearInterval(run);
```

```
            app.clearTimeOut(stoprun);
      }catch (e) {}
```

Here, we protect the "Stop" button code with a *try/catch*. If the user presses the "Stop" button without having first pressed the "Go", *run* and *stoprun* will be undefined, and the "Stop" code will throw an exception. When the exception is thrown, the catch code is executed. In the above example, code does nothing if the user presses "Stop" first.

Bookmark Object

A bookmark object represents a node in the bookmark tree that appears in the bookmarks navigational panel. Bookmarks are typically used as a “table of contents” allowing the user to navigate quickly to topics of interest.

Bookmark Object Properties

children

5.0			

Type: Array *Access: R*

Returns an array of bookmark objects that are the children of this bookmark in the bookmark tree. See also the parent property and the bookmarkRoot property of the Doc Object.

Example:

```
/* Dump all bookmarks in the document. */
function DumpBookmark(bm, nLevel)
{
    var s = "";
    for (var i = 0; i < nLevel; i++)
        s += " ";
    console.println(s + "+-" + bm.name);
    if (bm.children != null)
        for (var i = 0; i < bm.children.length; i++)
            DumpBookmark(bm.children[i], nLevel + 1);
}

console.clear();
console.show();
console.println("Dumping all bookmarks in the document.");
DumpBookmark(this.bookmarkRoot, 0);
```

color

5.0	💾		⊗

Type: Array *Access: R/W*

This property specifies the color for a bookmark. Values are defined by using 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. See also the style property.

Example: The following fun script will color the top level bookmark red, green and blue.

```
var bm = bookmarkRoot.children[0]
bm.color = color.black;
var C = new Array(1, 0, 0);
var run = app.setInterval(
    'bm.color = ["RGB",C[0],C[1],C[2]]; C.push(C.shift());', 1000);
var stoprun=app.setTimeOut(
    "app.clearInterval(run); bm.color=color.black",12000);
```

Note: *This property is read-only in Acrobat Reader.*

doc

5.0			

Type: Object *Access: R*

This property is the Doc Object that the bookmark resides in.

name

5.0	💾		

Type: String *Access: R/W*

This property is the text string for the bookmark that the user sees in the navigational panel.

open

5.0	💾		

Type: boolean *Access: R/W*

This property determines whether the bookmark shows its children in the navigation panel (open) or whether the children sub-tree is collapsed (closed).

parent

5.0			

Type: object | null *Access: R*

Returns the parent bookmark of the bookmark or *null* if the bookmark is the root bookmark. See also the children property and the bookmarkRoot property of the Doc Object.

style

5.0	💾		⊗

Type: Integer *Access: R/W*

This property specifies the style for the bookmark's font: 0 indicates normal, 1 is italic, 2 is bold, and 3 is bold-italic. See also the color property.

Note: *This property is read-only in Acrobat Reader.*

Bookmark Object Methods

createChild

5.0	💾		

Parameters: cName, [cExpr], [nIndex]
Returns: Nothing

Creates a new child bookmark at the specified location.

cName is the name of the bookmark that the user will see in the navigation panel.

cExpr is an expression to be evaluated whenever the user clicks on the bookmark. Default is no expression. This is equivalent to creating a bookmark with a JavaScript action, see the PDF Reference, "JavaScript Action" for more details.

nIndex is the zero-based index into the children array of the bookmark to create the new child at. Default is zero.

See also the children property and the insertChild and remove methods.

Example:
```
    // Create a bookmark at the top of the bookmark panel that takes you to the
    // next page in the document.
    bookmarkRoot.createChild("Next Page", "this.pageNum++");
```

execute

5.0			

Parameters: None
Returns: Nothing

Executes the action associated with this bookmark. This can have a variety of behaviors. See the PDF Reference, Section 7.5.3, "Actions Types" for a list of common action types.

See also the createChild property.

insertChild

5.0	💾		

Parameters: oBookmark, [nIndex]
Returns: Nothing

Inserts the specified bookmark as a child of this bookmark. If the bookmark already exists in the bookmark tree it is unlinked before inserting it back into the tree. In addition, the insertion is checked for circularities and disallowed if one exists. This prevents users from inserting a bookmark as a child or grandchild of itself.

bookmark is a bookmark object to add as the child of this bookmark.

nIndex is the zero-based index into the children array of the bookmark to insert the new child at. The default is zero.

See also the children property and the createChild and remove methods.

Example:
```
// Take the first child bookmark and move it to the end of the bookmarks.
var bm = bookmarkRoot.children[0];
bookmarkRoot.insertChild(bm, bookmarkRoot.children.length);
```

remove

5.0	💾		

Parameters: None
Returns: Nothing

Removes the bookmark (and all its children) from the bookmark tree.

See also the children property and the createChild and insertChild methods.

Example:

```
// Remove all bookmarks from the document.
bookmarkRoot.remove();
```

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	Equivalent JS	Version
Transparent	color.transparent	["T"]	
Black	color.black	["G" 0]	
White	color.white	["G" 1]	
Red	color.red	["RGB" 1 0 0]	
Green	color.green	["RGB" 0 1 0]	
Blue	color.blue	["RGB" 0 0 1]	
Cyan	color.cyan	["CMYK" 1 0 0 0]	
Magenta	color.magenta	["CMYK" 0 1 0 0]	
Yellow	color.yellow	["CMYK" 0 0 1 0]	
Dark Gray	color.dkGray	["G" 0.25]	4.0
Gray	color.gray	["G" 0.5]	4.0
Light Gray	color.ltGray	["G" 0.75]	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 */
f.target.textColor = event.value < 0 ? color.red : color.black;
```

Color Methods

convert

5.0			

Parameters: color array, cColorspace
Returns: color array

This method converts the colorspace and color values specified by the *color object* to the specified colorspace. Note that conversion to the gray colorspace is lossy in the same fashion that displaying a color TV signal on a black and white TV is lossy. For printing pundits: the conversion of RGB to CMYK does not take into account any black generation or under color removal parameters.

equal

5.0			

Parameters: color array 1, color array 2
Returns: bEqual

This method compares two color arrays to see if they are the same. The routine will perform conversions, if necessary, to determine if the two colors are indeed equal (i.e. ["RGB" 1 1 0] is equal to ["CMYK" 0 0 1 0]).

```
var f = this.getField("foo");
if (color.equal(f.textColor, f.fillColor))
   app.alert("Foreground and background color are the same!");
```

Connection Object

5.0			⊗

The Connection object is the object that encapsulates a session with a database. Connection objects are returned by the newConnection method of the ADBC Object.

Connection methods

newStatement

5.0			⊗

Parameters: None
Returns: a statement object | null

The *newStatement* method is used to create a Statement Object through which database operations may be performed. It returns a Statement object on success or *null* on failure.

Example:
```
// get a connection object, see newConnection
var con = ADBC.newConnection("q32000data");
// now get a statement object
var statement = con.newStatement();
var msg = (statement == null) ?
    "Failed to obtain newStatement!" : "newStatement Object obtained!";
console.println(msg);
```

getTableList

5.0			⊗

Parameters: None
Returns: An array of objects

The *getTableList* method is used to get information about the various tables in a database. It returns an array of TableInfo Objects. This method never fails but may return a zero-length array.

Below is a table that lists the properties of the TableInfo Object returned by the getTableList method.

TableInfo Object			
The TableInfo object contains basic information about a table.			
Property	**Type**	**Access**	**Description**
name	string	R	A string that represents the identifying name of a table. This string could be used in SQL statements to identify the table that the TableInfo object is associated with.
description	string	R	A string that contains database dependent information about the table.

Example

```
/* Assuming we have a Connection object (con) already in hand
   (see newStatement and newConnection), get the list of tables */
var tableInfo = con.getTableList();

console.println("A list of all tables in the database.");
for (var i = 0; i < tableInfo.length; i++) {
    console.println("Table name: "+ tableInfo[i].name);
    console.println("Description: "+ tableInfo[i].description);
}
```

getColumnList

5.0			⊗

Parameters: cName
Returns: An array of columninfo objects

The *getColumnList* method is used to get information about the various columns in the table

cName is the name of the table to get column information about.

Returns an array of ColumnInfo Objects. This method never fails but may return a zero-length array.

Below is a table that lists the properties of the ColumnInfo Object.

ColumnInfo Object			
The ColumnInfo object contains basic information about a column of data.			
Property	**Type**	**Access**	**Description**
name	string	R	A string that represents the identifying name of a column. This string could be used in getColumn calls to identify the column that the ColumnInfo object is associated with.
description	string	R	A string that contains database dependent information about the column.
type	number	R	A numeric value identifying the SQL Type of the data contained in the column associated with the ColumnInfo object.
typeName	string	R	A string identifying the type of the data contained in the column associated with the ColumnInfo object. This is NOT the same information contained in the *type* property as it is a database dependent string representing the data type. This property may give useful information about user defined data types.

Example:

```
/* Assuming we have a Connection object (con) already in hand (see newStatement
and newConnection), get list of all column names */
var con = ADBC.newConnection("q32000data");
var columnInfo = con.getColumnList("sales");
console.println("Column Information");
for (var i = 0; i < columnInfo.length; i++) {
    console.println(columnInfo[i].name);
    console.println("Description: "+ columnInfo[i].description);
}
```

Console Object

			⊗

The Console object is a static object to access the JavaScript console for displaying debug messages and executing JavaScript. It does not function in the Acrobat Reader.

Console Methods

show

Parameters: None
Returns: Nothing

This method shows the console window.

hide

Parameters: None
Returns: Nothing

This method closes the console window.

println

Parameters: cMessage
Returns: Nothing

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

```
// 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.

Data Object

5.0			

The Data object is the representation of an embedded file or data stream that is stored in the document. Data objects are stored in the name tree in the document. See the section on the *Names Tree* and *Embedded File Streams* in the *PDF Reference Manual* for more details.

Data objects can be inserted from the external file system, queried, and extracted. This is a good way to associate and embed source files, meta-data, and other associated data with a document.

See also the Doc Object dataObjects property, the Document createDataObject, exportDataObject, getDataObject, importDataObject, and removeDataObject methods and the Data Object.

Data Object Properties

creationDate

Type: Date *Access: R*

This property is the creation date of the file that was embedded.

modDate

Type: Date *Access: R*

This property is the modification date of the file that was embedded.

MIMEType

Type: String *Access: R*

This property is the MIME type assoicated with this data object.

name

Type: String *Access: R*

This property is the name associated with this data object.

Example:

```
console.println("Dumping all data objects in the document.");
var d = this.dataObjects;
for (var i = 0; i < d.length; i++)
  console.println("DataObject[" + i + "]=" + d[i].name);
```

path

Type: String *Access: R*

This property is the device independent path to the file that was embedded.

size

Type: Number *Access: R*

This property is the size, in bytes, of the uncompressed data object.

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 in a variety of ways. The most common method is through this Object, which usually points to the Doc object of the underlying document. Some properties and methods return *Doc objects*; for example, activeDocs, openDoc, or extractPages all return *Doc objects*.

Example:

```
// Access through "this"
var nPages = this.numPages;      // get number of pages in "this" document
var aCrop = this.getPageBox();   // get the crop box for "this" document

/* Access through return values: From one document, open, modify, save and
** close another. */
var myDoc = app.openDoc("myNovel.pdf", this); // path relative to "this" doc
myDoc.info.Title = "My Great Novel";
myDoc.saveAs(myDoc.path);
myDoc.closeDoc(true);
```

JavaScript is executed as a result of some event. For each event, an Event Object is created. A Doc object can often be accessed through the target property of the *Event object.* The *target* property returns the Field Object that initiated the event for all *mouse, focus, blur, calculate, validate,* and *format* events; *Doc object* access is then through the doc property of Field Object. For all other events, the *target* property points to the *Doc object*.

Example: Access through the Event Object.

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

// In all other events (e.g., batch or console events)
var myDoc = event.target;
```

Doc Object Properties

author

☹	💾		⊗

Type: String *Access: R/W*

This property defines the author of the document. See also the document info property which supersedes this property in later versions.

Note: *This property is read-only in Acrobat Reader.*

---

baseURL

5.0	💾		

Type: String *Access: R/W*

Base URL for the document. The base URL is used to resolve relative web links within the document.

```
console.println("Base URL was " + this.baseURL);
this.baseURL = "http://www.adobe.com/products/";
console.println("Base URL is " + this.baseURL);
```

See also the URL property.

bookmarkRoot

5.0			

Type: object *Access: R*

Returns the root bookmark for the bookmark tree. This bookmark is not displayed to the user; it is simply a programmatic construct to access the tree and access the child bookmarks.

See the Bookmark Object an example of usage.

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.). See also the document info property which supersedes this property in later versions.

creationDate

☹			

Type: Date *Access: R*

This property defines the document's creation date. See also the document info property which supersedes this property in later versions.

dataObjects

5.0			

Type: Array *Access: R*

Returns an array comprised of all the named data objects in the document.

Example:
```
    var d = this.dataObjects;
    for (var i = 0; i < d.length; i++)
       console.println("Data Object[" + i + "]=" + d[i].name);
```

See also the Document dataObjects property, the Document createDataObject, exportDataObject, getDataObject, importDataObject, and removeDataObject methods and the Data Object.

delay

4.0			

Type: Boolean *Access: R/W*

This property can delay 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.

icons

5.0			

Type: Array *Access: R*

Returns an array of named Icon Objects that are present in the document level named icons tree.

Example:

```
if (this.icons == null)
    console.println("No named icons in this doc");
else
```

```
        console.println("There are " + this.icons.length
          + " named icons in this doc");
```

Here is a summary listing of the properties of the icon object

Icon Object

An icon object is an opaque representation of a Form XObject appearance stored in the document. Icons are mostly used with Field Objects of type button.

Property	**Type**	**Access**	**Description**
name	string	R	This property returns the name of the icon. An icon may or may not have a name depending on whether it exists the document level named icons tree.

Example:

```
    // list all named icons
    for (var i = 0; i < this.icons.length; i++) {
        console.println("icon[" + i + "]=" + this.icons[i].name);
    }
```

See also the addIcon, getIcon, importIcon, and removeIcon methods of the Doc Object, the buttonGetIcon, buttonImportIcon, and buttonSetIcon methods of the Field Object, and the Icon Object.

info

5.0			

Type: object *Access: R*

For the Acrobat Reader, this property returns an object with properties from the document information dictionary in the PDF file. Standard entries include *Title*, *Author*, *Subject*, *Keywords*, *Creator*, *Producer*, *CreationDate*, *ModDate*, and *Trapped*. See Table 8.2 on page 475, *Entries in a document information dictionary*, in the PDF Reference, for more details.

Example:

```
    // get title of document
    var docTitle = this.info.Title;
```

5.0	💾		⊗

Type: object *Access: R/W*

Within Acrobat, properties for the *info* object are read/write access and setting a property in this object will dirty the document. Additional document information fields can be added by setting non-standard properties. Writing to any property in this object in the Acrobat Reader will throw an exception.

Example: The following script,

```
this.info.Title = "JavaScript, The Definitive Guide";
this.info.ISBN = "1-56592-234-4";
this.info.PublishDate = new Date();
for (var i in this.info)
    console.println(i + ": "+ this.info[i]);
```

could produce the following output:

```
CreationDate: Mon Jun 12 14:54:09 GMT-0500 (Central Daylight Time) 2000
Producer: Acrobat Distiller 4.05 for Windows
Title: JavaScript, The Definitive Guide
Creator: FrameMaker 5.5.6p145
ModDate: Wed Jun 21 17:07:22 GMT-0500 (Central Daylight Time) 2000
SavedBy: Adobe Acrobat 4.0 Jun 19 2000
PublishDate: Tue Aug 8 10:49:44 GMT-0500 (Central Daylight Time) 2000
ISBN: 1-56592-234-4
```

Note: *Standard entries are case insensitive, that is, doc.info.Keywords is the same as doc.info.keywords.*

---

keywords

☹	💾		⊗

Type: String *Access: R/W*

This property specifies the keywords that describe the document (e.g. "forms", "taxes", "government"). See also the document info property which supersedes this property in later versions.

Note: *This property is read-only in the Acrobat Reader.*

---

layout

5.0			

Type: String *Access: R/W*

Changes the page layout of the current document. Valid values for this property include "SinglePage", "OneColumn", "TwoColumnLeft", and "TwoColumnRight".

modDate

☹			

Type: Date *Access: R*

This property contains the date the document was last modified. See also the document info property which supersedes this property in later versions.

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). See also the document templates property which supersedes this property in later versions.

path

Type: String *Access: R*

This property defines the device independent path of the document, for example /c/Program Files/Adobe/Acrobat 5.0/Help/AcroHelp.pdf. See Section 3.10.1, “File Specification Strings”, page 108, in the PDF Reference for exact syntax of the path.

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 ;

// 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.). See also the document info property which supersedes this property in later versions.

securityHandler

5.0			

Type: String | null *Access: R*

This property returns the name of the security handler used to encrypt the document and returns *null* if there is no security handler (i.e. the document is not encrypted). For example,

```
console.println(this.securityHandler != null ?
    "This document is encrypted with " + this.securityHandler + " security." :
    "This document is unencrypted.");
```

could print out

```
Encrypted with Standard security.
```

if the document was encrypted with the standard security handler.

selectedAnnots

5.0			

Type: Array *Access: R*

This property returns an array of Annot Objects corresponding to every markup annotation the user currently has selected.

Example:
```
// show all the comments of selected annots in console
var aAnnots = this.selectedAnnots;
for (var i=0; i < aAnnots.length; i++)
    console.println(aAnnots[i].contents);
```

See also, getAnnot and getAnnots.

sounds

5.0			

Type: Array *Access: R*

Returns an array comprised of all of the named Sound Objects in the document.

Example:
```
var s = this.sounds;
for (i = 0; i < s.length; i++)
  console.println("Sound[" + i + "]=" + s[i].name);
```

See also the getSound, importSound, and deleteSound methods, and the Sound Object.

spellDictionaryOrder

5.0			

Type: Array *Access: R/W*

This property can be used to access or specify the dictionary array search order for this document. The Spelling plug-in will search for words first in this array and then it will search the dictionaries the user has selected on the Spelling Preference panel. The user's preferred order is available from the spell.dictionaryOrder property. An array of the currently installed dictionaries can be obtained using the spell.dictionaryNames property.

For example, if a user is filling out a Medical Form the form designer may want to specify a Medical dictionary to be searched first before searching the user's preferred order.

subject

☹	🖫		⊗

Type: String *Access: R/W*

This property defines the document's subject. See also the document info property which supersedes this property in later versions.

Note: *This property is read-only in Acrobat Reader.*

templates

5.0			

Type: Array *Access: R*

This property returns an array of all of the template objects in the document.

See also the createTemplate, getTemplate, and removeTemplate methods of the Doc Object, and the Template Object.

title

☹	🖫		⊗

Type: String *Access: R/W*

This property specifies the *title* of the document. See also the document info property which supersedes this property in later versions.

Note: *This property is read-only in Acrobat Reader.*

URL

5.0			

Type: String *Access: R*

This property specifies the document's URL. If the document is local, it will return an URL with a "file:///" scheme. This may be different than the baseURL.

zoom

Type: Integer *Access: R/W*

This property is used to get or set the current page *zoom* level. The values allowed are 8.33% and 1600% specified as an integer.

```
// 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

addAnnot

5.0	💾		⊗

Parameters: object literal
Returns: Annot Object

This method creates an Annot Object having the given *object literal*. An *object literal* is a *generic object* (see Parameter Specification for Methods) which specifies the *properties* of the Annot Object annotation, such as type, rect, and page, to be created.

Example
```
// This example creates a "Square" annotation.
var sqannot = this.addAnnot({type: "Square", page: 0});
```

The above is a minimal example; *sqannot* will be created as annotation of type "Square" located on page 0 (0 based page numbering).

Note: *Properties not specified in the* object literal *are given their default values for the specified* type *of annotation.*

Example:
```
var annot = this.addAnnot({
    page: 0,
    type: "Square",
    rect: [0, 0, 100, 100],
    name: "OnMarketShare",
    author: "A. C. Robat",
    contents: "This section needs revision."
});
```

addField

5.0	💾		⊗

Parameters: cName, cFieldType, nPageNum, oCoords
Returns: object

Creates a new form field and returns a Field Object.

cName is the name of the new field to create. This name can use the dot separator syntax to denote a hierarchy (e.g. name.last will create a parent node, name, and a child node, last).

cFieldType is the type of form field to create. Valid types include "text", "button", "combobox", "listbox", "checkbox", "radiobutton", or "signature".

nPageNum is the zero-based index of the page to add the field to.

oCoords is an array of four numbers in Rotated User Space that specifies the size and placement of the form field. These four numbers are, in this order: upper-left *x*, upper-left *y*, lower-right *x* and lower-right *y* coordinates. See also Field.rect.

Example: The following code might be used in a batch sequence to create a navigational icon on every page of a document, for each document in a selected set of documents.

```
var inch = 72;
for (var p = 0; p < this.numPages; p++) {
    var aRect = this.getPageBox( {nPage: p} );
    aRect[0] += .5*inch;          // position rectangle (.5 inch, .5 inch)
    aRect[2] = aRect[0]+.5*inch;  // from upper left hand corner of page.
    aRect[1] -= .5*inch;          // Make it .5 inch wide
    aRect[3] = aRect[1] - 24;     // and 24 points high

    // now construct button field with a right arrow from ZapfDingbats
    var f = this.addField("NextPage", "button", p, aRect )
    f.setAction("MouseUp", "this.pageNum++");
    f.delay = true;
    f.borderStyle = border.s;
    f.highlight = "push";
    f.textSize = 0;              // auto sized
    f.textColor = color.blue;
    f.fillColor = color.ltGray;
    f.textFont = font.ZapfD
    f.buttonSetCaption("\341")   // a right arrow
    f.delay = false;
}
```

See setAction for another example.

Note: *For developers that use the info panel to obtain the coordinates of the bounding rectangle, be warned, the info panel uses the upper left corner as the origin of its coordinate system. To transform from info space to rotated user space, simply subtract the info space y-coordinate from the onscreen page height.*

addIcon

Parameters: cName, icon object.

5.0	💾		

Returns: Nothing

This function adds a new named Icon Object to the document level icon tree, storing it under the name specified by *cName*.

Example: This example takes an icon already attached to a form button field in the document and assigns a name to it. This name can be used to retrieve the icon object with a getIcon for use in another button, for example.

```
    var f = this.getField("myButton");
    this.addIcon("myButtonIcon", f.buttonGetIcon());
```

See also the icons property and the getIcon, importIcon and removeIcon methods of the Doc Object, the buttonGetIcon, buttonImportIcon, and buttonSetIcon methods of the Field Object, and the Icon Object.

addThumbnails

5.0	💾		⊗

Parameters: [nStart], [nEnd]
Returns: Nothing

Creates thumbnails for the specified pages in the document.

nStart and *nEnd* are zero-based indexes that define an inclusive range of pages. If *nStart* and *nEnd* are not specified then the range of pages is for all pages in the document. If only *nStart* is specified then the range of pages is the single page specified by *nStart*. If only *nEnd* is specified then the range of a pages is 0 to *nEnd*.

See also the removeThumbnails method.

addWeblinks

5.0	💾		⊗

Parameters: [nStart], [nEnd]
Returns: integer

Scans the specified pages looking for instances of text with an http: scheme and converts them into links with URL actions.

nStart and *nEnd* are zero-based indexes that define an inclusive range of pages. If *nStart* and *nEnd* are not specified then the range of pages is for all pages in the document. If only *nStart* is specified then the range of pages is the single page specified by *nStart*. If only *nEnd* is specified then the range of a pages is 0 to *nEnd*.

Returns the number of web links added to the document.

See also the removeWeblinks method.

bringToFront

5.0			

Parameters: None
Returns: Nothing

This method brings the document, open in the Viewer, to the front, if it's not already there.

Example:
```
/* This example searches among the documents open in the Viewer for the
document with a title of "Annual Report" and brings it to the front */
var d = app.activeDocs;
for (var i = 0; i < d.length; i++)
if (d[i].info.Title == "Annual Report")
   d[i].bringToFront();
```

calculateNow

Parameters: None
Returns: Nothing

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

closeDoc

5.0			

Parameters: [bNoSave]
Returns: false

This method closes the document corresponding to the *Doc object*. If *bNoSave* is *false*, the default, the user is prompted to save the document, if it has been modified. If *bNoSave* is *true*, the document is closed without prompting the user and without saving, even if the document has been modified. Because this can cause data loss without user approval, use this feature judiciously.

Note: *It is important to use this method carefully as it is an abrupt change in the document state that can affect any JS executing after the close. Triggering this method off of a Page event or Document event could cause the application to behave strangely.*

createDataObject

5.0	💾		⊗

Parameters: cName, cValue, [cMIMEType]
Returns: nothing

Data objects can be constructed adhoc. This is useful if the data is being created in JavaScript from other sources instead of an external file (e.g. ADBC database calls).

cName is the name to associate with the data object.

cValue is a string containing the data to be embedded.

cMIMEType is the MIME type of the data. Default is "text/plain".

Example:
```
      this.createDataObject("MyData", "This is some data.");
```

See also the Document dataObjects property, the Document createDataObject, exportDataObject, getDataObject, importDataObject, and removeDataObject methods and the Data Object.

createTemplate

5.0	💾	🔑	⊗

Parameters: cName, [nPage]
Returns: template object

Use this function to create a visible template from the specified page.

cName is the name to be associated with this page.

nPage is the zero-based index of the page to operate on. If *nPage* is not specified then *nPage* is the first page in the document.

Returns the newly created Template Object.

See also the templates property, the getTemplate, and removeTemplate methods of the Doc Object, and the Template Object.

Security 🔑: *This method can only be executed during batch, console, or menu events. See the Event Object for a discussion of Acrobat JavaScript events.*

---

deletePages

5.0	💾		⊗

Parameters: [nStart], [nEnd]
Returns: Nothing

Deletes pages from the document.

nStart is the zero-based index of the first page in the range of pages to be deleted. The optional *nEnd* parameter indicates the last page in the range of pages to be deleted. If *nEnd* is not specified then only the page specified by *nStart* is deleted.

Both *nStart* and *nEnd* have a default value of zero, so if *this.deletePages()* is executed, the first page (page 0) will be deleted.

See also the insertPages, extractPages and replacePages methods.

Note: *You cannot delete all pages in a document: there must be at least one page remaining.*

---

deleteSound

5.0	💾		

Parameters: cName
Returns: Nothing

This method deletes the sound object with the specified name from the document.

Example:
```
this.deleteSound("Moo");
```

See also the sounds property, the getSound and importSound methods, and the Sound Object.

exportAsFDF

4.0			⊗

Parameters: [bAllFields], [bNoPassword], [aFields], [bFlags], [cPath]
Returns: Nothing

Use this method to export a FDF file to the local hard drive.

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 *bAllFields* or *bNoPassword*. If this parameter is omitted or is *null* then all fields in the form are exported (again subject to the restrictions of *bAllFields* and *bNoPassword*). This parameter can contain non-terminal field names in order to export an entire subtree of fields, see the example below.

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

5.0	Additions

The optional *cPath*, is a string specifying the device-independent pathname for the FDF. (See Section 3.10.1 of the PDF Reference for a description of the device-independent pathname format.) 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 to export to.

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.

Security 🔑: *If the "cPath" parameter is specified, then this method can only be executed during batch, console, or menu events. See the* Event Object *for a discussion of Acrobat JavaScript events.*

exportAsXFDF

5.0			⊗

Parameters: [bAllFields], [bNoPassword], [aFields], [cPath]
Returns: Nothing

Use this method to export an XFDF file to the local hard drive. XFDF is an XML representation of Acrobat form data. See the Adobe CD Documentation Forms System Implementation Notes for more details.

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 XFDF 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 *bAllFields* or *bNoPassword*. If this parameter is omitted or is *null* then all fields in the form are exported (again subject to the restrictions of *bAllFields* and *bNoPassword*).

The optional *cPath*, is a string specifying the device-independent pathname for the XFDF. (See Section 3.10.1 of the PDF Reference for a description of the device-independent pathname format.) 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. If this parameter is omitted, a dialog will be shown to let the user select the file to export to.

Security 🔑: *If the* cPath *parameter is specified, then this method can only be executed during batch, console or menu events. See the* Event Object *for a discussion of Acrobat JavaScript events.*

exportDataObject

5.0		🔑	⊗

Parameters: cName, [cDIPath]
Returns: nothing

This method extracts the specified data object to an external file.

cName is the name of the data object to extract.

cDIPath is optional and specifies a device independent path to extract the data object to. This path may be absolute or relative to the current document. If *cDIPath* is not specified then the user will be prompted to specify a save location. See *File Specification Strings* in the *PDF Reference Manual* for the exact syntax of the path.

Example:
```
// Prompt the user for a file and location to extract to.
this.extractDataObject("MyData");
// Extract to Foo.xml.
this.extractDataObject("MyData2", "../Foo.xml");
```

See also the Document dataObjects property, the Document createDataObject, exportDataObject, getDataObject, importDataObject, and removeDataObject methods and the Data Object.

Security 🔑: *If the* cDIPath *parameter is specified, then this method can only be executed during batch, console or menu events, or through an external call (e.g. OLE). See the Event Object for a discussion of Acrobat JavaScript events.*

---

extractPages

5.0	💾	🔑	⊗

Parameters: [nStart], [cEnd], [cPath]
Returns: doc object | null

Creates a new document consisting of pages extracted from the current document.

nStart and *nEnd* are zero-based indexes that define an inclusive range of pages in the source document to extract. If *nStart* and *nEnd* are not specified then the range of pages is for all pages in the document. If only *nStart* is specified then the range of pages is the single page specified by *nStart*. If only *nEnd* is specified then the range of pages is 0 to *nEnd*.

cPath specifies the device-independent pathname to save the new document to. See 3.10.1 of the PDF Reference Manual for a description of the device independent path name format. The path name may be relative to the location of the current document. The return value in this case is the *null* object as the new document has been saved to this path and closed.

If *cPath* is not specified then the new document is opened in the viewer and the Doc Object for the new document is returned by the call.

See also the deletePages, insertPages, and replacePages methods.

Example: The following batch sequence would take each of the selected files and extract each page and save the page to a folder with an unique name. This example may be useful in the following setting. Clients one-page bills are produced by an application and placed in a single PDF file. It is desired to separate the pages for distribution and/or separate printing jobs.

```
/* Extract Pages to Folder */
// regular expression acquire the base name of file
var re = /.*\/|\.pdf$/ig;

// filename is the base name of the file Acrobat is working on
var filename = this.path.replace(re,"");

try {
    for (var i = 0; i < this.numPages; i++)
        this.extractPages(
        {
            nStart: i,
            cPath: "/F/temp/"+filename+"_" + i +".pdf"
        });
} catch (e) {
    console.println("Aborted: "+e)
}
```

Security 🔑: *If the* cPath *parameter is specified, then this method can only be executed during batch, console or menu events, or through an external call (e.g. OLE). See the* Event Object *for a discussion of Acrobat JavaScript events.*

flattenPages

5.0	💾		⊗

Parameters: [nStart], [cEnd]
Returns: Nothing

Converts all annotations in the specified page range to page contents.

nStart and *nEnd* are zero-based indices that define an inclusive range in the current document. If only *nStart* is specified, then the page range is the single page specified by *nStart*. If neither parameter is specified, then the page range is all the pages in the current document.

Note: *Great care must be used when using this method. All annotations— including form fields, comments and links— on the specified range of pages will be flattened; they may have appearances, but they will no longer be annotations.*

getAnnot

5.0			

Parameters: nPage, cName
Returns: annot object | null

This method returns the Annot Object on the given page, *nPage*, and with the given name, *cName*. If there is no such annotation with the specified description, the method returns *null*.

Example:

```
var ann = this.getAnnot(0, "OnMarketShare");
if (ann == null)
    console.println("Not Found!")
else
    console.println("Found it! type: " + ann.type);
```

getAnnots

5.0			

Parameters: [nPage], [nSortBy], [bReverse], [nFilterBy]
Returns: array of annot objects

This method takes the criteria set down by the optional parameters and returns an array of Annot Objects satisfying this criteria.

If specified, *nPage* is the zero-based page number that causes the method to return only annotations on the given page. If *nPage* is not specified then the annotations from all pages are retrieved that meet the search criteria.

nSortBy, if specified, is a sort method applied to the array. The following table lists the valid values for *nSortBy*:

Values of *nSortBy*	
Name	**Description**
ANSB_None	default; do not sort; equivalent to leaving *nSortBy* out
ANSB_Page	use the page number as the primary sort criteria
ANSB_Author	use the author as the primary sort criteria
ANSB_ModDate	use the modification date as the primary sort criteria
ANSB_Type	use the annot type as the primary sort criteria

bReverse, if *true*, causes the array to be reverse sorted with respect to *nSortBy*.

nFilterBy, if specified, causes only annots satisfying certain criteria to appear in the resultant list. The valid values for *nFilterBy* are given below:

Values of *bFilterBy*	
Name	**Description**
ANFB_ShouldNone	default; equivalent to leaving *nFilterBy* out
ANFB_ShouldPrint	only include annots that should print
ANFB_ShouldView	only include annots that should view
ANFB_ShouldEdit	only include annots that should be editable
ANFB_ShouldAppearInPanel	only include annots that should appear in the annotations pane
ANFB_ShouldSummarize	only include annots that should be included in a summarization
ANFB_ShouldExport	only include annots that should be included in an export

Example:

```
this.syncAnnotScan();
var annots = this.getAnnots({
    nPage:0,
    nSortBy: ANSB_Author,
    bReverse: true
});
console.show();
console.println("Number of Annots: " + annots.length);
```

```
var msg = "%s in a %s annot said: \"%s\"";
for (var i = 0; i < annots.length; i++)
   console.println(util.printf(msg, annots[i].author, annots[i].type,
       annots[i].contents));
```

See also getAnnot and syncAnnotScan, especially the note that follows that method.

getDataObject

5.0			

Parameters: cName
Returns: data object

This method returns the data object corresponding to the specified name.

cName is the name of the data object to get.

Example:
```
var d = this.getDataObject("MyData");
console.show();
console.clear();
for (var i in d)
   console.println("MyData." + i + "=" + d[i]);
```

See also the Document dataObjects property, the Document createDataObject, exportDataObject, getDataObject, importDataObject, and removeDataObject methods and the Data Object.

getField

Parameters: cName
Returns: Field 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 Object* representing the form field in the PDF document.

Example:
```
// Make a text field multiline and triple its height
var f = this.getField("myText");
var aRect = f.rect;                 // get bounding rectangle
f.multiline = true;                 // make it multiline
var height = aRect[1]-aRect[3];     // calculate height
aRect[3] -= 2* height;              // triple the height of the text field
f.rect = aRect;                     // and make it so
```

getIcon

5.0			

Parameters: cName
Returns: icon object

This function returns an Icon Object associated with the specified name in the document or *null* if no icon of that name exists.

Example: The following is a custom keystroke script from a combobox. The face names of the items in the combobox are the names of some of the icons that populate the document. As the user chooses different items from the combobox, the corresponding icon appears as the button face of the field "myPictures".

