Adobe Acrobat 7.0.5

Adobe®PDF

Acrobat JavaScript Scripting Guide

September 27, 2005

Adobe Solutions Network — http://partners.adobe.com

Contents

Contents

Chapter 10   Modifying the User Interface 159

Chapter 11   Search and Index Essentials 191

Preface

Introduction

Welcome to the Adobe® *Acrobat*® *JavaScript Scripting Guide.* This scripting guide is designed to provide you with an overview of how you can use Acrobat JavaScript to develop and enhance standard workflows, such as:

- Printing and viewing
- Spell-checking
- Stamping and watermarking
- Managing document security and rights
- Accessing metadata
- Facilitating online collaboration
- Creating interactive forms
- Customizing interaction with Web Services
- Interacting with databases

Here you will find detailed information and examples of what the Acrobat JavaScript capabilities are and how to access them, as well as descriptions of the usage of the SDK tools. Acrobat JavaScript is a powerful means by which you can enhance and extend both Acrobat and PDF document functionality.

What is Acrobat JavaScript?

Acrobat JavaScript is a language based on the core of JavaScript version 1.5 of ISO-16262, formerly known as ECMAScript, an object-oriented scripting language developed by Netscape Communications. JavaScript was created to offload Web page processing from a server onto a client in Web-based applications. Acrobat JavaScript implements extensions, in the form of new objects and their accompanying methods and properties, to the JavaScript language. These Acrobat-specific objects enable a developer to manage document security, communicate with a database, handle file attachments, manipulate a PDF file so that it behaves as an interactive, web-enabled form, and so on. Because the Acrobat-specific objects are added on top of core JavaScript, you still have access to its standard classes, including **Math**, **String**, **Date**, **Array**, and **RegExp**.

PDF documents have great versatility since they can be displayed both within the Acrobat software as well as a Web browser. Therefore, it is important to be aware of the differences between Acrobat JavaScript and JavaScript used in a Web browser, also known as *HTML JavaScript*:

● Acrobat JavaScript does not have access to objects within an HTML page. Similarly, HTML JavaScript cannot access objects within a PDF file.

● HTML JavaScript is able to manipulate such objects as `Window`. Acrobat JavaScript cannot access this particular object but it can manipulate PDF-specific objects.

If you have used previous versions of Acrobat JavaScript, please note that there are a number of new JavaScript features and enhancements in version 7:

● JavaScript Byte Code

● Adobe PDF document creation

● Additional usage rights

● Engineering features

● File attachments

● Additional language support

● Forms authoring and management

● Review, markup, and approval

● Document security and digital signatures

● Accessibility

● Print production

● XML capabilities

Audience

It is assumed that you are an Acrobat solution provider or power user, and that you possess basic competency with JavaScript. If you would also like to take full advantage of Acrobat's web-based features, you will find it useful to understand XML, XSLT, SOAP, and Web services. Finally, if you would like to utilize Acrobat's database capabilities, you will need a basic understanding of SQL.

Purpose and Scope

The purpose of this guide is to:

- Describe how you can use the Acrobat JavaScript language as the primary vehicle with which to develop and deploy Acrobat workflow solutions.

- Provide you with easily understood, detailed information and examples of the Acrobat JavaScript scripting features.

- Provide you with references to other resources where you can learn more about Acrobat JavaScript and related technologies.

After reading this guide and completing the exercises, you should be equipped to start using Acrobat JavaScript. During the development process, you will find that the descriptions and examples in this guide will provide you with enough direction and background to enable you to successfully complete your projects.

Assumptions

This guide assumes that you are familiar with the non-scripting elements of the Acrobat 7 user interface that are described in Acrobat's accompanying online help documentation. To work through the exercises in this guide, you will need to use Acrobat 7 Professional.

How To Use This Guide

This guide includes exercises that give you an opportunity to work directly with Acrobat JavaScript. If you plan to practice any of the exercises, do the following:

1. Be sure that you have Acrobat Professional installed on your Windows® or Macintosh® workstation. The exercises are designed to work on Windows and Macintosh versions of Acrobat, unless otherwise noted.

2. Create and use the same directory on your machine for all the JavaScript exercises you would like to try. You will use this directory to store the PDF documents and other files used in the exercises.

3. The Acrobat SDK contains a set of `.zip` files you will need to work through the exercises. You should extract the contents of these files to your local directory.

Font Conventions Used in This Book

The Acrobat documentation uses text styles according to the following conventions.

Font	Used for	Examples
monospaced	Paths and filenames	`C:\templates\mytmpl.fm`
	Code examples set off from plain text	These are variable declarations: `AVMenu commandMenu,helpMenu;`
monospaced bold	Code items within plain text	The **`GetExtensionID`** method ...
	Parameter names and literal values in reference documents	The enumeration terminates if **`proc`** returns **`false`**.
monospaced italic	Pseudocode	`ACCB1 void ACCB2 ExeProc(void)` `{ do something }`
	Placeholders in code examples	`AFSimple_Calculate(cFunction,` `cFields)`
blue	Live links to Web pages	The Adobe Solutions Network URL is: http://partners.adobe.com/asn/
	Live links to sections within this document	See Using the SDK.
	Live links to code items within this document	Test whether an **ASAtom** exists.
bold	PostScript language and PDF operators, keywords, dictionary key names	The **setpagedevice** operator
	User interface names	The **File** menu
italic	Document titles that are not live links	*Acrobat Core API Overview*
	New terms	*User space* specifies coordinates for...
	PostScript variables	*filename* **deletefile**

Related Documents

This guide refers to the following sources for additional information about Acrobat JavaScript and related technologies:

● *Acrobat JavaScript Scripting Reference*

This document is the companion reference to this scripting guide. It provides detailed descriptions of all the Acrobat JavaScript objects.

● *Adobe Acrobat Help*

This online document is included with Acrobat.

● Acrobat Solutions Network

http://partners.adobe.com/asn/

● *PDF Reference*

● *Guide to SDK Samples*

● *Developing for Adobe Reader*

If, for some reason, you did not install the entire Acrobat SDK and you do not have all the documents, please visit the Adobe Solutions Network Web site to find the documents you need.

1 Acrobat JavaScript Overview

Introduction

This chapter introduces the Acrobat JavaScript objects and containment hierarchies, as well as the primary Acrobat and PDF capabilities related to Acrobat JavaScript usage.

Chapter Goals

At the end of this chapter, you will be able to:

● List the Acrobat JavaScript objects and describe their purposes.

● Describe how Acrobat JavaScript can be used to extend the functionality of Acrobat.

● Identify the primary workflows that may be achieved with Acrobat JavaScript.

Contents

Topics
Acrobat JavaScript Introduction
Acrobat JavaScript Object Summary
What Can You Do with Acrobat JavaScript?

Acrobat JavaScript Introduction

Most people know Acrobat as a medium for exchanging and viewing electronic documents easily and reliably, independent of the environment in which they were created. However, Acrobat provides far more capabilities than a simple document viewer.

You can enhance a PDF document so that it contains form fields to capture user-entered data as well as buttons to initiate user actions. This type of PDF document can replace existing paper forms, allowing employees within a company to fill out forms and submit them via PDF files, and connect their solutions to enterprise workflows by virtue of their XML-based structure and the accompanying support for SOAP-based Web Services.

Acrobat also contains functionality to support *online team review*. Documents that are ready for review are converted to PDF. When a reviewer views a PDF document in Acrobat and adds comments to it, those comments (or *annotations*) constitute an additional layer of information on top of the base document. Acrobat supports a wide variety of standard comment types, such as a note, graphic, sound, or movie. To share comments on a document with others, such as the author and other reviewers, a reviewer can export just the comment "layer" to a separate comment repository.

In either of these scenarios, as well as others that are not mentioned here, you can customize the behavior of a particular PDF document, implement security policies, interact with databases and web services, and dynamically alter the appearance of a PDF document by using Acrobat JavaScript. You can tie Acrobat JavaScript code to a specific PDF document, a particular page within a PDF document, or a form field or button in a PDF file. When an end user interacts with Acrobat or a PDF file displayed in Acrobat that contains JavaScript, Acrobat monitors the interaction and executes the appropriate JavaScript code.

Not only can you customize the behavior of PDF documents in Acrobat, but you can customize Acrobat itself. In earlier versions of Acrobat (prior to Acrobat 5), this type of customization could only be done by writing Acrobat plug-ins in a high-level language like C or C++. Now, much of that same functionality is available through Acrobat JavaScript extensions. You will find that using Acrobat JavaScript to perform a task such as adding a menu to Acrobat's user interface is much easier to do than writing a plug-in.

Acrobat JavaScripts can be created for batch processing of multiple documents, processing within a single document, processing for a given page, and processing for a single form field. For batch processing, it is possible to execute JavaScript on a set of PDF files, which enables tasks such as extracting comments from a comment repository, identifying spelling errors, and automatically printing PDF files.

Acrobat JavaScript Object Summary

Acrobat JavaScript defines several objects that allow your code to interact with Acrobat, a PDF document, or form fields within a PDF document. This section introduces you to the primary objects used to access and control the application and document, the development environment itself, and general-purpose JavaScript functionality.

The most significant objects available control Acrobat, the Acrobat JavaScript debugger, the Acrobat JavaScript console, the PDF document, the Adobe Dialog Manager®, Web services, databases, security, searches, and JavaScript events.

TABLE 1.1 *Primary Acrobat JavaScript Objects*

Object	Purpose
app	Acrobat
doc	PDF document
dbg	JavaScript debugger
console	JavaScript console
global	Persistent and cross-document information
util	JavaScript utility methods
dialog	Adobe Dialog Manager (ADM)
security	Encryption and digital signatures
SOAP	Web Services
search	Searching and indexing
ADBC	Database connections and queries
event	JavaScript events

app

The **app** object is a static object that represents the Acrobat application itself. It offers a number of Acrobat-specific functions in addition to a variety of utility routines and convenience functions. By interacting with the **app** object, you can open or create PDF and FDF documents, and customize the Acrobat interface by setting its viewing modes, displaying popup menus, alerts, and thermometers, displaying a modal dialog box, controlling time intervals, controlling whether calculations will be performed, performing email operations, and modifying its collection of toolbar buttons, menus, and menu items. You can also query **app** to determine which Adobe product and version the end user is using (such as Reader 6.0 or Acrobat Professional 7.0), as well as which printer names and color spaces are available.

doc

The **doc** object is the primary interface to the PDF document, and it can be used to access and manipulate its content. The **doc** object provides the interfaces between a PDF document open in the viewer and the JavaScript interpreter. By interacting with the **doc** object, you can get general information about the document, navigate within the document, control its structure, behavior and format, create new content within the document, and access objects contained within the document, including bookmarks, form fields, templates, annotations, and sounds.

FIGURE 1.1   Doc Object Containment Hierarchy
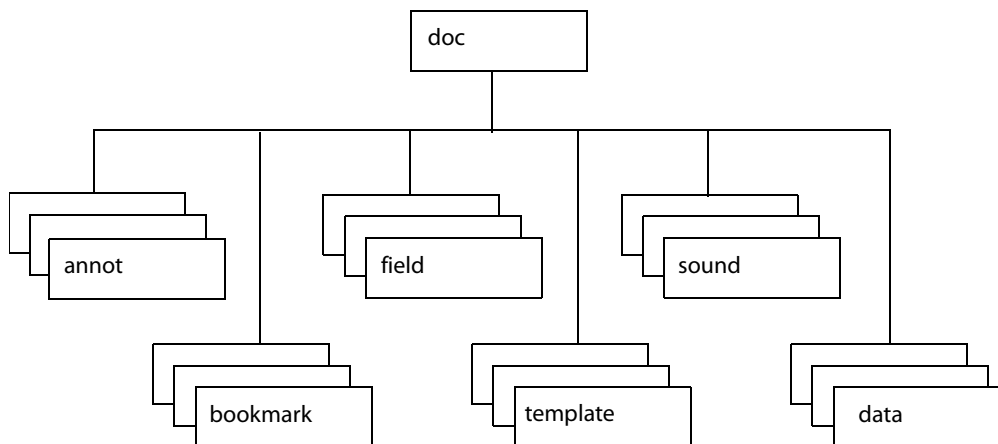
Figure 1.1 represents the containment hierarchy of objects related to the **doc** object.

Accessing the **doc** object from JavaScript can be done in a variety of ways. The most common method is using the **this** object, which is normally equivalent to the **doc** object of the current underlying document.

dbg

You can use the **dbg** object, available only in Acrobat Professional, to optionally control the JavaScript debugger from a command line standpoint while the application is not executing a modal dialog box. The **dbg** object methods offer the same functionality as the buttons in the JavaScript debugger dialog toolbar, which permit stepwise execution, setting, removing, and inspecting breakpoints, and quitting the debugger.

console

The **console** object is a static object available within Acrobat Professional that is used to access the JavaScript console for displaying debug messages and executing JavaScript. It does not function in Adobe Reader or Acrobat Standard. It is useful as a debugging aid and as a means of interactively testing code.

global

The **global** object is used to store data that is persistent across invocations of Acrobat or shared by multiple documents. Global data sharing and notification across multiple documents is done through a subscription mechanism, which enables monitoring of global variables and reporting of their values to all subscribing documents. In addition, **global** may be used to store information that pertains to a group of documents, a situation that occurs when a batch sequence runs. For example, batch sequence code often stores the total number of documents to be processed as a property of **global**. If information about the documents needs to be stored in a **report** object, it is assigned to a set of properties within **global** so it is accessible to the **report** object.

Util

The **Util** object is a static JavaScript object that defines a number of utility methods and convenience functions for number and date formatting and parsing. It can also be used to convert information between rich content and XML representations.

dialog

The **dialog** object is an object literal used by the **app** object's **execDialog()** method to present a modal dialog box identical in appearance and behavior to those used across all Adobe applications. The **dialog** object literal consists of a set of event handlers and properties which determine the behavior and contents of the dialog box, and may be comprised of the following elements: push buttons, checkboxes, radio buttons, listboxes, textboxes, popup controls, and containers and frames for sets of controls.

security

The `security` object is a static JavaScript object, available without restriction across all Acrobat applications including Adobe Reader, that employs a token-based security model to facilitate the creation and management of digital signatures and encryption in PDF documents, thus providing a means of user authentication and directory management. Its methods and properties are accessible during batch, console, menu, or application initialization events. The `security` object can be used to add passwords and set security options, add usage rights to a document, encrypt PDF files for a list of recipients, apply and assign security policies, create custom security policies, add security to document attachments, create and manage digital IDs using certificates, build a list of trusted identities, and check information on certificates.

SOAP

The `SOAP` object can be used to make remote procedure calls to a server and invoke Web Services described by the Web Services Description Language (WSDL), and supports both SOAP 1.1 and 1.2 encoding. Its methods are available from Acrobat Professional, Acrobat Standard, and for documents with Form export rights open in Adobe Reader 6.0 or later. The `SOAP` object makes it possible to share comments remotely and to invoke Web Services in form field events. It provides support for rich text responses and queries, HTTP authentication and WS-Security, SOAP headers, error handling, sending or converting file attachments, exchanging compressed binary data, document literal encoding, object serialization, XML streams, and applying DNS service discovery to find collaborative repositories on an Intranet. In addition the `XMLData` object can be used to evaluate XPath expressions and perform XSLT conversions on XML documents.

search

The `search` object is a static object that can be used to perform simple and advanced searches for text in one or more PDF documents or index files, create, update, rebuild, or purge indexes for one or more PDF documents, and search through document-level and object-level metadata. The `search` object has properties that may be used to fine-tune the query, such as a thesaurus, words with similar sounds, case-sensitivity, and settings to search the text both in annotations and in EXIF metadata contained in JPEG images.

ADBC

The `ADBC` object is used in conjunction with the `connection`, and `statement` objects to interface to a database. These Acrobat JavaScript objects constitute Acrobat Database Connectivity (ADBC), and provide a simplified means of utilizing ODBC calls to connect to a database and access its data. SQL statements are passed to the `statement` object's `execute()` method in order to insert, update, retrieve, and delete data.

event

All JavaScript actions are executed when a particular event occurs. For each event, Acrobat JavaScript creates an **event** object. When an event occurs, the **event** object can be used to obtain and manage any information associated with the state of that particular event. An **event** object is created for each of the following type of events: Acrobat Viewer initialization, batch sequences, mouse events on bookmarks, JavaScript console actions, document print, save, open, or close actions, page open and close events, form field mouse, keystroke, calculation, format, and validation events, and menu item selection events.

What Can You Do with Acrobat JavaScript?

Acrobat JavaScript enables you to do a wide variety of things within Acrobat and Adobe Reader, and within PDF documents. Acrobat JavaScript can be used to aid in the following workflows:

- Creating PDF documents
  – Create new PDF files
  – Control the appearance and behavior of PDF files
  – Convert PDF files to XML format
  – Create and spawn templates
  – Attach files to PDF documents

- Creating Acrobat forms
  – Create, modify, and fill in dynamically changing, interactive forms
  – Import and export form, attachment, and image data
  – Save form data in XML, XDP, or Microsoft Excel® format
  – Email completed forms
  – Make forms accessible to visually impaired users
  – Make forms Web-ready
  – Migrate legacy forms to dynamic XFA
  – Secure forms

- Facilitating review, markup, and approval
  – Setting comment repository preferences
  – Creating and managing comments
  – Approving documents using stamps

- Integrating digital media into documents
  – Controlling and managing media players and monitors
  – Adding movie and sound clips
  – Adding and managing renditions
  – Setting multimedia preferences

- Modifying the user interface
  – Using Adobe Dialog Manager (ADM)
  – Adding navigation to PDF documents
  – Managing PDF layers
  – Managing print production

- Searching and indexing of documents and document metadata
  – Perform searches for text in one or more documents
  – Create, update, rebuild, and purge indexes
  – Search document metadata

- Securing documents
  - Creating and managing digital signatures
  - Adding and managing passwords
  - Adding usage rights
  - Encrypting files
  - Managing digital certificates
- Managing usage rights
  - Writing JavaScript for Adobe Reader
  - Enabling collaboration
  - Security rights
  - Layer-specific rights
- Interacting with databases
  - Establishing an ADBC connection
  - Executing SQL statements
  - Support for ADO® (Windows only)
- Interacting with Web Services
  - Connection and method invocation
  - HTTP authentication and WS-Security®
  - SOAP header support
  - Error handling
  - Handling file attachments
  - Exchanging compressed binary data
  - Document literal encoding
  - Serializing objects
  - XML streams
  - Applying DNS service discovery to find collaborative repositories on an Intranet
- XML
  - Performing XSLT conversions on XML documents
  - Evaluating XPath expressions

2 Acrobat JavaScript Tools

Introduction

Acrobat provides an integrated development environment that offers several tools with which to develop and test Acrobat JavaScript functionality. These tools are the JavaScript Editor, Console, and Debugger. In addition, Acrobat supports the use of third-party editors for code development.