```
if (!event.willCommit) {
    var b = this.getField("myPictures");
    var i = this.getIcon(event.change);
    b.buttonSetIcon(i);
}
```

See buttonSetIcon for a more elaborate variation on this example.

See also the icons property and the addIcon, importIcon, and removeIcon methods of the Doc Object, the buttonGetIcon, buttonImportIcon, and buttonSetIcon methods of the Field Object, and the Icon Object.

getNthFieldName

4.0			

Parameters: nIndex
Returns: String

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: nIndex

Returns: String

Use this function to retrieve the name of the *n*th template within in the document.

This method is superceded by use of the Doc Object's templates property and the getTemplate method, and the Template Object object in later versions.

getPageBox

5.0			

Parameters: [cBox], [nPage]
Returns: array of four numbers

cBox can be one of "Art", "Bleed", "BBox", "Crop" (the default), or "Trim". For definitions of these boxes please see Section 8.6.1, "Page Boundaries", page 524, in the PDF Reference. Default is "Crop".

nPage is the zero-based index of the page to operate on. If *nPage* is not specified then *nPage* is the first page in the document.

Returns a rectangle in Rotated User Space that encompasses the named box for the page.

See also the setPageBoxes method.

Example: Get the dimensions of the Media box.

```
var aRect = this.getPageBox("Media");
var width = aRect[2] - aRect[0];
var height = aRect[1] - aRect[3];
console.println("Page 1 has a width of " + width + " and a height of " +
    height);
```

getPageLabel

5.0			

Parameters: [nPage]
Returns: String

Returns page label information for the specified page.

nPage is the zero-based index of the page to operate on. If *nPage* is not specified then *nPage* is the first page in the document.

See also the setPageLabels method for a good example.

getPageNthWord

5.0			

Parameters: [nPage], [nWord], [bStrip]
Returns: String

Returns the *n*th word on the page.

nPage is the zero-based index of the page to operate on. If *nPage* is not specified then *nPage* is the first page in the document.

nWord is the zero-based index of the word to obtain. If *nWord* is not specified then *nWord* is the first word on the page.

bStrip is a boolean indicating that punctuation and whitespace should be removed from the word before returning. Default is *true*.

See also the getNthTemplate, getPageNumWords, and selectPageNthWord methods.

Security 🔑*: This method will throw an exception if the document security is set to prevent content extraction.*

getPageNthWordQuads

5.0		🔑	

Parameters: [nPage], [nWord]
Returns: Array of quads

Returns the quads list for the *n*th word on the page. The quads can be used for constructing text markup annotations, *Underline, StrikeOut, Highlight* and *Squiggly*.

nPage is the zero-based index of the page to operate on. If *nPage* is not specified then *nPage* is the first page in the document.

nWord is the zero-based index of the word to obtain. If *nWord* is not specified then *nWord* is the first word on the page.

See also the B: This method will throw an exception if the document security is set to prevent content extraction. method.

Example: The following example underlines the fifth word on the second page of a document.

```
var annot = this.addAnnot({
    page: 1,
    type: "Underline",
    quads:  this.getPageNthWordQuads(1, 4),
    author: "A. C. Acrobat",
    contents: "Fifth word on second page"
});
```

See checkWord and Highlight, Strikeout, Underline and Squiggle for more interesting examples.

Security *: This method will throw an exception if the document security is set to prevent content extraction.*

getPageNumWords

5.0			

Parameters: [nPage]
Returns: number

Returns the number of words on the page.

nPage is the zero-based index of the page to operate on. If *nPage* is not specified then *nPage* is the first page in the document.

Example:
```
    // count the number of words in a document
    var cnt=0;
    for (var p = 0; p < this.numPages; p++)
        cnt += getPageNumWords(p);
    console.println("There are " + cnt + " words on this page.");
```

See also the getNthTemplate, getPageNthWord, and selectPageNthWord methods.

getPageRotation

5.0			

Parameters: [nPage]
Returns: integer

Gets the rotation of the specified page.

nPage is the zero-based index of the page to operate on. If *nPage* is not specified then *nPage* is the first page in the document.

Returns 0, 90, 180, or 270.

See also the setPageRotations method.

getPageTransition

5.0			

Parameters: [nPage]
Returns: array

Gets the transition of the specified page.

nPage is the zero-based index of the page to operate on. If *nPage* is not specified then *nPage* is the first page in the document.

The routine returns an array of three values: [nDuration, cTransition, nTransDuration].

nDuration is the maximum amount of time the page is displayed before the viewer automatically turns to the next page. A duration of -1 indicates that there is no automatic page turning.

cTransition is the name of the transition to apply to the page. See the application property transitions for a list of valid transitions.

cTransDuration is the duration (in seconds) of the transition effect.

See also the setPageTransitions method.

getSound

5.0			

Parameters: cName
Returns: Sound object

This method returns the sound object corresponding to the specified name.

Example:
```
var s = this.getSound("Moo");
console.println("Playing the " + s.name + " sound.");
s.play();
```

See also the sounds property, the importSound and deleteSound methods, and the Sound Object.

getTemplate

5.0			

Parameters: cName
Returns: template object | null

Use this function to retrieve the named template from the document. Returns *null* if the named template does not exist in the document.

cName is the name of the template to retrieve.

See also the templates property, the createTemplate, and removeTemplate methods of the Doc Object, and the Template Object.

getURL

4.0	💾		

Parameters: cURL, [bAppend]
Returns: Nothing

This method retrieves the specified URL over the internet using a GET.

cURL can either be fully qualified or a relative URL. It *is* permissible to have a query string at the end of the URL.

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 running inside Acrobat, then the URL of the current document is obtained either from the document's Base URL, or from the URL of page #0 (if the document was WebCaptured), or from the file system.

The *bAppend* optional parameter indicates, if*true* (the default), that the resulting page(s) should be appended to the current document. This flag is considered to be *false* if the document is running inside the web browser, the Acrobat Web Capture plug-in is not available, or if the URL is of type "file:///".

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 page 387 of the PDF Reference.

Example: The following example opens a document then goes to a named destination within that document.

```
// open new document
var myNovelDoc = app.openDoc("/c/fiction/myNovel.pdf");
// go to destination in this new doc
myNovelDoc.gotoNamedDest("chapter5");
// close old document
this.closeDoc();
```

importAnFDF

4.0	💾		⊗

Parameters: [cPath]
Returns: Nothing

This method imports the specified FDF file. The *cPath* parameter specifies the device-independent pathname to the FDF file. See Section 3.10.1 of the PDF Reference 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 submitForm method or via the "File->Export->Form Data" menu item. The pathname may be relative to the location of the current document. If this parameter is omitted a dialog will be shown to let the user select the file.

Example: The following code, which is an action of a Page Open event, checks whether a certain function, *ProcResponse*, is already defined, if not, it installs a document level JavaScript, which resides in an FDF file.

```
if (typeof ProcResponse == "undefined")
     this.importAnFDF("myDLJS.fdf");
```

Here, the pathname is a relative one. This technique may be useful for automatically installing document level JavaScripts for PDF files distilled from a PostScript file.

See also importAnXFDF and importTextData.

importAnXFDF

5.0	💾		⊗

Parameters: [cPath]
Returns: Nothing

This method imports the specified XFDF file containing XML form data. The *cPath* parameter specifies the device-independent pathname to the XFDF file. See Section 3.10.1 of the PDF Reference for a description of the device-independent pathname format. 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.

See also importAnFDF and importTextData. For a description of XFDF, please read the *Forms System Implementation Notes* in the Adobe CD Documentation.

importDataObject

5.0	💾	🔑	⊗

Parameters: cName, [cDIPath]
Returns: nothing

This method imports an external file into the document and associates the specified name with the "data object", Data objects can later be extracted or manipulated.

cName is the name to associate with the data object.

cDIPath is optional and specifies a device independent path to a data file on the user's hard drive. This path may be absolute or relative to the current document. If *cDIPath* is not specified then the user will be prompted to locate a data file. See *File Specification Strings* in the *PDF Reference Manual* for the exact syntax of the path.

Example:
```
function DumpDataObjectInfo(dataobj)
{
    for (var i in dataobj)
        console.println(dataobj.name + "[" + i + "]=" + dataobj[i]);
}
// Prompt the user for a data file to embed.
this.importDataObject("MyData");
DumpDataObjectInfo(this.getDataObject("MyData"));
// Embed Foo.xml (found in parent director for this doc).
this.importDataObject("MyData2", "../Foo.xml");
DumpDataObjectInfo(this.getDataObject("MyData2"));
```

See also the Document dataObjects property, the Document createDataObject, exportDataObject, getDataObject, importDataObject, and removeDataObject methods and the Data Object.

Security 🔑: *If the* cDIPath *parameter is specified, then this method can only be executed during batch, console or menu events, or through an external call (e.g. OLE). See the* Event Object *for a discussion of Acrobat JavaScript events.*

importIcon

5.0	💾	🔑	

Parameters: cName, [cDIPath], [nPage]
Returns: Integer

This method imports an icon into the document and associates it with the specified name.

cDIPath is optional and specifies a device independent path to a PDF file on the user's hard drive. This path may be absolute or relative to the current document. *cDIPath* may only be specified in a batch environment or from the console. See Section 3.10.1, "File Specification Strings" in the PDF Reference for the exact syntax of the path.

If *cDIPath* is not specified then the *nPage* parameter is ignored and the user will be prompted to locate a PDF file and browse to a particular page.

nPage is the zero-based index of the page in the PDF file to import as an icon. Default is 0.

This method returns a code indicating whether it was successful or not.

Return Codes

Code	Description
0	No error
1	The user cancelled the dialog
-1	The selected file couldn't be opened
-2	The selected page was invalid

This function is useful to populate a document with a series of named icons for later retrieval. For example, if a user of a document selects a particular state in a listbox, the author may want the picture of the state to appear next to the listbox. In prior versions of the application, this could be done using a number of fields that could be hidden and shown. This is difficult to author, however; instead, the appropriate script might be something like this:

```
var f = this.getField("StateListBox");
var b = this.getField("StateButton");
b.buttonSetIcon(this.getIcon(f.value));
```

This uses a single field to perform the same effect.

A simple user interface can be constructed to add named icons to a document. Assume the existence of two fields: a field called *IconName* which will contain the icon name and a field called *IconAdd* which will add the icon to the document. The mouse up script for *IconAdd* would be:

```
var t = this.getField("IconName");
```

```
    this.importIcon(t.value);
```

The same kind of script can be applied in a batch setting to populate a document with every selected icon file in a folder.

See also the icons property and the addIcon, getIcon and removeIcon methods of the Doc Object, the buttonGetIcon, buttonImportIcon, and buttonSetIcon methods of the Field Object, and the Icon Object.

Security 🔑: *If* cDIPath *is specified, this method can only be executed during batch, console or menu events. See the* Event Object *for a discussion of Acrobat JavaScript events.*

---

importSound

5.0	💾	🔑	

Parameters: cName, [cDIPath]
Returns: Nothing

This method imports a sound into the document and associates the specified name with the sound.

cName is the name to associate with the sound object.

cPath is optional and specifies a device independent path to a sound file on the user's hard drive. This path may be absolute or relative to the current document. If *cPath* is not specified then the user will be prompted to locate a sound file. See Section 3.10.1, "File Specification Strings", in the PDF Reference for the exact syntax of the path.

Example:
```
    this.importSound("Moo");
    this.getSound("Moo").play();
    this.importSound("Moof", "./moof.wav");
    this.getSound("Moof").play();
```

See also the sounds property, the getSound, and deleteSound methods, and the Sound Object.

Security 🔑: *If* cDIPath *is specified, this method can only be executed during batch, console, or menu events. See the* Event Object *for a discussion of Acrobat JavaScript events.*

---

importTextData

5.0	💾		⊗

Parameters: [cPath], [nRow]
Returns: Nothing

This method imports a row of data from a text file. Each row must be *tab delimited*. The entries in the first row of the text file are the column names of the tab delimited data. These names are also field names for text fields present in the PDF file. The data row numbers are 0-based; i.e., the first row of data is row zero (this does not include the column name row). When a row of data is imported, each column datum becomes the field value of the field that corresponds to the column to which the data belongs.

cPath is a relative device independent path to the text file. If not specified, the user is prompted to locate the text data file.

nRow is the zero-based index of the row of the data to import not counting the header row. If not specified, the user is prompted to select the row to import.

Example: Suppose there are text fields named "First", "Middle" and "Last", and there is also a data file, the first row of which consists of the three strings, *First, Middle* and *Last,* separated by tabs. Suppose there are four additional rows of name data, again separated by tabs.

```
First       Middle      Last
Al          Recount     Gore
George      Dubya       Bush
Alan        Cutrate     Greenspan
Bill        Outgoing    Clinton

// Import the first row of data from "myData.txt".
this.importTextData("/c/data/myData.txt", 0)

/* The following code is a mouse up action for a button. Clicking on the
** button cycles through the text file and populates the three fields
** "First", "Middle" and "Last" with the name data. */
if (typeof cnt == "undefined") cnt = 0;
    this.importTextData("/c/data/textdata.txt", cnt++ % 4)
```

The same functionality can be obtained using the ADBC Object and associated properties and methods. The data file can be a spreadsheet or a database.

insertPages

5.0	💾	🔑	⊗

Parameters: [nPage], cPath, [nStart], [nEnd]

Returns: Nothing

Inserts pages from the source document into the current document.

nPage is the zero-based index of the page to insert the source document pages after. To insert pages before the first page of the document *nPage* can be set to -1.

cPath specifies the device-independent pathname to the PDF file that will provide the inserted pages. See Section 3.10.1 of the PDF Reference for a description of the device-independent pathname format. The pathname may be relative to the location of the current document.

nStart and *nEnd* are zero-based indexes that define an inclusive range of pages in the source document to insert. If *nStart* and *nEnd* are not specified, then the range of pages is for all pages in the document. If only *nStart* is specified then the range of pages is the single page specified by *nStart*. If only *nEnd* is specified then the range of pages is 0 to *nEnd*.

See also the deletePages and replacePages methods.

Security 🔑*: This method can only be executed during batch, console, or menu events. See the Event Object for a discussion of Acrobat JavaScript events.*

---

mailDoc

4.0			⊗

Parameters: [bUI], [cTo], [cCc], [cBcc], [cSubject], [cMsg]
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* (the default) 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 *cMsg* 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: *On Windows, 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], [cMsg]
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 *cMsg* 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: *On Windows, the client machine must have its default mail program configured to be MAPI enabled in order to use this method.*

movePage

5.0	💾		⊗

Parameters: [nPage], [nAfter]
Returns: Nothing

Moves a page within the document.

nPage is the zero-based index of the page to move. Default is 0.

nAfter is the zero-based index of the page to move the page after. To move the page before the first page of the document *nAfter* can be set to -1. Default is the last page in the document.

Example: reverse the pages in the document.

```
for (i = this.numPages - 1; i >= 0; i--)
    this.movePage(i);
```

print

> *Parameters: [bUI], [nStart], [nEnd], [bSilent], [bShrinkToFit], [bPrintAsImage], [bReverse], [bAnnotations]*
> *Returns: Nothing*

Use this function to print all or a specific number of pages of the document.

bUI, if true (the default), will cause a UI to be pre-populated with any parameters supplied and presented to the user to obtain the missing information and confirm the action.

nStart and *nEnd* are zero-based indexes that define an inclusive range of pages. If *nStart* and *nEnd* are not specified then the range of pages is for all pages in the document. If only *nStart* is specified then the range of pages is the single page specified by *nStart*. If only *nEnd* is specified then the range of a pages is 0 to *nEnd*.

If *nStart* and *nEnd* parameters are used, *bUI* must be *false*.

bSilent, if *true*, suppresses the cancel dialog box while the document is printing. Default is *false*.

5.0 Additions

bShrinkToFit, if *true*, the page is shrunk (if necessary) to fit within the imageable area of the printed page. If *false*, it is not. The default is *false*.

bPrintAsImage, if *true*, print pages as an image. The default is *false*.

bReverse, if *true*, print from *nEnd* to *nStart*. The default is *false*.

bAnnotations, if *true* (the default), annotations are printed.

Example:

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

// print a file silently
this.print({bUI: false, bSilent: true, bShrinkToFit: true});
```

removeDataObject

5.0	💾		⊗

Parameters: cName
Returns: nothing

This method deletes the data object corresponding to the specified name from the document.

cName is the name of the data object to remove.

Example:
```
        this.removeDataObject("MyData");
```

See also the Document dataObjects property, the Document createDataObject, exportDataObject, getDataObject, importDataObject, and removeDataObject methods and the Data Object.

removeField

5.0	💾		⊗

Parameters: cName
Returns: Nothing

Removes the field specified by *cName* from the document. If the field appears on more than one page then all representations are removed.

removeIcon

5.0	💾		

Parameters: cName
Returns: Nothing

This function removes the specified named icon from the document.

See also the icons property and the addIcon, getIcon and importIcon methods of the Doc Object, the buttonGetIcon, buttonImportIcon, and buttonSetIcon methods of the Field Object, and the Icon Object.

removeTemplate

5.0	💾	🔑	⊗

Parameters: cName
Returns: Nothing

Use this function to remove the named template from the document.

cName is the name of the template to remove.

See also the templates property, the createDataObject and getSound methods of the Data Object, and the Template Object.

Security *: This method can only be executed during batch or console events. See the* Event Object *for a discussion of Acrobat JavaScript events.*

---

removeThumbnails

5.0	💾		⊗

Parameters: [nStart], [nEnd]
Returns: Nothing

Deletes thumbnails for the specified pages in the document.

nStart and *nEnd* are zero-based indexes that define an inclusive range of pages. If *nStart* and *nEnd* are not specified then the range of pages is for all pages in the document. If only *nStart* is specified then the range of pages is the single page specified by *nStart*. If only *nEnd* is specified then the range of a pages is 0 to *nEnd*.

See also the addThumbnails method.

removeWeblinks

5.0	💾		⊗

Parameters: [nStart], [nEnd]
Returns: integer

Scans the specified pages looking for links with actions to go to a particular URL on the web and deletes them.

nStart and *nEnd* are zero-based indexes that define an inclusive range of pages. If *nStart* and *nEnd* are not specified then the range of pages is for all pages in the document. If only *nStart* is specified then the range of pages is the single page specified by *nStart*. If only *nEnd* is specified then the range of a pages is 0 to *nEnd*.

Returns the number of web links removed from the document.

See also the addWeblinks method.

Note: *This method will only remove weblinks authored in the application using the UI. Web links that are executed via JavaScript (e.g. getURL) are not removed.*

---

replacePages

5.0	💾	🔑	⊗

Parameters: [nPage], cPath, [nStart], [nEnd]
Returns: Nothing

Replaces pages in the current document with pages from the source document.

nPage is the zero-based index of the page to start replacement at. Default is 0.

cPath specifies the device-independent pathname to the PDF file that will provide the replacement pages. See Section 3.10.1 of the PDF Reference for a description of the device-independent pathname format. The pathname may be relative to the location of the current document.

nStart and *nEnd* are zero-based indexes that define an inclusive range of pages in the source document to be used for replacement. If *nStart* and *nEnd* are not specified then the range of pages is for all pages in the document. If only *nStart* is specified then the range of pages is the single page specified by *nStart*. If only *nEnd* is specified then the range of pages is 0 to *nEnd*.

See also the deletePages, extractPages and insertPages methods.

Security 🔑*: This method can only be executed during batch, console, or menu events. See the Event Object for a discussion of Acrobat JavaScript events.*

---

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);
```

Note: *Resetting a field causes it to take on its default value which in the case of text fields is usually blank.*

saveAs

5.0		🔑	⊗

Parameters: cPath
Returns: nothing

This method saves the file to the device independent path specified by the required parameter, *cPath*. The file is not saved in linearized format.

Example: The following code could appear as a batch sequence. Assume there is a PDF file already open containing form files that needs to be populated from a database and saved. Below is an outline of the script:

```
var aDocs = app.activeDocs; // get all active docs
var myForm = aDocs[0];      // assume our file is the only one open in viewer
// code lines to read from a database and populate the form with data
// now save file to a folder; use customerID from database record as name
var row = statement.getRow();
.......
myForm.saveAs("/c/customer/invoices/" + row.customerID + ".pdf");
```

Example: You can use the newDoc and addField methods to dynamically layout a form, then populate it from a database and save.

```
var myDoc = app.newDoc()
// layout some dynamic form fields
// connect to database, populate with data, perhaps from a database
..........
// save the doc and/or print it; print it silently this time to default printer
myDoc.saveAs("/c/customer/invoices/" + row.customerID + ".pdf");
myDoc.closeDoc(true);                    // close the doc, no notification
```

Security 🔑: *This method can only be executed during batch, console, or menu events. See the Event Object for a discussion of Acrobat JavaScript events.*

scroll

Parameters: nX, nY
Returns: Nothing

Use this function to scroll the point on the current page specified by *nX* and *nY* into middle of the current view. These coordinates must be defined in Rotated User Space. Please refer to the PDF Reference, page 126, for more details on the user space coordinate system.

selectPageNthWord

5.0			

Parameters: [nPage], [nWord], [bScroll]
Returns: Nothing

Changes the current page number to *nPage* and selects the specified word on the page.

nPage is the zero-based index of the page to operate on. If *nPage* is not specified then *nPage* is the first page in the document.

nWord is the zero-based index of the word to obtain. If *nWord* is not specified then *nWord* is the first word on the page.

bScroll indicates whether or not to scroll the selected word into the view if it isn't already viewable. Default is true.

See also the getNthTemplate, getPageNthWord and getPageNumWords methods.

setPageBoxes

5.0	💾		⊗

Parameters: [cBox], [nStart], [nEnd], [rBox]
Returns: Nothing

Sets a rectangle that encompasses the named box for the specified pages.

nStart and *nEnd* are zero-based indexes that define an inclusive range of pages in the document to be operated on. If *nStart* and *nEnd* are not specified then the range of pages is for all pages in the document. If only *nStart* is specified then the range of pages is the single page specified by *nStart*.

cBox can be one of "Art", "Bleed", "Crop", "Media", or "Trim". Note that "BBox" is read-only and only supported in getPageBox. For definitions of these boxes please see Section 8.6.1, "Page Boundaries", page 524, in the PDF Reference.

rBox is an array of four numbers in Rotated User Space that the specified box will be set to. If *rBox* is not provided then the specified box is removed.

See also the getPageBox method.

setPageLabels

5.0	💾		⊗

Parameters: [nPage], [aLabel]
Returns: Nothing

This method establishes the numbering scheme for the specified page and all pages following it until the next page with an attached label is encountered.

nPage is the zero-based index that defines the page to be labelled.

If *aLabel* is not supplied, any page numbering for the page and any others up to the next specified label is removed.

When specified, *aLabel* is an array of three items [*cStyle*, *cPrefix*, *nStart*].

cStyle is the style of page numbering and one of "D" (decimal numbering), "R", "r" (roman numbering upper/lower), "A", "a" (alphabetic numbering upper/lower). See the PDF Reference, Section 7.3.1, for the exact definitions of these styles.

cPrefix is a string to prefix to the numeric portion of the page label.

nStart is the ordinal to start numbering the pages at.

Example: 10 pages in the document, label the first 3 with small roman numerals, the next 5 with numbers (starting at 1) and the last 2 with an "Appendix- prefix" and alphabetics.

```
this.setPageLabels(0, [ "r", "", 1]);
this.setPageLabels(3, [ "D", "", 1]);
this.setPageLabels(8, [ "A", "Appendix-", 1]);
var s = this.getPageLabel(0);
for (var i = 1; i < this.numPages; i++)
    s += ", " + this.getPageLabel(i);
console.println(s);
```

Would produce the following output on the console:

```
i, ii, iii, 1, 2, 3, 4, 5, Appendix-A, Appendix-B
```

Example: remove all page labels from a document.

```
for (var i = 0; i < this.numPages; i++) {
    if (i + 1 != this.getPageLabel(i)) {
```

```
                // Page label does not match ordinal page number.
                this.setPageLabels(i);
            }
        }
```

See also the getPageLabel method.

setPageRotations

5.0	💾		⊗

Parameters: [nStart], [nEnd], [nRotate]
Returns: Nothing

Rotates the specified pages in the current document.

nStart and *nEnd* are zero-based indexes that define an inclusive range of pages in the document to be operated on. If *nStart* and *nEnd* are not specified then the range of pages is for all pages in the document. If only *nStart* is specified then the range of pages is the single page specified by *nStart*. If only *nEnd* is specified then the range of pages is 0 to *nEnd*.

nRotate specifies the amount of rotation that should be applied to the target pages. It should be either 0, 90, 180, or 270. If *nRotate* is not specified then *nRotate* is zero.

See also the getPageRotation method.

setPageTransitions

5.0	💾		⊗

Parameters: [nStart], [nEnd], [aTrans]
Returns: Nothing

nStart and *nEnd* are zero-based indexes that define an inclusive range of pages in the document to be operated on. If *nStart* and *nEnd* are not specified then the range of pages is for all pages in the document. If only *nStart* is specified then the range of pages is the single page specified by *nStart*.

If *aTrans* is not present any page transitions for the pages are removed.

The page transition array consists of three values: [nDuration, cTransition, nTransDuration].

nDuration is the maximum amount of time the page is displayed before the viewer automatically turns to the next page. Setting *nDuration* to -1 indicates that automatic page turning should be turned off.

cTransition is the name of the transition to apply to the page. See transitions for a list of valid transitions.

nTransDuration is the duration (in seconds) of the transition effect.

See also the getPageTransition method.

spawnPageFromTemplate

☹	💾		⊗

Parameters: cTemplate, [nPage], [bRename], [bOverlay]
Returns: Nothing

Use this method with a template name, *cTemplate*, 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 (but see parameter *bOverlay*). 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*.

4.0 Addition

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

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

See also the Doc Object's templates property and the createTemplate method and the Template Object's spawn method which supersedes this method in later versions.

Note: *The template feature does not work in Acrobat Reader.*

---

submitForm

Parameters: cURL, [bFDF], [bEmpty], [aFields], [bGet], [bAnnotations], [bXML], [bIncrChanges], [bPDF], [bCanonical], [bExclNonUserAnnots], [bExclFKey], [cPassword]
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*).

4.0 Addition

The optional *bGet* parameter is a boolean that 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. parameters *bFDF*, *bXML* and *bPDF* should all be *false*).

5.0 Additions

The optional *bAnnotations* parameter is a boolean that indicates, if *true*, that the annotations should be included in the submitted FDF or XML. The default is *false*. Only applicable if *bFDF* or *bXML* are *true*.

The optional *bXML* parameter is a boolean that indicates, if *true*, to submit as XML. The default is *false*.

The optional *bIncrChanges* parameter is a boolean that indicates, if *true*, to include in the submitted FDF the incremental changes to the PDF. The default is *false*. Only applicable if *bFDF* is *true*. Not available in the Acrobat Reader.

The optional *bPDF* parameter is a parameter that indicates, if *true*, to submit the complete PDF document itself. The default is *false*. If *bPDF* is *true*, then the only other parameter that is relevant is *cURL*. Not available in the Acrobat Reader.

The optional *bCanonical* parameter is a boolean that indicates, if *true*, to convert any dates being submitted to standard format (i.e. D:YYYYMMDDHHmmSSOHH'mm' see the PDF Reference for more details). The default is *false*.

The optional *bExclNonUserAnnots* parameter is a boolean that indicates, if *true*, to exclude any annotations that are not owned by the current user. The default is *false*.

The optional *bExclFKey* parameter is a boolean that indicates, if *true*, to exclude the "F" key. The default is *false*.

If the FDF needs to be encrypted before getting submitted, then a password, *cPassword*, needs to be provided that will be used to generate the encryption key. Alternatively, a *boolean* can be passed instead: if *cPassword* is *true* (no quotes), then a dialog will be presented to the user requesting the password. This dialog will be skipped, however, if the user has previously (within this Acrobat session) entered a password as he submitted or received an encrypted FDF (in which case that password will be used instead). Regardless of whether an actual password is passed in, or one is requested from the user via dialog, this new password is remembered (within this Acrobat session) for future outgoing or incoming encrypted FDFs. This parameter is only valid if *bFDF* is *true*.

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, false, "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"*.

Example:
```
this.submitForm({
    cURL: "http://myserver/cgi-bin/myscript.cgi#FDF",
    bXML: true
});
```

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).*

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

syncAnnotScan

5.0			

Parameters: None
Returns: Nothing

In order to show or process annotations for the entire document all annotations must have been detected. Normally, a background task runs that examines every page and looks for annotations during idle time as this scan is a time consuming task. Calling this method simply guarantees that all annots will be scanned by the time this method returns.

Example:

```
this.syncAnnotScan();
annots = this.getAnnots({nSortBy:ANSB_Author});
// now, do something with the annotations.
```

The second line of code will not be executed until *syncAnnotScan* returns and this will not occur until the annot scan of the document is completed.

Note: *Much of the code in annots works gracefully even when the full list of annots is not yet acquired by background scanning. In general, you should probably do* syncAnnotScan *if want the entire list of annots.*

See also getAnnots.

Event Object

All JavaScripts are executed as the result of a particular event. Each event has a type and a name.The events detailed here are listed as type/name name pairs.

For each of these events, Acrobat JavaScript creates an *event object*. During the occurrence of each event, this *event object* can be accessed, information about the current state of the event can be obtained and possibly manipulated.

It is important for JavaScript writers to know when these events occur and in what order they are processed. Some methods or properties can only be accessed during certain events; therefore, a knowledge of these events will prove useful.

Event Type/Name Combinations

App/Init

When the Viewer is started, the *Application Initialization Event* occurs. Script files, called Folder Level JavaScripts, are read in from the application and user JavaScript folders. They load in the following order: *Config.js*, *glob.js*, all other files, then any user files.

This event defines the name and type properties for the event object.

This event does not listen to the rc return code.

Batch/Exec

5.0			

A batch event occurs during the processing of each document of a batch sequence. JavaScripts that authored as part of a batch sequence can access the event object upon execution.

This event defines the name, target, and type properties for the event object. The target in this event is the document object.

This event listens to the rc return code. If the return code is set to *false*, the batch sequence is stopped.

Bookmark/Mouse Up

5.0			

This event occurs whenever a user clicks on a bookmark that executes a JavaScript.

This event defines the name, target, and type properties for the event object. The target in this event is the bookmark object that was clicked.

This event does not listen to the rc return code.

Console/Exec

5.0			

A console event occurs whenever a user evaluates a JavaScript in the console.

This event defines the name, and type properties for the event object.

This event does not listen to the rc return code.

Doc/DidPrint

5.0			

This event is triggered after a document has printed.

This event defines the name, target, and type properties for the event object. The target in this event is the document object.

This event does not listen to the rc return code.

Doc/DidSave

5.0			

This event is triggered after a document has been saved.

This event defines the name, target, and type properties for the event object. The target in this event is the document object.

This event does not listen to the rc return code.

Doc/Open

This event is triggered whenever a document is opened. When a document is opened document level script functions are scanned and any exposed scripts are executed.

This event defines the name, target, targetName, and type properties for the event object. The target in this event is the document object.

This event does not listen to the rc return code.

Doc/WillClose

5.0			

This event is triggered before a document is closed.

This event defines the name, target, and type properties for the event object. The target in this event is the document object.

This event does not listen to the rc return code.

Doc/WillPrint

5.0			

This event is triggered before a document is printed.

This event defines the name, target, and type properties for the event object. The target in this event is the document object.

This event does not listen to the rc return code.

Doc/WillSave

5.0			

This event is triggered before a document is saved.

This event defines the name, target, and type properties for the event object. The target in this event is the document object.

This event does not listen to the rc return code.

External/Exec

5.0			

This event is the result of an external access, e.g. through OLE, AppleScript, or loading an FDF.

This event defines the name and type properties for the event object.

This event does not listen to the rc return code.

Field/Blur

4.05			

The *blur* event occurs after all other events just as the field loses focus. This event is generated regardless of whether or not a mouse click is used to deactivate the field (e.g. tab key).

This event defines the modifier, name, shift, target, targetName, type, and value properties for the event object. The target in this event is the field whose validation script is being executed.

This event does not listen to the rc return code.

Field/Calculate

This event is defined when a change in a form requires that all fields that have a calculation script attached to them be executed. All fields that depend on the value of the changed field will now be re-calculated. These fields may in turn generate additional Field/Validate, Field/Blur, and Field/Focus 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->Set Field Calculation Order...* menu item in Adobe Acrobat.

This event defines the name, source, target, targetName, type, and value properties for the event object. The target in this event is the field whose calculation script is being executed.

This event does listen to the rc return code. If the return code is set to *false*, the field's value is not changed. If true, the field takes on the value found in the value property.

Field/Focus

4.05			

The *focus* event occurs after the mouse down but before the mouse up after the field gains the focus. This routine is called whether or not a mouse click is used to activate the field (e.g. tab key) and is the best place to perform processing that must be done before the user can interact with the field.

This event defines the modifier, name, shift, target, targetName, type, and value properties for the event object. The target in this event is the field whose validation script is being executed.

This event does not listen to the rc return code.

Field/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.

This event defines the commitKey, name, target, targetName, type, value, and willCommit properties for the event object. The target in this event is the field whose format script is being executed.

This event does not listen to the rc return code. However, the resulting value property is used as the fields formatted appearance.

Field/Keystroke

The *keystroke* event occurs whenever a user types a keystroke into a *text box* or *combobox* (this includes cut and paste operations), or selects an item in a *combobox* 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 allows the author to specify a *Selection Change* script for listboxes. 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 combobox—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.

This event defines the commitKey, change, changeEx, keyDown, modifier, name, selEnd, selStart, shift, target, targetName, type, value, and willCommit properties for the event object. The target in this event is the field whose keystroke script is being executed.

This event does listen to the rc return code. If set to *false*, the keystroke is ignored. The resulting change property is used as the keystroke if the script desires to replace the keystroke code. The resultant selEnd and selStart properties can change the current text selection in the field.

Field/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.

This event defines the modifier, name, shift, target, targetName, and type properties for the event object. The target in this event is the field whose validation script is being executed.

This event does not listen to the rc return code.

Field/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.

This event defines the modifier, name, shift, target, targetName, and type properties for the event object. The target in this event is the field whose validation script is being executed.

This event does not listen to the rc return code.

Field/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.

This event defines the modifier, name, shift, target, targetName, and type properties for the event object. The target in this event is the field whose validation script is being executed.

This event does not listen to the rc return code.

Field/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 of a form. A *mouse up* event will not occur unless a *mouse down* event has already occurred.

This event defines the modifier, name, shift, target, targetName, and type properties for the event object. The target in this event is the field whose validation script is being executed.

This event does not listen to the rc return code.

Field/Validate

Regardless of the field type, user interaction with a 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.

This event defines the change, changeEx, keyDown, modifier, name, shift, target, targetName, type, and value properties for the event object. The target in this event is the field whose validation script is being executed.

This event does listen to the rc return code. If the return code is set to *false*, the field value is considered to be invalid and the value of the field is unchanged.

Link/Mouse Up

5.0			

This event is triggered when a link containing a JavaScript action is activated by the user.

This event defines the name, target, and type properties for the event object. The target in this event is the document object.

This event does not listen to the rc return code.

Menu/Exec

5.0			

A menu event occurs whenever JavaScript that has been attached to a menu item is executed. In Acrobat 5.0, the user can add a menu item and associate JavaScript actions with it. For example,

```
app.addMenuItem({ cName: "Hello", cParent: "File",
  cExec: "app.alert('Hello',3);", nPos: 0});
```

The script "`app.alert('Hello',3);`" will execute during a *menu event*. There are two ways for this to occur:

1. Through the user interface, the user can click on that menu item and the script will execute; and

2. Programmatically, when app.execMenuItem("Hello") is executed (perhaps, during a mouse up event of a button field), the script will execute.

This event defines the name, target, targetName, and type properties for the event object. The target in this event is the currently active document, if one is open.

This event listens to the rc return code in the case of the enable and marked proc for menu items. A return code of *false* will disable or unmark a menu item. A return code of *true* will Event Processing

Page/Open

4.05			

This event happens whenever a new page is viewed by the user and after page drawing for the page has occurred.

This event defines the name, target, and type properties for the event object. The target in this event is the document object.

This event does not listen to the rc return code.

Page/Close

4.05			

This event happens whenever the page being viewed is no longer the current page. I.e. the user has switched to a new page or closes the document.

This event defines the name, target, and type properties for the event object. The target in this event is the document object.

This event does not listen to the rc return code.

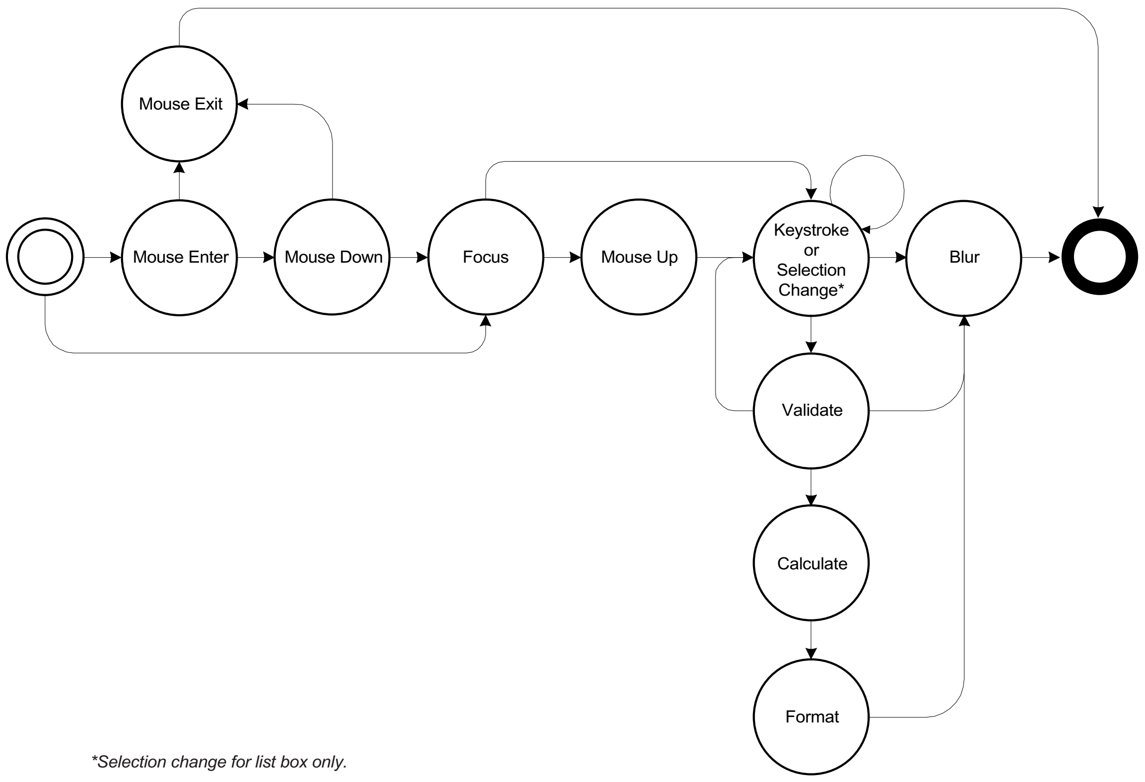
Document Event Processing

When a document is opened, the Doc/Open event occurs: functions are scanned, and any exposed scripts are executed. Next, if the *NeedAppearances* key in the PDF file is set to *true* in the *AcroForm* dictionary, the formatting scripts of all form fields in the document are executed. (See Section 3.6.1 and 7.6.1 of the PDF Reference for more information on the *NeedAppearances* key and the *AcroForm* dictionary.) Finally, the Page/Open event occurs.

Note: *For user's who create PDF files containing form fields with the* NeedAppearances *key set to* true*, be sure to do a "Save As" before posting such files on the Web. Performing a "Save As" on a file will generate the form appearances, which will be saved with the file. This increases the performance of Reader when it loads the file within a Web browser.*

Form Event Processing

The order in which the form events occur is illustrated in the state diagram below. This illustrates certain dependencies that are worth noting, e.g. the Mouse Up event cannot occur if the Focus event did not occur.



*Selection change for list box only.

Event Object Properties

change

Type: String *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. The change may take the form of an individual keystroke or a string of characters (for example if a paste into the field is performed).

changeEx

5.0			

Type: Various *Access: R*

This property contains the *export value* of the change and is available only during a Field/ Keystroke event for *listbox* and *combobox*. (For the former case, the keystroke script, if any, is entered under the *Selection Change* tab in the properties dialog.) The preceding description is true for *listbox*, but there are some additional qualifications for the *combobox*.

For the *combobox*, the changeEx property is *only* available if the pop-up part of it is used, i.e. a selection (with the mouse or the keyboard) is being made from the pop-up. If the combo is *editable* and the user types in an entry, then the above is not applicable, and the Field/Keystroke event behaves just like for a *text field* (i.e. there are no changeEx or keyDown event properties).

commitKey

4.0			

Type: Number *Access: R*

Use this property to determine how a form field will lose 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.");
```

keyDown

5.0			

Type: Boolean *Access: R*

This property is available only during a keystroke event for *listbox* and *combobox*, and is *true* if the arrow keys were used to make a selection, *false* otherwise. The preceding description is true for *listbox*, but there are some additional qualifications for the *combobox*.

For the *combobox*, however, the keyDown property is *only* available if the pop-up part of it is used, i.e. a selection (with the mouse or the keyboard) is being made from the pop-up. If the combo is *editable* and the user types in an entry, then the above is not applicable, and the Field/ Keystroke event behaves just like for a *text field* (i.e. there are no changeEx or keyDown event properties)

modifier

Type: Boolean *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.

name

4.05			

Type: String *Events: all* *Access: R*

This property is the name of the current event as a text string. The type and name together uniquely identify the event. Valid names are Keystroke, Validate, Focus, Blur, Format, Calculate, Mouse Up, Mouse Down, Mouse Enter, Mouse Exit, Open, Close, Will Save, Did Save, Will Print, Did Print, Init, and Exec.

rc

Type: Boolean *Events: Keystroke, Validate, Menu* *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*.

selEnd

Type: Integer *Access: R/W*

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

selStart

Type: Integer *Access: R/W*

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

shift

Type: Boolean *Access: R*

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

source

5.0			

Type: Object *Access: R*

This property contains the field object that triggered the calculation event. This is usually different from the target of event, that is, the field that is being calculated.

target

Type: Object *Access: R*

This property contains the target object that triggered the event. In all mouse, focus, blur, calculate, validate, and format 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.

targetName

Type: String *Access: R*

This property will try return the name of the JavaScript being executed and can be used for debugging purposes to help better identify the code causing exceptions to be thrown. Common values of *targetName* include:

- the folder-level script file name for App:Init events;
- the Doc-level script name for Doc:Exec events;
- the PDF file name being processed for Batch:Exec events;
- the Field name for Field:Exec events.
- the Menu item name for Menu:Exec events.

If there is an identifiable name, Acrobat EScript will report *targetName* in the case an exception is thrown.

Example: The first line of the folder level JavaScript file *conserve.js* has an error in it, when the Acrobat Viewer started, an exception is thrown. The standard message reveals quite clearly the source of the problem.

```
uncaught exception: conserve.js:App:Init:1: Missing required argument for
App.alert. ===> Parameter cMsg.
```

type

5.0			

Type: string *Access: R*

This property is the type of the current event as a text string. The type and name together uniquely identify the event. Valid types are Batch, Console, App, Doc, Page, External, Bookmark, Link, Field, and Menu.

value

Type: Various *Access: R/W*

For the Field/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 comboboxes, this is the *face value,* not the *export value* (see the changeEx property for the export value).

For example, the following JavaScript verifies that the field value is between zero and 100.

```
if (event.value < 0 || event.value > 100) {
      app.beep(0);
      app.alert("Invalid value for field " + event.target.name);
      event.rc = false;
}
```

For a Field/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 Field/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 comboboxes, this is the *face value,* not the *export value* (see the changeEx property for the export value).

For example, the following JavaScript formats the field as a currency type of field.

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

For the Field/Keystroke event, this is the current value of the field. If modifying a text field, for example, this will be the text in the text field before the keystroke is applied.

For Field/Blur and Field/Focus events, this is the current value of the field.

5.0 Addition

When dealing with a *listbox* that allows *multiple selections* (see the Field Object's multipleSelection property), if the field value is an array (that is, there are multiple selections currently selected), *event.value* will return an empty string when getting, and will not accept it when setting. This is reasonable behavior, because *event.value* is primarily meant for Field/Validate and Field/Format events, none of which are really useful for *listboxes*. However, Field/Calculate, Field/Focus and Field/Blur also set *field.value*; and for these cases that value can be used instead.

willCommit

Type: Boolean *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.
```

Field Object

The Field object represents an Acrobat form field (that is, a field created using the Acrobat form tool or the addField method of the Doc Object). 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.

In addition to the examples following the various properties and methods listed in this section, the sections How can I create a form field programmatically? and Quick Reference: Forms have extensive examples and discussion of how to create and control form fields through JavaScript.

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 method 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 = this.getField("Total");
```

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

Field Properties

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

buttonAlignX

5.0	💾		

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

The *buttonAlignX* alignment property defines how space is distributed from the left of the button face with respect to the icon. It is expressed as a percentage between 0 and 100 inclusive. The default value is 50. If the icon is scaled anamorphically (which results in no space differences) then this property is not used.

buttonAlignY

5.0	💾		

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

The *buttonAlignY* alignment property defines how unused space is distributed from the bottom of the button face with respect to the icon. It is expressed as a percentage between 0 and 100 inclusive. The default value is 50. If the icon is scaled anamorphically (which results in no space differences) then this property is not used.

buttonPosition

5.0	💾		

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

The *buttonPosition* property defines how the text and the icon of the button are positioned with respect to each other within the button face. The convenience *position* object defines all of the valid alternatives:

Icon/Text Placement	**Keyword**
Text Only	position.textOnly
Icon Only	position.iconOnly
Icon top, Text bottom	position.iconTextV
Text top, Icon bottom	position.textIconV
Icon left, Text right	position.iconTextH
Text left, Icon right	position.textIconH
Text in Icon (overlaid)	position.overlay

buttonScaleHow

5.0	💾		

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

The *buttonScaleHow* property defines how the icon is scaled (if necessary) to fit inside the button face. The convenience *scaleHow* object defines all of the valid alternatives:

How is Icon Scaled	**Keyword**
Proportionally	scaleHow.proportional
Non-proportionally	scaleHow.anamorphic

buttonScaleWhen

5.0	💾		

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

The *buttonScaleWhen* property defines when an icon is scaled to fit inside the button face. The convenience *scaleWhen* object defines all of the valid alternatives:

When is Icon Scaled	Keyword
Always	scaleWhen.always
Never	scaleWhen.never
If icon is too big	scaleWhen.tooBig
If icon is too small	scaleWhen.tooSmall

calcOrderIndex

	💾		

Type: Integer *Fields: combobox, text* *Access: R/W*

Use this property to change the calculation order of fields in the document. When a computable *Text* or *Combobox* field is added to a document, the field's name is appended to the *calculation order array*. The *calculation order array* determines the order fields are calculated in the document. The *calcOrderIndex* property works similarly to the *Calculate* tab used by the Acrobat 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.

currentValueIndices

5.0	💾		

Type: Integer | Array *Fields: combobox, listbox* *Access: R/W*

Read: This property returns the indices, in the options array, of the strings that are the value of a *listbox* or *combobox* field. These indices are zero-based. If the value of the field is a single string then it returns an integer. Otherwise, it returns an array of integers sorted in ascending order. If the current value of the field is *not* a member of the set of offered choices (as could happen in the case of an editable combobox) then it returns -1.

Example: Given a listbox field that allows multiple section, the following code, which is placed in the "Selection Change" script box, keeps a track of the current selection.

```
if (event.willCommit) {
    var f = event.target;
    var a = f.currentValueIndices;
    if (typeof a == "number")
        console.println("Selection: " + f.getItemAt(a, false));
    else {
        console.println("Selection:");
        for (var i = 0; i < a.length; i ++)
            console.println("   " + f.getItemAt(a[i], false));
    }
}
```

Write: This property is used to set the value of a *listbox* or *combobox*. It accepts either a single integer, or an array of integers, as an argument. If the listbox or combobox will have a single string as its value, then pass an integer, which is the index (zero-based) of that string in the options array. Note that in the case of an editable combobox, if the desired value is not a member of the set of offered choices, then you must set the Field.value property instead. Except for this case, *currentValueIndices* is the preferred way to set the value of a list/combobox. If the listbox (and this possibility exists *only* for listboxes) allows multiple selection, and the desired new value is, in fact, an array of strings, then pass as argument to this property an array containing the indices (which must be sorted in ascending order) of those strings in the options array. Setting the *currentValueIndices* property is the *only* way to invoke multiple selection for a listbox from JavaScript.

Example: The following code, selects the second and fourth (zero-based index values, 1 and 3, respectively) in a listbox.

```
var f = getField("myList");
f.currentValueIndices = [1,3];
```

The ability for a *listbox* to support multiple section can be set through the multipleSelection property.

defaultValue

	💾		

Type: String *Fields: all but button and signature* *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. For *comboboxes* and *listboxes* either an export or a user value can be used to set the default. In the case of a conflict (e.g. the field has an export value and a user value with the same string but these apply to different items in the list of choices), the export value is matched against first.

Example:
```
var f = this.getField("Name");
f.defaultValue = "Enter your name here.";
```

doNotScroll

5.0	💾		

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

When set *true*, the text field does not scroll and the user, therefore, is limited by the rectangular region designed for the field. Setting this property to *true* or *false* corresponds to checking or unchecking the “Do Not Scroll” field in the Options tab of the field.

doNotSpellCheck

5.0	💾		

Type: Boolean *Fields: combobox (editable), text* *Access: R/W*

When set to *true,* spell checking is *not* performed on this editable text field. Setting this property to *true* or *false* corresponds to checking or unchecking the “Do Not Spell Check” field in the Options tab of the field.

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.

Example:
```
// Set the display property
var f = getField("myField");
f.display = display.noPrint;

// Test whether field is hidden on screen and in print
if (f.display == display.hidden)
    console.println("hidden");
```

doc

Type: Object *Fields: all* *Access: R*

This property returns the Doc Object of the document to which the field belongs. Please refer to the Doc Object section for more details.

editable

	💾		

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

Comboboxes 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;
```

exportValues

5.0	💾		

Type: array *Fields: checkbox, radiobutton* *Access: R/W*

This property is the array of export values defined for the field. For radio button fields, this is necessary to make the field work properly as a group with the one button checked at any given time giving its value to the field as a whole. For checkbox fields, unless an export value is specified, the default used when the field is checked is "Yes" (or the corresponding localized string). When it is unchecked, its value is "Off" (this is also true for a radio button field when none of its buttons are checked). This property is expected to be an array of strings and have as many elements as there are annotations in the field. The elements of the array are distributed among the individual annotations comprising the field in the order that they got created (and is unaffected by tab-order).

Example:

```
var d = 40;
var f = this.addField("myRadio", "radiobutton", 0, [200, 510, 210, 500]);
this.addField("myRadio", "radiobutton", 0, [200+d, 510-d, 210+d, 500-d]);
this.addField("myRadio", "radiobutton", 0, [200, 510-2*d, 210, 500-2*d]);
this.addField("myRadio", "radiobutton", 0, [200-d, 510-d, 210-d, 500-d]);
f.strokeColor = color.black;
// now give each radio field an export value
f.exportValues = ["North", "East", "South", "West"];
```

fileSelect

5.0	💾	🔑	

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

The *fileSelect* property of the *text* field, when *true*, sets the “Field is Used for File Selection”, or more simply, the *file select flag*, in the Options tab of the field. This indicates that the value of the field represents a pathname of a file whose contents may be submitted with the form.

The pathname may be entered directly into the field by the user, or the user can browse for the file. (See the browseForFileToSubmit method.)

Note: *The* file select *flag is mutually exclusive with the* multiline, charLimit, password, *and* defaultValue *properties. Also, on the* Macintosh *platform, when setting the file select flag, the field gets treated as* read-only; *hence, the user must browse for the file to enter into the field. (See the* browseForFileToSubmit *method.)*

Security 🔑: *This property can only be set during batch, menu, or console events. See the* Event Object *for a discussion of Acrobat JavaScript events.*

---

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 rectangle of the field. 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 (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 convenience *highlight* object defines all the characteristics that a button can have. 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

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/W*

This property determines how the text is wrapped within the field. If multiline is *false,* the default, the text field can be a single line only; if *true*, multiple lines are allowed and wrap to field boundaries.

multipleSelection

5.0	💾		

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

This property, if *true*, indicates that the field, a listbox, allows multiple selection of the items.

See also Event.type, Field.value, and currentValueIndices.

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);          // displays "myField" in console window
```

numItems

Type: Integer *Fields: combobox, listbox* *Access: R*

The number of items in the *combobox* or *listbox*.

page

5.0			

Type: Integer | Array *Fields: all* *Access: R*

Returns the page number or an array of page numbers of a field. If the field has only one appearance in the document, the *page* property will return an integer representing the (0 based) page number of the page on which the field appears. If the field has multiple appearances, it will return an array of integers, each member of which is a (0 based) page number of an appearance of the field. The order in which the page numbers appear in the array is determined by the order in which the individual widgets of this field were created (and is unaffected by tab-order).

Example:
```
var f = this.getField("myField");
if (typeof f.page == "number")
   console.println("This field only occurs once on page " + f.page);
else
   console.println("This field occurs " + f.page.length + " times);
```

password

	💾		

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

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;
```

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.

rect

	💾		

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

This property takes (in the case of set), or returns (in the case of get) an array of four numbers in Rotated User Space that specifies the size and placement of the form field. These four numbers are, in this order: upper-left *x*, upper-left *y*, lower-right *x* and lower-right *y* coordinates.

Example:

```
// lay out a 2 inch wide text, field just to the right of the field "myText"
var f = this.getField("myText"); // get the field object
var myRect = f.rect;             // and get it’s rectangle

// make needed coordinate adjustments for new field
myRect[0] = f.rect[2];      // the ulx for new = lrx for old
myRect[2] += 2 * 72;        // move over two inches for lry
f = this.addField("myNextText", "text", this.pageNum, myRect);
f.strokeColor = color.black;
```

Example:

```
// move a button field that already exists over 10 points to the right.
var b = this.getField("myButton");
var aRect = b.rect;  // make a copy of b.rect
aRect[0] += 10;      // increment first x-coordinate by 10
aRect[2] += 10;      // increment second x-coordinate by 10
b.rect = aRect;      // update the value of b.rect
```

Note: *Note: Annot Object also has a rect property, but: 1) the coordinates are not in Rotated User Space, and 2) they are in different order than in Field.rect.*

---

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.

Example:

```
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 rectangle of the field 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, radiobutton* *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 table lists 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;
```

submitName

5.0	💾		

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

If nonempty, this property is used during form submission instead of the field name. Only applicable if submitting in HTML format (i.e. urlencoded).

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 *listbox* 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.

Note: *An exception will be thrown if a transparent color space is used to set* textColor.

---

textFont

	💾		

Type: String *Fields: button, combobox, listbox, text* *Access: R/W*

The *textFont* property determines the font that is used when laying out text in a *text field*, *combobox*, *listbox* 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;
```

5.0 Addition

An arbitrary font can be used when laying out a *text field*, *combobox*, *listbox* or *button* by setting the value of *textFont* equal to a string that represents the PostScript name of the font.

Example:

```
// set the font of "myField" to Viva-Regular
var f = this.getField("myField");
f.textFont = "Viva-Regular";
```

Note: *Use of arbitrary fonts as opposed to those listed in the font object will create compatibility problems with older versions of the Viewer.*

---

textSize

	💾		

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

This property determines the text size (in points) that is used in all controls. In *check 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 will allow all text data to still fit in the field’s rectangle should be used.

```
// set the text size of myField to 28 point
this.getField("myField").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", "signature" and "text".

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) * 1.0825;
```

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.

Note: *In the case of signature fields if the field has been signed then a non-null string is returned as the value.*

5.0 Addition

If the field is a *listbox* that accepts *multiple selection* (See multipleSelection), an *array* can be passed to set the *value* of the field. Similarly, *value* will return an array for a *listbox* with multiple values currently selected.

Also, see related notes on type for the Event Object. A related property is valueAsString.

Note: *However, the currentValueIndices property of a* listbox *that has* multiple selections *is still the preferred (and most efficient) way to get and set the value of this type of field.*

---

valueAsString

5.0	💾		

Type: Various *Fields: all but button* *Access: R*

The value property attempts to convert the contents of a field contents to an “accepted format”; for example, a field with a value of "020" is returned as "20", without the prefixed zero after being converted to an integer type. The *valueAsString* returns the value of the field as a JavaScript string; hence, in the example just cited, the value returned would be "020".

Field Methods

browseForFileToSubmit

5.0	💾		

Parameter: None
Returns: Nothing

When invoked on a *text field* for which the *file select* flag is set (checked), this methods pops up a standard file selection dialog. The path entered through the dialog is automatically assigned as the value of the text field

Example: The following code references a text field with the file select flag checked. This is a mouse up action of a button field.

```
var f = this.getField("resumeField");
f.browseForFileToSubmit();
```

The file select flag can be set using the field property exportValues.

Note: *If this method is invoked on a text field in which the exportValues flag is clear (unchecked), an exception is thrown.*

---

buttonGetCaption

5.0			

Parameter: [nFace]
Returns: string

This method returns the caption associated with a button. If the optional parameter *nFace* is specified, either the normal caption (0), down caption (1), or the rollover caption (2) can be retrieved.

Example: This example places pointing arrows to the left and right of the caption on a button field with icon and text.

```
// a mouse enter event
event.target.buttonSetCaption("=> "+event.target.buttonGetCaption()+" <=");

// a mouse exit event
var str = event.target.buttonGetCaption();
str = str.replace(/=> | <=/g, "");
event.target.buttonSetCaption(str);
```

The same effect can be created by having the same icon for rollover, and have the same text, with the arrows inserted, for the rollover caption. This approach would be slower and cause the icon to flicker. The above code gives a very fast and smooth rollover effect because only the caption is changed, not the icon.

buttonGetIcon

5.0			

Parameter: [nFace]
Returns: icon object

This method returns the Icon Object associated with a button. If the optional parameter *nFace* is specified, either the normal icon (0), down icon (1), or the rollover icon (2) can be retrieved. If *nFace* is not specified then it is assumed to be 0.

Example:
```
// Swap two button icons.
var f = this.getField("Button1");
var g = this.getField("Button2");
var temp = f.buttonGetIcon();
f.buttonSetIcon(g.buttonGetIcon());
g.buttonSetIcon(temp);
```

See also buttonSetIcon and buttonImportIcon for more examples of usage.

buttonImportIcon

Parameter: [cPath], [nPage]
Returns: Integer

This method imports the appearance of a button from another PDF file.

5.0 Additions

The *buttonImportIcon* method takes two optional parameters. If no parameters are passed, a dialog is presented that prompts the user to select a PDF file available on the system.

cPath is the device-independent pathname for the file. See Section 3.10.1 of the PDF Reference for a description of the device-independent pathname format.

nPage is the zero based page number from the file to turn into an icon. The default is 0.

This method returns an integer:

Return Codes	
Code	**Description**
1	The user cancelled the dialog
0	No error
-1	The selected file couldn’t be opened
-2	The selected page was invalid

Example: It is assumed that we are connected to an employee information database. We communicate with the database using the ADBC Object and related objects. An employee’s record is requested and three columns are utilized, *FirstName*, *SecondName* and *Picture*. The *Picture* column, from the database, contains a device independent path to the employee’s picture, stored in PDF format. The script might look like this:

```
var f = this.getField("myPicture");
f.buttonSetCaption(row.FirstName.value + " " + row.LastName.value);
if (f.buttonImportIcon(row.Picture.value) != 0)
    f.buttonImportIcon("/F/employee/pdfs/NoPicture.pdf");
```

The button field "myPicture" has been set to display both icon and caption. The employee’s first and last names are concatenated to form the caption for the picture. Note that if there is an error in retrieving the icon, a substitute icon could be imported.

buttonSetCaption

5.0	💾		

Parameter: cCaption, [nFace]
Returns: Nothing

This method sets the caption, *cCaption*, associated with a button. If the optional parameter *nFace* is specified, either the normal caption (0), down caption (1), or the rollover caption (2) can be set. If *nFace* is not specified then it is assumed to be 0. See the properties related to button/caption positioning for details on how the icon and caption actually get placed on the button face.

Example:
```
      var f = this.getField("myButton");
        f.buttonSetCaption("Hello");
```

A related method, buttonGetCaption, for a more extensive example.

buttonSetIcon

5.0	💾		

Parameter: oIcon, [nFace]
Returns: Nothing

This method sets the Icon Object, *oIcon*, associated with a button. If the optional parameter *nFace* is specified, either the normal icon (0), down icon (1), or the rollover icon (2) can be set. If *nFace* is not specified then it is assumed to be 0. See the properties related to button positioning and placement for details on how the icon actually gets rendered on the button face.

Example: This example takes every named icon in the document and creates a listbox using the names. Selecting an item in the listbox sets the icon with that name as the button face of the field "myPictures". What follows is the mouse up action of the button field "myButton".

```
var f = this.getField("myButton")
var aRect = f.rect;
aRect[0] = f.rect[2];         // place listbox relative to the
aRect[2] = f.rect[2] + 144;  // position of "myButton"
var myIcons = new Array();
var l = addField("myIconList", "combobox", 0, aRect);
l.textSize = 14;
l.strokeColor = color.black;
for (var i = 0; i < this.icons.length; i++)
    myIcons[i] = this.icons[i].name;
l.setItems(myIcons);
l.setAction("Keystroke",
'if (!event.willCommit) {\r\t'
+ 'var f = this.getField("myPictures");\r\t'
+ 'var i = this.getIcon(event.change);\r\t'
+ 'f.buttonSetIcon(i);\r'
+ '}');
```

The named icons themselves can be imported into the document through an interactive scheme, such as the example given in addIcon or through a batch sequence.

See also buttonGetIcon.

checkThisBox

5.0			

Parameters: nWidget, [bCheckIt]
Returns: Nothing

This method takes as first parameter the zero-based index, *nWidget*, of an individual *radio button* or *check box* widget for this field. The index is determined by the order in which the individual widgets of this field were created (and is unaffected by tab-order). The second, optional parameter is a boolean, indicating whether the widget in question should be checked. The default is *true*. Only checkboxes can be unchecked, radio buttons cannot. However, you can reset a radio button (e.g. use the doc-level method resetForm) if you really need to uncheck it, but this will only work if its default state is “unchecked” (see method defaultIsChecked).

Example:
```
// check the box "ChkBox"
var f = this.getField("ChkBox");
f.checkThisBox(0,true);
```

Note: *For a set of* radio buttons *that don't have duplicate export values, you can alternatively set the property* value *to the export value of the individual widget that should be checked (or pass an empty string if none should be).*

---

clearItems

	💾		

Parameters: None
Returns: Nothing

This method clears all the values in a *listbox* or *combobox*.

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

defaultIsChecked

5.0			

Parameters: nWidget, [bIsDefaultChecked]
Returns: Boolean

This method takes as first parameter the zero-based index, *nWidget*, of an individual *radio button* or *check box*. The index is determined by the order in which the individual widgets of this field were created (and is unaffected by tab-order). The second, optional parameter is a boolean, indicating whether the field in question should be checked by default (e.g. when the field gets reset). The default is *true*.

Example:
```
      // change the default of "ChkBox" to checked
      var f = this.getField("ChkBox");
      f.defaultIsChecked(0,true);
      this.resetForm(["ChkBox"]);
```

Note: *For a set of* radio buttons *that don't have duplicate export values, you can alternatively set the property* [defaultValue] *to the export value of the individual widget that should be checked by default (or pass an empty string if none should be).*

---

deleteItemAt

4.0	💾		

Parameters: [nIdx]
Returns: Nothing

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

If the current selection is deleted, for the case of a *listbox*, the field *no longer has a current selection*. Having no current selection can an lead to unexpected behavior by this method if is again invoked without parameters on this same field; there is no current selection to delete. It is important, therefore, to make a new selection so that this method will behave as documented. A new selection can be made by using the field property currentValueIndices.