In this chapter, you will have the opportunity to practice using the editor, console, and debugger to evaluate scripts.

Chapter Goals

At the end of this chapter, you will be able to:

- Invoke the JavaScript console and use it to interactively execute code and display print statements.
- Use the editor to create and modify Acrobat JavaScript code.
- Specify the default editor to be used in your development efforts.
- Identify the extra capabilities that Acrobat supports on some external editors.
- Identify and understand how to use the following debugger controls and features:
  - Start the debugger at any point within a script.
  - Interactively execute code and display output to the console window.
  - Set and manage customized watches and breakpoints.
  - Inspect the details of variables.
  - Start a new debugging session without exiting the debugger.

Contents

Using the Acrobat JavaScript Console

The Acrobat JavaScript Console provides an interactive and convenient interface for testing portions of JavaScript code and experimenting with object properties and methods. Because of its interactive nature, the console behaves as an editor that permits the execution of single lines or blocks of code.

There are two ways to activate the Acrobat JavaScript Console: either through an Acrobat menu command or through the use of the static `console` object within Acrobat JavaScript code. In either case, it appears as a component of the Acrobat JavaScript Debugger, and the primary means of displaying values and results is through the `console.println()` method.

Opening the JavaScript Console

To open the Acrobat JavaScript console from within Acrobat:

1. Open the debugger window using one of these methods:
   – Select **Advanced > JavaScript > Debugger**, or
   – Type **Ctrl-j** (Windows) or **Command-j** (Macintosh)

2. Select either **Console** or **Script and Console** from the debugger's **View** list.

To open and close the console from Acrobat JavaScript code, use `console.show()` and `console.hide()`, respectively.

Executing JavaScript

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

● To evaluate a portion of a line of code, highlight the portion and press either the **Enter** key on the numeric keypad or type **Ctrl+Enter** on the regular keyboard.

● To evaluate a single line of code, make sure the cursor is positioned on that line and press either the **Enter** key on the numeric keypad or type **Ctrl+Enter** on the regular keyboard.

● To evaluate multiple lines of code, highlight those lines and press either the **Enter** key on the numeric keypad or type **Ctrl+Enter** on the regular keyboard.

In all cases, the result of the most recent single JavaScript statement executed is displayed in the console.

Formatting Code

To indent code in the JavaScript console, use the **Tab** key.

● To indent four spaces to the right, position the cursor at the beginning of a single line or highlight the block of code, and press the **Tab** key.

● To indent four spaces to the left, position the cursor at the beginning of a single line or highlight a block of code and press **Shift+Tab**.

Exercise: Working with the JavaScript Console

NOTE: To complete this exercise, you will need Acrobat Professional installed on your machine.

In this exercise you will verify that JavaScript is enabled for Acrobat and begin working with the Acrobat JavaScript Console to edit and evaluate code.

At the end of the exercise you will be able to:

● Enable or disable Acrobat JavaScript.

● Enable or disable the JavaScript debugger.

● Open the Acrobat Console.

● Evaluate code in the console window.

Enabling JavaScript

In order to use Acrobat JavaScript, you must first verify that JavaScript has been enabled. In order to execute code from the Acrobat Console, you will also need to ensure that the JavaScript Debugger is enabled, since the console window is a component within the JavaScript Debugger interface.

Enable JavaScript, the Debugger, and the Console by performing the following steps (see Figure 2.1 below):

1. Launch Acrobat.

2. Select **Edit > Preferences** to open the **Preferences** dialog box.

3. Select **JavaScript** from the list of options on the left side of the dialog box.

4. Select **Enable Acrobat JavaScript** if it is not already selected.

5. In the **Preferences** dialog box, select **Enable JavaScript Debugger after Acrobat is restarted** from the JavaScript Debugger options.

6. Select **Enable interactive console**. This option enables you to evaluate code that you write in the console window.

7. Select **Show console on errors and messages**. This ensures that whenever you make mistakes, the console displays helpful information.

8. Click **OK** to close the Preferences dialog box.

9. Close and restart Acrobat.

FIGURE 2.1   ***Enabling JavaScript, Console, and Debugger***
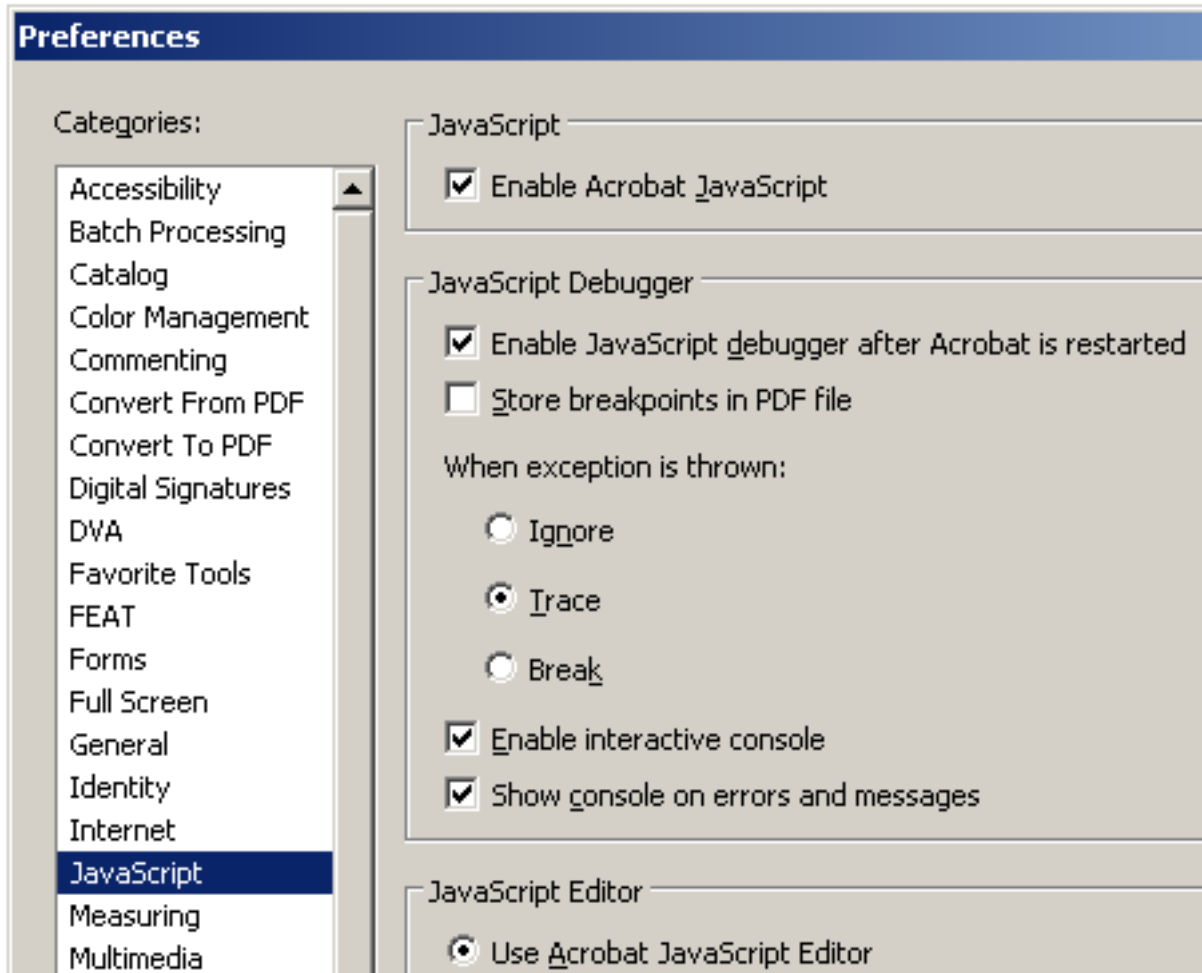
Trying out the JavaScript Console

1. Select **Advanced > JavaScript > Debugger** (**Ctrl+j**) to open the JavaScript debugger.

2. In the debugger, select **Console** from the **View** window.

   The console window should appear, as shown in Figure 2.2.

FIGURE 2.2 *Console window in the JavaScript Debugger*
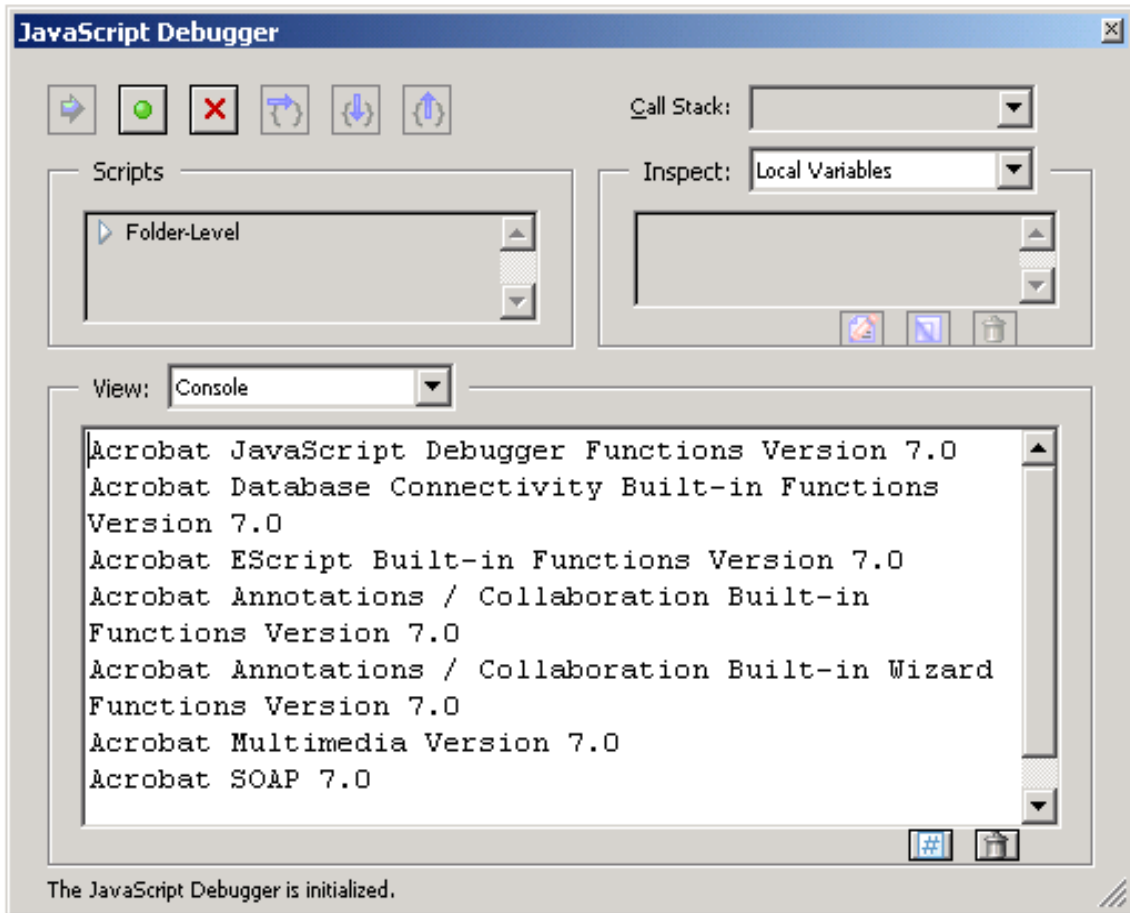
3. Click **Clear** (the trash can icon), located at the bottom right of the console, to delete any contents that appear in the window.

4. Type the following code into the console:
   ```
   var jsNum = 10;
   ```

5. With the mouse cursor positioned somewhere in this line of code, press **Enter** on the numeric keypad or **Ctrl+Enter** on the regular keyboard to evaluate the code. You should see the results shown in Figure 2.3 below.

FIGURE 2.3   ***Evaluating the variable declaration***
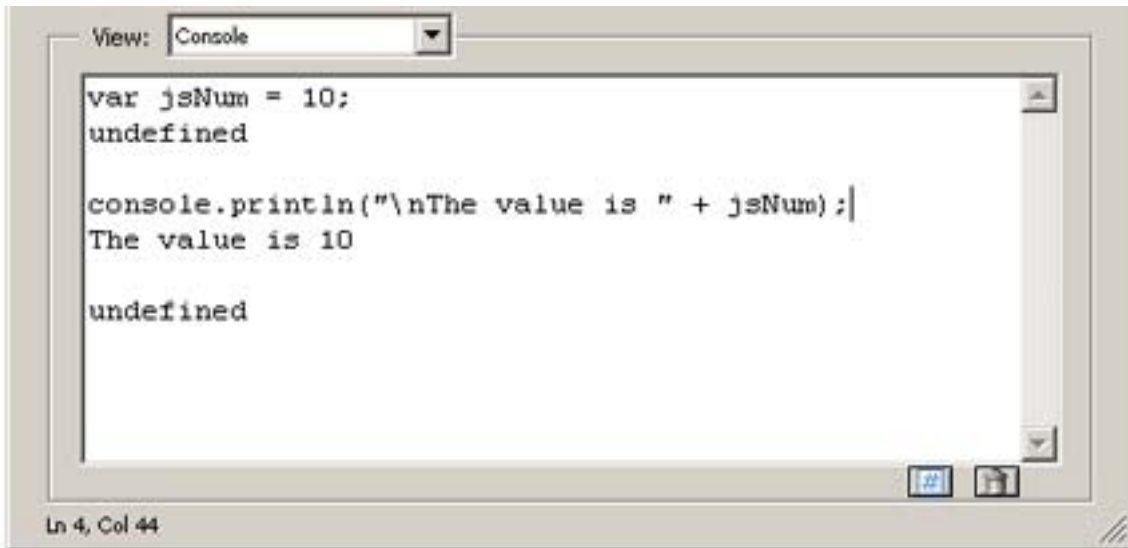
After each Acrobat JavaScript statement executes, the console window prints out **undefined,** which is the return value of the statement. Note that the result of a statement is not the same as the value of an expression within the statement. In this case, the return value **undefined** does not mean that the value of **jsNum** is undefined; it just means that the entire JavaScript statement's value is **undefined.**

6. A more convenient way to evaluate the **jsNum** variable is to highlight the variable name and execute it as a JavaScript expression, as shown below in Figure 2.4.

FIGURE 2.4   ***Evaluating jsNum***
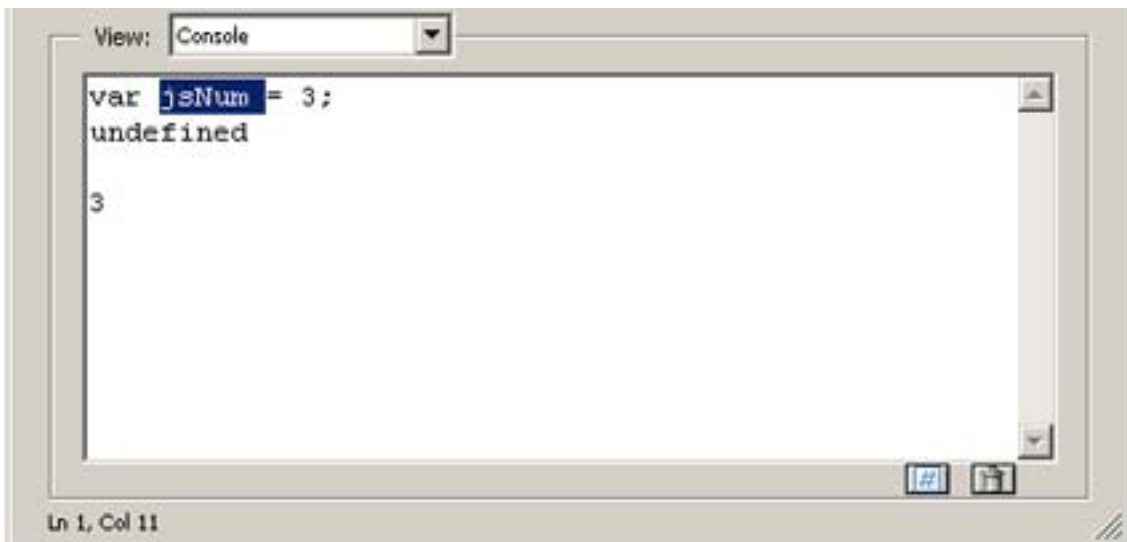
7. Click the **Close** button to exit the console and debugger, as shown in Figure 2.5 below:

FIGURE 2.5 ***Console and Debugger Close Button***

Using a JavaScript Editor

There are several ways to invoke the Arobat JavaScript Editor. To begin with, it is possible to select **JavaScripts** from the **Advanced** menu and choose one of the following options:

● **Edit all JavaScripts ...**

● **Document JavaScripts ...**

● **Set Document Actions ...**

A more basic approach, however, is to think of a JavaScript as an action associated with a part of the document, such as a page, bookmark, or form field. It would then make sense to select the object of interest and edit its particular JavaScript.

For example, the following are the steps to write a JavaScript for a bookmark:

1. Right-click a bookmark. This triggers a context menu.

2. Select **Properties** and choose the **Actions** tab, as shown below in Figure 2.6.

FIGURE 2.6   **Bookmark Properties**
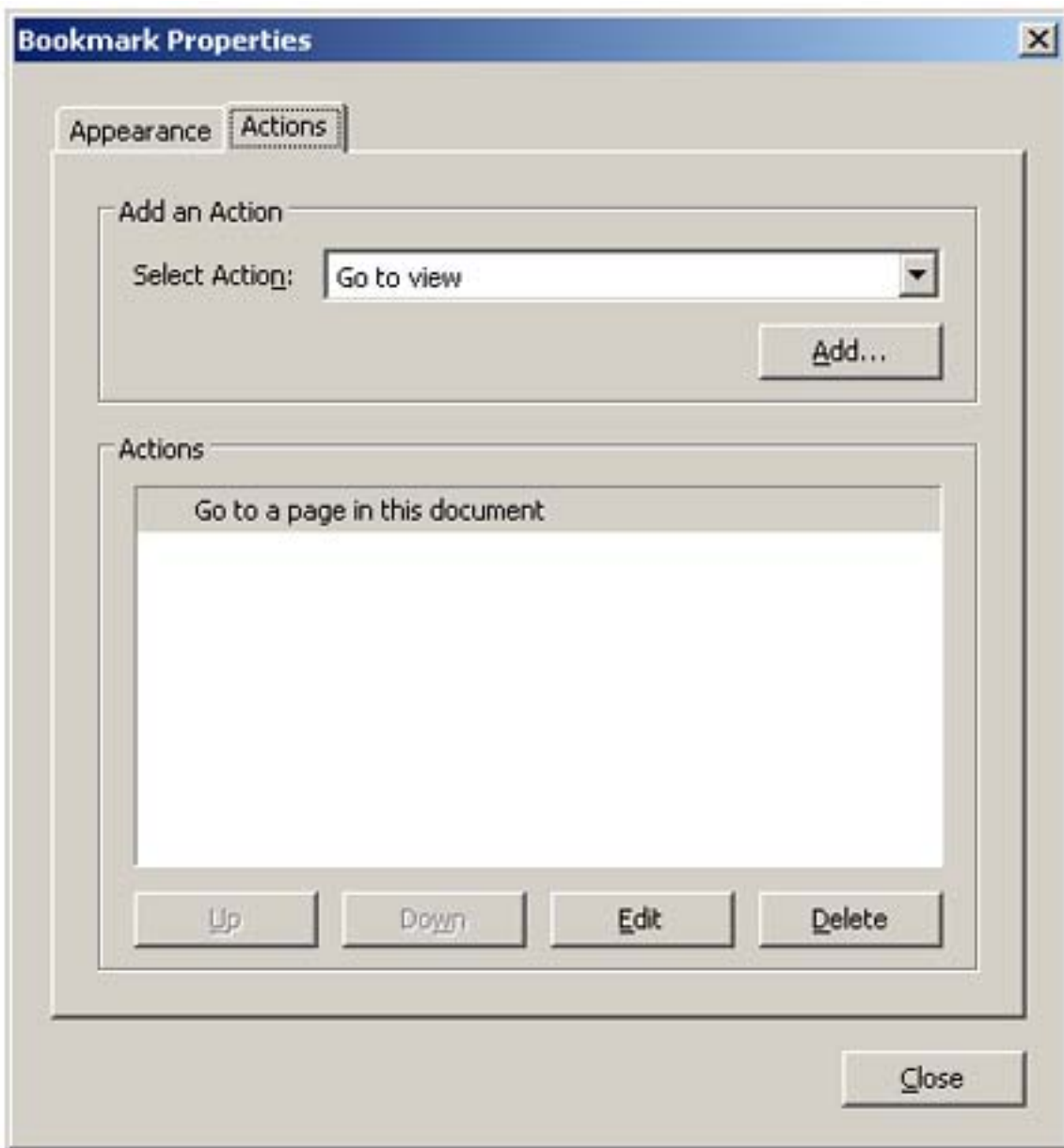
3. The **Select Action** drop-down list contains all the possible actions that can be associated with the object, such as **Run a JavaScript**, **Go to view**, or **Execute a menu item**.

4. Select **Run a JavaScript** from the **Select Action** drop-down list.

5. Click **Add** to open the JavaScript editor.

6. In the editor window, write the JavaScript code to be run when the user opens the page.

7. When the code is complete, click **Close** to close the editor.

If there are errors in your code, the JavaScript editor highlights the code line in question and display an error message, as shown below in Figure 2.7.

FIGURE 2.7   *Error detected by the JavaScript Editor*
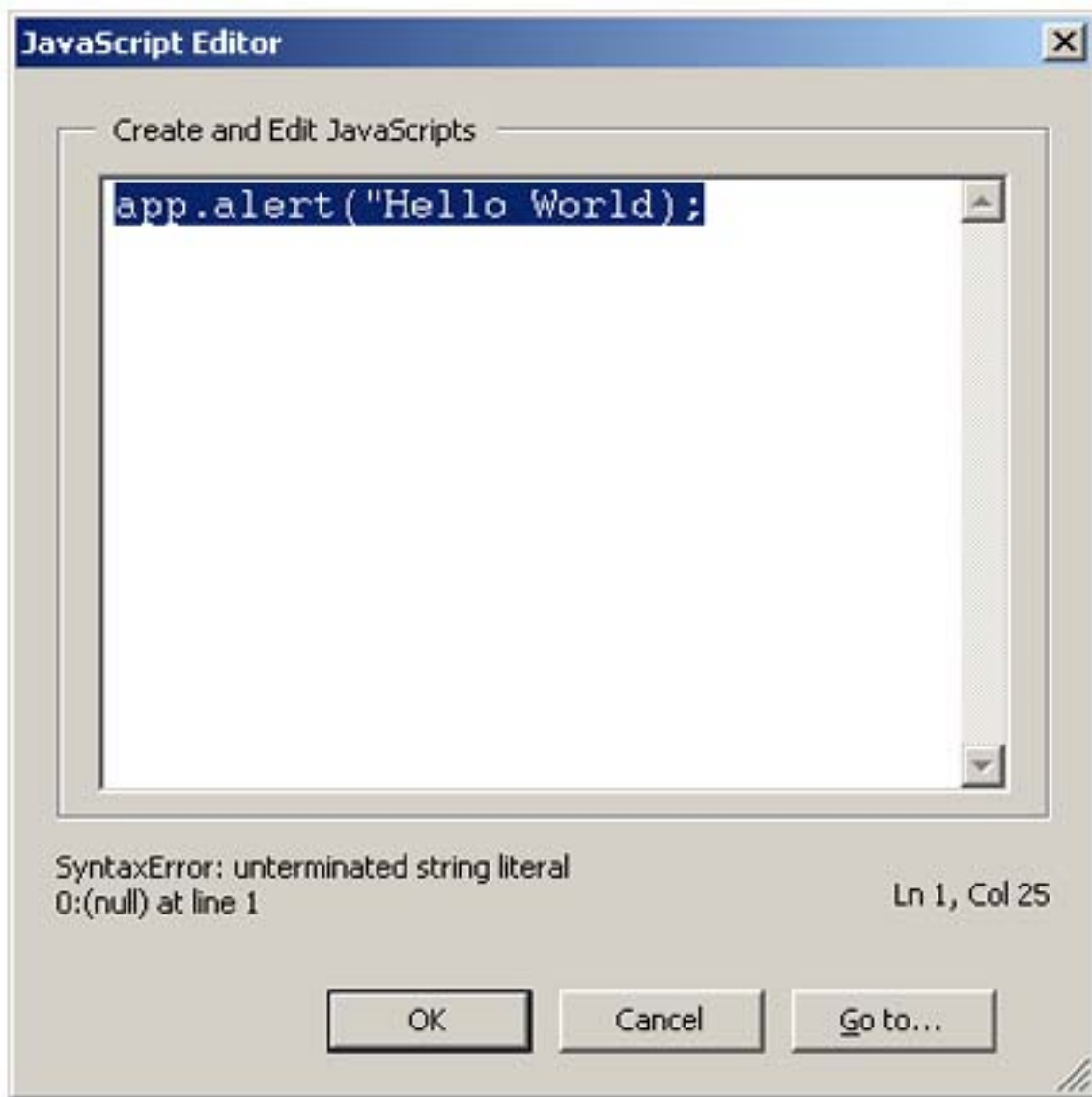
In Figure 2.7, the quotation mark to the right of the string is missing.

NOTE: JavaScript actions have a scope associated with various levels of objects in a PDF document, such as a form field, a page, or the entire document. For example, a script at the document level would be available from all other scriptable locations within the document.

Specifying the Default JavaScript Editor

You may choose whether to use the built-in JavaScript Editor that comes with Acrobat, or an external JavaScript editor of your choice. To set the default editor, invoke the **Preferences** dialog box as shown in Figure 2.1 above and in Figure 2.8 below:

1. Choose **Edit > Preferences** to open the **Preferences** dialog box.

2. Select **JavaScript** from the list of options on the left side of the dialog box.

   This brings up the **Preferences** dialog box.

3. In the **JavaScript Editor** section, select the editor you would like to use.

FIGURE 2.8   *Selecting the Editor in Preferences*
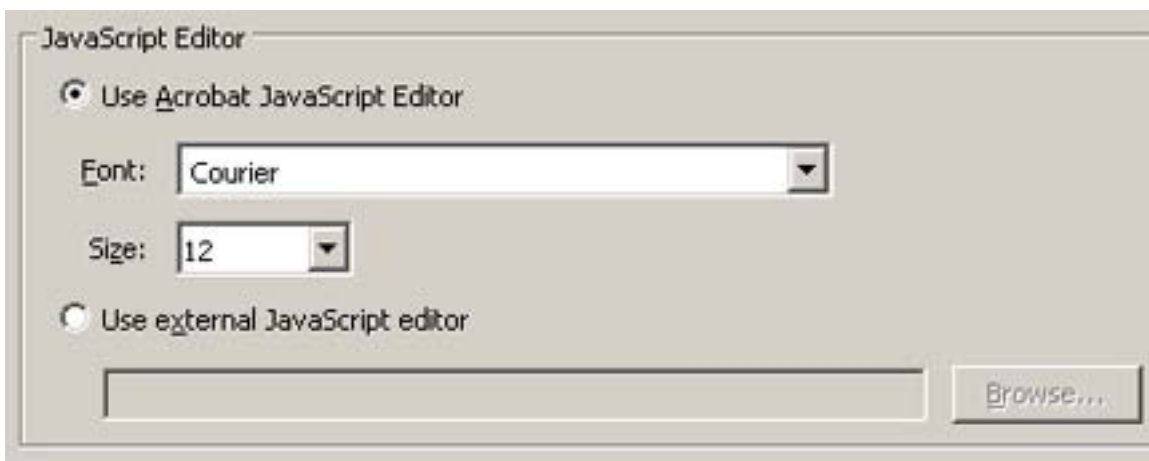
The **Acrobat JavaScript Editor** option sets the built-in Acrobat JavaScript Editor as the default.

The **External JavaScript Editor** option sets an external editor as the default. To choose this option, click **Browse...** to specify the path to the desired JavaScript editor.

NOTE: For some external editors, Acrobat provides extra command line options for invoking the editor. For details, see Additional Editor Capabilities.

Using the Built-in Acrobat JavaScript Editor

Like the Acrobat JavaScript Console, the built-in Acrobat JavaScript Editor can be used to evaluate portions of JavaScript code. Select a line or block of code to be evaluated, and press the **Enter** key on the numeric keypad or **Ctrl+Enter** on the regular keyboard.

In this case, the results of the JavaScript expressions or statements are displayed in the Console window, so you will also need to open the Acrobat JavaScript Console to see them. This is how you may keep the results separate from your Acrobat JavaScript code.

The Acrobat JavaScript Editor provides the same formatting options as those in the console window. For details, see Formatting Code.

Using an External Editor

If an external editor program has been specified as the default application for editing JavaScripts in Acrobat, Acrobat generates a temporary file and opens it in the external editor program. When editing a file in an external editor, note the following restrictions:

● You must save the file in order for Acrobat to detect the changes.

● Acrobat is inaccessible while the external editor is in use.

● Acrobat JavaScript code cannot be evaluated within the external editor.

Additional Editor Capabilities

Acrobat supports some additional command line editor capabilities for Windows-based applications, and provides support for two parameters in particular: the *file name* (`%f`) and the *target line number* (`%n`). Parameters for Macintosh-based editors are not supported.

Note that Acrobat launches a new instance of the editor for each new editing session. Some editors, if already running, load new files into the same session and may close the other open files without saving them. Thus, it is important to remember to take one of the following measures: save your changes before beginning a new editing session, close the editor application before starting a new editing session, or adjust its default preferences so that it always launches a new editor instance (this is the best course of action, if available).

If you are able to set the editor preferences to launch a new instance for each editing session, and if the editor requires a command line parameter in order to invoke a new editor instance, you may add that parameter to the editor command line specified in the **Edit > Preferences > JavaScript** dialog.

If your editor accepts a starting line number on the command line, Acrobat can start the editor on a line containing a syntax error by inserting the line number as a command line parameter (`%n`).

For your convenience, Acrobat provides predefined, command line templates for many current external editors. The external editor settings are defined in **Edit > Preferences > JavaScript**. If you use the **Browse** button to specify an external editor and it has a predefined command line template, the command line parameters and options appear to the right of the pathname for the editor application, and you may edit them. If no predefined template is available for your editor, you may still specify the appropriate command line parameters.

Specifying Additional Capabilities to Your Editor

Acrobat provides internal support for both of the commands described above on a few editors such as CodeWrite, Emacs, and SlickEdit (see Table 2.1below).

If your editor is not one that Acrobat currently supports, it will be necessary to check the editor's documentation. You will need to search for the following information:

● What are the command switches to tell the editor to always open a new instance?

Switches vary depending on the editor and include such parameters as **/NI** and **+new** followed by the file name (**"%f"**). Note that the quotes are required, because the file name that Acrobat sends to the editor may contain spaces.

● Is there a way to instruct the editor to open a file and jump to a line number?

Some line number command switches are **-#**, **-L, +**, and **-l**, each followed by the line number (**%n**). For most editors, the line number switch and **%n** should be enclosed in square brackets **[...]** . The text inside the square brackets will be used only when Acrobat requires that the editor jump to a specific line in order to correct an Acrobat JavaScript syntax error. You can use an editor that does not support a line number switch; in this case, you will need to scroll to the appropriate line in the event of a syntax error.

For example, Acrobat recognizes the Visual SlickEdit editor as `vs.exe` and automatically supplies this command line template:

```
"C:\Program Files\vslick\win\vs.exe" "%f" +new [-#%n]
```

When Acrobat opens the default JavaScript editor, it makes the appropriate substitutions in the command line and executes it with the operating sytem shell. In the above case, if the syntax error were on line 43, the command line generated would appear as follows:

```
"C:\Program Files\vslick\win\vs.exe" "C:\Temp\jsedit.js" +new -#43
```

NOTE: To insert **%**, **[**, or **]** as characters in the command line, precede each of them with the **%** escape character, thus using **%%**, **%[**, or **%]** respectively.

TABLE 2.1   *Supported external JavaScript editors with command linetemplates.*

Editor	Web site	Template Command Line Arguments
Boxer	http://www.boxersoftware.com	`-G -2 "%f" [-L%n]`
ConTEXT	http://fixedsys.com/context	`"%f" [/g1:%n]`
CodeWright	http://www.codewright.com	`-M -N -NOSPLASH "%f" [-G%n]`
Emacs	http://www.gnusoftware.com/Emacs	`[+%n] "%f"`
Epsilon	http://www.lugaru.com	`[+%n] "%f"`
Multi-Edit	http://www.multiedit.com	`/NI /NS /NV [/L%n] "%f"`
TextPad	http://www.textpad.com	`-m -q "%f"`
UltraEdit	http://www.ultraedit.com	`"%f" [-l%n]`
VEDIT	http://www.vedit.com	`-s2 "%f" [-l %n]`
Visual SlickEdit	http://www.slickedit.com	`+new "%f" [-#%n]`

Testing Whether Your Editor Will Open at Syntax Error Locations

To determine whether Acrobat can open your editor on a line number, do the following:

1. Open a script in your editor.

2. Add a syntax error.

3. Move the cursor to a line other than the one containing the syntax error.

4. Close and save the file.

If a dialog automatically appears prompting you to fix the syntax error, check whether it correctly specifies the line containing the error.

Saving and Closing a File with a Syntax Error

If you save and close a file containing a syntax error, Acrobat displays a dialog with a message asking if you would like to fix the error. For example, if there is an error on line 123, the following message appears:

There is a JavaScript error at line 123.
Do you want to fix the error?

NOTE: If you click **No**, Acrobat discards your file.

Always click **Yes**. Acrobat expands the path to the editor to include the line number in the specified syntax. The editor opens and the cursor is placed on line 123.

Using the Acrobat JavaScript Debugger

The Acrobat JavaScript Debugger is a fully capable JavaScript debugger that allows you to set breakpoints and inspect variable values while stepping through code. While it is normally accessed from the Acrobat Professional user interface, it can also be triggered to appear in Adobe Reader when an exception occurs.

Though fully supported Acrobat JavaScript debugging is only available in Acrobat Professional, the following instructions to make the complete debugger functionality available in Adobe Reader on Windows and Macintosh platforms are provided as a courtesy. Note that this procedure involves editing the registry. Adobe Systems Incorporated does not provide support for editing the registry, which contains critical system and application information. It is recommended that you back up the registry before modifying it.

1. The file `debugger.js`, available at http://partners.adobe.com/asn/acrobat/docs.jsp or in the SDK installation (`Acrobat 7.0 SDK/JavaScriptSupport/Debugger/debugger.js`), must be copied to the `Acrobat 7.0/Reader/JavaScripts` folder.

2. Create key/value pairs in the registry settings, starting at the location **HKEY_CURRENT_USER\Software\Adobe\Acrobat Reader\7.0\JSPrefs** on Windows as shown below in Table 2.2, or in the property list file **<user>:Library:Preferences:com.adobe.Reader7.0.plist** on the Macintosh. For the Macintosh, use an appropriate editor for the property list file, and add the following children under **JSPrefs**, using **Type : Array** in each case: **Console Open**, **Console Input**, **Enable Debugger**, and **Exceptions**. Under each of these children, add the following children: **0 (number)** and **1 (boolean)**.

3. Close and restart Adobe Reader. At this point the debugger will be available.

TABLE 2.2   ***Registry Key/Value Pairs for Windows***

`bConsoleInput`	`REG_DWORD`	`0x00000001`
`bEnableDebugger`	`REG_DWORD`	`0x00000001`
`iExceptions`	`REG_DWORD`	`0x00000002` (this will break into the debugger when exceptions occur)

NOTE: Since Adobe Reader does not provide access to the debugger through its menu items or the **Ctrl+j** key sequence, the only ways to access the debugger are to execute a JavaScript, cause an error, or customize the user interface (for example, you could add a button that runs a JavaScript causing the debugger to appear).

As you learned earlier when opening the Acrobat JavaScript Console, which is integrated with the debugger dialog box, the debugger may be activated in Acrobat Professional by selecting **Advanced > JavaScript > Debugger**. In addition, the debugger automatically opens if a running script throws an exception or encounters a previously set breakpoint.

NOTE: The Acrobat JavaScript Debugger cannot be used to analyze JavaScript stored in HTML pages viewed by Web browsers such as NetScape or Internet Explorer, or any other kind of scripting languages.