Example:
```
      var a = this.getField("MyListBox");
      a.deleteItemAt();          // delete current item, and...
      a.currentValueIndices = 0; // select top item in list
```

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.

```
// 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: nIdx, [bExportValue]
Returns: export value or item name in a list or combobox

This function gets the internal value of an item in a *combobox* or a *listbox*. The parameter *nIdx* is the index of the item in the list to obtain. If *nIdx* is set to -1, it returns the value of the last item in the list.

5.0 Additions

The value of the optional second parameter, *bExportValue*, determines the type of value returned. If *bExportValue* is set to *true*, the default, *and* the requested item has an export value, the export value is returned; otherwise, the item name is returned. If *bExportValue* is set to *false,* then the item name is always returned.

Example: In the two examples that follow, assume there are three items on "myList": "First", with an export value of 1; "Second", with an export value of 2; and "Third" with no export value.

```
// returns value of first item in list, which is 1
var f = this.getField("myList");
var v = f.getItemAt(0);
```

The following example illustrates the use of the second optional parameter.

```
for (var i=0; i < f.numItems; i++)
    console.println(f.getItemAt(i,true) + ":  " + f.getItemAt(i,false));
```

The output to the console reads:

```
1:        First
2:        Second
```

```
Third:    Third
```

Thus, by putting the second parameter to *false* the item name (face value) can be obtained, even when there is an export value.

insertItemAt

	💾		

Parameters: cName, [cExport], [nIdx]
Returns: Nothing

This function inserts a new item into a *combobox* or a *listbox*.

cName is the *item name*, that is, the name that will appear in the form. This methods works only for a *listbox* or *combobox*.

cExport is the export value of the field when this item is selected. If no export value is provided, the *cName* is used as the export value.

nIdx is the index in the list to insert the item at. If *nIdx* is 0, *cName* is inserted at the top of the list. If *nIdx* is –1, *cName* is inserted at the end of the list. The default value for *nIdx* is 0.

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

isBoxChecked

5.0			

Parameters: nWidget
Returns: Boolean

This method takes as parameter the zero-based index, *nWidget*, of an individual *radio button* or *check box* widget for this field. The index is determined by the order in which the individual widgets of this field were created (and is unaffected by tab-order). This method returns a *Boolean*, indicating whether the widget in question is currently checked.

Example:
```
var f = this.getField("ChkBox");
if(f.isBoxChecked(0))
   app.alert("The Box is Checked");
else
   app.alert("The Box is not Checked");
```

Note: *For a set of* radio buttons *that don't have duplicate export values, you can alternatively get the property* value*, which is equal to the export value of the individual widget that is currently checked (or returns an empty string, if none is).*

isDefaultChecked

5.0			

Parameters: nWidget
Returns: Boolean

This method takes as parameter the zero-based index, *nWidget*, of an individual *radio button* or *check box* widget for this field. The index is determined by the order in which the individual widgets of this field were created (and is unaffected by tab-order). This method returns a *Boolean*, indicating whether the widget in question is checked by default (e.g. when the field gets reset).

Example:
```
var f = this.getField("ChkBox");
if (f.isDefaultChecked(0))
   app.alert("The Default: Checked");
else
   app.alert("The Default: Unchecked");
```

Note: *For a set of* radio buttons *that don't have duplicate export values, you can alternatively get the property;* defaultValue*, which is equal to the export value of the individual widget that is checked by default (or returns an empty string, if none is).*

setAction

	💾		

Parameters: cTrigger, cScript
Returns: Nothing

This method sets the action of the field for a given trigger.

cTrigger: A string that sets the trigger for the action. The values of this first parameter are "MouseUp", "MouseDown", "MouseEnter", "MouseExit", "OnFocus", "OnBlur", "Keystroke", "Validate", "Calculate" and "Format".

cScript: The JavaScript code that is to be executed when the trigger is activated.

Example:
```
var f = this.addField("actionField", "button", 0 , [20, 100, 100, 20]);
f.setAction("MouseUp", "app.beep(0);");
f.fillColor = color.ltGray;
f.buttonSetCaption("Beep");
f.borderStyle = border.b;
f.lineWidth = 3;
f.strokeColor = color.red;
f.highlight = highlight.p;
```

The example following buttonSetIcon is another illustration of this method.

setFocus

4.05			

Parameters: None
Returns: Nothing

This method sets the keyboard focus to this field. This can involve changing the page that the user is currently on and/or causing the view to scroll to a new position in the document. This method automatically brings the document that the field resides in to the front, if it's not already there.

Example:
```
// Search for a certain open doc, then focus in on the field of interest.
var d = app.activeDocs;
for (var i = 0; i < d.length; i++) {
    if (d[i].info.Title == "Response Document") {
        d[i].getField("name").value="Enter your name here: "
        d[i].getField("name").setFocus(); // also brings the doc to front.
        break;
    }
}
```

See also the bringToFront method.

setItems

4.0	💾		

Parameters: oArray
Returns: Nothing

This method sets the list of items for a *combobox* or a *listbox*. The single parameter, *oArray*, necessary to call this method must be an array. Each element in *oArray* 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.

Examples:

```
      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.

signatureInfo

5.0			⊗

Parameters: [oSig]
Returns: Object *Access: R*

Returns a signatureInfo Object that enumerates the properties of the signature. A signature handler may specify additional properties specific to the signature handler. This type of generic object is used when signing as well.

By default this method uses the handler that created the signature to retrieve the *signatureInfo object*. Optionally, an *oSig* parameter, a signature handler object, can be specified; in this case, *oSig* is used to acquire the *signatureInfo object*, even if *oSig* is not an engine for the same handler as was used to create the signature.

All signature handlers define the following properties:

signatureInfo Object

Property	Type	Access	Description
date	date	R	Date the signature was signed in PDF date format.
handlerName	string	R	The language *independent* name of the handler used to apply the signature.
handlerUserName	string	R	The language *dependent* name of the signature handler.

location	string	R/W	User specified location when signing.
name	string	R	Name of the user that signed the field.
numFieldsAltered	number	R	Number of fields altered between the previous signature and 'this' signature.
numFieldsFilledIn	number	R	Number of fields filled in between the previous signature and 'this' signature.
numPagesAltered	number	R	Number of pages altered between the previous signature and 'this' signature.
numRevisions	number	R	Number of revisions in the document.
reason	string	R/W	User specified reason for signing.
revision	number	R	The revision that this signature corresponds to.
status	number	R	The status of the last call to the processor intensive signatureValidate method. See the Return Codes of the status Property table below.
statusText	string	R	The language *dependent* text string suitable for user display denoting the status of the last call to the processor intensive signatureValidate method.

Writable properties can be specified when signing the object. See the signatureSign method.

The following table list the codes returned by the Field.signatureInfo.status, and their meaning.

Return Codes of the status Property

Status Code	**Meaning**
-1	Not a signature field
0	Signature is blank
1	Unknown status
2	Signature is invalid
3	Signature of document is valid, identity of signer could not be verified
4	Signature of document is valid and identity of signer is valid.

Example:
```
var f = this.getField("mySignature"); // get signature field

var sigInfo = f.signatureInfo();
// returns "Acrobat Self-Sign Security"
console.println(sigInfo.handlerName);
```

```
var msg = "Status = " + sigInfo.status +
    " ("+ sigInfo.statusText + ")";
console.println(msg);

console.println(sigInfo.name);
console.println(sigInfo.date);
console.println(sigInfo.location);
console.println(sigInfo.reason);
```

Note: *Some properties of a signature handler, for example,* certificates, *may return a null value until the signature is validated.* Therefore, signatureInfo *should be called again after* signatureValidate. (certificates *is a property of the* PPKLite Signature Handler Object.)

---

signatureSign

5.0	💾	🔑	⊗

Parameters: oSig, [oInfo], [cDIPath]
Returns: bSuccess

Signs the field with the specified signature handler.

oSig is the signature handler object to sign with. See the Security Object's handlers property and the getHandler method. Some signature handlers require that the user be logged in before signing can occur.

oInfo is a generic object specifying the writable properties of the signature. See also the signatureInfo method and the PPKLite Signature Handler Object.

cDIPath is the device independent path to the file to save to following the application of the signature. If *cDIPath* is not specified, the file is saved back to its original location.

The method returns *true* if the signature was applied successfully, *false* otherwise. Before a signature field can be signed, it must be cleared. See the resetForm method of the Doc Object.

The following example signs the "Signature" field with the PPKLite signature handler:

```
var ppklite = security.getHandler("Adobe.PPKLite");    // choose handler
ppklite.login("dps017", "/C/signatures/DPSmith.apf");  // login
var f = this.getField("mySignature");                  // get signature field
this.resetForm(["mySignature"]);                       // clear it, and ...
f.signatureSign(ppklite,                               // sign it
  { password: "dps017",                               // provide password
    location: "San Jose, CA",                         // ... see note below
```

```
        reason: "I am approving this document",
        contactInfo: "dpsmith@adobe.com",
        appearance: "Fancy"});
```

See the getHandler and login methods from the PPKLite Signature Handler Object.

Note: *In the above example, a password was provided. This may or may not have been necessary depending whether the* Password Timeout *had expired. The* Password Timeout *can be set through the user interface (Tools > Self-Sign Security > Users Settings) or, programmatically, by the* setPasswordTimeout *method.*

Security *: This method can only be executed during batch, console, menu, or application initialization events. See the* Event Object *for a discussion of Acrobat JavaScript events.*

---

signatureValidate

5.0			⊗

Parameters: [oSig]
Returns: nStatus

Returns the validity status of the signature. This routine can be computationally expensive and take a significant amount of time depending on the signature handler used to sign the signature. The return value is described in signatureInfo.status. See also the signatureInfo.statusText property.

By default this method uses the handler that created the signature by default. Optionally, the *oSig* parameter, a signature handler object, can be specified. In this case, *oSig* is used to validate the signature even if *oSig* is not an engine for the same handler as was used to create the signature.

If the original handler used to create the signature is not installed, *oSig* can be used to validate the signature.

If *oSig* is from same handler as was used to create signature then different logon contexts for validating can be used.

Example:
```
    var f = this.getField("mySignature") // get signature field
    var sigInfo = f.signatureInfo();
    if (f.signatureValidate())
          if (sigInfo.status < 3)
              var msg = "Signature not valid!  " + sigInfo.statusText;
          else
              var msg  = "Signature valid!  " + sigInfo.statusText;
    else
        var  msg = "Validation failed!";
    app.alert(msg);
```

FullScreen Object

5.0			

The FullScreen object is the interface to fullscreen (presentation mode) preferences and properties. To acquire a FullScreen object, use the fs property of the App Object .

FullScreen Properties

backgroundColor

Type: color array *Access: R/W*

This property determines the background color of the screen in full screen mode. See Color Arrays for more details.

Example
```
app.fs.backgroundColor = color.ltGray;
```

clickAdvances

Type: Boolean *Access: R/W*

This property determines whether or not a mouse click anywhere on the page will cause the viewer to advance one page.

cursor

Type: Number *Access: R/W*

This property determines the behavior of the mouse pointer in full screen mode. The convenience *cursor* object defines all the valid cursor behaviors:

Cursor Behavior	Keyword
Always hidden	cursor.hidden
Hidden after delay	cursor.delay
Visible	cursor.visible

Example:
```
app.fs.cursor = cursor.visible;
```

defaultTransition

Type: Number *Access: R/W*

This property determines the default transition to use when advancing pages in full screen mode. See the transitions property for a valid list of transition names.

Example: Put document into presentation mode

```
app.fs.defaultTransition = "WipeDown";
app.fs.isFullScreen = true;
```

Note: No Transition *is equivalent to setting* app.fs.defaultTransition = `"";`

---

escapeExits

Type: Boolean *Access: R/W*

This property determines whether or not the escape key can be used to exit full screen mode.

isFullScreen

Type: Boolean *Access: R/W*

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

Example:
```
app.fs.isFullScreen = true;
```

In the above example, the Adobe Acrobat viewer is set to fullscreen mode when *isFullScreen* is set to *true*. If *isFullScreen* was *false* then the 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.

Fullscreen only works if there are documents open in the Acrobat viewer window.

See defaultTransition for an example.

Note: *A PDF document being viewed from within a web browser* cannot *be put into fullscreen mode.*

---

loop

Type: Boolean *Access: R/W*

This property determines whether or not the document will loop around to the beginning of the document in response to a page advance (mouse click, keyboard, and/or timer generated) in full screen mode.

timeDelay

Type: Number *Access: R/W*

This property determines the default number of seconds before the page automatically advances in full screen mode. See useTimer to activate/deactivate automatic page turning.

Example:
```
app.fs.timeDelay = 5;        // delay 5 seconds
app.fs.useTimer = true;      // activate automatic page turning
app.fs.usePageTiming = true; // allow page override
app.fs.isFullScreen = true;  // go into fullscreen
```

transitions

Type: Array *Access: R*

This property returns an array of strings representing valid transition names implemented in the viewer. The script:

```
console.println("[" + app.fs.transitions + "]");
```

would produce the following results:

```
[Replace,WipeRight,WipeLeft,WipeDown,WipeUp,SplitHorizontalIn,
SplitHorizontalOut,SplitVerticalIn,SplitVerticalOut,BlindsHorizontal,
BlindsVertical,BoxIn,BoxOut,GlitterRight,GlitterDown,GlitterRightDown,
Dissolve,Random]
```

Note: No Transition *is equivalent to setting* app.fs.defaultTransition = `"".`

See also defaultTransition.

usePageTiming

Type: Boolean *Access: R/W*

This property determines whether or not automatic page turning will respect the values specified for individual pages in full screen mode. Programmatically, transition properties of individual pages can be set using setPageTransitions

useTimer

Type: Boolean *Access: R/W*

This property determines whether or not automatic page turning is enabled in full screen mode. See timeDelay to set the default time interval before proceeding to the next page.

Global Object

The Global object is a static JavaScript object that allows you to share data *between* documents and to 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 *subscribe* mechanism. This mechanism gives you the ability to monitor global data variables and report their value changes across documents.

Global Object Properties

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 "radius" and to allow all document scripts to have access to this variable, a script would simply define it as

```
global.radius = 8;
```

The global variable "radius" is now known across documents throughout the *current* viewer session. To clarify further, suppose two files, A.pdf and B.pdf, are open in the Viewer. Assume it is in A.pdf that the above declaration is made. From within file A.pdf *or* B.pdf, you can then calculate the volume of a sphere whose radius is global.radius

```
var V = (4/3) * Math.PI * Math.pow(global.radius, 3);
```

and obtain the same result 2144.66058. If the value of *global.radius* is changed and the above script is executed again, the value of *V* will be changed accordingly.

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 Core JavaScript 1.4 Documentation. For example, to remove the property "radius" from the global object, call the following script:

```
delete global.radius
```

Global Object Methods

setPersistent

	👥		

Parameters: cVariable, bPersist
Returns: Nothing

This method sets *cVariable* to be persistent. It requires that *bPersist* is set *true*. This means the *cVariable* will exist *across invocations* of Acrobat. 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 "radius" property persistent and accessible for other documents you could use:

```
global.radius = 8;                      // declare radius to be global
global.setPersistent("radius", true); // now say it’s persistent
```

The volume calculation, defined above, will now yield the same result across viewer sessions, or until the value of *global.radius* is changed.

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, 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: *The global variables that are persistent are recorded in the* glob.js *file located in the user's folder for* Folder Level JavaScripts *upon application exit and re-loaded at application start.*

---

subscribe

5.0			

Parameters: cVariable, fCallback
Returns: Nothing

This method subscribes to the global property *cVariable*. If this property is changed, even in another document, the function specified by *fCallback* will be called. Multiple subscribers are allowed for a published property.

The *subscribe* methods enables you to *automatically* update one or more fields when the value of the subscribed global variable changes, as the following example attempts to illustrate.

Example: Suppose there are two files, setRadius.pdf and calcVolume.pdf, open in Acrobat or Reader.

- In setRadius.pdf there is a single button with the code: `global.radius = 2;`
- In calcVolumne.pdf there there a Document-Level JavaScript named *subscribe*:

```
// In the Tools > JavaScripts > Document JavaScripts
global.subscribe("radius", RadiusChanged);
```

```
function RadiusChanged(x)                     // callback function
{
    var V = (4/3) * Math.PI * Math.pow(x,3);
    getField("MyVolume").value = V;           // put value in text field
}
```

- Open both files in the Viewer, now, clicking on the button in setRadius.pdf file immediately gives an update in the text field "MyVolume" in calcVolume.pdf of 33.51032 (as determined by *global.radius = 2*).

The syntax of the callback function is as follows:

```
function fCallback(newval) {
// newval is the new value of the global variable you have subscribed to.
< code to process the new value of the global variable >
}
```

Identity Object

5.0		🔑	

The Identity object is a static object that identifies the current user of the application.

Identity Object Properties

Security 🔑 *These properties are only accessible during batch, console, menu, and application initialization events in order to protect the privacy of the user.*

---

corporation

Type: String *Access: R*

This property is the corporation name that the user has entered in the identity preferences panel.

email

Type: String *Access: R*

This property is the email address that the user has entered in the identity preferences panel.

loginName

Type: String *Access: R*

This property is the login name as registered by the operating system.

name

Type: String *Access: R*

This property is the user name that the user entered in the identity preferences panel.

Example:
```
console.println("Your name is " + identity.name);
console.println("Your e-mail is " + identity.email);
```

Index Object

5.0			

The Index object is a non-creatable object returned by various methods of the Search Object.

Index Object Properties

available

Type: Boolean *Access: R*

This property indicates whether or not the index is available for selection and searching. An index may be unavailable if a network connection is down or a CD-ROM is not inserted, or if the index administrator has brought the index down for maintenance purposes.

name

Type: String *Access: R*

This property is the name of the index as specified by the index administrator at indexing time.

Example:
```
// Enumerate all of the indexes and dump their names
for (var i = 0; i < search.indexes.length; i++) {
    console.println("Index[" + i + "] = " + search.indexes[i].name);
}
```

See the indexes property, which returns an array of the index objects currently accessed by the search engine, in the section devoted to the Search Object for a description of this method.

path

Type: String *Access: R*

This property is the device dependent path where the index resides. See Section 3.10.1, “File Specification Strings”, in the PDF Reference for exact syntax of the path.

selected

Type: Boolean *Access: R/W*

This property indicates whether the index is to participate in the search. If *selected* is *true*, the index will be searched as part of the query, if *false* it will not be. Setting or unsetting this property is equivalent to checking the selection status in the index list dialog.

PlugIn Object

5.0			

A plug-in object represents an extension to the viewer that extends its functionality. See also the pluglns method of the App Object.

PlugIn Object Properties

certified

Type: Boolean *Access: R*

If *true* indicates that the plug-in is certified by Adobe. Certified plug-ins have undergone extensive testing to ensure that breaches in application and document security do not occur. The user can configure the viewer to only load certified plug-ins.

Example:
```
var aPlugins = app.plugIns;
var j=0;
for (var i=0; i < aPlugins.length; i++)
    if (!aPlugins[i].certified) j++;
console.println("Report: There are " + j + " uncertified plugins loaded.");
```

loaded

Type: Boolean *Access: R*

If *true* indicates that the plug-in was loaded.

name

Type: String *Access: R*

Returns the name of the plug-in.

Example:
```
// get array of PlugIn Objects
var aPlugins = app.plugIns;
// get number of plugins
var nPlugins = aPlugins.length;
// enumerate names of all plugins
for (var i = 0; i < nPlugins; i++)
          console.println("Plugin \#" + i + " is " + aPlugins[i].name);
```

path

Type: string *Access: R*

This property defines the device independent path to the plug-in. See “File Specification Strings“, Section 3.10.1, in the PDF Reference for the exact syntax of the path.

version

Type: number *Access: R*

Returns the version number of the plug-in. The integral part of the version number indicates the major version, the decimal part indicates the minor and update versions. For example, 5.11 would indicate that the plug-in is major version 5, minor version 1, and update version 1.

PPKLite Signature Handler Object

The *PPKLite Signature Handler Object* exposes the functionality of Acrobat Self-Sign Security to JavaScript.

This object can be obtained by using the getHandler method of the Security Object.

In addition to the standard set of properties for the signatureInfo Object, the *PPKLite Signature Handler* exposes the following:

Property	**Type**	**Access**	**Description**
appearance	string	R/W	Appearance to use when signing this field
contactInfo	string	R/W	User specified contact information for determining trust, e.g. email address or telephone number
certificates	array	R	Array specifying a hierarchy of certificates that identify the signer. The first element is the signer’s certificate and subsequent elements are certificate authorities that issued the certificate in the previous element. For PPKLite, there is only one entry because certificates are self-issued.
password	string	W	Password required to sign a signature field

The certificate object is a generic object comprised of six read-only properties:

Property	**Type**	**Access**	**Description**
binary	string	R	Returns the return the binary certificate as a hex string
issuerDN	object	R	Distinguished name of the issuer of the certificate
MD5Hash	string	R	MD5 hash of the certificate. This provides a unique fingerprint for this certificate.
serialNumber	string	R	In conjunction with the issuerDN uniquely identifies this certificate
SHA1Hash	string	R	SHA1 hash of the certificate. This provides a unique fingerprint for this certificate.
subjectCN	string	R	Common name of the signer
subjectDN	object	R	Distinguished name of the signer

The *subjectDN* and *issuerDN* are themselves RDN objects (Relative Distinguished Name objects) which have the following properties.

RDN Object			
Property	**Type**	**Access**	**Description**
c	string	R	Country
cn	string	R	Common name
o	string	R	Organization name
ou	string	R	Organizational unit

Examples: The following illustrates how to access these properties.

```
var f = this.getField("mySignature"); // uses the ppklite sig handler
var Info = f.signatureInfo();

// some standard signatureInfo properties
console.println("name = " + Info.name);
console.println("reason = " + Info.reason);
console.println("date = " + Info.date);

// additional signatureInfo properties from PPKLite
console.println("contact info = " + Info.contactInfo);

// get the certificate; first (and only) one
var certificate = Info.certificates[0];

// common name of the signer
console.println("subjectCN = " +  certificate.subjectCN);
console.println("serialNumber = " +  certificate.serialNumber);

// now display some information about this the distinguished name of signer
console.println("subjectDN.cn = " + certificate.subjectDN.cn);
console.println("subjectDN.o = " +  certificate.subjectDN.o);
```

PPKLite Object Properties

appearances

5.0			⊗

Type: Array *Access: R*

Returns an array containing the language independent names of the available appearances for the specified signature handler. If the signature handler does not support alternate appearances then it will return a *null* object.

isLoggedIn

5.0			⊗

Type: Boolean *Access: R*

Returns *true* if currently logged into the PPKLite Signature Handler object.

Example:

```
var ppklite = security.getHandler("Adobe.PPKLite", true);
console.println( "Is logged in = " + ppklite.isLoggedIn ); // false
ppklite.login( "dps017", "/C/signatures/DPSmith.apf");
console.println( "Is logged in = " + ppklite.isLoggedIn ); // true
```

loginName

5.0			⊗

Type: String *Access: R*

Name of the user logged into the signature handler. This is an empty string if no one is logged in.

loginPath

5.0			⊗

Type: String *Access: R*

Device independent path to the user's profile file used to login to the signature handler. This is an empty string if no one is logged in.

name

5.0			⊗

Type: String *Access: R*

The *name* property returns the language independent name of the signature handler. This is always "Adobe.PPKLite".

signInvisible

5.0			⊗

Type: Boolean *Access: R*

Indicates that the signature handler is capable of generating invisible signatures.

signVisible

5.0			⊗

Type: Boolean *Access: R*

Indicates that the signature handler is capable of generating visible signatures.

uiName

5.0			⊗

Type: String *Access: R*

Returns the language dependent string for the signature handler. This string is suitable for user interfaces.

PPKLite Object Methods

login

5.0			⊗

Parameters: [cPassword], [cDIPath]
Returns: bSuccess

Logs into the signature handler.

cPassword is the password necessary to access the user profile.

cDIPath is a device independent path to the user's profile file.

Returns *true* if the login succeeded, *false* otherwise.

Example:

```
// use "Adobe.PPKLite" handler engine for the UI
var ppklite = security.getHandler("Adobe.PPKLite");
// login
ppklite.login("dps017", "/C/signatures/DPSmith.apf");
```

```
..... make a signature field and sign it ......
ppklite.logout();
```

See signatureSign for more details about signing a signature field.

logout

5.0			⊗

Parameters: None
Returns: Nothing

Logs out of the signature handler. See login, above.

newUser

5.0			⊗

Parameters: cPassword, cDIPath, oRDN
Returns: Nothing

This method will create a self-sign signature profile file.

cPassword, a required parameter, is the password for the profile to be created

The parameter, *cDIPath*, is the device independent path to the new user profile.

oRDN is a generic object or a RDN object. A RDN (relative distinguished name) is the object containing the issuer or subject name for a certificate. (See RDN Object.) In the RDN or generic object the only field that is *required* to be filled in is *cn*. If *c* is provided then it must be 2 characters, using the ISO 3166 standard (e.g., 'US').

Example:

```
// Third parameter specified as a generic object
var ppklite = security.getHandler("Adobe.PPKLite");
ppklite.newUser( "testpasswd", "/d/temp/FredNewUser.apf",
    { cn: "Fred NewUser", c: "US" } );
```

Example: Once you get a certificate from a verified signature you can do the following to create a new user profile:

```
var f = this.getField( "mySignature" );
f.signatureValidate();
var sigInfo = f.signatureInfo();
var certs = sigInfo.certificates;
var issuerDN = certs[0].issuerDN;
var subjectDN = certs[0].subjectDN;
```

```
// issuerDN and subjectDN have get/set properties cn, o, ou, c
subjectDN.cn = "DP Smith";
ppklite.newUser( "dps017", "/d/profiles/DPSmith.apf", subjectDN );
```

setPasswordTimeout

5.0			⊗

Parameters: cPassword, iTimeout
Returns: Nothing

Sets number of seconds, *iTimeout*, after which password should expire between signatures. Set *iTimeout* to 0 for always expire (i.e. password always required); set *iTimeout* to 0x7FFFFFFF for password to never expire. Note that for new users, timeout is 0 (password always required).

cPassword is the Self-Sign Security password needed to set the timeout value.

setPasswordTimeout will throw an exception if the user has not logged in to the PPKLite Signature Handler

Example: This example logs in to the PPKLite Signature Handler and sets the password timeout to 30 seconds. If the password timeout has expired—30 seconds in this example—the signer must provide a password. The password is not necessary if the password has not timed out.

```
var ppklite= security.getHandler( "Adobe.PPKLite" );
ppklite.login( "dps017", "/d/profiles/DPSmith.apf" );
ppklite.setPasswordTimeout( "dps017", 30 );
```

Report Object

The Report object allows the user to programmatically generate PDF documents suitable for reporting with JavaScript. Use the Report constructor to create a Report object; for example,

```
var rep = new Report();
```

The properties and methods can then be used to write and format a report.

Report Object properties

size

5.0	💾		⊗

Type: Number *Access: R/W*

This property reflects the size of any text created by writeText. It is a multiplier. Text size is determined by multiplying the *size* property by the default size for the given style.

Example:

```
var rep = new Report();
rep.size = 1.2;
rep.writeText("Hello World!");
```

absIndent

5.0	💾		⊗

Type: Number *Access: R/W*

This property reflects the absolute indentation level. It is desirable to use indent/outdent only whenever possible, as those calls correctly handle indentation overflows.

If a report is indented past the middle of the page, the effective indent is set to the middle. Note that Report.divide does a little squiggly bit to indicate that it's been indented too far.

color

5.0	💾		⊗

Type: Color *Access: R/W*

Read and set the color of any text and any divisions written into the report.

Example:

```
var rep = new Report();
rep.size = 1.2;
rep.color = color.blue;
rep.writeText("Hello World!");
```

Text is written to the report with writeText and divisions (horizontal rules) are written using divide.

Report Object Methods

breakPage

5.0	💾		⊗

Parameters: None

Ends the current page and begins a new one.

divide

5.0	💾		⊗

Parameters: [nWidth]

Writes a horizontal rule across the page at the current location with the given width. The rule will go from the current indent level to the right most edge of the bounding box. If the indent level is past the middle of the bounding box, the rule will have a squiggle bit in it to indicate this.

indent

5.0	💾		⊗

Parameters: [nPoints]

Increments the current indentation mark by *nPoints* or the default amount.

If a report is indented past the middle of the page, the effective indent is set to the middle. Note that Report.divide() does a little squiggly bit to indicate that it's been indented too far.

See writeText for an example of usage.

outdent

5.0	💾		⊗

Parameters: [nPoints]

The opposite of indent; i.e. decrements the current indentation mark by *nPoints* or the default amount.

See writeText for an example of usage.

open

5.0	💾		⊗

Parameters: cTitle
Returns: a doc object

Ends report generation, opens the report in Acrobat and returns a Doc Object that can be used to perform additional processing of the report.

Example:
```
var docRep = rep.open("myreport.pdf");
docRep.info.Title = "End of the month report: August 2000";
docRep.info.Subject = "Summary of comments at the August meeting";
```

See writeText for a more complete example.

save

5.0	💾	🔑	⊗

Parameters: cDIPath, [cFS]

Ends report generation and saves the report to the device independent path, *cDIPath*.

Example:
```
rep.save("/c/myReports/myreport.pdf");
```

The optional *cFS* specifies a file system. The only value for *cFS* is "CHTTP"; in this case, the *cDIPath* parameter should be an URL. Note that the second parameter is only relevant if the web server supports WebDAV.

Example:
```
rep.save("http://www.mycompany/reports/myreport.pdf", "CHTTP");
```

Security : *This method can only be executed during batch or console events. See the* Event Object *for a discussion of Acrobat JavaScript events.*

mail

5.0	💾		⊗

Parameters: [bUI], [cTo], [cCc], [cBcc], [cSubject], [cMsg]

Ends report generation and mails the report. Parameters are just like mailDoc.

Report

5.0	💾		⊗

Parameters: [aMedia], [aBBox]

A Constructor. Creates a new report object with the given media and bounding boxes (values are defined in points or 1/72 of an inch). Defaults to a 8.5 x 11 inch media box and a bounding box that is indented .5 inches on all sides from the media box.

writeText

5.0	💾		⊗

Parameters: String

Writes out a block of text to the report. Every call to writeText is guaranteed to begin on a new line at the current indentation mark. *writeText* correctly wraps Roman, CJK, and WGL4 text.

Example:
```
// Get the comments in this document, and sort by author
this.syncAnnotScan();
annots = this.getAnnots({nSortBy: ANSB_Author});

// open a new report
var rep = new Report();

rep.size = 1.2;
rep.color = color.blue;
rep.writeText("Summary of Comments: By Author");
rep.color = color.black;
rep.writeText(" ");
rep.writeText("Number of Comments: " + annots.length);
rep.writeText(" ");

var msg = "\200 page %s: \"%s\"";
var theAuthor = annots[0].author;
rep.writeText(theAuthor);
rep.indent(20);
for (var i=0; i < annots.length; i++) {
   if (theAuthor != annots[i].author) {
       theAuthor = annots[i].author;
       rep.writeText(" ");
       rep.outdent(20);
       rep.writeText(theAuthor);
       rep.indent(20);
   }
rep.writeText(util.printf(msg, 1 + annots[i].page, annots[i].contents));
}

// now open the report
var docRep = rep.open("myreport.pdf");
docRep.info.Title = "End of the month report: August 2000";
docRep.info.Subject = "Summary of comments at the August meeting";
```

See the file *Annots.js* for additional examples of the Report object.

Search Object

5.0			

The Search object is a static object that accesses the functionality provided by the Acrobat Search plug-in. This plug-in must be installed in order to interface with the Search object (see the available property).

A related is the Index Object, which is returned by some of the methods of the Search object.

Search Object Properties

available

Type: Boolean *Access: R*

This property is *true* if the Search plug-in is loaded and query capabilities are possible. A script author should check this boolean before performing a query or other search object manipulation.

Example:
```
// Make sure the search object exists and is available.
if (typeof search != "undefined" && search.available) {
    search.query("Cucumber");
}
```

indexes

Type: Array *Access: R*

This property returns an array of all of the Index Objects currently accessible by the search engine.

Example:
```
// Enumerate all of the indexes and dump their names
for (var i = 0; i < search.indexes.length; i++) {
    console.println("Index[" + i + "]=", search.indexes[i].name);
}
```

matchCase

Type: Boolean *Access: R/W*

This property indicates whether or not the search query is case sensitive. The default is *false*.

maxDocs

Type: Integer *Access: R/W*

This property indicates the maximum number of documents that will be returned as part of the search query. The default is 100 documents.

proximity

Type: Boolean *Access: R/W*

This property indicates whether or not the search query will reflect the proximity of words in the results ranking when performing the search that contains *AND* boolean clauses. The default is *false.* Please see the sections in the Acrobat Online Guide pertaining to Search capabilities for a more thorough discussion of proximity.

refine

Type: Boolean *Access: R/W*

This property indicates whether or not the search query will take the results of the previous query and refine the results based on the next query. The default is *false.* Please see the sections in the Acrobat Online Guide pertaining to Search capabilities for a more thorough discussion of refining queries.

soundex

Type: Boolean *Access: R/W*

This property indicates whether or not the search query will take the sound of words (e.g. MacMillan, McMillan, McMilon) into account when performing the search. The default is *false.* Please see the sections in the Acrobat Online Guide pertaining to Search capabilities for a more thorough discussion of soundex.

stem

Type: Boolean *Access: R/W*

This property indicates whether or not the search query will take the stemming of words (e.g. run, runs, running) into account when performing the search. The default is *false.* Please see the sections in the Acrobat Online Guide pertaining to Search capabilities for a more thorough discussion of stemming.

thesaurus

Type: Boolean *Access: R/W*

This property indicates whether or not the search query will find *similar* words. For example, searching for "embellish" might yield "enhanced", "gracefully", or "beautiful". The default is *false.* Please see the sections in the Acrobat Online Guide pertaining to Search capabilities for a more thorough discussion of the thesaurus option.