In order to make the debugger available for use, you will need to enable both JavaScript and the debugger. As you did earlier, use the **Edit > Preferences** dialog to control the behavior of the Acrobat JavaScript development environment. Enabling JavaScript and the JavaScript editor are described in Enabling JavaScript. To enable the debugger, select JavaScript from the list on the left in the **Preferences** dialog and make sure the **Enable JavaScript debugger...** option is checked. Note that you must restart Acrobat for this option to take effect.

The debugger options are located in the **JavaScript Debugger** section of the **Preferences** dialog, as shown above in Figure 2.1, and are explained below in Table 2.3.

TABLE 2.3   *JavaScript debugger options*

Option	Meaning
Enable Javascript debugger after Acrobat is restarted	To enable the debugger, check this option, which makes all debugger features available the next time Acrobat is launched.
Store breakpoints in PDF file	This option enables you to store breakpoints so they are available the next time you start Acrobat or open the PDF file. To remove the breakpoints, do the following: ● Turn this option off. ● Select **Advanced > JavaScript > Document JavaScripts** and delete the `ACRO_Breakpoints` script. ● Save the file.
When an exception is thrown	This option provides three choices for actions when an exception is thrown: ● **Ignore**: ignores the exception. ● **Trace**: displays a stack trace. ● **Break**: stops execution and displays a message window that gives you the option to start the debugger at the line where the exception occurred.
Enable interactive console	This option allows you to enter JavaScript commands in the console window. If this option is not checked and you click in the console window, the following dialog box appears: The interactive console is not enabled. Would you like to enable it now? Click **Yes** to enable this option from within the debugger. In **Preferences** you will now see this option checked.
Show console on errors and messages	This option opens the console window in the debugger dialog box. Regardless of whether the debugger is enabled, this option causes the debugger dialog box to open when an error occurs and displays the error message to the console window.

Acrobat JavaScript Debugger

You can open the Acrobat JavaScript Debugger at any time by selecting the Acrobat menu item **Advanced > JavaScript > Debugger**. Familiarize yourself with the parts of the window and the controls as described here before you attempt interactive debugging of a script.

The section Accessing Scripts in the Scripts Window outlines the types and locations of scripts that may be debugged. The section Starting the Debugger describes how to automatically start the debugger for a script.

IMPORTANT: *On Windows, while the Debugger is open and a debugging session is in progress, Acrobat will be unavailable.*

Main Groups of Controls

The debugger dialog, shown below in Figure 2.9, consists of three main groups of controls. The toolbar on the top left contains six button controls that provide basic debugging session functionality. See Figure 2.10 below for details.

Immediately below the toolbar, a **Scripts** window displays the names of scripts available for debugging. These are organized in a tree hierarchy, such as the one shown below in Figure 2.9, and may be accompanied by the **Script** window below, which shows the code for a single script corresponding to the one highlighted in the **Scripts** window.

The **Call Stack** and **Inspect** drop-down lists are located at the top right of the debugger dialog. Selecting entries in these lists enables you to view the nesting order of function calls, and enable you to inspect the details of variables, watches, and breakpoints in the **Inspect** details window.

Debugger View Windows

Below the main group of controls, the debugger provides a **Views** drop-down list with the following choices:

- **Script**: view a single JavaScript script selected from the Scripts hierarchy window.
- **Console**: view the output of a selected script as it executes in the JavaScript console window. The console may also be used to run scripts or individual commands. See Using the Acrobat JavaScript Console.
- **Script and Console**: view both the console and script windows at the same time. The script window displays above the console window, as shown in Figure 2.9.

FIGURE 2.9   **Debugger Dialog Box**
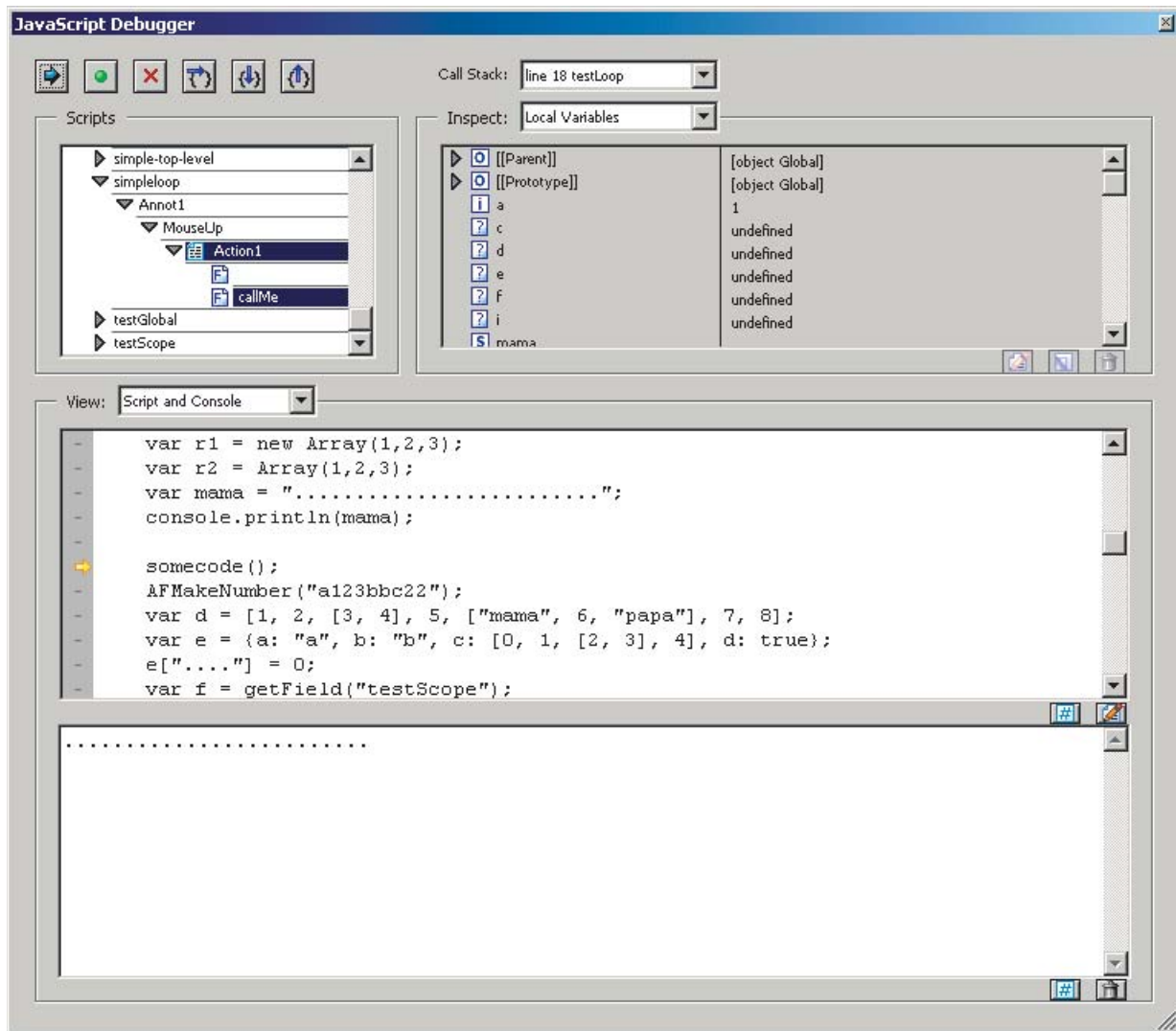
Debugger Buttons

Figure 2.10 shows the debugger buttons on the toolbar, and Table 2.4 summarizes the functionality of each button, followed by detailed descriptions below. An overview of their usage is provided in Exercise: Calculator.

FIGURE 2.10   Debugger Buttons
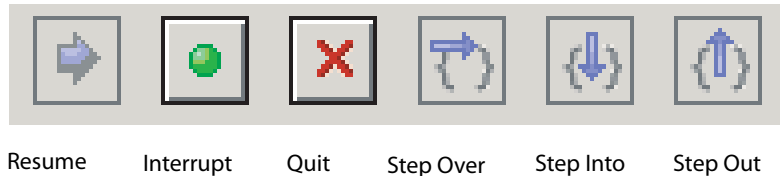
Resume   Interrupt   Quit   Step Over   Step Into   Step Out

TABLE 2.4   Debugger Buttons Summary

Button	Description
Resume Execution	Runs a script stopped in the debugger.
Interrupt	Halts execution.
Quit	Closes the debugger and terminates script execution.
Step Over	Executes the next instruction, but does not enter a function call if encountered.
Step Into	Executes the next instruction, and enters a function call if encountered.
Step Out	Executes the remaining code in a function call, and stops at the next instruction in the calling script.

Resume Execution

When the script is stopped, the **Resume Execution** button cause the script to continue execution until it reaches one of the following:

● The next script to be executed

● The next breakpoint encountered

● The next error encountered

● The end of the script

Interrupt

The **Interrupt** button halts execution of the current script. When clicked, it appears in red, which indicates that it has been activated and causes execution to stop at the beginning of the next script that is run. If this occurs, the **Interrupt** button is automatically deactivated and returns to its green color. It must be activated again in order to interrupt another script.

Quit

The **Quit** button terminates the debugging session and closes the debugger.

Step Over

The **Step Over** button executes a single instruction, and if it is a function call, it executes the entire function in a single step, rather than stepping into the function. For example, the position indicator (yellow arrow) in the debugger is to the left of a function call, as shown below in Figure 2.11:

FIGURE 2.11   ***Position Indicator at a Function Call***


```
callMe();
```

Exccution is currently halted before the call to **callMe()**. Assuming that there are no errors or breakpoints in **callMe()**, clicking **Step Over** executes the entire **callMe()** function, and advances the position indicator to the next script instruction following the function call.

If the statement at the position indicator does not contain a function call, **Step Over** simply executes that statement.

Step Into

The **Step Into** button executes the next statement, and if it is a function call, it proceeds to the first statement within the function. Refer again to Figure 2.11. Clicking **Step Into** at this point advances the position indicator to the first statement within the **callMe()** function.

NOTE: Be aware that it is not possible to step into native functions, since they have no JavaScript implementation. This applies to Acrobat native functions as well as core JavaScript functions.

Step Out

The **Step Out** button executes the remaining code within the current function call and stops at the instruction immediately following the call. This button provides a convenient means of eliminating cumbersome, stepwise execution of functions that do not contain bugs. If you are not inside a function call and there are no errors, the **Step Out** button continues executing code to the end of the current script or until a breakpoint is encountered.

Debugger Scripts Window

All scripts associated with a PDF file are available in the debugger dialog. The debugger displays these in the **Scripts** window (see Figure 2.12 below for an example).

FIGURE 2.12   Scripts window
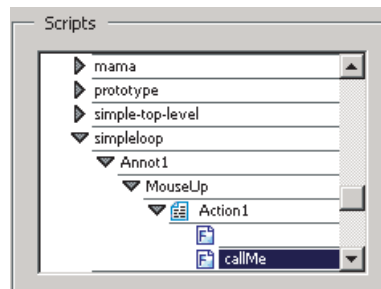
Accessing Scripts in the Scripts Window

To display the content a script, click the triangle to its left in the **Scripts** window. Each triangle opens the next level in the containment hierarchy. A script icon indicates the lowest level, which means that the code for the given function is available. As shown above in Figure 2.12, a function has been defined for a mouse-up action on a button named **Button1**. Click on the script icon to display its code.

Acrobat JavaScripts can be stored in several places, which may be either inside or outside PDF files. The following sections describe their possible locations.

Scripts Inside PDF Files

Table 2.5 below lists the types of scripts contained in PDF files. These can be accessed from the Scripts window within the debugger dialog. You can edit them from inside the debugger, and set breakpoints as described in Breakpoints on page 58.

NOTE: Changes to scripts do not take effect until the scripts are re-run; changes cannot be applied to a running script.

TABLE 2.5   *Scripts inside PDF files*

Location	Access
Document level	**Advanced > JavaScript > Document JavaScripts**
Document actions	**Advanced > JavaScript > Set Document Actions**
Page actions	Click the page on the **Pages** tab; select **Options > Page Properties**
Forms	Double-click the form object in form editing mode (see below) to bring up the **Form Properties** dialog
Bookmarks	Click the bookmark on the **Bookmarks** tab; select **Options > Bookmark Properties**
Links	Double-click the link object in form editing mode to bring up the **Link Properties** dialog

Form Editing mode — To switch to form editing mode, use the Acrobat Advanced Editing toolbar. If it is not visible, select **Tools > Advanced Editing > Show Advanced Editing Toolbar**. You may want to dock this toolbar, as you will use it frequently.

Scripts Outside PDF Files

Scripts outside of Acrobat are also listed in the Scripts window and are available for debugging in Acrobat. Table 2.6 lists these script types and how to access them.

TABLE 2.6   *Scripts outside PDF files*

Location	Access
Folder level	Stored as JavaScript (`.js`) files in the `App` or `User` folder areas
Console	Typed and evaluated in the console window
Batch	Choose **Advanced > Batch Processing**

Folder-level scripts normally can be viewed and debugged but not edited in Acrobat. Console and batch processing scripts are not visible to the debugger until they are executed. For this reason, you cannot set breakpoints prior to executing these scripts. You can access the scripts either using the **Debug From Start** option or by using the `debugger` keyword. See Starting the Debugger for details.

Call Stack List

To the right of the debugger control buttons is the **Call Stack** drop-down list which displays the currently executing function and its associated state within the current set of nested calls. An example is shown below in Figure 2.13. When the debugger has been used to suspend execution at a given statement, the call stack displays text indicating the current function call (stack frame). Each entry shows the current line number and function name. The most recent stack frame is displayed at the top of the **Call Stack** drop-down list. To inspect the local variables of a particular frame in the stack, click on that entry. They appear in the **Inspect** details window immediately below the call stack list, as shown below in Figure 2.13.

FIGURE 2.13   ***Call stack***
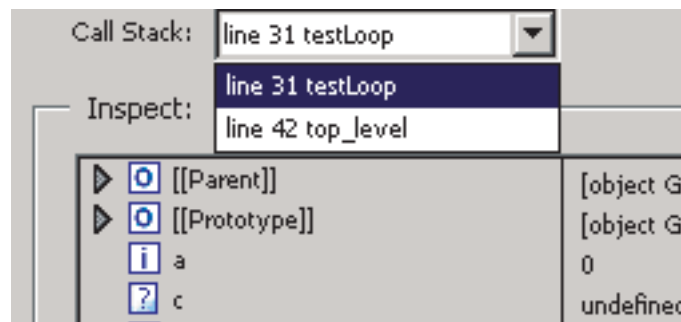
You may select any function in the call stack. Doing so selects that stack frame, and its location is shown in the **Inspect** details window. When **Local Variables** is selected in the **Inspect** drop-down list, the variables specific to that active frame are displayed in the **Inspect** details window.

Inspect Details Window

The **Inspect** details window is located to the right of the **Scripts** window and below the **Call Stack**. Its purpose is to help you inspect the values of variables, customize the way in which variables are inspected (setting watches), and obtain detailed information about breakpoints.

Inspect Details Window Controls

The three buttons at the bottom right of the **Inspect** details window, shown in Figure 2.14 below, can be used to edit, create, or delete items. The **Edit**, **New**, and **Delete** buttons become active when items in the **Inspect** drop-down list are selected.

FIGURE 2.14   *Inspect details window button controls*
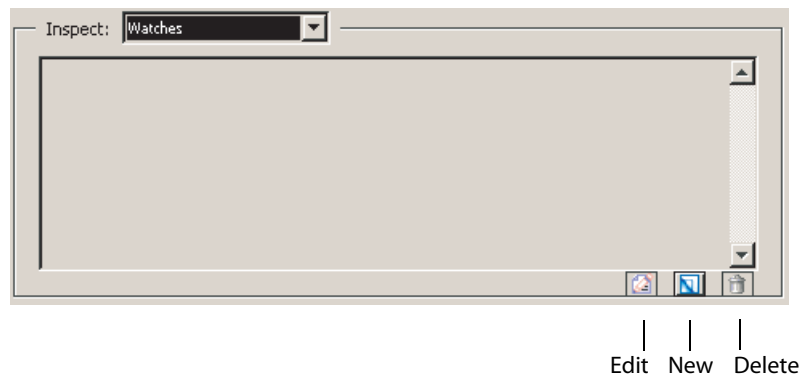
Inspecting Variables

The **Inspect** details window is a powerful tool that you can use to examine the current state of JavaScript objects and variables. It enables you to inspect any objects and properties in a recursive manner within the current stack frame in the debugging session.

To inspect a variable, select **Local Variables** from the **Inspect** drop-down list, which displays a list of variable and value pairs in the **Inspect** details window. To place a value in a variable, highlight the variable in the details window (this activates the **Edit** button). Click the **Edit** button. An **Edit Variable** dialog appears, allowing you to enter a new value for the variable as shown below in Figure 2.15.

A triangle next to a name indicates that an object is available for inspection. If you would like to view its properties, click the triangle to expand the object.

FIGURE 2.15   Local variable details
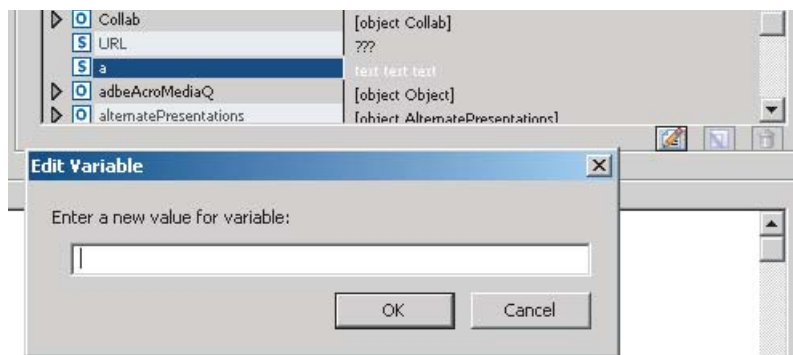
Watches

The **Watches** list enables you to customize how variables are inspected. Watches are JavaScript expressions evaluated when the debugger encounters a breakpoint or a step in execution. The **Watches** list provides you with the ability to edit, add, or delete watches using the three buttons just below the **Inspect** details window. All results are displayed in the **Inspect** details window in the order in which they were created.

To set a watch, select **Watches** from the **Inspect** drop-down list. Click the **New** button, and a dialog prompts you for the JavaScript variable or expression to be evaluated.

To change the value of a watch, select the watch from the list. Click the **Edit** button, which displays a dialog prompting you to specify a new expression for evaluation. To delete a watch, select it from the **Inspect** drop-down list and click the **Delete** button.