Search Object Methods

addIndex

5.0	👥		

Parameters: cDIPath, [bSelect]
Returns: Index object

This method adds the index referred to by the path to the list of searchable indexes.

cDIPath specifies a device independent path to an index file on the user's hard drive. See "File Specification Strings", Section 3.10.1, in the PDF Reference for the exact syntax of the path.

The optional parameter *bSelect* indicates whether the index should be selected for searching.

```
// Adds the standard help index for Acrobat to the index list:
search.addIndex("/c/program files/adobe/acrobat 5.0/help/exchhelp.pdx", true);
```

getIndexForPath

Parameters: cDIPath
Returns: Index object

Searchs the index list and returns the Index Object whose path corresponds to the specified path.

cDIPath specifies a device independent path to an index file on the user's hard drive. See "File Specification Strings", Section 3.10.1, in the PDF Reference for the exact syntax of the path.

query

Parameters: cQuery
Returns: Number

This method asks the Search engine to search the list of indexes for the text specified in *cQuery*. It returns the number of documents found by the search. Note that the properties associated with the Search object (i.e. *stem*, *soundex*, *thesaurus*, *proximity*, *refine*, *maxDocs*) may affect the results.

Example:

```
var nDocs = search.query("Acrobat");
app.alert("You found " + nDocs + " documents that match your query.");
```

removeIndex

5.0	👥		

Parameters: index
Returns: Nothing

This method removes the Index Object, *index*, from the index list.

Security Object

5.0		🔑	⊗

The security object is a static JavaScript object which encapsulates the encryption and digital signature capabilities of Acrobat.

Security 🔑*: The methods and properties of the Security Object can only be executed during batch, console, menu, or application initialization events. See the* Event Object *for a discussion of Acrobat JavaScript events.*

Security Object Properties

handlers

Type: Array *Access: R*

Returns an array containing the language independent names of the available signature handlers. See also the getHandler method.

validateSignaturesOnOpen

5.0	👥		

Type: boolean *Access: R/W*

Gets or sets the user level preference that validates signatures when a document is opened.

Security Object Methods

getHandler

Parameters: cName, [bUIEngine]
Returns: Sig Object

Returns the signature handler object specified by *cName*. If the signature handler is not present then a *null* object is returned. See also the handlers property.

The second optional parameter, *bUIEngine*, is boolean that defaults to *false*, if not present. If *true* then *getHandler* returns the existing signature handler engine that is also hooked up to the UI (eg can login via UI). If *false* (default) then returns a new engine. The caller can create as many new engines as desired and each call to *getHandler* will create a new engine; however, there is only one UI engine

Example: This code selects the PPKLite Signature Handler Object

```
// validate signatures on open
security.validateSignaturesOnOpen = true;

// list all available signature handlers
var a = security.handlers;
for (var i = 0; i < a.length; i++)
   console.println("a["+i+"] = "+a[i]);

// use "Adobe.PPKLite" handler engine for the UI
var ppklite = security.getHandler("Adobe.PPKLite", true);
// login
ppklite.login("dps017", "/C/signatures/DPSmith.apf");
```

See also the example following signatureSign for a continuation of this example.

Sound Object

5.0			

The Sound object is the representation of a sound that is stored in the document. The array of all sound objects can be obtained from the Doc.sounds property.

See also the getSound, importSound, and deleteSound methods of the Doc Object.

Sound Object Properties

name

Type: String *Access: R*

This property is the name associated with this sound object.

Example:
```
console.println("Dumping all sound objects in this document.");
var s = this.sounds;
for (var i = 0; i < this.sounds.length; i++)
   console.println("Sound[" + i + "]=" + s[i].name);
```

Sound Object Methods

play

Parameters: None
Returns: Nothing

This method plays the sound asynchronously.

pause

Parameters: None
Returns: Nothing

This method pauses the currently playing sound. If the sound is already paused then the sound play is resumed.

stop

Parameters: None
Returns: Nothing

This method stops the currently playing sound.

Spell Object

5.0			⊗

The JavaScript Spell object allows users to check the spelling of form and annotation text fields and other spelling domains. To be able to use the Spell object, the user must have installed the Acrobat Spelling plug-in and the spelling dictionaries.

Spell Object Properties

available

5.0			⊗

Type: Boolean *Access: R*

This property is *true* if the Spell object is available.

Example:
```
      console.println("Spell checking available: " + spell.available);
```

dictionaryNames

5.0			⊗

Type: Array *Access: R*

This property returns the array of available dictionary names. A subset of this array can be passed to the check, checkText, and checkWord methods, and to the Doc.spellDictionaryOrder property to force the use of a specific dictionary or dictionaries and the order they should be searched.

Depending on the user's installation, valid dictionary names can include "English USA", "German Traditional", "French", "Spanish", "Italian", "English UK", "Swedish", "Danish", "Norwegian", "Dutch", "Portuguese", "Portuguese Brazilian", "French Canadian", "German Swiss", "Norwegian Nynorsk", "Finnish", "Catalan", "Russian", "Ukrainian", "Czech", "Polish", "English UK Legal", "English UK Medical", "German Reformed", "German Old Swiss", "English USA Legal", "English USA Medical", "English USA Sci/Tech", and "English USA Geo/Bio".

dictionaryOrder

5.0			⊗

Type: Array *Access: R*

This property returns the dictionary array search order specified by the user on the Spelling Preferences panel. The Spelling plug-in will search for words first in the Doc.spellDictionaryOrder array if it has been set for the document, and then it will search this array of dictionaries.

domainNames

5.0			⊗

Type: Array *Access: R*

This property returns the array of spelling domains that have been registered with the Spelling plug-in by other plug-ins. A subset of this array can be passed to the check method to limit the scope of the spell check.

Depending on the user's installation, valid domains can include "Everything", "Form Field", "All Form Fields", "Comment", "All Comments".

Spell Object Methods

addDictionary

5.0	👥		⊗

Parameters: cFile, cName, [bShow]
Returns: Boolean

Use this method to add a dictionary to the list of available dictionaries. The required *cFile* parameter is the device independent path to the dictionary files.

The required *cName* parameter will be the dictionary name used in the spelling dialog and can be used as the input parameter to the check, checkText, and checkWord methods.

When the optional *bShow* paramenter is *true*, the default, Spelling will combine the *cName* parameter with "User: " and show that name in all lists and menus. For example if cName is "Test", Spelling will add "User: Test" to all lists and menus. When bShow is *false*, Spelling will not show this custom dictionary in any lists or menus.

The *addDictionary* method returns *true* on success, *false*, otherwise

A dictionary actually consists of four files: DDDxxxxx.hyp, DDDxxxxx.lex, DDDxxxxx.clx, and DDDxxxxx.env. The *cFile* parameter must be the device independent path of the .hyp file. For example, "/c/temp/testdict/TST.hyp". Spelling will look in the parent directory of the TST.hyp file for the other three files. All four file names must start with the same unique 3 characters to associate them with each other, and they must end with the dot three extensions listed above, even on a Macintosh.

Example

```
    /* Get dictionary path and name from the user */
    var dictPath = this.getField("dictPath");
```

```
var dictName = this.getField("dictName");

/* now add this dictionary, if possible */
if ( spell.available ) {
    spell.addDictionary(dictPath.value, dictName.value);
}
```

addWord

5.0	👥		⊗

Parameters: cWord, cName
Returns: Boolean

Use this method to add a new word, *cWord*, to a dictionary. If successful it returns *true*, otherwise, *false*.

The required *cName* parameter is the dictionary name. An array of the currently installed dictionaries can be obtained using the dictionaryNames method.

See also the removeWord method.

Note: *Internally the Spell Check Object will scan the user "Not-A-Word" dictionary and remove the word if it is listed there. Otherwise, the word is added to the user dictionary. The actual dictionary is not modified.*

---

check

5.0			⊗

Parameters: [aDomain], [aDictionary]
Returns: Boolean

This method presents the Spelling dialog to allow the user to correct misspelled words in form fields, annotations, or other objects. This method returns *true* if the user changed or ignored all of the flagged words. When the user dismisses the dialog before checking everything the method returns *false*.

The optional *aDomain* parameter is an array of document objects that should be checked by the Spelling plug-in, for example form fields or comments. When you do not supply an array of domains the "EveryThing" domain will be used. An array of the domains that have been registered can be obtained using the domainNames method.

The optional *aDictionary* parameter is the array of dictionary names that the spell checker should use. The order of the dictionaries in the array is the order the spell checker will use to check for misspelled words. An array of the currently installed dictionaries can be obtained using the dictionaryNames method. When this parameter is omitted the Doc.spellDictionaryOrder list will be searched followed by the spell.dictionaryOrder list.

Example:

```
var dictionaries = ["English USA Medical", "English USA"];
var domains = ["All Form Fields", "All Annotations"];
if (spell.check(domains, dictionaries) )
   console.println("You get an A for spelling.");
else
   console.println("Please spell check this form before you submit.");
```

checkText

5.0			⊗

Parameters: cText, [aDictionary]
Returns: String

This method presents the spelling dialog to allow the user to correct misspelled words in the string *cText*. This method returns the result from the spelling dialog in a new string.

The optional *aDictionary* parameter is the array of dictionary names that the spell checker should use. The order of the dictionaries in the array is the order the spell checker will use to check for misspelled words. An array of the currently installed dictionaries can be obtained using the dictionaryNames method. When this parameter is omitted the Doc.spellDictionaryOrder list will be searched followed by the spell.dictionaryOrder list.

Example:

```
var f = this.getField("Text Box")   /* a form text box */
f.value = spell.checkText(f.value); /* let the user pick the dictionary */
```

checkWord

5.0			⊗

Parameters: cWord, [aDictionary]
Returns: null | Array

This method checks the spelling of a word, *cWord*. If the word is correct a *null* object is returned, otherwise an array of alternative spellings for the unknown word is returned.

The optional *aDictionary* parameter is the array of dictionary names that Acrobat should use. The order of the dictionaries in the array is the order Acrobat will use to check for misspelled words. An array of the currently installed dictionaries can be obtained using the dictionaryNames method. When this parameter is omitted the Doc.spellDictionaryOrder list will be searched followed by the spell.dictionaryOrder list.

Example:

```
var word = "subpinna"; /* misspelling of "subpoena" */
var dictionaries = ["English USA Legal", "English USA"];
var f = this.getField("Alternatives") /* alternative spellings listbox */
f.clearItems();
f.setItems(spell.checkWord(word, dictionaries));
```

Example: The following script goes through the document and marks with a squiggle annot any misspelled word. The contents of the squiggle annot contains the suggested alternative spellings. The script can be executed from the console, as a mouse up action within the document, or as a batch sequence.

```
var ckWord, numWords;
for (var i = 0; i < this.numPages; i++ )
{
    numWords = this.getPageNumWords(i);
    for (var j = 0; j < numWords; j++)
    {
        ckWord = spell.checkWord(this.getPageNthWord(i, j))
        if ( ckWord != null )
        {
           this.addAnnot({
               page: i,
               type: "Squiggly",
               quads: this.getPageNthWordQuads(i, j),
               author: "A. C. Acrobat",
               contents: ckWord.toString()
           });
        }
    }
}
```

removeDictionary

5.0	👥		⊗

Parameters: cName
Returns: Boolean

This method will remove a user dictionary that was added via addDictionary. *cName* must be the same name as was used with *addDictionary*.

The *removeDictionary* method returns *true* on success, *false*, otherwise

removeWord

5.0	👥		⊗

Parameters: cWord, cName
Returns: Boolean

Use this method to remove the word *cWord* from a dictionary. If successful it returns *true*, otherwise, *false*.

The required *cName* parameter is the dictionary name. An array of the currently installed dictionaries can be obtained using the dictionaryNames method.

See also the addWord method.

Note: *Internally the SpellCheck Object will scan the user dictionary and remove the previously added word if it is there. Otherwise the word will be added to the user's "Not-A-Word" dictionary. The actual dictionary is not modified.*

---

userWords

5.0	👥		⊗

Parameters: cName, bAdded
Returns: Array

This method returns the array of words a user has added or removed from a dictionary. See also the addWord and checkWord methods.

The required *cName* parameter is the dictionary name. An array of the currently installed dictionaries can be obtained using the dictionaryNames property.

The required *bAdded* parameter indicates which of the two arrays should be returned. When *true*, the user's array of added words are returned. When *false*, the user's array of removed words are returned.

Statement Object

5.0			⊗

Statement objects are the heart of ADBC. Through Statement objects, one can execute SQL updates and queries, and retrieve the results of these operations. Statement objects can be created through calls to newStatement.

Note: *NOTE: once a column is retrieved with any of the methods below, any future calls attempting to retrieve the same column may fail!*

Statement properties

columnCount

Type: Number *Access: R*

The *columnCount* property is the number of columns in each row of results returned by a query. It is undefined in the case of an update operation.

rowCount

Type: Number *Access: R*

The *rowCount* property is the number of rows affected by an update. It is *not* the number of rows returned by a query. Its value is undefined in the context of a query.

Statement methods

execute

Parameters: cSQL
Returns: Boolean

The *execute* method executes an SQL statement through the context of the Statement object. It returns *true* on success and *false* on failure.

Example:

```
statement.execute("Select * from ClientData");

/* if the name of the database table or column name contains spaces, they
   need to be enclosed in escaped quotes. */
statement.execute("Select firstname, lastname, ssn from \"Employee Info\"");
statement.execute("Select \"First Name\"  from \"Client Data\"");
```

```
/* A cleaner solution would be to enclose the whole SQL string with single
   quotes, then table names and column names can be enclosed with double
   quotes. */
statement.execute(' Select  "First Name","Second Name" from "Client Data" ');
```

See the getRow and nextRow methods for extensive examples.

Note: *There is no guarantee that a client can do anything on a statement if an* execute *has neither failed nor returned all of its data.*

getColumn

Parameters: nColumn, [nDesiredType]
Returns: column object | null

The *getColumn* method returns a Column Object representing the data in the column identified by the *nColumn* parameter. The *nColumn* parameter may be either a number, in which case it returns a *Column object* based on its number, or a string in which case the method returns a *Column object* based on the name of the column. The optional *nDesiredType* parameter can be used to specify what JavaScript Type best represents the data in the column. The *getColumn* method will return *null* on failure.

The properties of the Column Object are listed in the table below.

Column Object

The Column object is a generic object that contains the data from every row in a column. Column objects can be obtain through calls to getColumn or getColumnArray.

Property	Type	Access	Description
columnNum	number	R	The number identifying the column in the Statement from which the Column was retrieved.
name	string	R	The name of the column. It is similar to the *name* property of the ColumnInfo Object.
type	number	R	The SQL Type of the data in the column. It is similar to the *type* property of the ColumnInfo Object.
typeName	string	R	The name of the type of data the column contains. It is similar to the *typeName* property of the ColumnInfo Object.
value	various	R/W	The value of the data in the column in whatever format the data was originally retrieved in.

getColumnArray

Parameters: None
Returns: An array of column objects | null

The *getColumnArray* method returns an array of *Column Objects*, one for each column in the result set. A "best guess" is used to decide on the type that best represents which JavaScript Type should be used to represent the data in the column. This method may return *null* on failure as well as a zero-length array.

The properties of the Column Object are listed in the table following the getColumn method.

getRow

Parameters: None
Returns: A row object

The *getRow* method returns a Row Object representing the current row. This object contains information from each column. Like getColumnArray, column data is captured in the "best guess" format.

The table below, and the discussion that follows, identifies the properties of the Row Object.

Row Object

The Row object is a generic object that contains the data from every column in a row. Row objects are created through calls to getRow.

Property	Type	Access	Description
columnArray	array	R	The *columnArray* property is an array that is equivalent to what getColumnArray would return if called on the same Statement at the same time that the Row object was created.

In addition to the *columnArray* property, the Row Object has properties representing the column of each column selected by the query.

Every Row object contains a property for each column in a row of data. Consider the following example:

```
statement.execute("SELECT firstname, lastname, ssn FROM \"Employee Info\"");
statement.nextRow();
row = statement.getRow();
console.println("The first name of the first person retrieved is: "
    + row.firstname.value);
console.println("The last name of the first person retrieved is: "
    + row.lastname.value);
console.println("The ssn of the first person retrieved is: "+ row.ssn.value);
```

Example: If the column name contains spaces, then the above syntax for accessing the row properties (E.g., row.firstname.value) does not work. Alternatively,

```
Connect = ADBC.newConnection("Test Database");
statement = Connect.newStatement();
statement.execute(' Select  "First Name","Second Name"  from "Client Data" ');
statement.nextRow();

// Populate this PDF file
this.getField("name.first").value = row["First Name"].value;
this.getField("name.last").value = row["Second Name"].value;
```

nextRow

Parameters: None
Returns: Nothing

The *nextRow* method obtains data about the next row of data generated by a previously executed query. This must be called following a call to execute to acquire the first row of results. This method throws an exception on *failure* (possibly because there is no next row).

Example: The following example is a rough outline of how to create a series of buttons and Document Level JavaScripts to browse a database and populate a PDF form.

For the getNextRow button, defined below, the *nextRow* is used to retrieve the next row from the database, unless there is an exception thrown (indicating that there is no next row), in which case, we reconnect to the database, and use *nextRow* to retrieve the first row of data (again).

```
/* Button Script */
// getConnected button
if (getConnected())
   populateForm(statement.getRow());

// a getNextRow button
try {
    statement.nextRow();
}catch(e){
    getConnected();
}
var row = statement.getRow();
populateForm(row);

/* Document Level JavaScript */
// getConnected() Doc Level JS
function getConnected()
```

```
{
    try  {
        ConnectADBCdemo = ADBC.newConnection("ADBCdemo");
        if (ConnectADBCdemo == null)
            throw "Could not connect";
        statement = ConnectADBCdemo.newStatement();
        if (statement == null)
            throw "Could not execute newStatement";
        if (statement.execute("Select * from ClientData"))
            throw "Could not execute the requested SQL";
        if (statement.nextRow())
            throw "Could not obtain next row";
        return true;
    } catch(e) {
        app.alert(e);
        return false;
    }
}
// populateForm()
/* Maps the row data from the database, to a corresponding text field in the
   PDF file. */
function populateForm(row)
{
    this.getField("firstname").value = row.FirstName.value;
    this.getField("lastname").value = row.LastName.value;
    this.getField("address").value = row.Address.value;
    this.getField("city").value = row.City.value;
    this.getField("state").value = row.State.value;
    this.getField("zip").value = row.Zipcode.value;
    this.getField("telephone").value = row.Telephone.value;
    this.getField("income").value = row.Income.value;
}
```

Template Object

Template objects are named pages within the document. These pages may be hidden or visible and can be copied or “spawned”. They are typically used to dynamically create content (e.g. adding pages to an invoice on overflow).

Template Object Properties

hidden

5.0	💾		⊗

Type: Boolean *Access: R/W*

This property indicates whether the template is hidden or not. Hidden templates cannot be seen by the user until they are spawned or are made visible. When an invisible template is made visible it is appended to the document.

See also the templates property, the createTemplate, getTemplate, and removeTemplate methods of the Doc Object, and the Template Object.

Note: *Although reading this property is valid, setting this property in Acrobat Reader will generate an exception.*

---

name

5.0			

Type: String *Access: R*

This property returns the name of the template which was supplied when the template was created.

See also the templates property, the createTemplate, getTemplate, and removeTemplate methods of the Doc Object, and the Template Object.

Template Object Methods

spawn

5.0	💾		⊗

Parameters: [nPage], [bRename], [bOverlay]
Returns: Nothing

Creates a new page in the document based on the template.

nPage represents the page number (zero-based) on which the template will be overlaid. The default for *nPage* is zero.

bRename is a boolean that indicates whether form fields on the page should be renamed. The default for *bRename* is *true*.

bOverlay is a boolean that indicates whether the template should be overlaid on the page or whether is should be inserted as a new page before the specified page. The default for *bOverlay* is *true*. To append a page to the document, set *bOverlay* to *false* and set *nPage* to the number of pages in the document.

Example:
```
var a = this.templates;
for (i = 0; i < a.length; i++)
   a[i].spawn(this.numPages, false, false);
```

This example spawns all templates and appends them one by one to the end of the document.

See also the templates property, the createTemplate, getTemplate, and removeTemplate methods of the Doc Object, and the Template Object.

TTS Object

4.05			

The JavaScript TTS object allows users to transform text into speech. To be able to use the TTS object, the user's machine must have a Text-To-Speech engine installed on it. The Text-To-Speech engine will render text as digital audio and then "speak it". It has been implemented mostly with accessibility in mind but it could potentially have many other applications, bringing to life PDF documents.

This is currently a Windows-only feature and requires that the MicroSoft Text to Speech engine be installed in the operating system.

The Tts Object is present on both the Windows and Mac platforms (since it is a JavaScript Object), it is disabled on the Mac, however.

Note: *Acrobat 5.0 has taken a very different approach to providing accessibility for disabled users by integrating directly with popular screen readers. Some of the features and methods defined in 4.05 for the TTS object have been deprecated as a result as they conflict with the screen reader. The TTS object remains, however, as it still has useful functionality in its own right that might be popular for multi-media documents.*

TTS Properties

available

Type: Boolean *Access: R*

This property returns true if the TTS object is available and the Text-To-Speech engine can be used.

```
      console.println("Text to speech available: " + tts.available);
```

numSpeakers

Type: Integer *Access: R*

Returns the number of different speakers available to the current text to speech engine. See also the speaker property and the getNthSpeakerName method.

pitch

Type: Integer *Access: R/W*

This property sets the baseline pitch for the voice of a speaker. The valid range for pitch is from 0 to 10, with 5 being the default for the mode.

soundCues

☹			

Type: Boolean *Access: R/W*

This property has been deprecated. It now returns only *false*.

speaker

Type: String *Access: R/W*

This property allows users to specify different speakers with different tone qualities when performing text-to-speech. See also the numSpeakers property and the getNthSpeakerName method.

speechCues

☹			

Type: Boolean *Access: R/W*

This property has been deprecated. It now returns only *false*.

speechRate

Type: Integer *Access: R/W*

This property sets the speed at which text will be spoken by the Text-To-Speech engine. The value for speechRate is expressed in number of words per minute.

volume

Type: Integer *Access: R/W*

This property sets the volume for the speech. Valid values are from 0 (mute) to 10 (loudest).

TTS Methods

getNthSpeakerName

Parameters: nIndex
Returns: cName

Use this function to obtain the *n*th speaker name in the currently installed text to speech engine (see also the numSpeakers and speaker properties).

Example:

```
// Enumerate through all of the speakers available.
for (var i = 0; i < tts.numSpeakers; i++) {
    var cSpeaker = tts.getNthSpeakerName(i);
    console.println("Speaker[" + i + "] = " + cSpeaker);
    tts.speaker = cSpeaker;
    tts.qText ("Hello");
    tts.talk();
}
```

pause

Parameters: None
Returns: Nothing

This method immediately pauses text-to-speech output on a TTS object. Playback of the remaining queued text can be resumed via the resume method.

qSilence

Parameters: nDuration
Returns: Nothing

This method queues a period of silence into the text. *nDuration* specifies the amount of silence in milliseconds.

qSound

Parameters: cSound
Returns: Nothing

This method puts a sound into the queue in order to be performed by talk. It accepts one parameter, *cSound,* from a list of possible sound cue names. These names map directly to sound files stored in the SoundCues folder, if it exists.

```
tts.qSound("DocPrint");         // Plays DocPrint.wav
```

The SoundCues folder should exist at the program level for the viewer, e.g. C:\Program Files\Adobe\Acrobat 5.0\SoundCues.

Note: *Windows only--qSound can handle only 22KHz,16 bit PCM .wav files. These should be at least one second long in order to avoid a queue delay problem in MS SAPI. In case the sound lasts less than one second, it should be edited and have a silence added to the end of it.*

---

qText

Parameters: cText
Returns: Nothing

This method puts text into the queue in order to be performed by talk.

cText is the text to convert to speech.

```
tts.qText("Hello, how are you?");
```

reset

Parameters: None
Returns: Nothing

This method stops playback of current queued text and flushes the queue. Playback of text cannot be resumed via the resume method. Additionally, it resets all the properties of the TTS object to their default values.

resume

Parameters: None
Returns: Nothing

This method resumes playback of text on a paused TTS object.

stop

Parameters: None
Returns: Nothing

This method stops playback of current queued text and flushes the queue. Playback of text cannot be resumed with the resume method.

talk

Parameters: None
Returns: Nothing

This method sends whatever is in the queue to be spoken by the Text-To-Speech engine. If text output had been paused, talk resumes playback of the queued text.

```
tts.qText("Hello there!");
tts.talk();
```

this Object

In JavaScript the special keyword *this* refers to the current object. In Acrobat 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 in one of the* Folder Level JavaScripts files, *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.

Acrobat JavaScript programmers should avoid using property and method names from the Doc Object as variable names. Use of method names when after the reserved word "var" will throw an exception, as the following line illustrates:

```
var getField = 1; // TypeError: redeclaration of function getField
```

Use of property names will not throw an exception, but the value of the property may not be altered if the property refers to an object:

```
// "title" will return "1", but the document will now be named "1".
var title = 1;

// property not altered, info still an object
var info = 1; // "info" will return [object Info]
```

The following was taken from the example that follows the signatureInfo Object table and is an example of avoiding variable name clash.

```
var f = this.getField("mySignature"); // uses the ppklite sig handler

// use "Info" rather than "info" to avoid a clash
var Info = f.signatureInfo();

// some standard signatureInfo properties
console.println("name = " + Info.name);
```

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 and parsing.

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. This method converts and formats incoming arguments into a result string (*cResult*) according to a format string (*cFormat*). The format string consists of two types of objects: ordinary characters, which are copied to the result string, and conversion specifications, each of which causes conversion and formatting of the next successive argument to printf. Each conversion specification is constructed as follows:

%[,*nDecSep*][*cFlags*][*nWidth*][.*nPrecision*]*cConvChar*

cDecSep, preceded by a comma character (,), is a digit from 0 to 3 which indicates the decimal/ separator format:

0 - comma separated, period decimal point.

1 - no separator, period decimal point.

2 - period separated, comma decimal point.

3 - no separator, comma decimal point.

cFlags is only valid for numeric conversions and consists of a number of characters (in any order), which will modify the specification:

+ - specifies that the number will always be formatted with a sign.

space - if the first character is not a sign, a space will be prefixed.

0 - specifies padding to the field with with leading zeros.

# - which specifies an alternate output form. For f the output will always have a decimal point.

nWidth is a number specifying a minimum field width. The converted argument will be formatted in so that it is at least this many characters wide, including the sign and decimal point, and may be wider if necessary. If the converted argument has fewer characters than the field width it will be padded on the left to make up the field width. The padding character is normally a space, but is 0 if zero padding flag is present.

nPrecision, preceded by a period character (.), is a number which specifies the number of digits after the decimal point for float conversions.

cConvChar is one of:

d - integer, interpret the argument as an integer (truncating if necessary).

f - float, interpret the argument as a number.

s - string, interpret the argument as a string.

x - hexadecimal, interpret the argument as an integer (truncating if necessary) and format in unsigned hexadecimal notation.

Example:
```
var n = Math.PI * 100;
console.clear();
console.show();
console.println(util.printf("Decimal format: %d", n));
console.println(util.printf("Hex format: %x", n));
console.println(util.printf("Float format: %.2f", n));
console.println(util.printf("String format: %s", n));
```

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

printd

Parameters: cFormat, oDate
Returns: String

Use this method to format a date according to a format. *cFormat* be either a string or a number; *date* must be a date object

Valid string format values for the *cFormat* parameter 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

String	Effect	Example
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));
```

5.0 Additions

A variety of addition "quick" formats are possible using numeric values.

Value	Description	Example
0	PDF date format	D:20000801145605+07'00'
1	Universal	2000.08.01 14:56:05 +07'00'
2	Localized string	2000/08/01 14:56:05

Example:

```
// display date in a local format
var d = new Date();
console.println(util.printd(2, d));
```

printx

Parameters: cFormat, [cSource]
Returns: String

This method formats a source string, *cSource*, according to a formatting string, *cFormat*. A valid format is any string which may contain special masking characters:

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);
```

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 scand. 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.

A Short Acrobat JavaScript FAQ[1]

Where can JavaScripts be found and how are they used?

JavaScripts work with Acrobat on a variety of levels: the *folder* level, *document* level, *field* level and *batch* level. (See also the subsection entitled Where Can You Use JavaScript?) These levels restrict the type of processing that can occur and are loaded at different times.

Folder Level JavaScripts

JavaScripts can be placed in individual files with the "*.js*" extension. For such files to be read by the viewer at startup they need to be placed in either the Acrobat Application *JavaScripts* folder or in the user's *JavaScripts* folder. See App/Init for a discussion of the order these files are loaded into the application on startup.

Variables and functions that might be generally useful to the application should be kept in these folders. Note that some JavaScripts methods can only be executed from within one of these JavaScript files at startup; e.g., App.addMenuItem and App.addSubMenu.

There are some restrictions when writing JavaScript files, particularly with respect to the use of this Object.

The standard Acrobat JavaScript implementation comes with three JavaScript files: *Aform.js* which contains built-in pre-canned functions, *Annots.js* which is used by the Annotations plug-in and *ADBC.js* used by the ADBC plug-in. These are located in the application *JavaScripts* folder.

4.0	👥		

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).

[1] Frequently Asked Questions

Document level

By using the Adobe Acrobat menu item *Tools->JavaScript->Document JavaScripts…*, 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. 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.

How should I name my form fields?

Form fields typically have names like *FirstName*, *LastNam*e, 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.Las***t** 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.

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";
```

How do I use date objects?

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.*

Tip: *When using the Date object, do not use the getYear() method which returns the current year - 1900. Instead use the getFullYear() method which always returns a four digit year.*

Converting from a Date to a String

Acrobat Forms provides several date related methods in addition to the ones provided by the JavaScript Date object. These are the preferred methods 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 scand method of the Util Object is used. Unlike the built-in conversion of the Date object to a string, scand 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 s = [ "mm/dd/yy", "yy/m/d", "mmmm dd, yyyy", "dd-mmm-yyyy" ];
for (var i = 0; i < s.length; i++) {
  /* print these strings to the console */
  console.println("Format " + s[i] + " looks like: " + util.printd(s[i], d));
}
```

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
```

Tip: *Given the ever increasing length of the human lifespan and the lessons we've learned from Y2K coding issues, it is advised that you output dates with a four digit year to avoid ambiguity.*

Converting from 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
```

How can I make my document secure?

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.

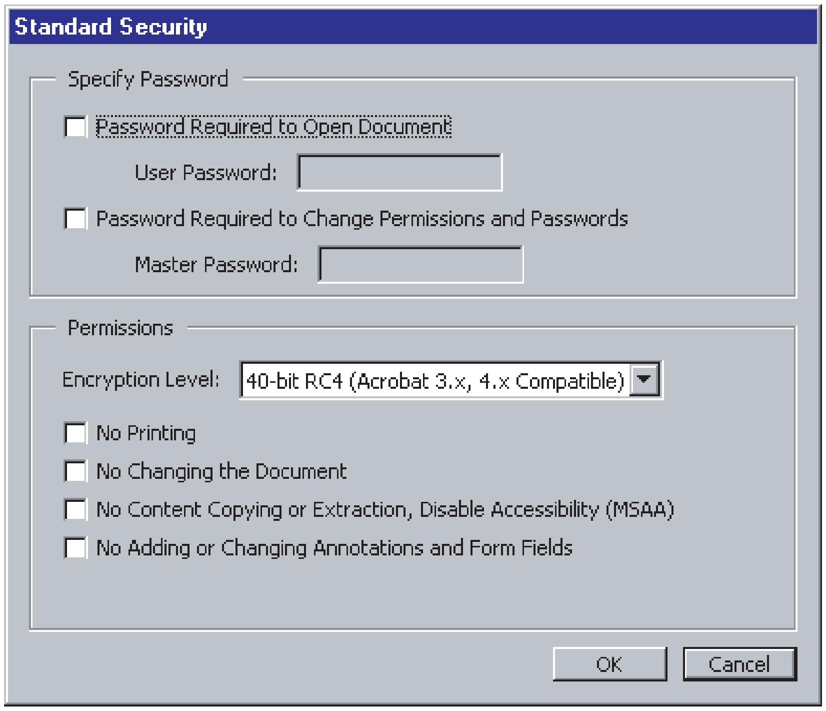
Restricting Access to the Document

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 by choosing *File > Document Security...* from the Acrobat menu, then from the drop-down menu entitled "Document Secured with:", by choosing the appropriate security handler, *Acrobat Standard Security,* for example. You can now set the password and permissions as desired.