Breakpoints

The Breakpoints option in the **Inspect** drop-down list enables you to manage program breakpoints, which in turn make it possible to inspect the values of local variables once execution is halted. A breakpoint may be defined so that execution halts at a given line of code, and conditions may be associated with them (see Using Conditional Breakpoints).

When a breakpoint is reached, JavaScript execution halts and the debugger displays the current line of code.

To add a breakpoint, click on the gray strip to the left of the code in the script view, which should cause a red dot to appear. The lines at which breakpoints are permitted have small horizontal lines immediately to their left in the gray strip. To remove the breakpoint, click on the red dot, which should subsequently disappear.

Coding Styles and Breakpoints

Placement of the left curly brace (**{**) in a function definition is a matter of style.

Style 1: Place the left curly brace on the same line as the function name, for example,

```
function callMe() { // curly brace on same line as function name
    var a = 0;
}
```

Style 2: Place the left curly brace on a separate line, for example

```
function callMe()
{ // curly brace is on a separate line
    var a = 0;
}
```

If you would like to set a breakpoint at the function heading, use Style 1. Note that the Acrobat JavaScript debugger does not set a breakpoint at the function heading for Style 2. It is only possible to set a breakpoint from the line of code containing the left curly brace. This is illustrated below in Figure 2.16. It is possible to set the breakpoint on the line below the function heading for **callMe()**, and on the line containing the function heading for **testLoop()**. Setting a breakpoint at a function heading causes execution to stop at the first statement within the function.

FIGURE 2.16   ***Setting a Breakpoint at a Function Heading***


```
function callMe()
{
    var a = 0;
    b = 0;
}

function testLoop(max) {
    var a = 1;
    b = 1;
    var c;
```

Listing Breakpoints

To view the list of all breakpoints set for the debugging session, select the **Breakpoints** option from the **Inspect** drop-down list. You may edit and delete breakpoints using the button controls just beneath the **Inspect** details window, as shown above in Figure 2.14.

Using Conditional Breakpoints

A conditional breakpoint causes the interpreter to stop the program and activate the debugger only when a specified condition is true. Conditional breakpoints are useful for stopping execution when conditions warrant doing so, and streamline the debugging process by eliminating needless stepwise execution. For example, if you are only interested in debugging after 100 iterations in a loop, you can set a breakpoint that only becomes active when the looping index reaches the value of 100.

The condition is a JavaScript expression. If the expression evaluates to `true`, the interpreter stops the program at the breakpoint. Otherwise, the interpreter does not stop the program. An unconditional breakpoint, the default, always causes the interpreter to stop the program and to activate the debugger when it reaches the breakpoint, because its condition is always set to `true`.

To change a breakpoint condition, select **Breakpoint** from the **Inspect** drop-down list and click **Edit**. A dialog appears, prompting you to change the breakpoint condition.

Starting the Debugger

There are four ways to invoke the Acrobat JavaScript Debugger. Two of these ways begin the debugging session from the start of execution, and the other two begin the session from a specified line of code.

Debugging From the Start of Execution

There are two ways to start the debugger from the start of execution. In either case, use the Step Into button to proceed with the debugging session.

Debug From Start

Choose **Advanced > JavaScript** and, if the option is not already checked, click on **Debug From Start**.

This option causes the debugging session to begin at the start of execution of any new script.

NOTE: **Debug From Start** does not turn off automatically. Be sure to turn off this option when you have finished debugging, otherwise it continues to stop on every new script you execute in Acrobat.

Click Interrupt

Open the debugger window and click the Interrupt button, which displays in red. At this point, performing any action that runs a script causes execution to stop at the beginning of the script.

Unlike **Debug From Start**, the **Interrupt** button is automatically deactivated after being used. To stop at the beginning of a new script, you must reactivate it by clicking it again.

Debugging From an Arbitrary Point in the Script

Define a Breakpoint

To start debugging from a specific point in your script, you can set a breakpoint. See Breakpoints for details on how to set a breakpoint.

Using the debugger keyword

You can also insert the **`debugger`** keyword in any line of your code to stop execution and enter the debugger when that particular line is reached.

NOTE: Breakpoints created using the **`debugger`** keyword are not listed in the **Inspect** details window when you select Breakpoints from the **Inspect** drop-down list.

Exercise: Calculator

NOTE: To complete the following exercises, unzip the file `TestDebugger.zip` and open `Calc.pdf`.

In this exercise, you will set breakpoints in a script and create watches to view how a variable changes as the script executes. At the end of this exercise you will be able to:

● Start the debugger from a location within a script.

● Interpret the contents of the **Inspect** details window.

● Create, edit, and delete watches.

● Set and clear breakpoints.

● Use the debugger buttons to control how the interpreter executes the code in the script.

Calculator

`Calc.pdf` uses document-level and form field-level scripts to simulate the behavior of a simple calculator. The keypad consists of numeric and function keys, the period (**.**), the equals sign (=), and Cancel (**C**) and Cancel Entry (**CE**) keys.

How the Calculator Displays Output

`Calc.pdf` uses the Acrobat Document Object's **getField()** method to place values in the calculator display window, which is the text field **display**. In this statement, **getField()** binds the variable **display** to this field:

```
var display = this.getField("Display");
```

Every time a numeric key is clicked, its value should be shown in the calculator display window. To do this, we define a **Mouse Up** action for each numeric key in which the number shown on the key is assigned to the calculator display window. For example, when the user clicks the **7** button, the **Mouse Up** action associated with the button could assign the value returned by **digit_button(7)** to the calculator window as shown below:

```
display.value = digit_button(7);
```

`Calc.pdf` uses a similar mechanism to display the strings representing arithmetic operations to the **func** text field, which is used to display the operation (PLUS, MINUS, MULT, DIV). In this statement, **getField()** binds the variable **func** to this field:

```
var func = this.getField("Func");
```

The code below is contained in the **Mouse Up** action for the division (/) key, and represents a call to the **func_button()** script:

```
func_button("DIV");
```

The action passes the string **"DIV"** to the **func_button()** function's parameter named **req_func**, which is thus bound to that value.

Later in the function, this statement is encountered:

```
func.value = req_func
```

Because **getField()** has bound the **func** variable to the **"Func"** field, this statement displays the string **"DIV"** in the field.

Values With More Than 1 Digit

To adjust for values greater than or equal to 10, the calculator multiplies the current value by 10 and adds the value of the numeric key selected. For example, if the current value is 2 and the user clicks the **3** button, the new current value is 23 (10 times 2 plus 3).

For decimal values, a **Mouse Up** action is associated with the **.** button. In this action, the calculator multiplies the divisor by 10. Subsequently, when other numeric keys are selected, the calculator divides the selected value by the divisor, and adds that amount to the current value. For example, if the current divisor is 1 and the user presses the **2**, **.**, and **4** buttons, respectively, the current value is first updated to 2, the divisor becomes 10, and 4 is divided by the divisor to obtain 0.4, which is added to 2 to obtain 2.4, which is assigned to the new current value and shown in the display area.

Testing the Calculator

To familiarize yourself with the basic calculator operations, open `Calc.pdf` in Acrobat and try entering expressions and evaluating their results.

When you are familiar with the calculator's operation, close this file.

There are limitations to debugging scripts in Acrobat from inside a browser, because not all scripts contained in a PDF file may be available if the PDF file has not been completely downloaded.

Debugging is not possible if a modal dialog is running. This may occur when debugging a batch sequence. If a modal dialog is running during a debugging session and the program stops responding, press the **Esc** key.

Debugging scripts with an event initiated by either the **`app.setInterval()`** or **`app.setTimeOut()`** method may trigger the appearance of a series of recurring alert messages. If this occurs, press the **Esc** key after the modal dialog has exited.

Summary

The Acrobat JavaScript Debugger is a fully capable tool for troubleshooting problems in Acrobat JavaScript scripts. In combination with the **Edit > Preferences** dialog box, the debugger enables you to make decisions about how to control the behavior of your Acrobat JavaScript development environment, which consists of three main sets of controls for selecting and executing scripts. The six buttons provide basic control over execution, the **Scripts** window enables you select a script for debugging, the **Call Stack** and **Inspect** lists, in combination with an **Inspect** details window, provide a view of nested function calls (stack frames) as well as details related to objects, variables, watches, and breakpoints. You may utilize the **Inspect** details window controls to set and clear breakpoints, inspect variable values while stepping through JavaScript code, and manage custom watches for examination during script execution.

3 Acrobat JavaScript Contexts

Introduction

Acrobat JavaScript can be used to create scripts that are used for batch sequences and at various levels within documents. This chapter discusses how to determine the appropriate level for a JavaScript, and how to create and access Acrobat JavaScripts at all levels.

Chapter Goals

At the end of this chapter, you will be able to:

- List the Acrobat JavaScript contexts and describe their purposes.
- Describe how to create and edit scripts at all levels.

Contents

Topics
Introduction to Acrobat JavaScript Contexts
Folder Level JavaScripts
Document Level JavaScripts
Field Level JavaScripts
Batch Level JavaScripts

Introduction to Acrobat JavaScript Contexts

Acrobat JavaScripts can be applied at a variety of levels:

- *folder* level
- *document* level
- *field* level
- *batch* level.

Each of these levels represents a context in which processing occurs, which has implications about when the scripts are loaded and their accessibility inside and outside documents.

In addition, the placement of a JavaScript at a given level determines its reusability. Folder level scripts are available within all documents, document level scripts are available to all form fields within a given document, field level scripts are only visible to the form fields with which they are associated.

> **NOTE:** For instructions on how to disallow changes to scripts or hide scripts, see How can I disallow changes in scripts contained in my document? and How can I hide scripts contained in my document?

Folder Level JavaScripts

Folder level JavaScripts contain variables and functions that may be generally useful to Acrobat, and are visible from all documents. There are two kinds of folder level scripts: *App* and *User*. For example, if you would like to add specialized menus and menu items to be available to all documents opened in Acrobat, you may create and store the scripts for these in folder level JavaScripts.

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

When Acrobat is installed on your machine, it provides you with several standard folder level JavaScript files, including `Aform.js,`, `Annots.js`, and `ADBC.js`. In addition, your `User` folder may also contain `glob.js`, which is programmatically generated and contains cross-session application preferences set using the **global** object's **setPersistent()** method, and `Config.js`, which is used to customize the look of the viewer by adjusting its toolbar buttons and menu items. For example, **app.addMenuItem()** calls would be locaed in `Config.js`.

To create folder level JavaScripts, use an external editor running in parallel to Acrobat. Note that the external editor cannot be invoked from Acrobat.

Document Level JavaScripts

Document level JavaScripts are function and variable definitions that are generally useful to a given document, but are not applicable outside the document.

To create or access document level JavaScripts in Acrobat, select **Advanced > JavaScript > Document JavaScripts…**, which enables you to add, modify, or delete document level scripts. Document level scripts are executed after the document has opened, and are stored within the PDF document.

Field Level JavaScripts

You can associate a JavaScript with any component within a document. In the case of dynamic XML forms, such scripts are only visible to that component and execute as actions or events. For example, you may associate an action page open or page close events, as well as a `MouseUp` event for a bookmark, page, or form field. Field level scripts are executed as the events occur, and are usually applied to validate, format, or calculate form field values. Like document level scripts, field level scripts are stored within the PDF document.

There are several ways to create or edit field level JavaScripts. The most straightforward manner is to right-click the form field, select **Properties** context menu item and choose the **Actions** tab, as shown above in Figure 2.6. Additionally, you may select **Advanced > JavaScripts > Edit all JavaScripts …**.

Batch Level JavaScripts

A batch level JavaScript is a script that can be applied to a collection of documents, and operates at the application level. For example, you may define a batch script to print a series of documents or apply security restrictions to them.

To create or edit a batch level JavaScript in Acrobat, select **Advanced > Batch Processing…**.

Summary

Acrobat JavaScripts can be applied either within a document or outside a document. Inside a document, scripts can be created so that they are visible throughout the document (document level JavaScripts), or visible only to a given form field within the document (field-level JavaScripts). Outside a document, scripts can be created so that they provide generally useful functions or variables available to all documents (folder level JavaScripts), or they can be created for the purpose of processing a collection of documents (batch level JavaScripts).

The decision to place a JavaScript at a given level is based on its general applicability. If a script is to have a specialized function applicable to a single component within a document, it would be appropriate to create a field level JavaScript. If a script would be generally useful to several form fields within a document, it would be appropriate to create a document level JavaScript. If a function or variable can be used in a variety of situations within many documents, it would be appropriate to create a folder level JavaScript.

4 Creating and Modifying PDF Documents

Introduction

This chapter provides a detailed overview of how to apply Acrobat JavaScript in order to dynamically create PDF files, modify them, and convert PDF files to XML format.

Chapter Goals

At the end of this chapter, you will be able to:

● Describe how to use Acrobat JavaScript to create and modify PDF files.

● Convert PDF documents to XML format.

Contents

Topics
Creating and Modifying PDF Files
Creating PDF Files from Multiple Files
Converting PDF Documents to XML Format

Creating and Modifying PDF Files

Acrobat JavaScript provides support for dynamic PDF file creation and content generation. This means that it is possible to dynamically create a new PDF file and modify its contents in an automated fashion. This can help make a document responsive to user input and enhance the workflow process.

To create a new PDF file, invoke the **app** object's **newDoc()** method, as shown in the example below:

```
var myDoc = app.newDoc();
```

Once this statement has been executed, you may add content by invoking methods contained within the **doc** object, as indicated below in Table 4.1:

TABLE 4.1   *Acrobat JavaScript Usage for Creating Content in a PDF Document*

Content	Object	Methods
page	**doc**	**newPage, insertPages, replacePages**
page	**template**	**spawn**
annot	**doc**	**addAnnot**
field	**doc**	**addField**
icon	**doc**	**addIcon**
link	**doc**	**addLink**
document-level JavaScript	**doc**	**addScript**
thumbnails	**doc**	**addThumbnails**
bookmark	**doc.bookmarkRoot**	**createChild, insertChild**
web link	**doc**	**addWebLinks**
template	**doc**	**createTemplate**

Combining PDF Documents

You can take advantage of Acrobat JavaScript to customize and automate the process of combining PDF documents.

If you would like to combine multiple PDF files into a single PDF document, you may do so through a series of calls to the **doc** object's **insertPages** method. For example, the following code creates a new document that is composed from two other documents:

```
// Create a new PDF document:
var newDoc = app.newDoc();

// Insert doc1.pdf:
newDoc.insertPages({
    nPage : -1,
    cPath : "/c/doc1.pdf",
});

// Insert doc2.pdf:
newDoc.insertPages({
    nPage : newDoc.numPages,
    cPath : "/c/doc2.pdf",
});

// Save the new document:
newDoc.saveAs({
    "/c/myNewDoc.pdf");
});

// Close the new document without notifying the user:
newDoc.closeDoc(true);
```

Creating PDF Files from Multiple Files

It is possible to dynamically add content from other sources into a new PDF file. The sources may include files whose types conform to Multipurpose Internet Mail Extensions (MIME[®]) type definitions. One simple approach would be to invoke the **app** object's **openDoc** method to convert and open other files with supported formats, and then save or combine files as a PDF document.

Another way to import an external file into a PDF document is to invoke the **doc** object's **importDataObject** method. After doing this, it is possible to extract information from the external file for placement and presentation within the PDF document. For example, the following code illustrates how to import an XML file and extract its content:

```
this.importDataObject("myFile", "/c/myFile.xml");
var myData = this.getDataObject("myFile");
for (var i in myData)
    console.println(myData[i]);
```

If you would like to automate the insertion of multiple PDF files into a single PDF document, you may do so through a series of calls to the **doc** object's **insertPages** and **replacePages** methods. For example, the following code inserts a cover page at the beginning of the current document:

```
this.insertPages({
    nPage : -1,
    cPath : "/c/myCoverPage.pdf",
    nStart: 0
});
```

Finally, if you would like to use a portion of the current document to create a new document, you may do so by invoking the **doc** object's **extractPages** method. For example, suppose the document consists of a sequence of invoices, each of which occupies one page. The following code creates separate PDF files, one for each invoice:

```
var filename = "invoice";
for (var i = 0; i < this.numPages; i++)
    this.extractPages({
        nStart: i,
        cPath : filename + i + ".pdf"
    });
```

Cropping and Rotating Pages

Cropping Pages

The Acrobat JavaScript **doc** object provides methods for setting and retrieving the page layout dimensions. These are the **setPageBoxes** and **getPageBox** methods. There are five types of boxes available:

● Art

● Bleed

● Crop

● Media

● Trim

The **setPageBoxes** method accepts the following parameters:

● **cBox**: the type of box

● **nStart**: the zero-based index of the beginning page

● **nEnd**: the zero-based index of the last page

● **rBox**: the rectangle in rotated user space

For example, the following code crops pages 2-5 of the document to a 400 by 500 pixel area:

```
this.setPageBoxes({
    cBox: "Crop",
    nStart: 2,
    nEnd: 5,
    rBox: [100,100,500,600]
});
```

The **getPageBox** method accepts the following parameters:

● **cBox**: the type of box

● **nPage**: the zero-based index of the page

For example, the following code retrieves the crop box for page 3:

```
var rect = this.getPageBox("Crop", 3);
```

Rotating Pages

You may use Acrobat JavaScript to rotate pages in 90-degree increments in the clockwise direction relative to the normal position. This means that if you specify a 90-degree rotation, no matter what the current orientation is, the upper portion of the page is placed on the right side of your screen.

The **doc** object's **setPageRotations** and **getPageRotation** methods are used to set and retrieve page rotations.

The **setPageRotations** method accepts three parameters:

● **nStart**: the zero-based index of the beginning page

● **nEnd**: the zero-based index of the last page

● **nRotate**: 0, 90, 180, or 270 are the possible values for the clockwise rotation

In the following example, pages 2 and 5 are rotated 90 degrees in the clockwise direction:

```
this.setPageRotations(2,5,90);
```

To retrieve the rotation for a given page, invoke the **doc** object's **getPageRotation** method, which requires only the page number as a parameter. The following code retrieves and displays the rotation in degrees for page 3 of the document:

```
var rotation = this.getPageRotation(3);
console.println("Page 3 is rotated " + rotation + " degrees.");
```

Extracting, Moving, Deleting, Replacing, and Copying Pages

The Acrobat JavaScript **doc** object, in combination with the **app** object, can be used to extract pages from one document and place them in another, and moving or copying pages within or between documents.