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.


Standard Security

Specify Password

☐ Password Required to Open Document

User Password:

☐ Password Required to Change Permissions and Passwords

Master Password:

Permissions

Encryption Level: 40-bit RC4 (Acrobat 3.x, 4.x Compatible)

☐ No Printing

☐ No Changing the Document

☐ No Content Copying or Extraction, Disable Accessibility (MSAA)

☐ No Adding or 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 *No 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 *No Changing the Document* and *No Adding or 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.

Also, see the section on the signature field for a discussion on how to create and sign a digital signature field programmatically.

How can I lock a document after a signature field has been signed?

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 Where can JavaScripts be found and how are they used?) 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.

How can I make my documents accessible?

4.05			

Accessibility of electronic information is an ever-increasingly important issue. Creating forms that adhere to the accessibility tips below will make your forms more easily usable by all users. The 4.05 release of Acrobat is intended to allow motion and vision impaired users to fill out Acrobat Forms. Future versions of Acrobat will be more fully speech-enabled.

The following is a set of guidelines which must be followed in order to make a form minimally accessible given the default behavior of Acrobat 4.05.

Document Meta-Data

The meta-data for a document can be specified using the File->Document Info->General dialog.

When a document is opened, saved, printed, or closed, the document title is spoken to the user. If the title has not been specified then the filename is used. Often, file names are abbreviated or changed and as such it is highly encouraged that the document author specify a title for the document. For example, if a document has a file name of “IRS1040.pdf” a good document title would be “Form 1040: U.S. Individual Income Tax Return for 1998”.

Filling all of the additional meta-data associated with a document (Author, Subject, Keywords) also makes it more easily searchable using Acrobat Search and Internet search engines.

Short Description

If a field name is not user friendly, it must contain a user name (short description) that is user friendly if the field is likely to be interacted with (i.e. it is not a read-only or hidden field). This name is spoken when a user acquires the focus to that field and should give an indication of the field's purpose. For example, if a field is named "name.first" a good short description would be "First Name". This description is also displayed as a "tooltip" when the user positions his mouse over the field making working with the form a more productive experience.

Setting tab order

In order to traverse the document in a reasonable manner, the tab order for the fields must be set in a logical way. This is important as most users use the tab key to move through the document. For visually impaired users this is a necessity as they cannot rely on mouse movements or visual cues.

Use the TTS object

In cases that the default behavior of Acrobat with respect to forms is insufficient to give context or indication of the current state of the form fields in the document, the author is encouraged to make use of the TTS Object to supplement that behavior.

Default Behavior

The default behavior of Acrobat 4.05 with respect to accessibility is as follows:

1. Tab key: pressing the tab (shift-tab) key when there is no form field that has the keyboard focus will cause the first (last) field in the tab order on the current page to become active. If there are no form fields on the page then Acrobat will inform the user of this via a speech cue.

   Using tab (shift-tab) while a field has the focus tabs forward (backward) in the tab order to the next (previous) field. If the field is the last (first) field on the page and the tab (shift-tab) key is pressed, the focus is set to the first (last) field on the next (previous) page if one exists. If such a field does not exist, then the focus "loops" to the first (last) field on the current page.

2. Sound Cues: sound cues are user-configurable sounds that play when certain events occur and give the user an indication of context. The following actions have sound cues attached to them:

- *Document open, close, activate, save, print*
- *Page turn*
- *Keystroke handling when interacting with a field*

3. Speech cues: speech cues are user configurable strings that are spoken by the text to speech engine to give the disabled user an indication of context. The following actions have speech cues attached to them:
   - *Application initialize*
   - *Document open, close, activate, save, print*
   - *Page turn*
   - *Field focus, blur*
   - *Alert dialogs*
   - *Keystroke handling when interacting with a field.*

How can I define globals in JavaScript?

The Acrobat extensions to JavaScript define a Global object to which you can attach global variables as properties. To define a new global called 'myVariable' and set it equal to the null string, you would type:

```
global.myVariable = "";
```

All of your scripts, no matter where they are in a document, will now be able to reference this variable.

Making Globals Persistent

Global data does not persist across user sessions unless you specifically make your globals persistent. The predefined Global object has a method designed to do this. To make a variable named 'myVariable' persist across sessions, do this:

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

In future sessions, the variable will still exist (with its previous value intact).

How can I send form data to an e-mail address?

You can use the submitForm method of the Field Object to accomplish this:

```
var url = "mailto:johndoe@doe.net";
```

```
    this.submitForm(url,false);
```

In this instance, the form contents will be sent to the address given in the variable *url*. The second argument of submitForm() determines whether the form contents are sent as url-encoded data using the POST method, or sent as FDF (Forms Data Format). A value of "false" means the data will be sent in url-encoded fashion.

How can I hide a field based on the value of another?

Use the display property of the Field object.

```
    var title = this.getField("title");
    if (this.getField("showTitle").value == "Off")
       title.display = display.hidden;
    else
       title.display = display.visible;
```

How can I query a field value in another open form from the form I'm working on?

One way would be to use the Global Object's subscribe method to make the field(s) of interest available to others at runtime. For example, a form could contain a document-level script (invoked when that document is first opened) that defines a global field value of interest:

```
    function PublishValue( xyzForm_fieldValue_of_interest ) {
      global.xyz_value = xyzForm_fieldValue_of_interest;
    }
```

Then, when your document (Document A) wants to access the value of interest from the other form (Document B), it can subscribe to the variable in question:

```
    global.subscribe("xyz_value", ValueUpdate);
```

In this case, ValueUpdate refers to a user-defined function that gets called automatically whenever `xyz_value` changes. If you were using *xyz_value* in Document A as part of a field called MyField, you might define the callback function this way:

```
    function ValueUpdate( newValue ) {
      this.getField("MyField").value = newValue;}
```

How can I intercept keystrokes one by one as they occur?

Create a Custom Keystroke Filter script (see the Format tab in the Field Properties dialog for any text or combo box field) in which you examine the value of *event.change*. By altering this value, you can alter the user's input as it takes place.

How can I build a nested popup menu?

Use the *app.popUpMenu()* method. Create an array of menu selections, then call *app.popUpMenu(arrayName)* from the mouse-down or mouse-up event of a given field to pop the menu.

Example:
```
      var cChoice = app.popUpMenu("one", "two", "-",
        [ "three", "three.one", "three.two" ] );
      app.alert("You chose " + cChoice);
```

How can I construct my own colors?

Colors are Array objects in which the first item in the array is a string describing the color space ('G' for grayscale, 'RGB' for RGB, 'CMYK' for CMYK) and the following items are numeric values for the respective components of the color space. Hence:

```
      color.blue = new Array("RGB", 0, 0, 1);
      color.cyan = new Array("CMYK", 1, 0, 0, 0);
```

To make a custom color, just declare an array containing the color-space type and channel values you want to use.

How can I prompt the user for a response in a dialog?

Use the response method defined in the 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. class. This method displays a dialog box containing a question and an entry field for the user to reply to the question. (Optionally, the dialog can 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 dialog's Cancel button, the response is the null object.

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

How can I fetch an URL from JavaScript?

Use the getURL method of the Data Object class. 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.

How can I change the hot-help text for a field dynamically?

The userName property of the Field Object returns/sets the “short description” of the field in question, as a string. This property is intended to be used as a tool tip or hot-help popup whenever the mouse cursor loiters over a field. It can be a great help to the user if you put useful suggestions (or descriptive language of some sort) in the userName property.

How can I change the zoom factor programmatically?

Use the zoom method of the Data Object class. For example, the following code shows two ways to set the zoom factor of the current page:

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

// This sets the zoom to 73%:
this.zoom = 73;
```

How can I determine if the mouse has entered/left a certain area?

Create an invisible, read-only text field at the place where you want to detect mouse entry or exit. Then attach JavaScripts to the mouse-enter and/or mouse-exit actions of the field.

What are Rotated User Space and Default User Space?

Two terms used frequently in this specification are *rotated user space* and *default user space.* These two arise in connection with forms and annots, respectively. Forms always uses rotated user space and annots always uses the default user space.

Rotated User Space

In rotated user space (which is a coordinate system), the origin (or reference point) is the lower left-hand corner of the *crop box*, *even if the page has been rotated*. The positive *x*-axis points to the right from the origin, the positive *y*-axis moves up from the origin. The length of one unit along both the *x* and *y* axes is 1/72 inch.

Default User Space

Default user space is the coordinate system described in PDF Reference, page 126. For an *unrotated page*, the origin is the lower left corner of the *media box* with the positive *x*-axis extending horizontally to the right and the positive *y*-axes extending vertically upward. The length of one unit along both the *x* and *y* axes is 1/72 inch. When the page is rotated, however, the axis system is rotated as well, in which case, the *x*- and *y*- axes may have a different orientation from the standard one just described.

Graphical Relationship

Below is a depiction of a page that has been cropped and rotated 90 degrees clockwise. In the *rotated user space*, the origin is in the lower left-hand corner of the crop box, the axes are in standard mathematical orientation. The origin for the *default user space*, in this case, is in the upper left-hand corner, remembering the page has been rotated 90 degrees clockwise, with the positive *x*-axis pointing downward, and the positive *y*-axis pointing to the right.

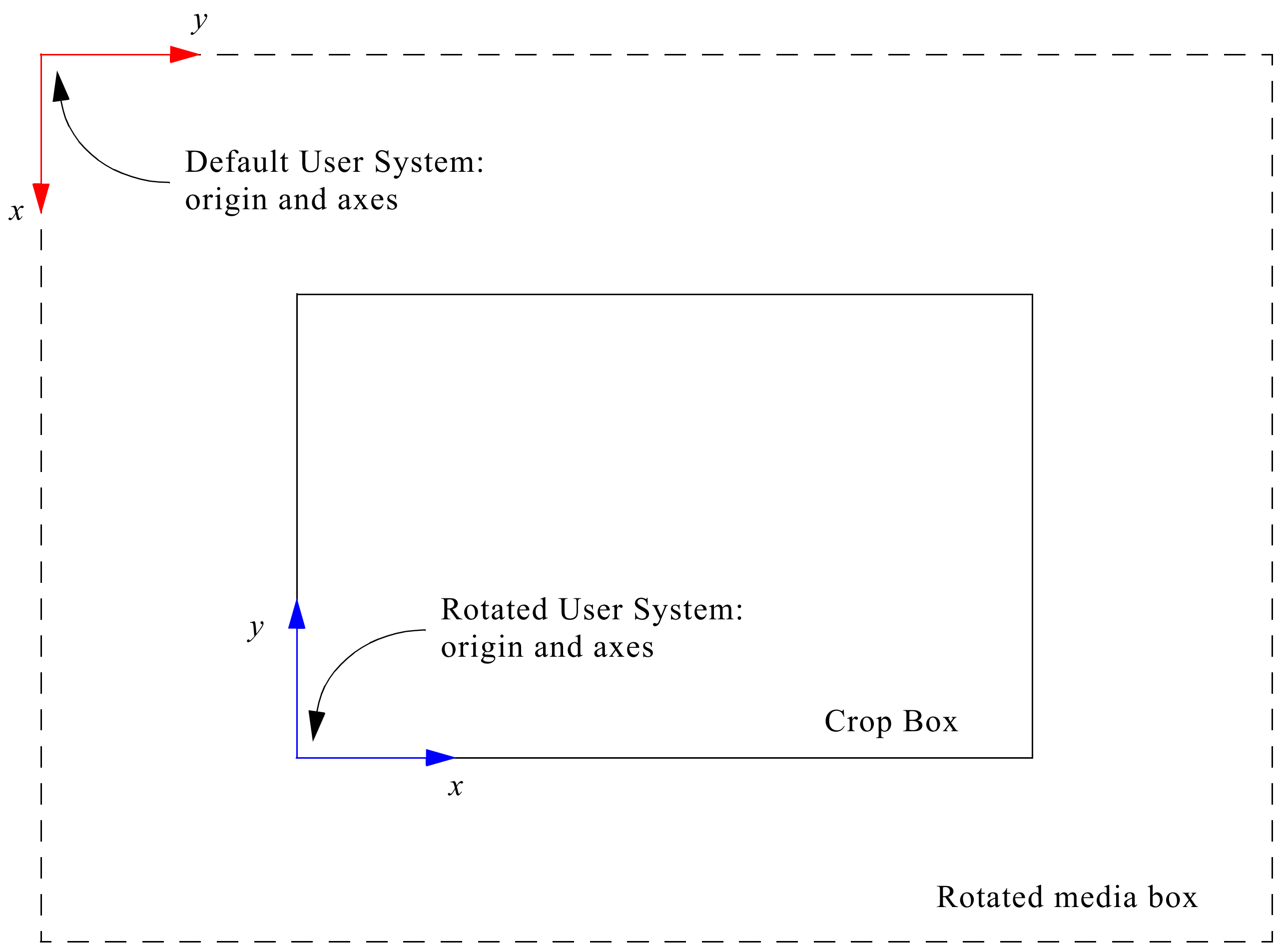
Annots and Forms elements having the same coordinates (bounding rectangles), will not necessarily be located in identical places on the page. For example, if we execute

```
    this.addField("myButton", "button", 0, [0,0,20,20] )
```

and executed the script

```
    this.addAnnot({ page: 0, type: "Square", rect: [0,0,20,20] });
```

we would get a button located in the lower left-hand corner of the crop box, and a "Square" comment in the upper left-hand corner of the media box! This should always be kept in mind when placing forms or annot objects on a page.



Example: The following code first places a button field in the lower left-hand corner of the screen page (the cropped page), then places a "Square" annot in exactly the same location. This script works correctly even if the page is cropped and/or rotated. The script uses some undocumented JavaScript methods to make the conversion from *Rotated User Space* to *Default User Space*.

```
var f = this.addField("myButton", "button", 0, [36,36,72,72] );
f.strokeColor = color.black;
var m = (new Matrix2D).fromRotated(this,0);
r = m.transform(f.rect);
this.addAnnot( { type:"Square", rect: r, page: 0 });
```

How can I create a form field programmatically?

Acrobat provides a large number of JavaScript properties and methods for creating a form field and to set its appearance and associated action, if any. In this section, an attempt at organizing all the new (and old) properties and methods is made. The organizational scheme used is the Acrobat user interface; a correspondence is made between items in the user interface and the associated JavaScript property or method.

There are seven types of form fields:

- Button
- Check Box
- Combo Box
- List Box
- Radio Button
- Signature
- Text

Each of these is discussed in turn.

Button

A form field may be created either by the UI for Acrobat, or by the addField method of the Data Object. Programmatically, a button field is created as follows:

```
var f = this.addField("myButton", "button", 0, [400, 436, 472, 400]);
```

This would create a button on page 0, and located at `[400, 436, 472, 400]`; i.e., the button would be one inch wide (72 points) and 0.5 inch high (36 points). A default appearance is given to the field. The return value of this method is a field object that will be used throughout the rest of this section.

The field object for a field that is already existent can be obtained by the getDataObject method of the doc object:

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

Field Properties

The top of the UI, above the tab fields, there are listed three general properties.

General Field Properties		
Name	**Use**	**Example**
Name	name	console.println(f.name);
Type	type	console.println(f.type);
Short Description	userName	f.userName = "Submit Button"

Table Notes

- The *name* and *type* are read-only properties. They can be set at creation time, either through the UI, or programmatically, with addField. See the example above.

It is possible through the UI in Acrobat to specify the *Appearance* of the field, its *Options* and its *Actions*, all of which appear as tabs in the UI for a button field. All these properties can be accessed through field level JavaScript properties and methods.

Appearance

The appearance tab allows you to set the basic appearance of the field. The table below summarizes how the appearance can be set programmatically using JavaScript. In the table, it is assumed that the variable *f* is a field object.

The Appearance Tab		
Region	**Use**	**Example**
Border		

Border Color
Background Color
Width
Style
	strokeColor
	fillColor
	lineWidth
	borderStyle
		f.strokeColor = color.black;
		f.fillColor = color.ltGray;
		f.lineWidth = 1;
		f.borderStyle = style.b

Text		

Text Color
Font
Size
	textColor
	textFont
	textSize
		f.textColor = color.blue;
		f.textFont = font.Times;
		f.textSize = 16;

Common Properties		

Read Only
Form Field is
Orientation
	readonly
	display
		f.readonly = true;
		f.display = display.visible
		See tables notes

Table Notes

- There is no property to set the orientation of a form field; however, by rotating the page, creating the button, then rotating back again, a rotated button can be created.

```
this.setPageRotations(this.pageNum, this.pageNum, 90);
var f = this.addField("actionField", "button", 0, [200, 250, 300, 200]);
f.delay = true;
f.strokeColor = color.black;
f.fillColor = color.ltGray;
f.borderStyle = style.b;
f.delay=false;
this.setPageRotations(this.pageNum, this.pageNum);
```

Options

The option tab allows you to set the highlighting, the layout and the button-face attributes.

The Options Tab		
Highlight	highlight	f.hightlight = hightlight.p
Layout	buttonPosition	f.buttonPosition = position.iconOnly;
Advanced Layout		

Scale When
Scale How
Button
	buttonScaleWhen
	buttonScaleHow
	buttonAlignX and buttonAlignY
		f.buttonScaleWhen = scaleHow.always
		f.buttonScaleHow = scaleHow.proportional
		f.buttonAlign = 50;
		f.buttonAlign = 50;

Button Face Attributes		
Text Select Icon	buttonSetCaption and buttonGetCaption buttonSetIcon	f.buttonSetCaption("Push Me"); See table notes

Table Notes

- For “Select Icon” under “Button Face Attributes”, the basic tool for associating a icon with a button face is buttonSetIcon; however, this assumes there is a named icon already in the PDF file. See the doc object addIcon for a complete example of inserting an icon into a button face.
- buttonImportIcon can be used to introduce named icons into the document.

Actions

The action of the button can be set with the field level setAction method with trigger names "MouseUp", "MouseDown", "MouseEnter", "MouseExit", "OnFocus", "OnBlur". For example,

```
f.setAction("MouseUp", "app.beep(0);")
```

Check Box

A form field may be created either by the UI for Acrobat, or by the addField method of the Data Object. Programmatically, a check box is created as follows:

```
    var f = this.addField("myCheck", "checkbox", 0, [400, 412, 412, 400]);
```

This would create a check box on page 0, and located at `[400, 412, 412, 400]`; i.e., the check box would be 12 points wide and 12 points high. A default appearance is given to the field. The return value of this method is a field object that will be used throughout the rest of this section.

The field object for a field that is already existent can be obtained by the getField method of the doc object:

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

Field Properties

The top of the UI, above the tab fields, there are listed three general properties. The properties are same for button, see the Field Properties for buttons.

Appearance

The Appearance attributes are the same as in the case of button. There are two differences, however. First, there is no choice for textFont; in the case of a check box, the font is always *Zapf Dingbats*. Secondly, under the **Common Properties** of the Appearance tab, *Required* check box is active.

The Appearance Tab		
Region	**Use**	**Example**
Common Properties		
Required	required	f.required = true;

Options

The Options tab allows you to set the style of check used in the field and the export value. The table below makes the connection between the UI and JavaScript.

The Options Tab		
Check Style	style	f.style = style.ci;
Export Value	setFocus	f.setExportValues(["buy"]);
Default is Checked	defaultIsChecked	f.defaultIsChecked(0); // see table notes

Table Notes

- Default is Checked: After using defaultIsChecked, the field is not necessarily checked. To check the field, either reset the field, `this.resetForm([f.name])`, or apply the checkThisBox method: `f.checkThisBox(0);`
- Default is Checked: To determine if the “Default is Checked” check box on the Options tab is “checked”, use isDefaultChecked.
- To determine if a check box is “checked”, use isBoxChecked.

Actions

The action of the button can be set with the field level setAction method with trigger names "MouseUp", "MouseDown", "MouseEnter", "MouseExit", "OnFocus", "OnBlur". For example,

```
f.setAction("MouseUp", "app.beep(0);")
```

Combo Box

A form field may be created either by the UI for Acrobat, or by the addField method of the Doc Object. Programmatically, a combo box is created as follows:

```
var f = this.addField("myCombo", "combobox", 0, [200, 436, 400, 400]);
```

This would create a check box on page 0, and located at `[200, 436, 400, 400]`; i.e., the combo box would be 200 points wide and 36 points high. A default appearance is given to the field. The return value of this method is a field object that will be used throughout the rest of this section.

The field object for a field that is already existent can be obtained by the getField method of the doc object:

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

Field Properties

The top of the UI, above the tab fields, there are listed three general properties. The properties are same for button, see the Field Properties for buttons.

Appearance

The appearance tab allows you to set the basic appearance of the field. The table below summarizes how the appearance can be set programmatically using JavaScript. In the table, it is assumed that the variable f is a field object.

The Appearance Tab		
Region	**Use**	**Example**
Border		

Border Color
Background Color
Width
Style
	strokeColor
	fillColor
	lineWidth
	borderStyle
		f.strokeColor = color.black;
		f.fillColor = color.ltGray;
		f.lineWidth = 1;
		f.borderStyle = style.b

Text		

Text Color
Font
Size
	textColor
	textFont
	textSize
		f.textColor = color.blue;
		f.textFont = font.Times;
		f.textSize = 16;

Common Properties		

Read Only
Required
Form Field is
Orientation
	readonly
	required
	display
		f.readonly = true;
		f.required = true;
		f.display = display.visible
		See tables notes

Table Notes

- The field can be reoriented as described in the Table Notes of the section on the button field.

Options

In the Options tab, the item name and its corresponding export value can be set.

The Options Tab		
Item	setItems	see table notes
Export Value	setItems	see table notes
Sort Items		see table notes
Editable	editable	f.editable = true;
Do Not Spell Check	doNotSpellCheck	f.doNotSpellCheck = true;

Table Notes

- Item and Export Value: After a combo box is created with addField. the item names their export values can easily be introduced using setItems; for example

```
f.setItems([ ["California", "CA"], ["Ohio", "OH"], ["Arizona", "AZ"] ]);
```

- Sort: There is no direct hook to this check box on the Options tab. This check box is directed at the Acrobat to sort the list as it is entered in the UI. In the above example of setItems, the items are not entered in alphabetical order. Programmatically, the list can be sorted using the sort method of the array object. For example:

```
function compare (a,b) {              // define a compare function
  if (a[0] < b[0] ) return -1;
  if (a[0] > b[0] ) return 1;
  return 0;
}
var tmp = new Array();
var f = this.getField("myCombo");
for (var i = 0; i < f.numItems; i++)   // load [item, exportvalue]
  tmp[i] = [f.getItemAt(i,false), f.getItemAt(i)];
tmp.sort(compare);                     // sort of first component
f.clearItems();                        // out with the old
f.setItems(tmp);                       // in with the new
```

Actions

The action of the button can be set with the field level setAction method with trigger names "MouseUp", "MouseDown", "MouseEnter", "MouseExit", "OnFocus", "OnBlur". For example,

```
f.setAction("MouseEnter", "app.beep(0);")
```

Format

The action of the button can be set with the field level setAction method and a trigger name of "Format". The UI has several categories of formatting, the JavaScript counterparts are listed in the table below. Except for custom formatting, all formats can be realized by using the formatting functions contained in *Aform.js*.

The Format Tab

Number	AFNumber_Format	f.setAction("Format", 'AFNumber_Format(2, 0, 0, 0, "\240", true)');
Percentage	AFPercent_Format	
Date	AFDate_FormatEx	
Time	AFTime_Format	
Special	AFSpecial_Format	
Custom		see table notes

Table Notes

- Number: The example in the table corresponds to a comma delimited euro currency with two decimal points in the UI.
- Custom: Any format script that does not use the above mentioned format functions is classified as custom formatting script. Custom keyboard script is set using the setAction method with a trigger name of "Keystroke".

Validate

The action of the button can be set with the field level setAction method and a trigger name of "Validate". The UI has several categories of Validate, the JavaScript counterparts are listed in the table below. Except for custom formatting, all formats can be realized by using the formatting functions contained in *Aform.js*.

The Validate Tab		
Value must be greater than or equal to and less than or equal to	AFRange_Validate	f.setAction("Validate", 'AFRange_Validate(true, 0, true, 100)'); /* value between 0 and 100, inclusive */
Custom		see table notes

Table Notes

- Custom: Any validate script that does not use the AFRange_Validate function is classified as custom.

Calculate

The action of the button can be set with the field level setAction method and a trigger name of "Calculate". The UI has several categories of Calculate, the JavaScript counterparts are listed in the table below. Except for custom formatting, all formats can be realized by using the formatting functions contained in *Aform.js*.

The Calculate Tab		
Value is the sum (product, average, minimum, maximum) of the following fields:	AFSimple_Calculate	f.setAction("Calculate", 'AFSimple_Calculate("SUM", new Array ("line.1", "line.3"))');
Custom		see table notes

Table Notes

- Custom: Any calculate script that does not use the AFSimple_Calculate function is classified as custom.

Miscellaneous Programming Notes

- The number of items in a combo (or list) box can be queried using the property numItems.
- getItemAt can be used to get the face name (the item name) and/or the export value of that item.
- insertItemAt can be used to insert a new item into a combo (or list) box.
- deleteItemAt can be used to delete an item from a combo (or list) box.
- clearItems can be used to delete the whole list from the combo (or list) box.
- currentValueIndices can be used to change the current value of the combo (or list) box. For example, putting f.currentValueIndices = 2 will make the third item (0 based) the current value of combo box. (Its export value will be exported, if the form is submitted.)

List Box

A form field may be created either by the UI for Acrobat, or by the addField method of the Doc Object. Programmatically, a list box is created as follows:

```
    var f = this.addField("myList", "listbox", 0, [400, 445, 544, 400]);
```

This would create a check box on page 0, and located at `[200, 445, 544, 400]`; i.e., the combo box would be 2 inches wide (144 point) wide and high enough for three item at 12 point type. A default appearance is given to the field. The return value of this method is a field object that will be used throughout the rest of this section.

The field object for a field that is already existent can be obtained by the getField method of the doc object:

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

Field Properties

The top of the UI, above the tab fields, there are listed three general properties. The properties are same for button, see the Field Properties for buttons.

Appearance

The Appearance is the same as that of the combo box.

Options

The Options are the same as that of the comobox, with one exception

The Options Tab		
Item	setItems	see table notes

The Options Tab		
Export Value	setItems	see table notes
Sort Items		see table notes
Multiple Selection	multipleSelection	f.multipleSelection = true;

Table Notes

- Item and Export Value: See Table Notes of the combo box.
- Sort Items: See Table Notes of the combo box.

Actions

The action of the button can be set with the field level setAction method with trigger names "MouseUp", "MouseDown", "MouseEnter", "MouseExit", "OnFocus", "OnBlur". For example,

```
f.setAction("MouseUp", "app.beep(0);");
```

Selection Change

The action of the button can be set with the field level setAction method and a trigger name of "Keystroke". The UI has several categories of formatting, the JavaScript counterparts are listed in the table below. Except for custom formatting, all formats can be realized by using the formatting functions contained in *Aform.js*

Example:

```
f.setAction("Keystroke", "ProcessSelection();");
```

Miscellaneous Programming Notes

See the Miscellaneous Programming Notes of the combo box.

Radio Button

A form field may be created either by the UI for Acrobat, or by the addField method of the Doc Object. Programmatically, a radio button field is created as follows:

```
this.addField("myRadio", "radiobutton", 0, [400, 442, 412, 430]);
this.addField("myRadio", "radiobutton", 0, [400, 427, 412, 415]);
var f = this.addField("myRadio", "radiobutton", 0, [400, 412, 412, 400]);
f.setExportValues(["Yes", "No", "Sometimes"]);
```

This would create a series of three radio buttons on page 0; each radio button would be 12 points wide and 12 points high. A default appearance is given to the field. The return value of this method is a field object that will be used throughout the rest of this section. The export values of the different buttons are defined by using setFocus.

The field object for a field that is already existent can be obtained by the getDataObject method of the doc object:

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

The UI for a radio button is exactly the same as that of a check box. See the section on Check Box to see how to change the appearance, set the options and actions of a radio button field.

Field Properties

The top of the UI, above the tab fields, there are listed three general properties. The properties are same for button, see the Field Properties for buttons.

Appearance

The Appearance is the same as that of the combo box.

Options

The Options of a radio button field is the same as that of a check box. The Table Notes are applicable as well.

Actions

The action of the button can be set with the field level setAction method with trigger names "MouseUp", "MouseDown", "MouseEnter", "MouseExit", "OnFocus", "OnBlur". For example,

```
f.setAction("MouseUp", "app.beep(0);")
```

Signature

A form field may be created either by the UI for Acrobat, or by the addField method of the Doc Object. Programmatically, a combo box is created as follows:

```
var f = this.addField("mySignature", "signature", 0, [200, 500, 500, 400]);
```

This would create a signature field on page 0, and located at `[200, 500, 500, 400]`; i.e., the signature field would be 300 points wide and 100 points high. A default appearance is given to the field. The return value of this method is a field object that will be used throughout the rest of this section.

The field object for a field that is already existent can be obtained by the getField method of the Doc Object:

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

Field Properties

The top of the UI, above the tab fields, there are listed three general properties. The properties are same for button, see the Field Properties for buttons.

Appearance

The Appearance is the same as that of the combo box.

Actions

The action of the button can be set with the field level setAction method with trigger names "MouseUp", "MouseDown", "MouseEnter", "MouseExit", "OnFocus", "OnBlur". For example,

```
f.setAction("MouseUp", "app.beep(0);")
```

Signed

The action of the button can be set with the field level setAction method and a trigger name of "Format". The UI has several categories of Signed, the JavaScript counterparts are listed in the table below. Except for custom formatting, all formats can be realized by using the formatting functions contained in *Aform.js*

The Signed Tab		
Lock all fields (just these fields, all fields except these)	AFSignature_Format	f.setAction("Format", 'AFSignature_Format("THESE", new Array ("mySignature"));');
Custom		see table notes

Table Notes

- Custom: Any script that does not use AFSignature_Format is classified as custom.

An Example

Here is a complete example to create, sign, and lock a signature field using JavaScript

```
// Create signature field dynamically
var f = this.addField("mySignature", "signature", 0, [200, 500, 500, 400]);
f.strokeColor = color.black;

// set it to lock when signed
f.setAction("Format",
  'AFSignature_Format("THESE", new Array ("mySignature"));' );

var ppklite = security.getHandler("Adobe.PPKLite");    // choose handler
```

```
ppklite.login("dps017", "/C/signatures/DPSmith.apf");  // login
f.signatureSign(ppklite,                               // sign it
  { password: "dps017",
    location: "San Jose, CA",
    reason: "I am approving this document",
    contactInfo: "dpsmith@adobe.com",
    appearance: "Fancy"});
ppklite.logout();                                      // logout
```

Text

A form field may be created either by the UI for Acrobat, or by the addField method of the Doc Object. Programmatically, a combo box is created as follows:

```
var f = this.addField("myText", "text", 0, [200,516,344,500])
```

This would create a text field on page 0, and located at [200, 516, 344, 500]; i.e., the text field would be 144 points wide and 16 points high. A default appearance is given to the field. The return value of this method is a field object that will be used throughout the rest of this section.

The field object for a field that is already existent can be obtained by the getField method of the Doc Object:

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

Field Properties

The top of the UI, above the tab fields, there are listed three general properties. The properties are same for button, see the Field Properties for buttons.

Appearance

The Appearance is the same as that of the combo box.

Options

In the Options tab, the default text can be entered as various text field attributes can be set.

The Options Tab		
Default	defaultValue	f.defaultValue = "Name: ";
Alignment	alignment	f.alignment = "center";
Multiline	multiline	f.multiline = true;
Limit of # Characters	charLimit	f.charLimit = 40;

The Options Tab		
Password	password	f.password = true;
Field is used for File Selection	exportValues	f.fileSelect = false;
Do Not Spell Check	doNotSpellCheck	f.doNotSpellCheck = true;

Actions

The action of the button can be set with the field level setAction method with trigger names "MouseUp", "MouseDown", "MouseEnter", "MouseExit", "OnFocus", "OnBlur". For example,