The **app** object an be used to create or open any document. To create a new document, invoke its **newDoc** method, and to open an existing document, invoke its **openDoc** method.

The **doc** object offers three useful methods for handling pages:

● **insertPages**: inserts pages from the source document into the current document

● **deletePages**: deletes pages from the document

● **replacePages**: replaces pages in the current document with pages from the source document.

These methods enable you to customize the page content within and between documents.

Suppose you would like to remove pages within a document. Invoke the **doc** object's **deletePages** method, which accepts two parameters:

● **nStart**: the zero-based index of the beginning page

● **nEnd**: the zero-based index of the last page

For example, the following code deletes pages 2 through 5 of the current document:

```
this.deletePages({nStart: 2, nEnd: 5});
```

Suppose you would like to copy pages from one document to another. Invoke the **doc** object's **insertPages** method, which accepts four parameters:

● **nPage**: the zero-based index of the page after which to insert the new pages

● **cPath**: the device-independent pathname of the source file

● **nStart**: the zero-based index of the beginning page

● **nEnd**: the zero-based index of the last page

For example, the following code inserts pages 2 through 5 from `mySource.pdf` at the beginning of the current document:

```
this.insertPages({
    nPage: -1,
    cPath: "/C/mySource.pdf",
    nStart: 2,
    nEnd: 5
});
```

You can combine these operations to extract pages from one document and move them to another (they will be deleted from the first document). The following code will extract pages 2 through 5 in `mySource.pdf` and move them into `myTarget.pdf`:

```
// this represents myTarget.pdf
//
// First copy the pages from the source to the target document
this.insertPages({
    nPage: -1,
    cPath: "/C/mySource.pdf",
    nStart: 2,
    nEnd: 5
});
//
// Now delete the pages from the source document
var source = app.openDoc("/C/mySource.pdf");
source.deletePages({nStart: 2, nEnd: 5});
```

To replace pages in one document with pages from another document, invoke the target document's **replacePages** method, which accepts four parameters:

- **nPage**: the zero-based index of the page at which to start replacing pages

- **cPath**: the device-independent pathname of the source file

- **nStart**: the zero-based index of the beginning page

- **nEnd**: the zero-based index of the last page

In the following example, pages 2 through 5 from `mySource.pdf` replace pages 30 through 33 of `myTarget.pdf`:

```
// this represents myTarget.pdf
this.replacePages({
    nPage: 30,
    cPath: "/C/mySource.pdf",
    nStart: 2,
    nEnd: 5
});
```

To safely move pages within the same document, it is advisable to perform the following sequence:

1. Copy the source pages to a temporary file.

2. Insert the pages in the temporary file at the new desired location in the original document.

3. Delete the source pages from the original document.

The following example moves pages 2-5 after page 30 in the document:

```
// First create the temporary document:
var tempDoc = app.newDoc("/C/temp.pdf");

// Copy pages 2-5 into the temporary file
tempDoc.insertPages({
    cPath: "/C/mySource.pdf",
    nStart: 2,
    nEnd: 5
});

// Copy all of the temporary file pages back into the original:
this.insertPages({
    nPage: 30,
    cPath: "/C/temp.pdf"
});

// Now delete pages 2-5 from the source document
this.deletePages({nStart: 2, nEnd: 5});
```

Adding Watermarks and Backgrounds

The **doc** object's **addWatermarkFromText** and **addWatermarkFromFile** methods create watermarks within a document, and place them in optional content groups (OCGs).

The **addWatermarkFromFile** method adds a page as a watermark to the specified pages in the document. The example below adds the first page of watermark.pdf as a watermark to the center of all pages within the current document:

```
this.addWatermarkFromFile("/C/watermark.pdf");
```

In the next example, the **addWatermarkFromFile** method is used to add the second page of watermark.pdf as a watermark to the first 10 pages of the current document. It is rotated counterclockwise by 45 degrees, and positioned one inch down and two inches over from the top left corner of each page:

```
this.addWatermarkFromFile({
    cDIPath: "/C/watermark.pdf",
    nSourcePage: 1,
    nEnd: 9,
    nHorizAlign: 0,
    nVertAlign: 0,
    nHorizValue: 144,
    nVertValue: -72,
    nRotation: 45
});
```

It is also possible to use the **addWatermarkFromText** method to create watermarks. In this next example, the word **Confidential** is placed in the center of all the pages of the document, and its font helps it stand out:

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

Adding Headers and Footers

As you learned in Adding Watermarks and Backgrounds, you may use Acrobat JavaScript's watermarking functionality to add headers and footers to your documents by invoking the **doc** object's **addWatermarkFromText** method, which can be applied as a header or footer by specifying the placement of the text at the top or bottom of the page. The following example places a multiline header, one inch down and one inch over from the top right corner of all the pages within the current document:

```
this.addWatermarkFromText({
    cText: "Confidential Document",
    nTextAlign: 2,
    nHorizAlign: 2,
    nVertAlign: 0,
    nHorizValue: -72,
    nVertValue: -72
});
```

Converting PDF Documents to XML Format

Since XML is often the basis for information exchange within Web Services and enterprise infrastructures, it may often be useful to convert your PDF documents into XML format.

It is a straightforward process to do this using the **doc** object's **saveAs** method, which not only performs the conversion to XML, but also to a number of other formats.

In order to convert your PDF document to a given format, you will need to determine the device-independent path to which you will save your file, and the conversion ID used to save in the desired format. A list of conversion IDs for all formats is provided in the ASN documentation. For XML, the conversion ID is `"com.adobe.acrobat.xml-1-00"`.

The following code converts the current PDF file to `C:\test.xml`:

```
this.saveAs("/c/test.xml", "com.adobe.acrobat.xml-1-00");
```

5 **Print Production**

Introduction

This chapter will provide you with an in-depth understanding of the ways in which you may manage print production workflows for PDF documents.

Chapter Goals

At the end of this chapter, you will be able to:

- Use Acrobat JavaScript to automate and control print quality.
- Determine whether a file will be sent to a printer or a PostScript file.
- Control how PDF layers are printed.

Contents

Topics
Printing PDF Documents
Printing Documents with Layers
Setting Advanced Print Options

Print Production

Since printing involves sending pages to an output device, there are many options that can affect print quality. Acrobat JavaScript can be used to enhance and automate the use of these options in print production workflows, primarily through the use of the **printParams** object, whose properties and methods are described below in Table 5.1:

TABLE 5.1   *PrintParams Properties*

Property	Description
binaryOK	Binary printer channel is supported
bitmapDPI	DPI used for bitmaps or rasterizing transparency
colorOverride	Uses color override settings
colorProfile	Color profile based on available color spaces
constants	Wrapper object for **printParams** constants
downloadFarEastFonts	Sends Far East fonts to the printer
fileName	Filename is used when printing to a file instead of a printer
firstPage	The first zero-based page to be printed
flags	A bit field of flags to control printing options
fontPolicy	Used to determine when fonts are emitted
gradientDPI	The DPI used for rasterizing gradients
interactive	Sets the level of interaction for the user
lastPage	The last zero-based page to be printed
pageHandling	How pages will be handled (fit, shrink, or tiled)
pageSubset	Even, odd, or all pages are printed
printAsImage	Sends pages as large bitmaps
printContent	Determines whether form fields and comments will be printed
printerName	The name of the destination printer
psLevel	The level of PostScript emitted to the printer
rasterFlags	A bit field of flags for outlines, clips, and overprint

TABLE 5.1 *PrintParams Properties*

Property	Description
reversePages	Prints pages in reverse order
tileLabel	Labels each page of tiled output
tileMark	Output marks to cut the page and where overlap occurs
tileOverlap	The number of points that tiled pages have in common
tileScale	The amount that tiled pages are scaled
transparencyLevel	The degree to which high level drawing operators are preserved
userPrinterCRD	Determines whether the printer Color Rendering Dictionary is used
useT1Conversion	Determines whether Type 1 fonts will be converted

In addition to the **printParams** object's properties, the **app** object's **printColorProfiles** and **printerNames** properties provide a list of available color spaces and printer names, respectively. Also, the **field** object's **print** method can be used to determine whether an individual field will be printed.

Printing PDF Documents

It is possible to use Acrobat JavaScript to specify whether a PDF document is sent to a printer or to a PostScript file. In either case, to print a PDF document, invoke the **doc** object's **print** method. Its parameters are described below in Table 5.2:

TABLE 5.2 *Print Method Parameters*

Parameter	Description
bUI	Determines whether to present a user interface to the user
nStart	The zero-based index of the beginning page
nEnd	The zero-based index of the last page
bSilent	Suppresses the **Cancel** dialog box while the document is printed
bShrinkToFit	Determines whether the page is shrunk to fit the imageable area of the printed page

TABLE 5.2   *Print Method Parameters*

Parameter	Description
`bPrintAsImage`	Determines whether to print each page as an image
`bReverse`	Determines whether to print in reverse page order
`bAnnotations`	Determines whether to print annotations
`printParams`	The **printParams** object containing the printing settings

In the first example below, pages 1-10 of the document are sent to the default printer, print silently without user interaction, and are shrunk to fit the imageable area of the pages:

```
this.print({
    bUI: false,
    bSilent: true,
    bShrinkToFit: true,
    nStart: 1,
    nEnd: 10
});
```

To print the document to a PostScript file, obtain the **printParams** object by invoking the **doc** object's **getPrintParams** method. Set its **printerName** property to the empty string, and set its **fileName** property to a string containing the device-independent path of the PostScript file to which it will be printed, as shown in the following example:

```
var pp = this.getPrintParams();
pp.printerName = "";
pp.fileName = "/C/myPSDoc.ps";
this.print(pp);
```

If you would like send the file to a particular printer, you may specify the printer by setting the **printerName** property of the **printParams** object, as shown in the following example:

```
var pp = this.getPrintParams();
pp.interactive = pp.constants.interactionLevel.automatic;
pp.printerName = "hp officejet d series";
this.print(pp);
```

Silent Printing

There are various ways to print a document without requiring user interaction. One way is to use the **doc** object's **print** method and set the **bSilent** attribute to **true**, as shown in "Printing PDF Documents" on page 79 and in the following example:

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

If you would like to print without requiring user interaction, but would like to display a progress monitor and automatically disappearing cancel dialog box, use the interactive property as shown in the following example:

```
var pp = this.getPrintParams();
pp.interactive = pp.constants.interactionLevel.automatic;
```

There are many options you may choose without requiring user interaction. For example, you can select the paper tray:

```
var fv = pp.constants.flagValues;
pp.flags |= fv.setPageSize;
```

These coding approaches may be used in menus or buttons within a PDF file, may exist at the folder or batch levels, and are available through Acrobat or Adobe Reader 6.0 or later. For more information, see the *Acrobat JavaScript Scripting Reference*, as well as the Acrobat SDK samples SDKSilentPrint.js and SDKJSSnippet1.pdf.

Printing Documents with Layers

The **printParams** object's **printContent** property can be used to control whether document content, form fields, and comments will be printed. In the following example, only the form field contents will be printed (this is useful when sending data to preprinted forms):

```
var pp = this.getPrintParams();
pp.interactive = pp.constants.interactionLevel.silent;
pp.printContent = pp.constants.printContent.formFieldsOnly;
this.print(pp);
```

Setting Advanced Print Options

You can set the **printParams** object's properties to specify advanced options including output, marks and bleeds, transparency flattening, PostScript options, and font options.

Specifying Output Settings

You may obtain a listing of printer color spaces available by invoking the **app** object's **printColorProfiles** method. You may then assign one of these values to the **printParams** object's **colorProfile** property.

In addition, you may set the **printParams** object's **flags** property to specify advanced Output settings, such as applying proof settings, shown in the example below:

```
var pp = this.getPrintParams();
var fv = pp.constants.flagValues;
pp.flags |= fv.applySoftProofSettings;
this.print(pp);
```

Specifying Marks and Bleeds

You can specify the types of tile marks and where overlap occurs by setting the **printParams** object's **tileMark** property. For example, in the following code, Western style tile marks are printed:

```
var pp = this.getPrintParams();
pp.tileMark = pp.constants.tileMarks.west;
this.print(pp);
```

Setting PostScript Options

You may set the **printParams** object's **flags** property to specify advanced PostScript settings, such as emitting undercolor removal/black generation, shown in the example below:

```
var pp = this.getPrintParams();
var fv = pp.constants.flagValues;
pp.flags &= ~(fv.suppressBG | fv.suppressUCR);
this.print(pp);
```

In addition, you may set the **printParams** object's **psLevel** property to specify the level of PostScript emitted to PostScript printers. In addition, if the printer only supports PostScript level 1, set the **printParams** object's **printAsImage** property to **true**.

Setting Font Options

You can control the font policy by setting the **printParams** object's **fontPolicy** property. There are three values that may be used:

- **everyPage**: emit needed fonts for every page, freeing fonts from the previous page. This is useful for printers having a small amount of memory.
- **jobStart**: emit all fonts at the beginning of the print job, free them at the end of the print job. This is useful for printers having a large amount of memory.
- **pageRange**: emit the fonts needed for a given range of pages, free them once those pages are printed. This may be used to optimize the balance between memory and speed constraints.

In the following example, all the fonts are emitted at the beginning of the print job, and freed once the job is finished:

```
var pp = this.getPrintParams();
pp.fontPolicy = pp.constants.fontPolicy.jobStart;
this.print(pp);
```

You may also control whether Type 1 fonts will be converted to alternative font representations, by setting the **printParams** object's **useT1Conversion** property. There are three values that may be used:

- **auto**: let Acrobat decide whether to disable the conversion, based on its internal list of printers that have problems with these fonts.
- **use**: allow conversion of Type 1 fonts.
- **noUse**: do not allow conversion of Type 1 fonts.

In the following example, conversion of Type 1 fonts is set to automatic:

```
var pp = this.getPrintParams();
pp.useT1Conversion = pp.usages.useT1Conversion.auto;
this.print(pp);
```

Finally, it is possible to send Far East fonts to the printer by setting the **printParams** object's **downloadFarEastFonts** property to **true**.

6 Using Acrobat JavaScript in Forms

Introduction

In this chapter you will learn how to extend the functionality of Acrobat forms through the application of Acrobat JavaScript. You will learn how to generate, modify, and enhance all types of PDF forms and the elements they contain, and ensure the proper collection and export of information in various formats relevant to your workflow needs. In addition, you will understand how to leverage the XML Forms Architecture (XFA) so that your presentation format will be not only responsive to user input, but will also ensure that the information can be exchanged with Web Services and enterprise infrastructures.

Chapter Goals

At the end of this chapter, you will be able to:

● Use Acrobat JavaScript to create and enhance all types of PDF forms.

● Create and modify form fields and their properties.

● Use Acrobat JavaScript to make your forms secure, accessible, and web-ready.

● Create XML forms and migrate legacy forms to XFA format.

● Automate the collection and export of form data.

● Make forms accessible, secure, and web-ready.

Contents

Topics
Forms Essentials
Filling in PDF Forms
Making Forms Accessible
Using JavaScript to Secure Forms

Forms Essentials

Introduction

Acrobat JavaScript can be integrated into your forms to enhance their interactive capabilities. You can extend the capability of your forms by using Acrobat JavaScript to automate formatting, calculations, and data validation. In addition, you can develop customized actions assigned to user events. Finally, it is possible for your forms to interact with databases and Web services.

Forms Essentials Topics
About PDF Forms
Creating Acrobat Form Fields
Setting Acrobat Form Field Properties
Making a Form Fillable
Setting the Hierarchy of Form Fields
Creating Forms From Scratch
Using Custom JavaScripts in Forms
Introduction to XML Forms Architecture (XFA)
Forms Migration: Working with Forms Created in Acrobat 6.0 or Earlier

About PDF Forms

Types of PDF Forms

There are two types of PDF forms: Acrobat forms and XML forms.

Acrobat forms present information using form fields. They are useful for providing the user with a structured format within which to view or print information. Forms permit the user to fill in information, select choices, and digitally sign the document. Once the user has entered in data, the information within the PDF can be sent to the next step in the workflow for extraction or, in the case of browser-based forms, immediately transferred to a database. If you are creating a new form, the most recommended type is a XML form since its format readily allows for Web service interactions and compatibility with document processing needs within enterprise-wide infrastructures.

The new Adobe XML forms model uses a Document Object Model (DOM) architecture to manage the components that comprise a form. These include the base template, the form itself, and the data contained within the form fields. In addition, all calculations, validations, and formatting are specified and managed within the DOM and XML processes.

Static XML forms were supported in Acrobat 6.0, and *dynamic XML forms* are now supported in Acrobat 7.0. Both types are created using Adobe LiveCycle Designer. A static XML form presents a fixed set of text, graphics, and field areas at all times. Dynamic XML forms are created by dividing a form into a series of subforms and repeating subforms. They support dynamically changing fields that can grow or shrink based on content, variable-size rows and tables, and intelligent data import/export features.

Elements of Acrobat Forms

The form fields used in Acrobat forms are the basis of interaction with the user. They include buttons, check boxes, combo boxes, list boxes, radio buttons, text fields, and digital signature fields. In addition, you can enhance the appearance and value of your forms through the use of tables, templates, watermarks, and other user interface elements such as bookmarks, thumbnails, and dialogs. Finally, the Acrobat JavaScript methods you define in response to events will help customize the utility and behavior of the form within the context of its workflow.

Text fields can be useful for either presenting information or collecting data entered by the user, such as an address or telephone number.

Digital signature fields can be used to ensure the security of a document.

When presenting the user with decisions or choices, you may use check boxes and radio buttons for a relatively small set of choices, or list boxes and combo boxes for a larger set of dynamically changing choices.

Guidelines for Creating a New Form

When designing a PDF form, consider first its purpose and the data it must manage. It may be that the same page is used in multiple contexts, depending on user interactions and decisions. In this case, there may be multiple sets of form fields. When this occurs, treat each set of form fields as a different problem, as though each set had its own page. This will also require extra logic applied to visibility settings. Your form design may have dynamically changing features such as the current date, as well as convenience options such as automatic generation of email messages. It may even have a dynamically changing appearance and layout which is responsive to user interactions.

Usability is a major factor in the design of forms since they are essential graphical user interfaces, so layout and clarity will be a major consideration. Finally, consider the medium on which the form will be presented: screens with limited resolution may affect your decisions, and printing characteristics may also be relevant.

When creating forms programmatically, consider the form elements that will be needed for a given area. Declare those variables associated with the form elements, and apply logical groupings to those elements that belong to the same collections, such as radio buttons or check boxes. This will simplify the task of assigning properties, formatting options, validation scripts, calculation scripts, and tabbing order to each of the individual form elements.

The creation of a new form, whether done through the Acrobat layout tools or LiveCycle™ Designer, or programmatically through Acrobat JavaScript, will require that you consider the following:

● How the form fields will be positioned.

● Which form fields will be associated in collections so that their properties can be set with consistency and efficiency.

● How size, alignment, and distribution of form fields within the document will be determined.

● When and how to set up duplicate form fields so that when the user types information into one form field, that information automatically appears in the duplicate form fields.

● When to create multiple form fields for array-based access and algorithms.

● The tab order of form fields.

Creating Acrobat Form Fields

There are seven types of Acrobat form fields, each associated with a field type value as shown below in Table 6.1:

TABLE 6.1 *Acrobat Form Field Types*

Form Field	Field Type Value
Button	`button`
Check Box	`checkbox`
Combo Box	`combobox`
List Box	`listbox`
Radio Button	`radiobutton`
Text Field	`text`
Digital Signature	`signature`

You can use Acrobat JavaScript to create a form field by invoking the **doc** object's **addField()** method, which returns a **field** object. This method permits you to specify the following information:

1. The field name. This may include hierarchical syntax in order to facilitate logical groupings. For example, the name **myGroup.firstField** implies that the form field **firstField** belongs to a group of fields called **myGroup**. The advantage of creating logical hierarchies is that you can enforce consistency among the properties of related form fields by setting the properties of the group, which automatically propagate to all form fields within the group.

2. One of the seven field type values listed above, surrounded by quotes.

3. The page number where the form field is placed, which corresponds to a zero-based indexing scheme. Thus, the first page is considered to be page 0.

4. The location, specified in rotated user space (the origin is located at the bottom left corner of the page), on the page where the form field is placed. The location is specified through the use of an array of 4 values. The first two values represent the coordinates of the upper left corner, and the second two values represent the coordinates of the lower right corner: [*upper-left x, upper-left y, lower-right x, lower-right y*].

For example, suppose you would like to place a Button named **myButton** on the first page of the document. Assume that the button is one inch wide, one inch tall, and located 100 points in from the left side of the page and 400 points up from the bottom of the page (there are 72 points in 1 inch). The code for creating this button would appear as follows:

```
var name = "myButton";
var type = "button";
var page = 0;
var location = [100, 472, 172, 400];
var myField = this.addField(name, type, page, location);
```

This approach to creating form fields is applicable to all fields, but it should be noted that radio buttons require special treatment. Since a set of radio buttons represents a set of mutually exclusive choices, they belong to the same group. Because of this, the names of all radio buttons in the same group must be identical. In addition, the export values of the set of radio buttons must be set with a single statement, in which an array of values are assigned through a call to the radio button collection's **setExportValues** method.

For example, suppose we would like to create a set of three radio buttons, each 12 points wide and 12 points high, all named **myRadio**. We will place them on page 5 of the document, and their export values will be **"Yes"**, **"No"**, and **"Cancel"**. They can be created as shown in the code given below:

```
var name = "myRadio";
var type = "radiobutton";
var page = 5;
this.addField(name, type, page, [400, 442, 412, 430]);
this.addField(name, type, page, [400, 427, 412, 415]);
var rb = this.addField(name, type, page, [400, 412, 412, 400]);
rb.setExportValues(["Yes", "No", "Cancel"]);
```

This code actually creates an array of 3 radio buttons, named **myRadio.0**, **myRadio.1**, and **myRadio.2**.

Setting Acrobat Form Field Properties

Acrobat Javascript provides a large number of properties and methods for determining the appearance and associated actions of form fields. In this section you will learn what properties and methods are available, and how to write JavaScripts that control the appearance and behavior of form fields.

Form Field Properties Topics
Format Options
Button Properties
Checkbox Properties
Combo Box Properties
Listbox Properties
Radio Button Properties
Signature Properties
Text Field Properties
Validation Options
Calculation Options
Highlighting Required Form Fields
Alerting Users Automatically for Required Form Fields

Format Options

Format options applicable to form fields are applied to appearance, validation of data, and calculation of results, and differ according to field type. The **field** class contains many properties that may be used to set the format of a given form field. The most basic property of every form field is its **name**, which provides the reference necessary for subsequent access and modification.

General formatting options that apply to all form fields include the display rectangle, border style, border line thickness, stroke color, orientation, background color, and tooltip. In addition, you can choose whether it should be read only, have the ability to scroll, and be visible on screen or in print.

There are also specific settings you can apply to text characteristics, button and icon size and position relationships, button appearance when pushed, check box and radio button glyph appearance, and the number and selection options for combo box and list box items.

All formatting options are listed and described below in Table 6.2.

TABLE 6.2 *Format Options*

Format	Description	Field Properties
display rectangle	position and size of field on page	`rect`
border style	rectangle border appearance	`borderStyle`
stroke color	applied to edge of surrounding rectangle	`strokeColor`
border thickness	width of the edge of the surrounding rectangle	`lineWidth`
orientation	rotation of field in 90-degree increments	`rotation`
background color	background color of field (gray, transparent, RGB, or CMYK)	`fillColor`
tooltip	short description of field that appears on mouse-over	`userName`
read only	whether the user may change the field contents	`readonly`
scrolling	whether text fields may scroll	`doNotScroll`
display	visible or hidden on screen or in print	`display`

TABLE 6.2   *Format Options*

Format	Description	Field Properties
text	font, color, size, rich text, comb format, multiline, limit to number of characters, file selection format, or password format	`textFont, textColor, textSize, richText, richValue, comb, multiline, charLimit, fileSelect, password`
text alignment	controls text layout in text fields	`alignment`
button alignment	controls alignment of icon on button face	`buttonAlignX, buttonAlignY`
button icon scaling	relative scaling of an icon to fit inside a button face	`buttonFitBounds, buttonScaleHow, buttonScaleWhen`
highlight mode	indicates how a button will appear when pushed	`highlight`
glyph style	for checkbox and radio buttons	`style`
number of items	number of items in a combo box or list box	`numItems`
editable	whether the user can type in a combo box	`editable`
multiple selection	whether multiple listbox items may be selected	`multipleSelection`

Button Properties

We will begin by creating a button named **myButton**:

```
var f = this.addField("myButton", "button", 0, [200, 250, 250, 400]);
```

To create a blue border along the edges of its surrounding rectangle, we will set its **strokeColor** property:

```
f.strokeColor = color.blue;
```

In addition, you may select from one of the following choices to specify its border style: *solid* (**border.s**), *beveled* (**border.b**), *dashed* (**border.d**), *inset* (**border.i**), or *underline* (**border.u**). In this case we will make the border appear beveled by setting its **borderStyle** property:

```
f.borderStyle = border.b;
```

To set the line thickness (in points) of the border, set its **lineWidth** property:

```
f.lineWidth = 1;
```

To set its background color to yellow, we will set its **fillColor** property:

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

To specify the text that appears on the button, invoke its **buttonSetCaption** method:

```
f.buttonSetCaption("Click Here");
```

You may set the text size, color, and font:

```
f.textSize = 16;
f.textColor = color.red;
f.textFont = font.Times;
```

To create a tooltip that appears when the mouse hovers over the button, set its **userName** property:

```
f.userName = "This is a button tooltip for myButton.";
```

In addition to the text, it is also possible to specify the relative positioning of the icon and text on the button's face. In this case, we will set the layout so that the icon appears to the left of the text:

```
f.buttonPosition = position.iconTextH;
```

To specify whether the button should be visible either on screen or when printing, set its **display** property:

```
f.display = display.visible;
```

To set the button's appearance in response to user interaction, set its **highlight** property to one of the following values: *none* (**highlight.n**), *invert* (**highlight.i**), *push* (**highlight.p**), or *outline* (**highlight.o**). In this example , we will specify that the button appears to be pushed:

```
f.highlight = highlight.p;
```

It is possible to specify the scaling characteristics of the icon within the button face. You may determine when scaling takes place by setting the button's **buttonScaleWhen** property to one of the following values: *always* (**scaleWhen.always**), *never* (**scaleWhen.never**), if the icon is *too big* (**scaleWhen.tooBig**), or if the icon is *too small* (**scaleWhen.tooSmall**). In this case, we will specify that the button always scales:

```
f.buttonScaleWhen = scaleWhen.always;
```

You may also determine whether the scaling will be proportional by setting the **buttonScaleHow** property to one of the following values: **buttonScaleHow.proportional** or **buttonScaleHow.anamorphic**. In this case, we will specify that the button scales proportionally:

```
f.buttonScaleHow = buttonScaleHow.proportional;
```

To guarantee that the icon scales within the bounds of the rectangular region for the button, set the **buttonFitBounds** property:

```
f.buttonFitBounds = true;
```

You can specify the alignment characteristics of the icon by setting its **buttonAlignX** and **buttonAlignY** properties. This is done by specifying the percentage of the unused horizontal space from the left or the vertical space from the bottom that is distributed. A value of **50** would mean that the 50 percent of the unused space would be distributed to the left or bottom of the icon (centered). We will center our icon in both dimensions:

```
f.buttonAlignX = 50;
f.buttonAlignY = 50;
```

Now that you have prepared the space within the button for the icon, you can import an icon into the document and place it within the button's area. To import an icon, invoke the **doc** object's **importIcon** method. To place the icon in the button, invoke the button object's **setIcon** method. Assuming you have already made the icon available in the document and that its variable name is **myIcon**, the following code would cause the icon to appear on the button's face:

```
f.setIcon(myIcon);
```

To rotate the button counterclockwise, set its **rotation** property:

```
f.rotation = 90;
```

Finally, you will undoubtedly wish to associate an action to be executed when the button is clicked. You may do this by invoking the button's **setAction** method, which requires a trigger (an indication of the type of mouse event) and an associated script. The possible triggers are **MouseUp**, **MouseDown**, **MouseEnter**, **MouseExit**, **OnFocus**, and **OnBlur**. The following code displays a greeting when the button is clicked:

```
f.setAction("MouseUp", "app.alert('Hello');" );
```

Checkbox Properties

The checkbox field supports many of the same properties as the button, and actions are handled in the same manner. The properties common to both form fields are:

- **userName**
- **readonly**
- **display**
- **rotation**
- **strokeColor**
- **fillColor**
- **lineWidth**
- **borderStyle**
- **textSize**
- **textColor**

In the case of **textFont**, however, the font is always set to **Adobe Pi**.

You may choose from six different glyph styles (the appearance of the check symbol that appears when the user clicks in the check box): *check* (**style.ch**), *cross* (**style.cr**), *diamond* (**style.di**), *circle* (**style.ci**), *star* (**style.st**), and *square* (**style.sq**). For example, the following code causes a check to appear when the user clicks in the check box:

```
f.style = style.ch;
```

When the user selects a check box, an export value may be associated with the option invoking the **field** object's **setExportValues** property. For example, the code below associates the export value **"buy"** with the check box:

```
f.setExportValues(["buy"]);
```

If there are several check box fields, you may wish to indicate that one particular form field is always checked by default. To do this, you must do two things:

1. Invoke the field's **defaultIsChecked** method. Note that since there may be several check boxes that belong to the same group, the method requires that you specify the zero-based index of the particular check box.

2. Reset the field to ensure that the default is applied by invoking the **doc** object's **resetForm** method.

This process is shown in the following code:

```
f.defaultIsChecked(0); // 0 means that check box #0 is checked
this.resetForm([f.name]);
```

Combo Box Properties

The combo box has the same properties as the button and check box fields. Its primary differences lie in its nature. Since the combo box maintains an item list in which the user may be allowed to enter custom text, it offers several properties that support its formatting options.

If you would like the user to be permitted to enter custom text, set the field's **editable** property, as shown in the following code:

```
f.editable = true;
```

You may decide whether the user's custom text will be checked for spelling by setting its **doNotSpellCheck** property. The following code indicates that the spelling is not checked:

```
f.doNotSpellCheck = true;
```

A combo box may interact with the user in one of two ways: either a selection automatically results in a response, or the user first makes their selection and then takes a subsequent action, such as clicking a **Submit** button.

In the first case, as soon as the user clicks on an item in the combo box, an action may automatically be triggered. If you would like to design your combo box this way, then set its **commitOnSelChange** property to **true**. Otherwise, set the value to **false**. The following code commits the selected value immediately:

```
f.commitOnSelChange = true;
```

To set the export values for the combo box items, invoke its **setItems** method, which can be used to set both the user and export values. In this case, the user value (the value that appears in the combo box) is the first value in every pair, and the export value is the second. The following code results in the full state names appearing in the combo box, with abbreviated state names as their corresponding export values:

```
f.setItems( ["Ohio", "OH"], ["Oregon", "OR"], ["Arizona", "AZ"] );
```

In many cases, it is desirable to maintain a sorted collection of values in a combo box. In order to do this, you will need to write your own sorting script. Recall that the JavaScript **Array** object has a **sort** method that takes an optional argument which may be a comparison function.

This means that you must first define a **compare** function that accepts two parameters. The function must return a negative value when the first parameter is less than the second, **0** if the two parameters are equivalent, and a positive value if the first parameter is greater.

In the following example, we define a **compare** function that accepts two parameters, both of which are user/export value pairs, and compares their user values. For example, if the first parameter is ["Ohio", "OH"] and the second parameter is ["Arizona", "AZ"], the **compare** function returns **1**, since "Ohio" is greater than "Arizona":

```
function compare (a,b)
{
    if (a[0] < b[0]) return -1; // index 0 means user value
    if (a[0] > b[0]) return 1;
    return 0;
}
```

Create a temporary array of values and populate it with the user/export value pairs in your combo box field. The following code creates the array, iterates through the combo box items, and copies them into the array:

```
var arr = new Array();
var f = this.getField("myCombobox");
for (var i = 0; i < f.numItems; i++)
    arr[i] = [f.getItemAt(i,false), f.getItemAt(i)];
```

At this point you may invoke the **Array** object's **sort** method and replace the items in the combo box field:

```
arr.sort(compare); // sort the array using your compare method
f.clearItems();
f.setItems(arr);
```

To specify whether the combo box automatically formats its entries as numbers, percentage values, or other specialized formats, you may use the functions shown below in Table 6.3. Their definitions are located in the file Javascripts\aform.js:

TABLE 6.3 **Combobox Formatting Functions**

Format	Function
Number	**AFNumber_Format**
Percentage	**AFPercent_Format**
Date	**AFDate_FormatEx**
Time	**AFTime_Format**
Special	**AFSpecialFormat**

In all cases, invoke the method and pass in the **"Format"** trigger name as the first parameter, followed by a script containing a call to one of the functions. For example, the code below sets up the Number format:

```
f.setAction("Format", 'AFNumberFormat(2,0,0,0,"\u20ac",true)');
```

NOTE: You may also create Custom formatting.

Listbox Properties

A list box has many of the same properties as buttons and combo boxes, except for the fact that the user may not enter custom text and, consequently, that spellchecking is not available.

However, the user may select multiple entries. To enable this feature, set its **multipleSelection** property to **true**, as shown in the code below:

```
f.multipleSelection = true;
```

To set up an action associated with a selected item, invoke its **setAction** method and pass in the **"Keystroke"** trigger name as the first parameter, as shown in the code below:

```
f.setAction( "Keystroke", "myListboxJavascript();" );
```

Radio Button Properties

The unique nature of radio buttons is that they are always created in sets, and represent a collection of mutually exclusive choices. This means that when you create a set of radio buttons, you must give all of them identical names and access them with zero-based numeric indices, as you learned earlier in Creating Acrobat Form Fields.

Radio buttons have many of the same properties as buttons and check boxes.

Signature Properties

Signature fields have many of the same properties as buttons. You may set the action of a signature field by invoking its **setAction** method and passing in the **"Format"** trigger name as the first parameter. When the user signs the form, you may reformat other form fields with the script you pass into the **setAction** method.

Once a document is signed, you may wish to lock certain form fields within the document. You may do so by creating a script containing a call to the signature field's **setLock** method and passing that script as the second parameter to the signature field's **setAction** method.

The **setLock** method requires a **Lock** object, which you will obtain by invoking the form field's **getLock** method. Once you obtain the **Lock** object, set its **action** and **fields** properties. The **action** property can be set to one of 3 values: **"All"** (lock all fields), **"exclude"** (lock all fields except for these), or **"include"** (lock only these fields). The **fields** property is an array of fields.

For example, suppose you created a signature and would like to lock the form field whose name is **myField** after the user signs the document. The following code would lock **myField**:

```
var oLock = f.getLock();
oLock.action = "include";
oLock.fields = new Array("myField");
f.setLock(oLock);
```

To actually sign a document, you must do two things: choose a security handler, and then invoke the signature field's **signatureSign** method. The following code is an example of how to choose a handler and actually sign the document:

```
var ppklite = security.getHandler("Adobe.PPKLite");
var oParams = {
    password: "myPassword",
    cDIPath: "/C/signatures/myName.pfx" // digital signature profile
};
ppklite.login(oParams);
f.signatureSign(ppklite,
    {
        password: "myPassword",
        location: "San Jose, CA",
        reason: "I am approving this document",
        contactInfo: "userName@adobe.com",
        appearance: "Fancy"
    }
); //end of signature
```

Text Field Properties

The text field has many of the same properties as buttons and combo boxes. In addition, it offers the following specialized properties shown below in Table 6.4:

TABLE 6.4   *Text Field Formatting*

Property	Description	Example
alignment	justify text	`f.alignment = "center";`
defaultValue	set a default text string	`f.defaultValue = "Name: ";`
multiline	allow multiple lines in the area	`f.multiline = true;`
doNotScroll	permit scrolling of long text	`f.doNotScroll = true;`
richText	set rich text formatting	`f.richText = true;`
charLimit	limit on number of characters in area	`f.charLimit = 40;`
password	use special formatting to protect the user's password	`f.password = true;`
fileSelect	format field as a file pathname	`f.fileSelect = true;`
doNotSpellCheck	set spell checking	`f.doNotSpellCheck = true;`
comb	comb of characters subject to limitation set by **charLimit**	`f.comb = true;`

Validation Options

You may use validation to enforce valid ranges, values, or characters entered in form fields. The main reason to use validation is to ensure that users are only permitted to enter valid data into a form field.

To apply validation to a field action, invoke the field's **setAction** method, and pass **"Validate"** as the first parameter and a script containing a call to **AFRange_Validate** as the second parameter. The function **AFRange_Validate** is located in `Javascripts\aform.js`, and requires 4 parameters: **bGreaterThan** (boolean value), **nGreaterThan** (first value in the range), **bLessThan** (boolean value), and **nLessThan** (last value in the range).