```
f.setAction("MouseEnter", "app.beep(0);")
```

Format

The Format tab is the same as that of the combo box.

Validate

The Validate tab is the same as that of the combo box.

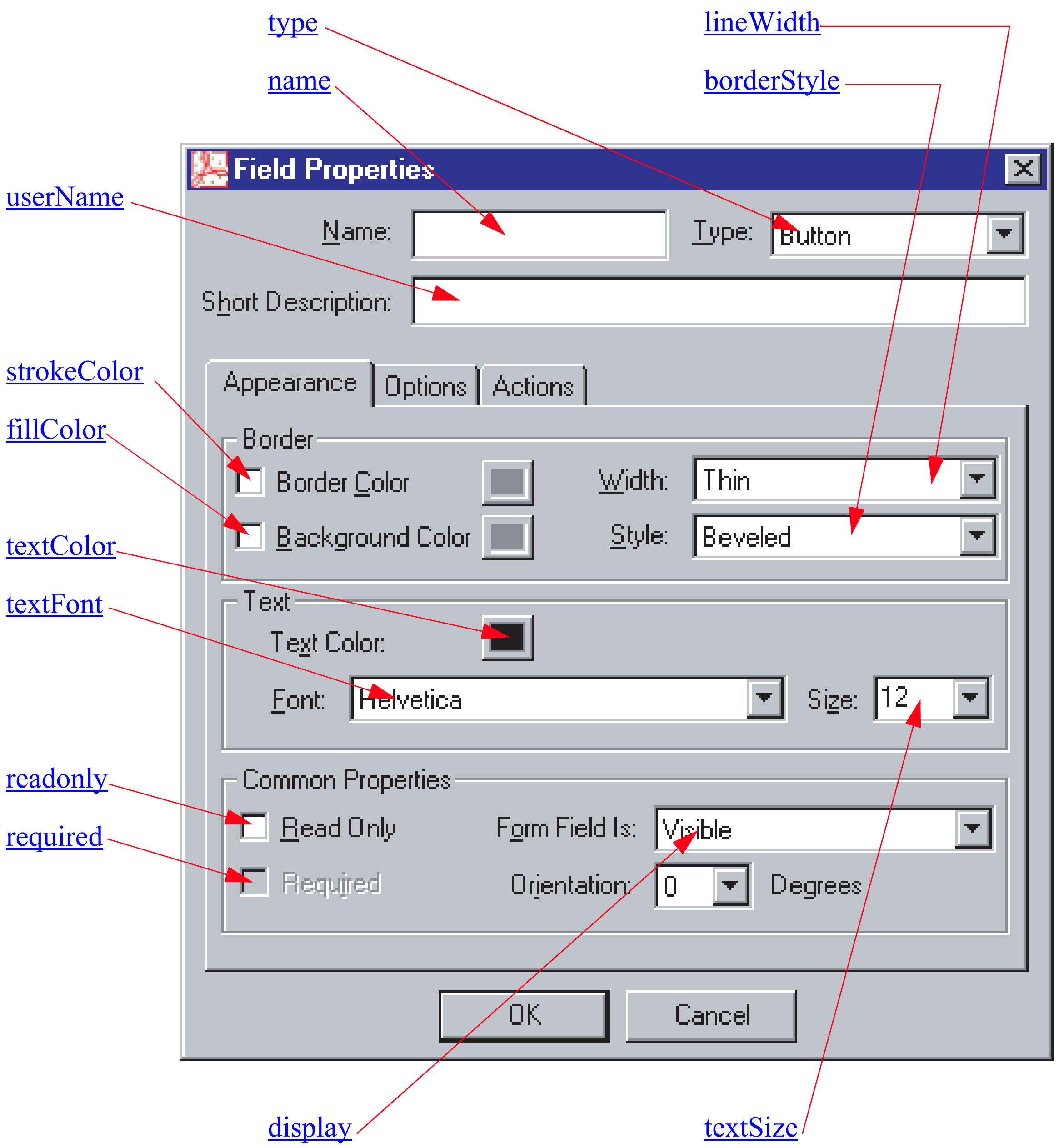
Calculate

The Calculate tab is the same as that of the combo box.

Quick Reference: Forms

Appearance: All Fields

The appearance tab is much the same for all field types.

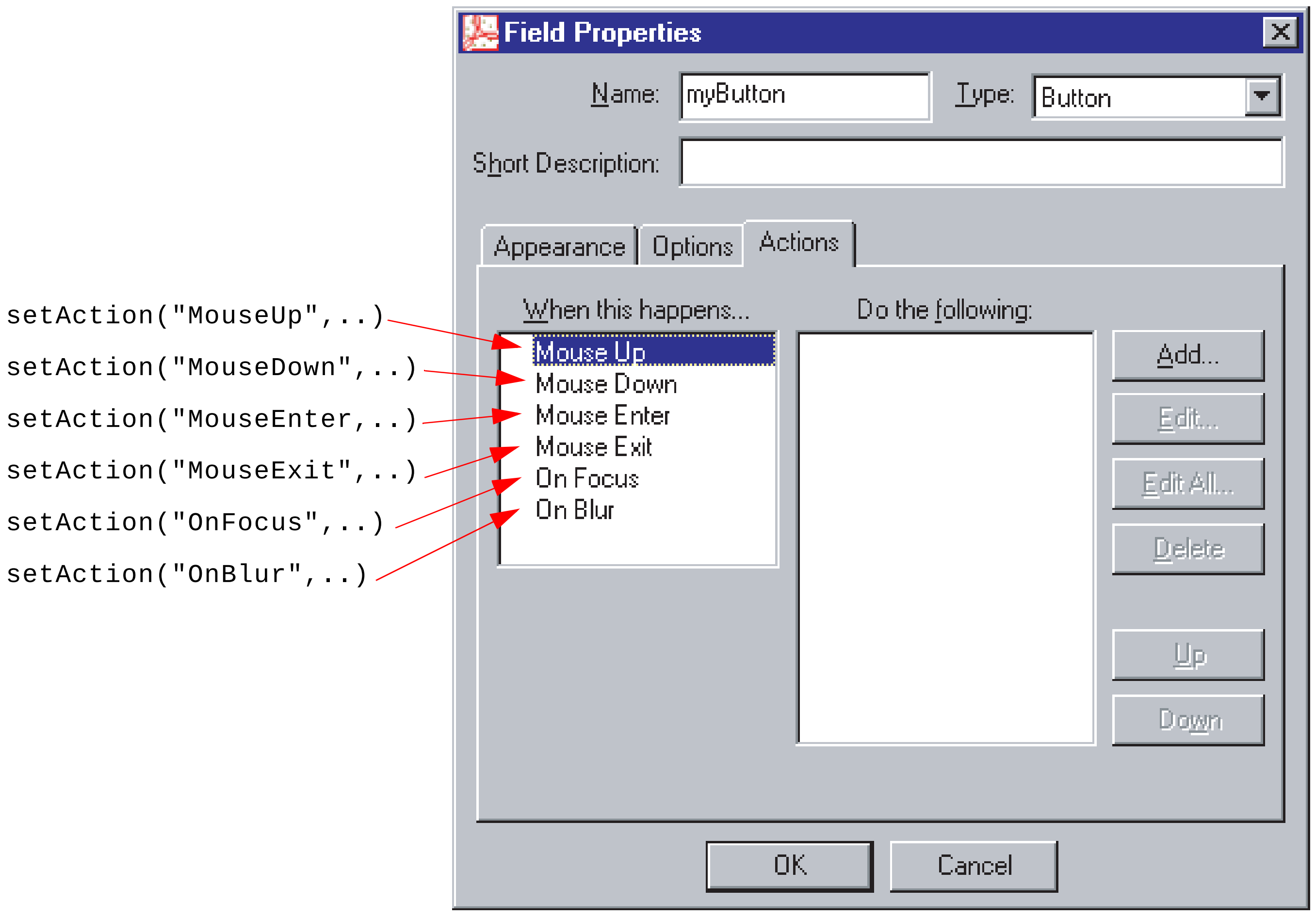


Action: All Fields

Actions can be set using the Field level method setAction, for example

```
        f.setAction("MouseUp", "app.beep(0);");
```

The first argument is the action trigger, and the second argument is the JavaScript to be executed when the trigger event occurs.



```
setAction("MouseUp",..)
setAction("MouseDown",..)
setAction("MouseEnter,..)
setAction("MouseExit",..)
setAction("OnFocus",..)
setAction("OnBlur",..)
```

Field Properties

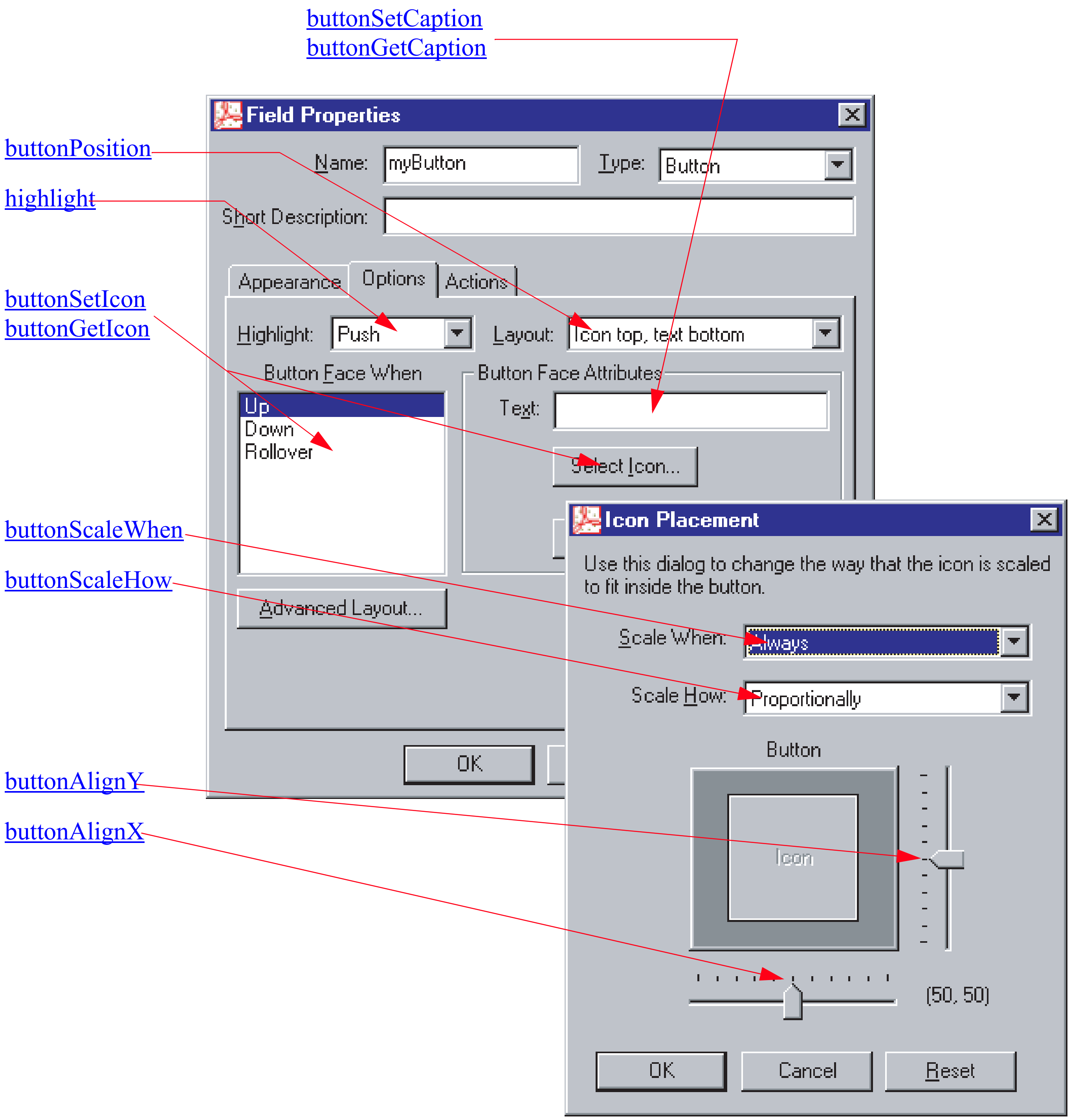
Name: myButton

Type: Button

Short Description:

Options: Buttons

For more details and examples, see the Options tab for the Button field.



Field Properties

Name: myButton
Type: Button
Short Description:

Appearance | Options | Actions

Highlight: Push
Layout: Icon top, text bottom

Button Face When
Up
Down
Rollover

Button Face Attributes
Text:
Select Icon...

Advanced Layout...

OK

Icon Placement

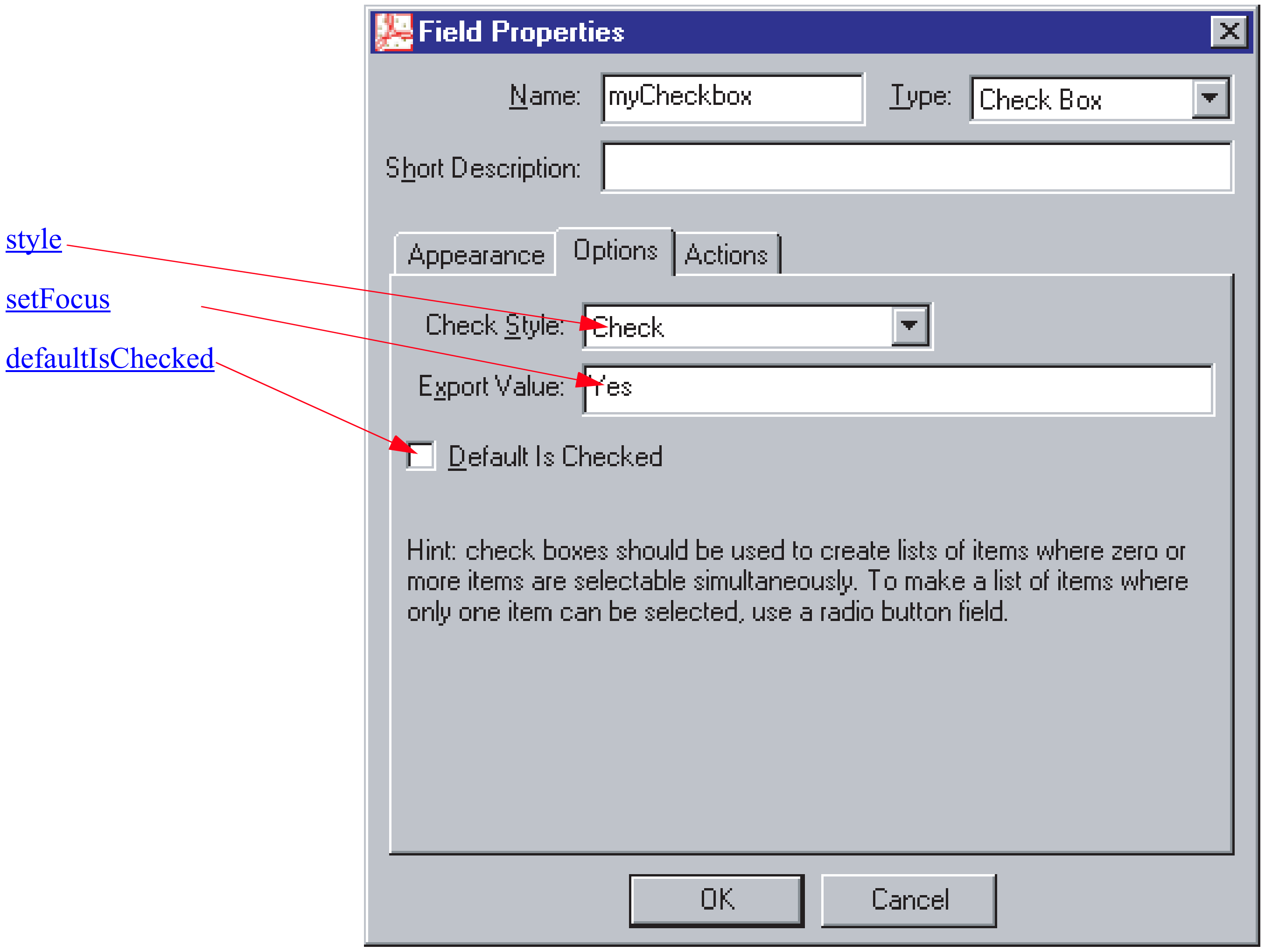
Use this dialog to change the way that the icon is scaled to fit inside the button.

Scale When: Always
Scale How: Proportionally

Options: Check Box and Radio Button

For more details and examples, see the Options tab for the Check Box and the Options tab for the Radio Button

.



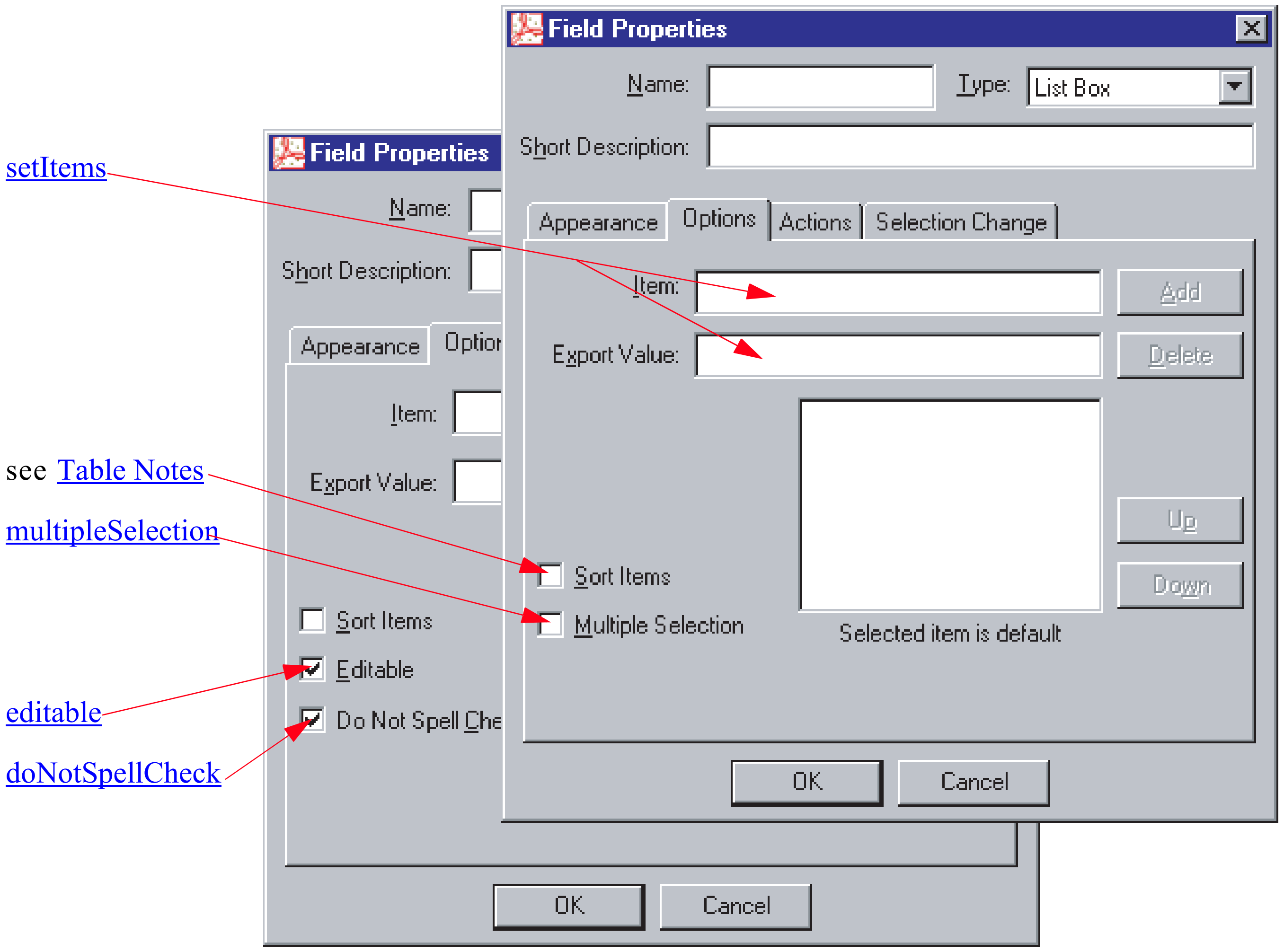
Name: myCheckbox
Type: Check Box
Short Description:
Check Style: Check
Export Value: Yes
Default Is Checked

Hint: check boxes should be used to create lists of items where zero or more items are selectable simultaneously. To make a list of items where only one item can be selected, use a radio button field.

Options: Combo and List Boxes

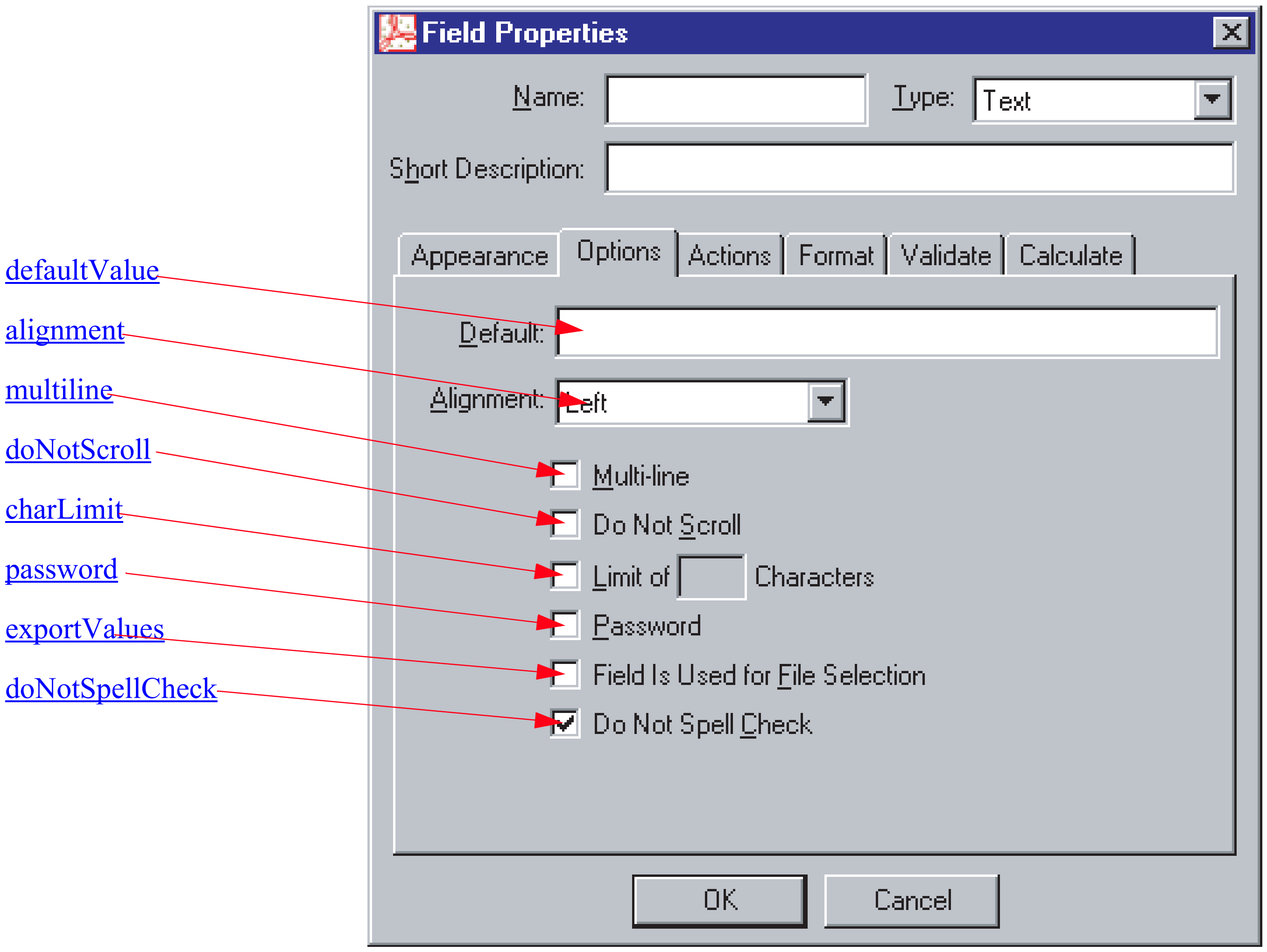
For more details and examples, see the Options tab for the Combo box and the Options tab for the List box.

Note: The Options tab for the List Box is shown on top, and the Options tab for the Combo Box is shown beneath it.



Options: Text Fields

For more details and examples, see the Options tab for the Text field.



Name:
Type: Text
Short Description:
Default:
Alignment: Left
☐ Multi-line
☐ Do Not Scroll
☐ Limit of Characters
☐ Password
☐ Field Is Used for File Selection
☑ Do Not Spell Check

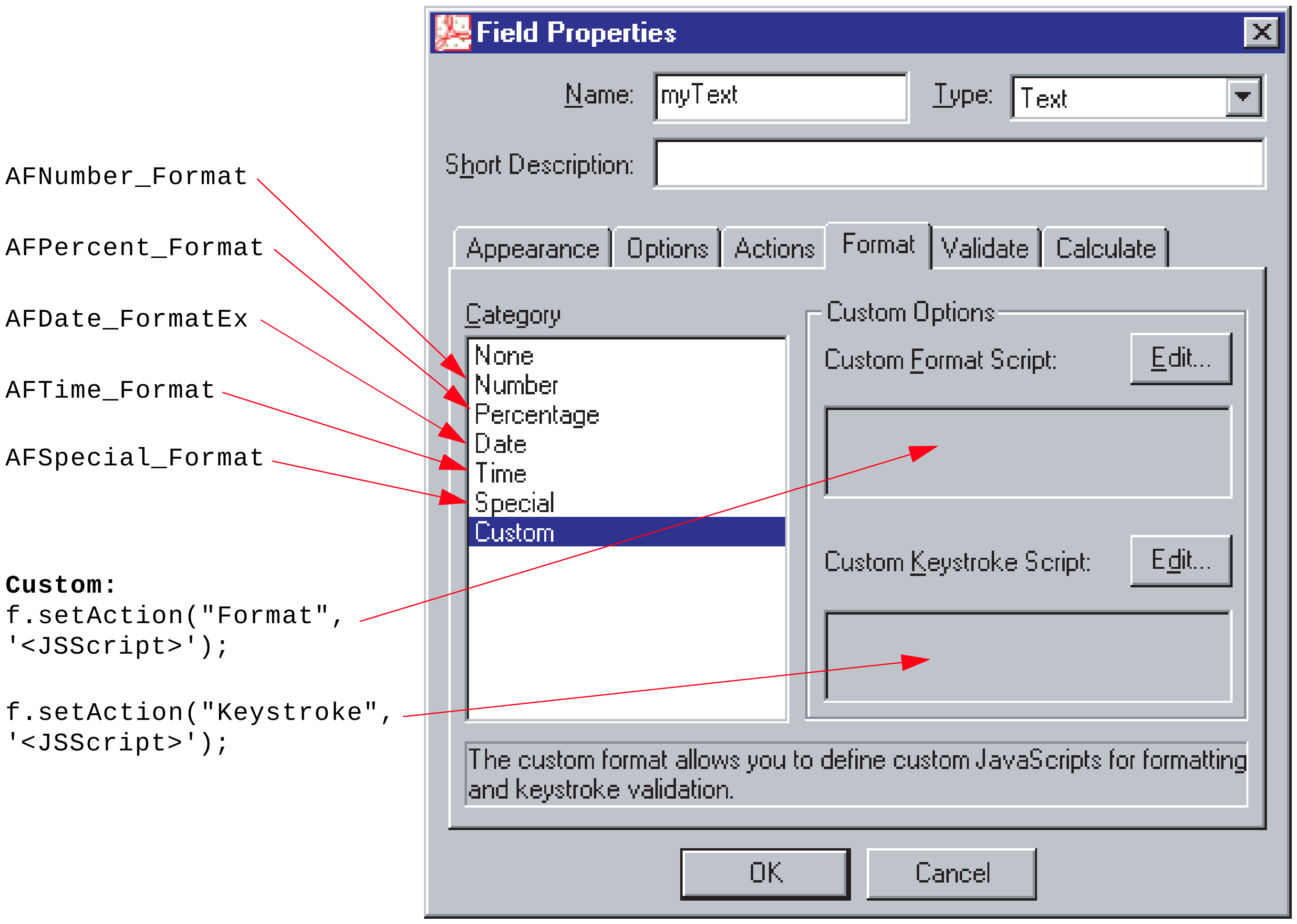
Format: Combo and Text

For more details and examples, see the Format tab for the Combo Box and the Format tab for the Text field.

Except for the custom keystroke scripts, all format categories are set using the field method setAction:

```
        f.setAction("Format", '<JSScript>');
```

Use the following JavaScript functions defined in aform.js for the JScript action



```
AFNumber_Format

AFPercent_Format

AFDate_FormatEx

AFTime_Format

AFSpecial_Format
```

Custom:

```
f.setAction("Format",
'<JSScript>');

f.setAction("Keystroke",
'<JSScript>');
```

Field Properties

Name: myText

Type: Text

Short Description:

Category: Custom

The custom format allows you to define custom JavaScripts for formatting and keystroke validation.

Validate: Combo and Text

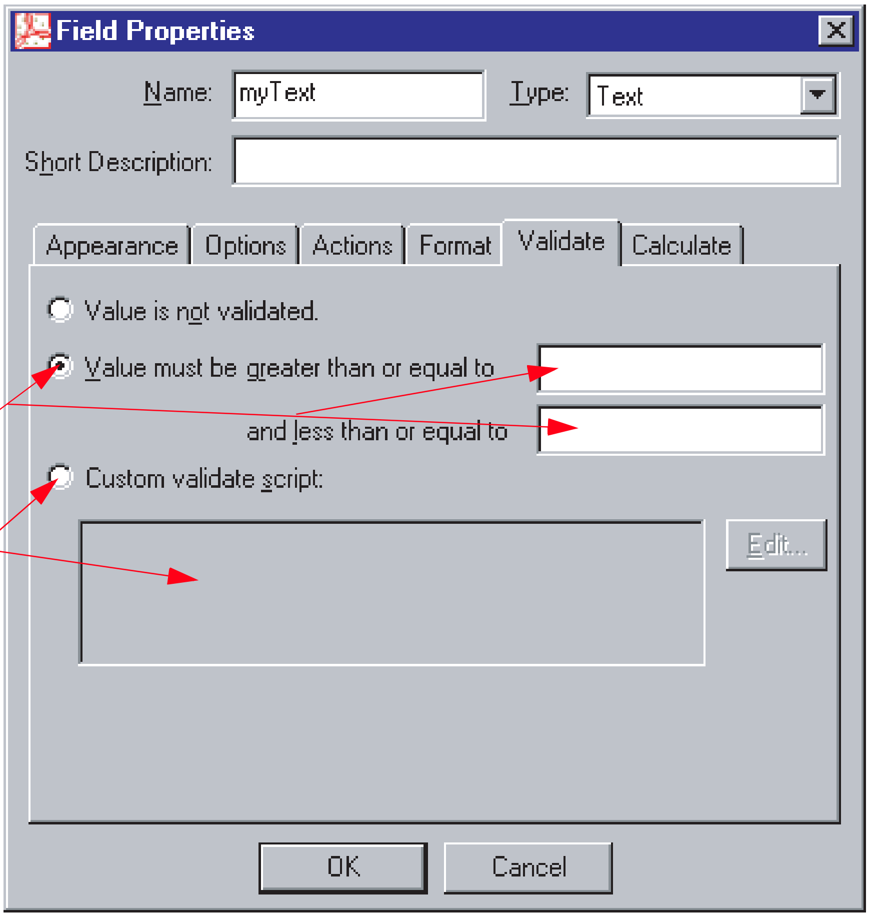
For more details and examples, see the Validate tab for the Combo Box and the Validate tab for the Text field.

Validate scripts are set using the field method setAction:

```
        f.setAction("Validate", '<JScript>');
```

```
f.setAction("Validate",
'AFSimple_Calculate(..)');
```

```
f.setAction("Validate",
'<JScript>');
```



Calculate: Combo and Text

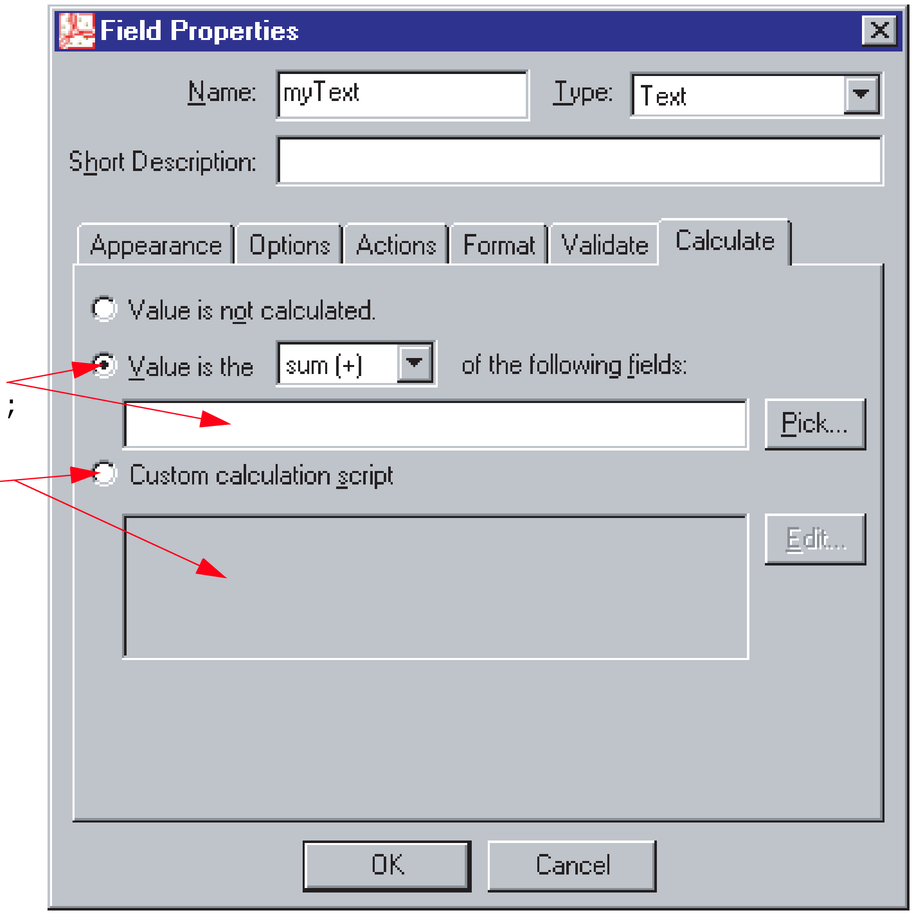
For more details and examples, see the Calculate tab for the Combo Box and the Calculate tab for the Text field.

Calculate scripts are set using the field method setAction:

```
        f.setAction("Calculate", '<JScript>');
```

```
f.setAction("Calculate",
'AFSimple_Calculate(...)');
```

```
f.setAction("Calculate",
'<JScript>');
```



Name: myText
Type: Text
Short Description:

Value is not calculated.
[✓ Value is the sum (+) of the following fields:]
Custom calculation script

Signed

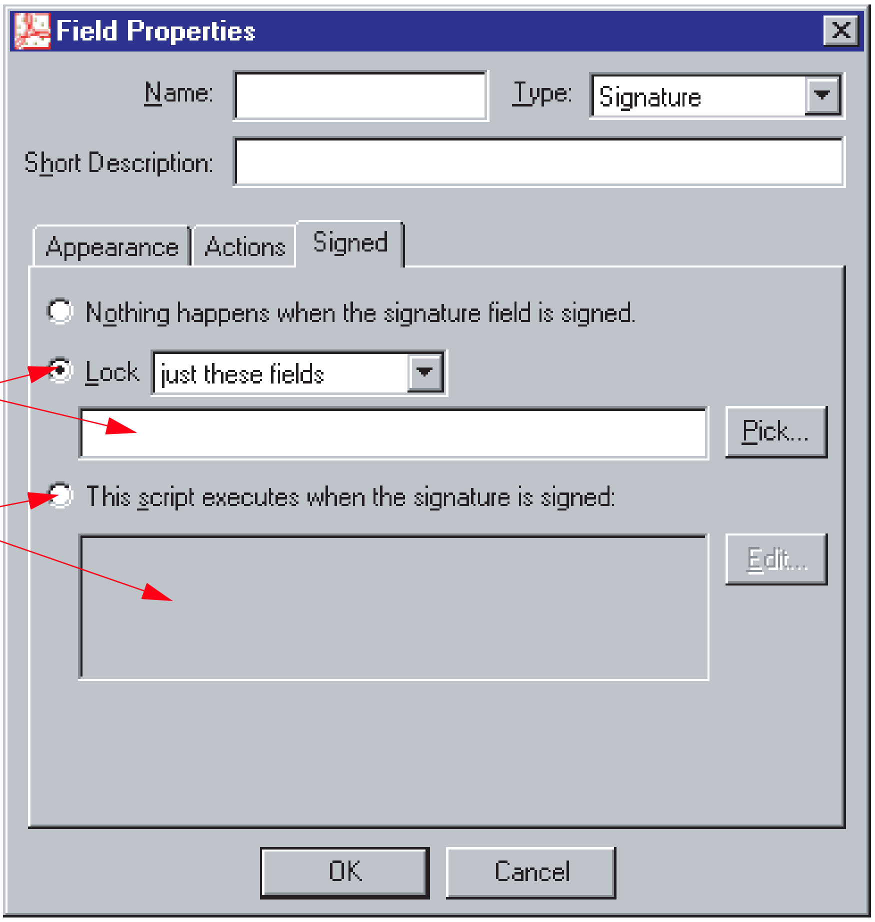
For more details and examples, see the Signed tab for the Signature field.

Signed scripts are set using the field method setAction:

```
        f.setAction("Format", '<JScript>');
```

```
f.setAction("Format",
'AFSignature_Format(...)');
```

```
f.setAction("Format",
'AFSignature_Format(...)');
```



Field Properties

Name:

Type: Signature

Short Description:

Appearance | Actions | Signed

Nothing happens when the signature field is signed.

[✓ Lock] just these fields

This script executes when the signature is signed:

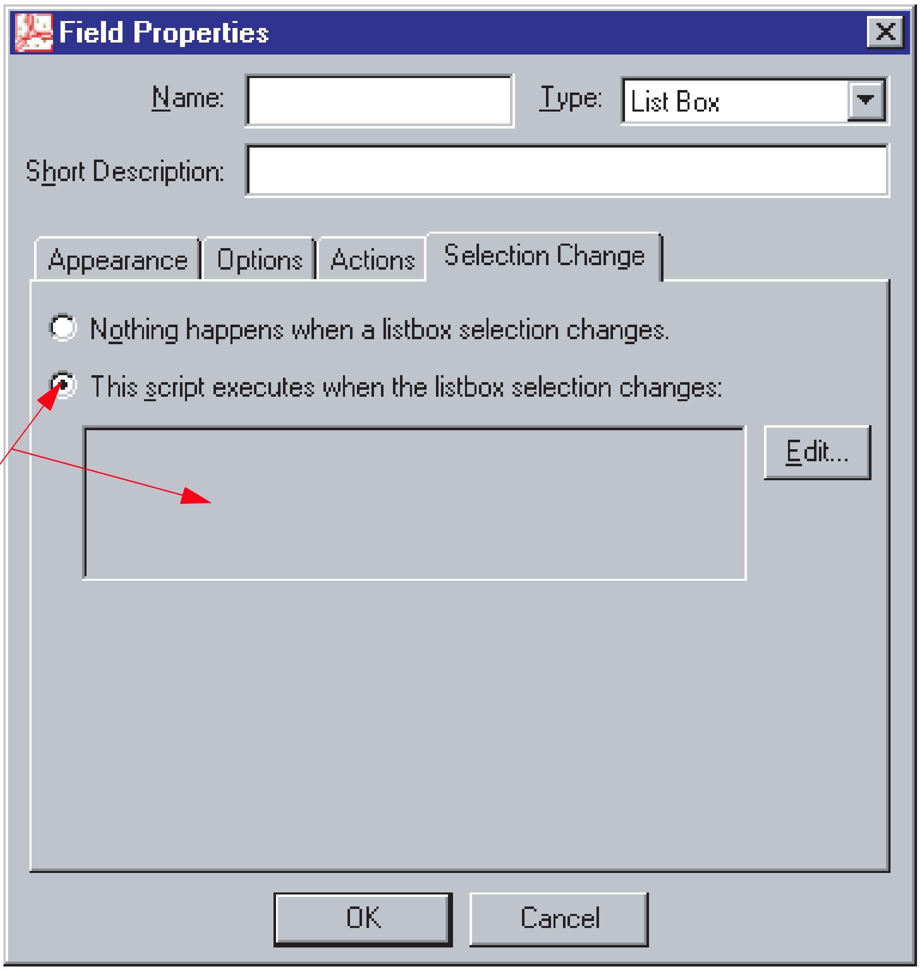
Selection Change

For more details and examples, see the Selection Change tab for the List Box field.

Signed scripts are set using the field method setAction:

```
        f.setAction("Keystroke", '<JScript>');
```

```
f.setAction("Keystroke",
'<JScript>');
```



( ) Nothing happens when a listbox selection changes.

(●) This script executes when the listbox selection changes:

How can I create an Annotation programmatically?

Acrobat provides a large number of JavaScript properties and methods for creating an annotation. The following is a quick outline of these methods.

There are thirteen types of annotations (only eleven of which are discussed here). They can be grouped by similarity of properties:

- Circle and Square Annotations
- Line Annotations
- Stamp Annotations
- FreeText Annotations
- Text Annotations
- Ink Annotations
- Highlight, Strikeout, Underline and Squiggle

Circle and Square Annotations

A circle annotation can be constructed with the addAnnot

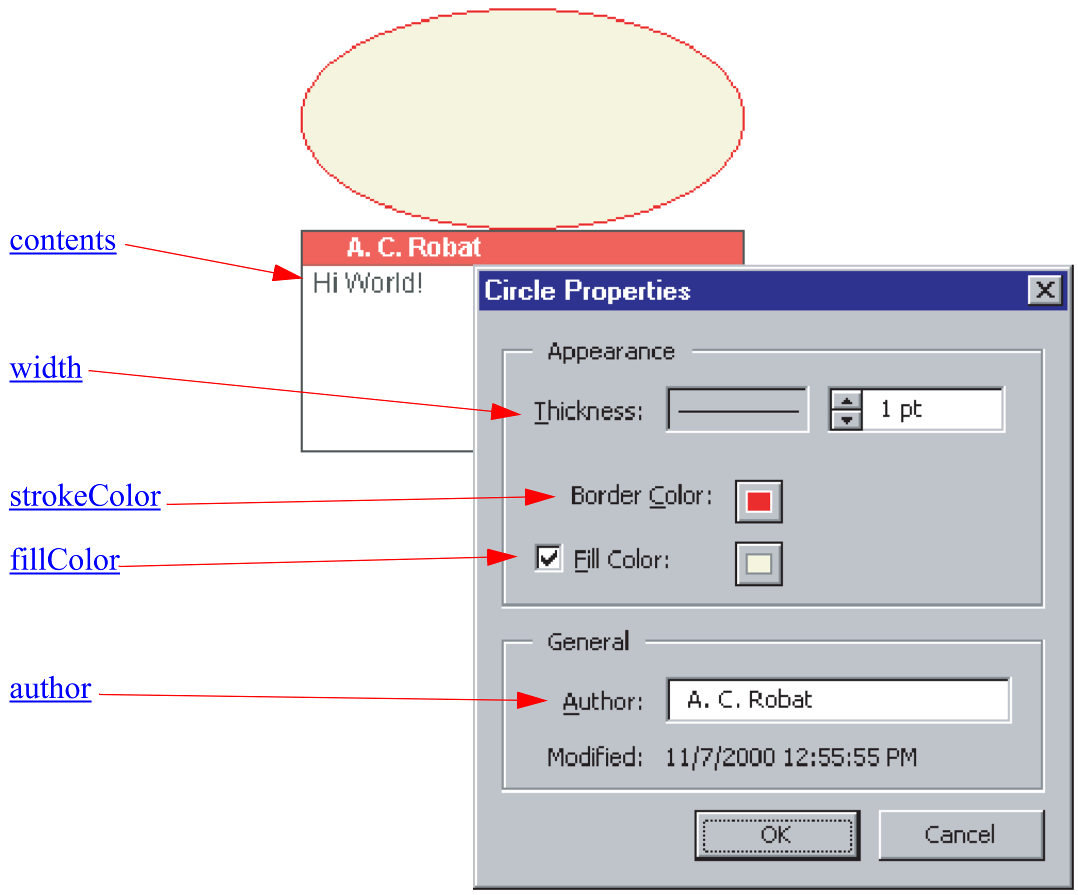
Example:

```
var annot = this.addAnnot(
{
    type: "Circle",
    page: 0,
    rect: [200,200,400,300],
    author: "A. C. Robat",
    name: "myCircle",
    popupOpen: true,
    popupRect: [200,100,400,200],
    contents: "Hi World!",
    strokeColor: color.red,
    fillColor: ["RGB",1,1,.855]
});
```

Here is a slight variation on the previous example; we use addAnnot and setProps.

Example:

```
var annot = this.addAnnot
    ({ type: "Square" });
annot.setProps
({
    page: 0,
    rect: [200,200,400,400],
    author: "A. C. Robat",
    name: "mySquare",
    popupOpen: true,
    popupRect: [200,100,400,200],
    contents: "Hi World!",
    strokeColor: color.red
});
```



Hi World!

Thickness: 1 pt

Border Color: 🟥

[✓] Fill Color:

Author: A. C. Robat

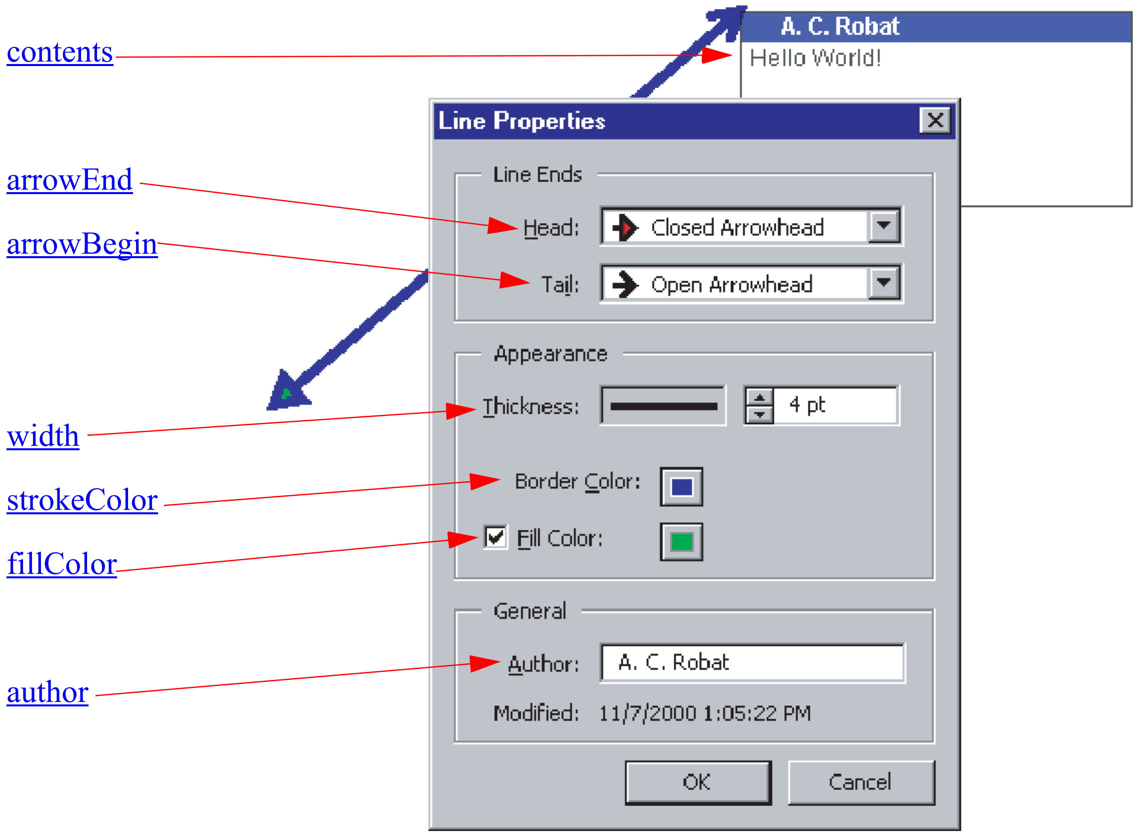
Modified: 11/7/2000 12:55:55 PM

Line Annotations

A line annotation can be constructed using addAnnot, or by using a combination of addAnnot and setProps (see Circle and Square Annotations for an example).

Example:

```
var annot = this.addAnnot
({
    type: "Line"
    page: 0,
    points: [[10,40],[200,200]],
    author: "A. C. Robat",
    name: "myLine",
    popupOpen: true,
    popupRect: [200, 100, 400, 200],
    arrowBegin: "ClosedArrow",
    arrowEnd: "OpenArrow",
    width: 4,
    contents: "Hello World!",
    strokeColor: color.red,         // border and text color
    fillColor: color.green          // background color
});
```



Hello World!

Head: Closed Arrowhead

Tail: Open Arrowhead

Thickness: 4 pt

Border Color:

Fill Color: ✓

Author: A. C. Robat

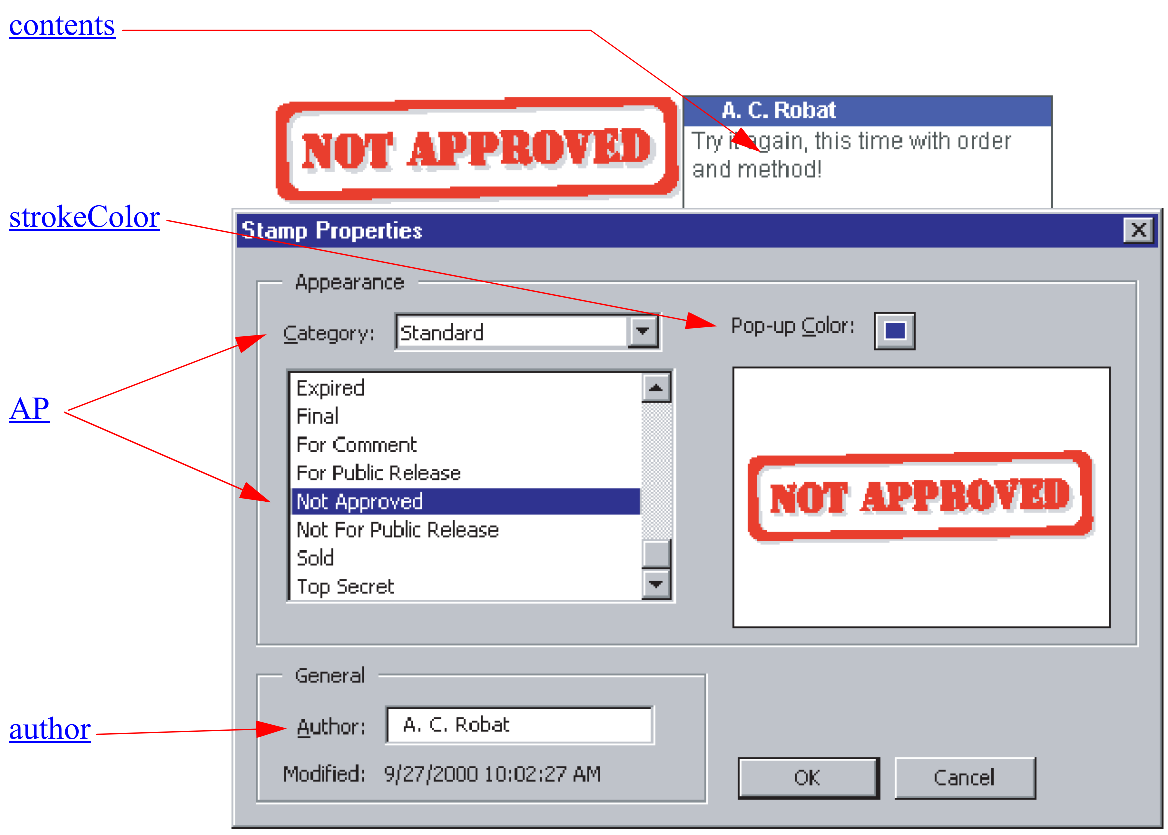
Modified: 11/7/2000 1:05:22 PM

Stamp Annotations

A line annotation can be constructed using addAnnot, or by using a combination of addAnnot and setProps (see Circle and Square Annotations for an example).

Example:

```
var annot = this.addAnnot
({
    page: 0,
    type: "Stamp",
    name: "myStamp",
    author: "A. C. Robat",
    rect: [400, 400, 550, 500],
    contents: "Try it again, this time with order and method!",
    strokeColor: color.blue,
    AP: "NotApproved"
});
```



A. C. Robat
Try it again, this time with order and method!

Category: Standard

Expired
Final
For Comment
For Public Release
Not Approved
Not For Public Release
Sold
Top Secret

Pop-up Color: 🟦

Author: A. C. Robat

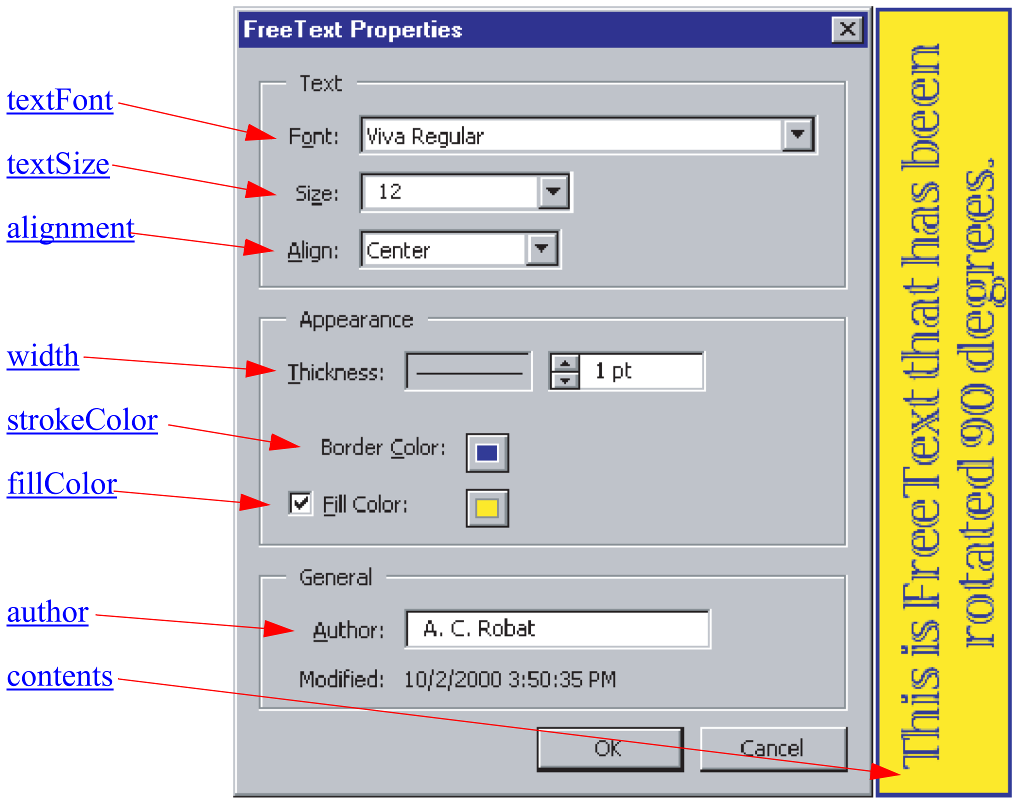
Modified: 9/27/2000 10:02:27 AM

FreeText Annotations

A FreeText annotation can be constructed using addAnnot, or by using a combination of addAnnot and setProps (see Circle and Square Annotations for an example).

Example:

```
var annot = this.addAnnot
({
    page: 0,
    type: "FreeText",
    author: "A. C. Robat",
    textFont: "Viva-Regular",
    textSize: 12,
    alignment: 1,
    rect: [10, 10, 42, 200],
    fillColor: ["RGB", 1, 1, 0],
    strokeColor: color.blue,
    name: "myFreeText",
    contents: "This is FreeText that has \
              been rotated 90 degrees.",
    rotate: 90 // no GUI for rotation
});
```



Font: Viva Regular
Size: 12
Align: Center
Thickness: 1 pt
Border Color:
Fill Color: ✓
Author: A. C. Robat
Modified: 10/2/2000 3:50:35 PM

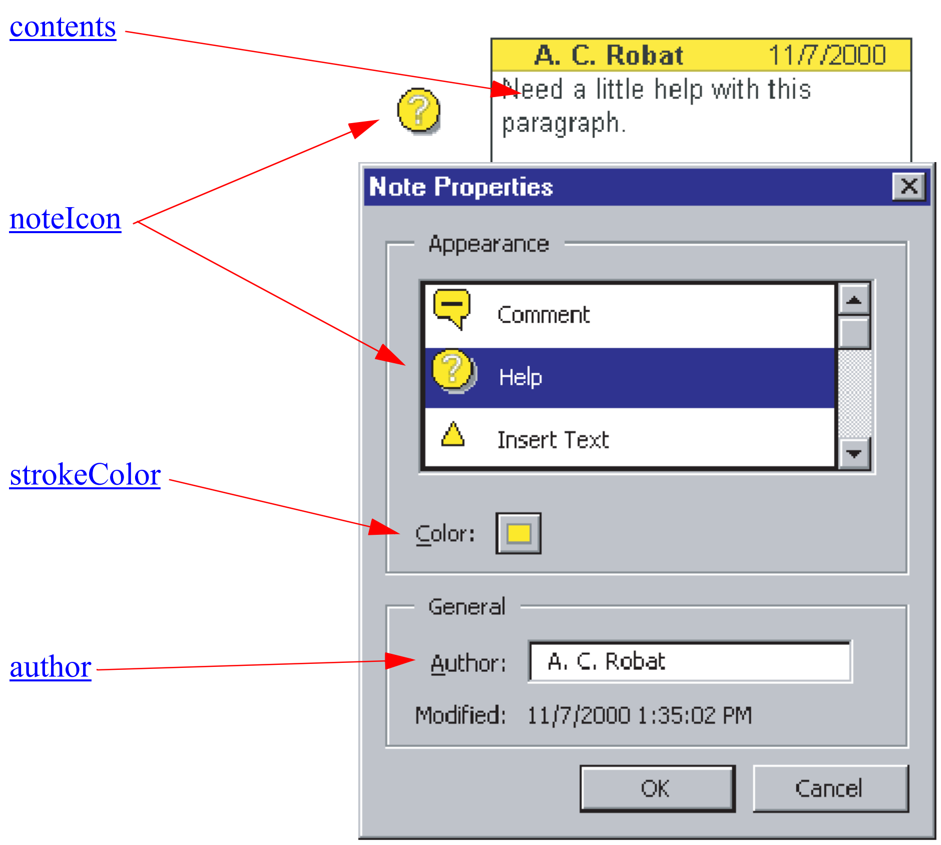
This is FreeText that has been rotated 90 degrees.

Text Annotations

A Text annotation can be constructed using addAnnot, or by using a combination of addAnnot and setProps (see Circle and Square Annotations for an example).

Example:

```
     var annot = this.addAnnot
     ({
         page: 0,
         type: "Text",
         author: "A. C. Robat",
         point: [300,400],
         strokeColor: color.yellow,
         name: "myHelp",
         contents: "Need a little help with this paragraph.",
         noteIcon: "Help"
     });
```



A. C. Robat 11/7/2000
Need a little help with this paragraph.

Author: A. C. Robat
Modified: 11/7/2000 1:35:02 PM

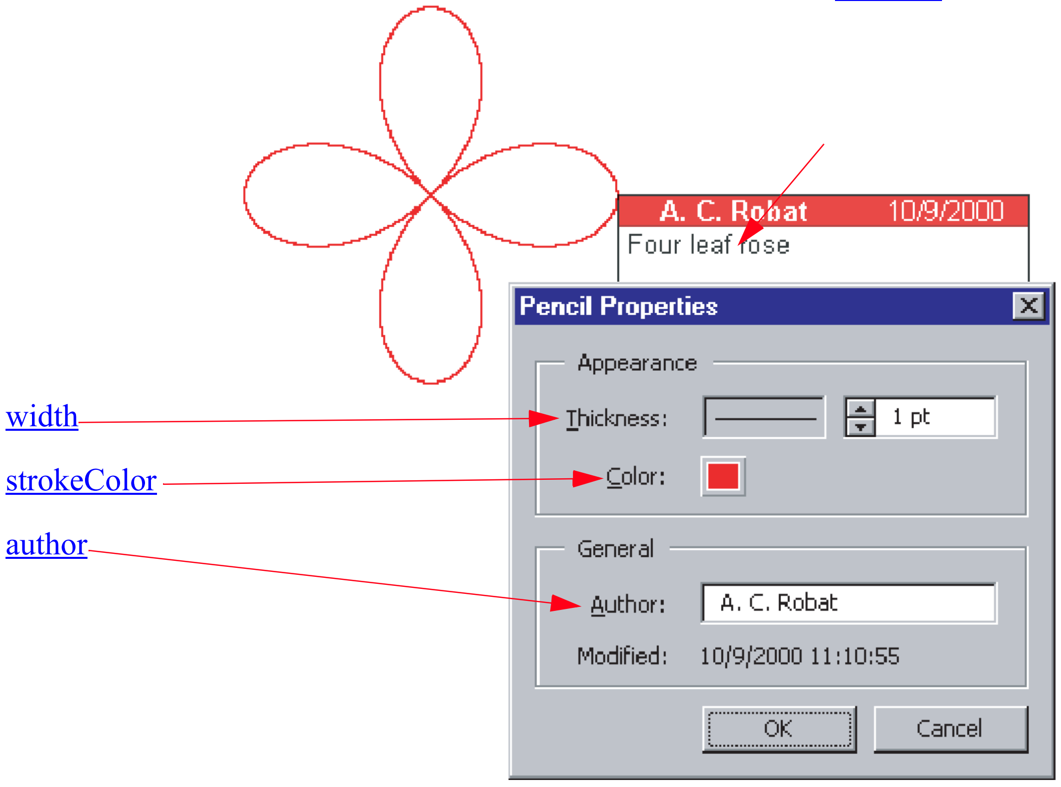
Ink Annotations

An Ink annotation can be constructed using addAnnot, or by using a combination of addAnnot and setProps (see Circle and Square Annotations for an example).

Example:

```
var inch = 72, x0 = 2*inch, y0 = 4*inch;
var scaledInch = .5*inch;
var nNodes = 60;
var theta = 2*Math.PI/nNodes;
var points = new Array();
for (var i = 0; i <=  nNodes; i++) {
    Theta = i*theta;
    points[i] = [x0 + 2*Math.cos(2*Theta)*Math.cos(Theta)*scaledInch,
      y0 + 2*Math.cos(2*Theta)*Math.sin(Theta)*scaledInch];
}
var annot = this.addAnnot({
    type: "Ink",
    page: 0,
    name: "myRose",
    author: "A. C. Robat",
    contents: "Four leaf rose",
    gestures: [points],
    strokeColor: color.red,
    width: 1
});
```

contents



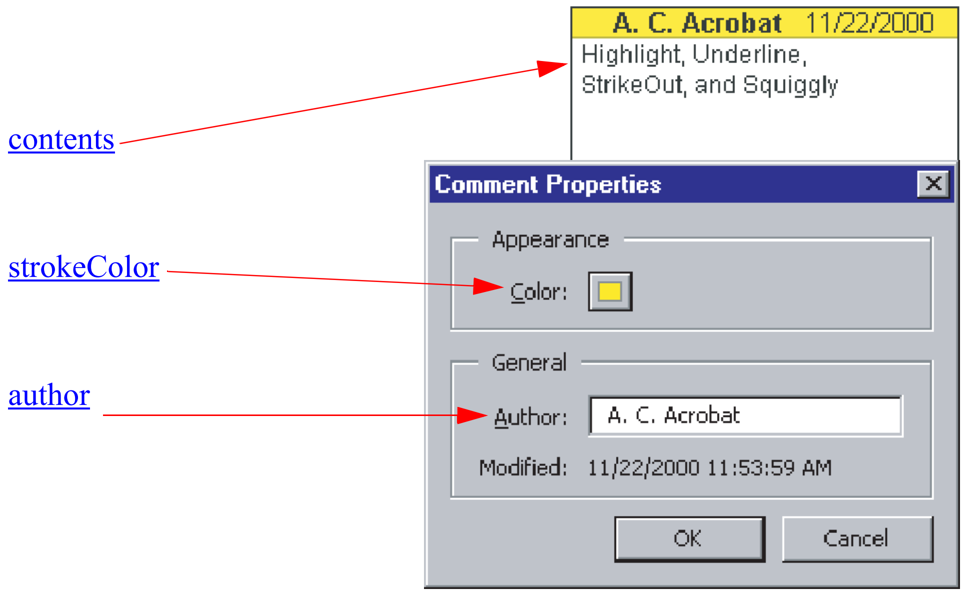
Highlight, Strikeout, Underline and Squiggle

A mark up annotations can be constructed using addAnnot, or by using a combination of addAnnot and setProps (see Circle and Square Annotations for an example).

Example: The following code would highlight the three words “mark up annotations” in the above sentence. The script searches through the current page for the first occurrence of the word “mark”. The quads are obtained for this word as well as the second word after “mark”, that’s the word “annotations”. The two sets of quads are combined to form a single set of quads for all words between “mark” and “annotation”. This is not a practical example, but illustrates some of the mechanics of working with quads.

```
var thisPage = this.pageNum;
var numWords = this.getPageNumWords(thisPage);
for ( var j = 0; j < numWords; j++) {
    nthWord = this.getPageNthWord(thisPage,j)
    if ( nthWord == "mark" ) {
        aQuadsFirst = this.getPageNthWordQuads(thisPage,j);
        aQuadsLast  = this.getPageNthWordQuads(thisPage,j+2);
        annot = this.addAnnot({
            page: thisPage,
            type: "Highlight",     // "Underline", "StrikeOut", or "Squiggly"
            strokeColor: color.yellow,
            quads: [[ aQuadsFirst[0][0], aQuadsFirst[0][1],
                       aQuadsLast[0][2], aQuadsLast[0][3],
                       aQuadsFirst[0][4], aQuadsFirst[0][5],
                       aQuadsLast[0][6], aQuadsLast[0][7]
                    ]],
            author: "A. C. Acrobat",
         contents: "Highlight, Underline,\rStrikeOut, and Squiggly"
        });
        break;
    }
}
```

Try it: Copy this code and paste it into the console. Highlight all the code and execute it by pressing Ctrl-Enter, or the Enter key on the number pad. The phrase “mark up annotation” above should be highlighted in yellow. See checkWord for a spell check example.



A. C. Acrobat 11/22/2000
Highlight, Underline,
StrikeOut, and Squiggly

Author: A. C. Acrobat
Modified: 11/22/2000 11:53:59 AM