For example, the following code ensures that all values entered in the form field are from 0 to 100, inclusive:

```
f.setAction(
    "Validate",
    'AFRange_Validate(true, 0, true, 100)'
);
```

Calculation Options

Calculation options make it possible to automate mathematical calculations associated with form fields. To apply a calculation to a form field action, invoke the form field's **setAction** method, pass **"Calculate"** as the first parameter, and pass a script containing a call to a calculation script as the second parameter.

If you would like to perform simple calculations on an array of form field values, you may use a convenient script already written for you called **AFSimple_Calculate** in the second parameter. The function **AFSimple_Calculate** is located in `Javascripts\aform.js`, and requires 2 parameters: **cFunction** (the type of calculation to be performed) and **cFields** (a field list that may be separated by commas or spaces, or may be an array). The first parameter specifies the type of calculation, which may be a sum (**"SUM"**), product (**"PRD"**), average (**"AVG"**), minimum (**"MIN"**), or maximum (**"MAX"**).

For example, the following code calculates the sum of the values entered into two text areas on a form:

```
var arr = new Array("line.1", "line.2");
f.setAction(
    "Calculate",
    'AFSimple_Calculate("SUM",arr)'
);
```

Highlighting Required Form Fields

Some text fields on a form may not be left blank: these are called required form fields. It is helpful to the user to highlight them so that they can be easily recognized. To do this, create a Highlight annotation as shown in the following example:

```
this.addAnnot({
    page: 2,
    strokeColor: color.yellow,
    type: "Highlight",
    quads: this.getPageNthWordQuads(2, 3),
});
```

Alerting Users Automatically for Required Form Fields

A user action such as clicking a **Submit** button or leaving a page could trigger validation scripts detecting either bad input or blank form fields that alert the user to the fact that some required form fields require their attention.

For example, suppose the user has forgotten to fill in the text field **myText** and has clicked the **Submit** button. The following script would alert the user to the problem, and could be followed with code that highlights the text:

```
app.alert("You forgot the field shown in yellow.");
```

Making a Form Fillable

In order for a form to be fillable, its text fields or combo boxes must be formatted so that the user can edit them.

Enabling Typing

If you would like a text area to be enabled for typing, set its **readonly** property to **false**, as shown in the following code:

```
f.readonly = false;
```

If you would like a combo box to be enabled for typing, set its **editable** property to **true**, as shown in the following code:

```
f.editable = true;
```

Setting the Hierarchy of Form Fields

The hierarchy of form fields is determined according to a naming strategy that uses "dot" notation. For example, suppose you have 4 radio buttons that all belong to the same group. The group could be named **myGroup**. The 4 radio buttons would then be named **myGroup.0**, **myGroup.1**, **myGroup.2**, and **myGroup.3**. The convenience of this naming system becomes most apparent when you decide that all members of the group should have the same property characteristics. For example, to set the glyph style of all 4 radio buttons, you can do this with one line of code as shown below:

```
var f = this.getField("myGroup");
f.style = style.ch; // glyph style for all radio buttons
```

This notation is certainly more convenient than typing in 4 nearly identical lines of code.

Suppose that you have 3 different groups, each of which has several radio buttons. You can extend the hierarchical naming to this situation as well. Suppose that you would like to be able to assign common property values to all 3 groups. To do this, you could create a parent name for all the groups, such as **mySet**. The 3 groups would be named **mySet.0**, **mySet.1**, and **mySet.2**. Suppose the first group, **mySet.0**, has 2 radio buttons in it. They would be named **mySet.0.0** and **mySet.0.1**. To set the glyph style for all 3 groups of radio buttons, you can do this, again, with just a single line of code as shown below:

```
var f = this.getField("mySet");
f.style = style.ch; // glyph style for all 3 groups
```

You can also differentiate the groups within the hierarchy. Suppose you would like to designate a yellow background color for the third set. You could do so with the following statements:

```
var f = this.getField("mySet.2");
f.fillColor = color.yellow; // affects only the 3rd group
```

Creating Forms From Scratch

Positioning Form Form Fields

Remember that form field positioning takes place in *Rotated User Space*, in which the origin of a page is located at the bottom left corner. This differs from *Info Space* and may require that you occasionally perform transformations.

For example, suppose that you use the **Info** panel to obtain the coordinates of a given rectangle. Use the **doc** object's **getPageBox** method to obtain the coordinates in Rotated User Space for the page, and then subtract the Info Space y-coordinates from the page height provided by the **getPageBox** method.

If you are accustomed to calculating the positions of form fields from the top left corner of a page, the following example will serve as a template for obtaining the correct position. In this example, we will position a 1 inch by 2 inch form field 0.5 inches from the top of the page and 1 inch from the left side:

```
// 1 inch = 72 points
var inch = 72;

// obtain the page coordinates in Rotated User Space
var aRect = this.getPageBox({nPage: 2});

// position the top left corner 1 inch from the left side
aRect[0] += 1 * inch;

// make the rectangle 1 inch wide
aRect[2] = aRect[0] + 1*inch;

// top left corner is 0.5 inch down from the top of the page
aRect[1] -= 0.5*inch;

// make the rectangle 2 inches tall
aRect[3] = aRect[1] - 2*inch;

// draw the button
var f = this.addField("myButton", "button", 2, aRect);
```

Duplicating Form Fields

It may sometimes be useful to duplicate information typed in by the user in other pages of the document. For example, you might wish to display the user's name on every page of the document.

To automate this, give all such form fields the same name and actions. Then whenever the user triggers a related action, the same information appears in all form fields containing that name.

To duplicate form fields in general, assign the same name and actions to each of them. In the example below, we will create duplicate text fields, each named **myField**, on page 2 of the document, and we will set the background color of every instance to yellow:

```
for (var i = 0; i < 5; p++)
{
    var aRect = [36, 36+100*i, 72, 144+100*i];
    var f = this.addField("myField", "text", 2, aRect);
    f.fillColor = yellow;
}
```

Creating Multiple Form Fields

The best approach to creating a row, column, or grid of form fields is to use array notation in combination with hierarchical naming.

For example, the following code creates a column of 3 text fields:

```
var myColumn = new Array(3);
myColumn[0] = "myFieldCol.name";
myColumn[1] = "myFieldCol.birthday";
myColumn[2] = "myFieldCol.ssn";
var aRect = [36, 36, 72, 144];
for (var i=0; i<myColumn.length; i++)
{
    aRect[1] += 100; // move the next field down 100 points
    aRect[3] += 100; // move the next field down 100 points
    var f = this.addField(myColumn[i], "text", 0, aRect);
```

Creating Form Fields That Span Multiple Pages

From a programmatic standpoint, duplicating form fields across multiple pages requires the same steps as duplicating form fields in general (see Duplicating Form Fields). The only difference is specifying the page number.

To duplicate form fields in general, assign the same name and actions to each of them. In the example below, we will create duplicate text fields, each named **myField**, on every page of the document, and we will set the background color of every instance to yellow:

```
for (var p = 0; p < this.numPages; p++)
{
    var aRect = [36, 36, 72, 144];
    var f = this.addField("myField", "text", p, aRect);
    f.fillColor = yellow;
}
```

Defining the Tabbing Order

You may specify the tabbing order on a given page by invoking the **doc** object's **setPageTabOrder** method, which requires two parameters: the page number and the order to be used.

There are three options for tabbing order: you may specify tabbing by rows (**"rows"**), columns (**"columns"**), or document structure (**"structure"**).

For example, the following code sets up tabbing by rows for page 2 of the document:

```
this.setPageTabOrder(2, "rows");
```

Defining Form Field Calculation Order

When you add a text field or combo box that has a calculation script to a document, the new form field's name is appended to the *calculation order array*. When a calculation event occurs, the calculation script for each of the form fields in the array runs, beginning with the first element in the array (array index **0**) and continuing in sequence to the end of the array.

If you would like one form field to have calculation precedence over another, you can change its calculation index, accessed through the **field** object's **calcOrderIndex** property. A form field script with a lower calculation index executes first. The following code guarantees that the calculation script for form field **a** will run before the one for form field **b**:

```
b.calcOrderIndex = a.calcOrderIndex + 1;
```

Making PDF Forms Web-Ready

PDF forms can be used in workflows that require the exchange of information over the Web. You can create forms that run in Web browsers, and can submit and retrieve information between the client and server by making a **Submit** button available in the form. The button can perform similar tasks to those of HTML scripting macros.

You will need a CGI application on the Web server that can facilitate the exchange of your form's information with a database. The CGI application must be able to retrieve information from forms in HTML, FDF, or XML formats.

In order to enable your PDF forms for data exchange over the Web, be sure to name your form fields so that they match those in the CGI application. In addition, be sure to specify the export values for radio buttons and check boxes.

The client side form data may be posted to the server using the HTML, FDF, XFDF, or PDF formats. Note that the use of XFDF format results in the submission of XML-formatted data to the server, which will need an XML parser or library to read the XFDF data. The equivalent MIME types for all posted form data are shown below in Table 6.5.

TABLE 6.5   *MIME Types for Data Formats*

Data Format	MIME Type
HTML	application/x-www-form-urlencoded
FDF	application/vnd.fdf
XFDF	application/vnd.adobe.xfdf
PDF	application/pdf
XML	application/xml

Creating Submit Buttons

When you use the Button tool, use the `MouseUp` trigger and select **Submit a Form**. You may specify which data format is used when you select the **Export Format** option. If it is necessary for the server to be able to recognize and reconstruct a digital signature, it is advisable that you choose the **Incremental Changes to the PDF** option.

Creating Reset Form Buttons

In this case, when you create the `MouseUp` trigger, select **Reset a Form**. Now you will be able to specify which form fields are reset.

Creating Import Data Buttons

First set up an FDF file with common data. In this case, when you create the `MouseUp` trigger, select **Import Form Data**.

Defining CGI Export Values

If you are storing the form data in a database and the data is either different from the item designated by the form field (such as those in combo boxes or list boxes) or the form field is a radio button or check box (all of which must have different export values), then you will need to define CGI export values, which represents identifying information about the form field to the CGI application.

Using Custom JavaScripts in Forms

The most common uses for Acrobat JavaScript in enhancing the interactive behavior of forms are in formatting, calculating, and validating data. In addition, you may write custom scripts for different types of mouse actions, as well as database connectivity.

Introduction to XML Forms Architecture (XFA)

The XML Forms Architecture (XFA) is an XML-based architecture which supports the production of business form documents through the use of templates based on the XML language. Its features address a variety of workflow needs including dynamic reflow, dynamic actions based on user interaction or automated server events, headers, footers, and complex representations of forms capable of large-scale data processing.

XFA can be understood in terms of two major components: templates and content. The templates define presentation, calculation, and interaction rules, and are based on XML. Content is the static or dynamic data, stored in the document, that is bound to the templates.

Dynamic XFA indicates that the content will be defined later after binding to a template. This also means that the following is possible:

● Form fields may be moved or resized.

● Form fields automatically grow or shrink according to the amount of text added or removed.

● As a form field grows, it can span multiple pages.

● Repeating subforms may be spawned as needed, and page contents shift accordingly.

● Elements on the page are shown or hidden as needed.

To take advantage of the rich forms functionality offered by the XFA plug-in, use Adobe LiveCycle Designer® to create or edit the templates and save your forms in the XML Data Package format (XDP) or as a PDF document. Use XDP if the data is processed by server applications, and PDF if the data is processed by Acrobat.

Enabling Dynamic Layout and Rendering

In order to enable dynamic layout and rendering for a form, save it from LiveCycle Designer as a "Dynamic PDF Form File".

Growable Form Fields

The elements, which include Fields, Subforms, Areas, Content Areas, and Exclusion Groups, expand to fit the data they contain. They may relocate in response to changes in the location or extent of their containing elements, or if they flow together with other elements in the same container.

If the element reaches the nominal content region of the containing page, then it splits so that it may be contained across both pages.

Variable-Size Rows and Tables

Subforms may repeat to accommodate incoming data. For example, when importing a variable number of subforms containing entries for name, address, and telephone number, form fields need to be added or removed based on the volume of data. This means that the number of rows in a table may increase.

Multiple Customized Forms within a Form Based on User Input

Subforms may also be subject to conditions. For example, form fields for dependent children would become visible if the user checks a box indicating that there are dependent children. In addition, XFA allows multiple form fields with the same name and multiple copies of the same form.

Handling Image Data

Images are handled as data and are considered to have their own field type. There is automatic support for browsing images in all standard raster image formats (including PNG, GIF, TIFF, and JPEG).

Dynamic Tooltips

XFA forms support dynamic tooltips, including support for individual radio buttons in a group.

XFA-Specific JavaScript Methods

Acrobat JavaScript provides access to the XFA **appModel** container, which provides the properties and methods indicated below in Table 6.6 and Table 6.7.

TABLE 6.6   *appModel Properties*

appModel Property	Description
aliasNode	The node represented by the alias for this model
all	Returns all nodes with the same name or class
appModelName	Returns **xfa**

TABLE 6.6 *appModel Properties*

appModel Property	Description
`classAll`	Returns all nodes with the same class name
`classIndex`	Returns the position of this node in the collection of nodes having the same class name
`className`	Returns the class name of the object
`context`	The current node (needed for resolveNode and resolveNodes)
`id`	The ID of the current node
`index`	Returns the position of this node in the collection of nodes having the same name
`isContainer`	Returns true if this is a container object
`isNull`	Returns true if the node has a null value
`model`	Returns the XFA model for this node
`name`	The name of this node
`nodes`	A list of child nodes for this node
`ns`	The namespace for this node (or XFAModel)
`oneOfChild`	Retrieves or sets the child that has the XFA::oneOfChild relationship to its parent
`parent`	Retrieves the parent of this node
`somExpression`	Retrieves the SOM expression for this node
`this`	Retrieves the current node (starting node for resolveNode and resolveNodes)

TABLE 6.7 *XFA appModel Methods*

appModel Method	Description
`applyXSL`	Performs an XSL transformation of the current node
`assignNode`	Sets the value of the node, and creates one if necessary
`clearErrorList`	Clears the current list of errors
`clone`	Clones a node (and its subtree if specified)
`createNode`	Creates a new XFA node based on a valid classname
`getAttribute`	Retrieves a specified attribute value
`getElement`	Retrieves a specified property element
`isCompatibleNS`	Determines if two namespaces are equivalent
`isPropertySpecified`	Checks if a specific property has been defined for the node
`loadXML`	Loads and appends the current XML document to the node
`resolveNode`	Obtains the node corresponding to the SOM expression
`resolveNodes`	Obtains the XFATreeList corresponding to the SOM expression
`saveXML`	Saves the current node to a string
`setAttribute`	Sets a specified attribute value
`setElement`	Sets a specified property element

The XFA DOM model contains a root object that consists of either a **treeList** or a **Tree**. A **treeList** consists of a list of nodes (which is why it is sometimes called a **NodeList**). A **Tree** consists of a hierarchical structure of nodes, each of which may contain content, a model, or a **textNode**.

The following properties and methods are available in a **treeList** object:

> Properties:
> **length**
>
> Methods:
> **append**, **insert**, **item**, **namedItem**, and **remove**.

The following properties and methods are available in a **Tree** object:

> Properties:
> **all**, **classAll**, **classIndex**, **index**, **name**, **nodes**,
> **parent**, **somExpression**
>
> Methods:
> **resolveNode**, **resolveNodes**

Each **Node** element represents an element and its children in the XML DOM. To obtain a string representation of the node, invoke its **saveXML** method. To apply an XSL transform (XSLT) to the node and its children, invoke its **applyXSL** method. The following properties and methods are available in a **Node** object:

> Properties:
> **id**, **isContainer**, **isNull**, **model**, **ns**, **oneOfChild**
>
> Methods:
> **applyXSL**, **assignNode**, **clone**, **getAttribute**, **getElement**,
> **isPropertySpecified**, **loadXML**, **saveXML**, **setAttribute**,
> **setElement**

There are two approaches to accessing content in an XML stream. In the first approach, XFA syntax may be used to manipulate the XML DOM. In the second approach, you may use standard XPath syntax.

The Acrobat JavaScript **XML** object provides two methods useful for manipulating XML documents: **applyXPath** and **parse**.

The **applyXPath** permits the manipulation of an XML document via an XPath, and the **parse** method creates an object representing an XML document tree. Both of these return an **XFA** object.

The first approach involves the usage of the **XFL** object's **parse** method for accessing and manipulating XML streams. The second approach involves the usage of the **XFL** object's **applyXPath** method.

Both approaches are illustrated below.

In this first example, the usage of the **parse** method is illustrated below:

```
// Create the XML stream
var parseString = "<purchase>";
parseString += "<product>";
parseString += "<price>299.00</price>";
parseString += "<name>iPod</name>";
parseString += "</product>";
parseString += "<product>";
parseString += "<price>49.95</price>";
parseString += "<name>case</name>";
parseString += "</product>";
parseString += "</purchase>";

// Now create the DOM:
var x = XML.parse(parseString);

// Use the XFA API to obtain the XFA tree list for products:
var products = x.resolveNodes("product");

// Obtain the iPod product:
var iPod = products.item(0);

// Obtain the iPod price:
var price = iPod.getElement("price").value;

// Convert the price to a string:
var priceString = price.saveXML();
```

This next example accomplishes the same task through the usage of the **applyXPath** method:

```
// Create the XML stream
var parseString = "<purchase>";
parseString += "<product>";
parseString += "<price>299.00</price>";
parseString += "<name>iPod</name>";
parseString += "</product>";
parseString += "<product>";
parseString += "<price>49.95</price>";
parseString += "<name>case</name>";
parseString += "</product>";
parseString += "</purchase>";

// Now create the DOM:
var x = XMLData.parse(parseString,false);

// Set up the XPath expression:
var xPathExpr = "//purchase/product/[name='iPod']/price";

// Now get the iPod price:
var price = XMLData.applyXPath(x, xPathExpr);
```

JavaScript Methods Not Enabled in XML Forms

The following Acrobat JavaScript **doc** methods are not available from an XML form:

- **getField**
- **getNthFieldName**
- **addNewField**
- **addField**
- **removeField**
- **setPageTabOrder**

ADO Support for Windows

It is now possible to access both individual and multiple records. Forms can be enabled with ADO support for more direct database interaction.

Forms Migration: Working with Forms Created in Acrobat 6.0 or Earlier

Detecting XML Forms and Earlier Form Types

To determine whether a document is an XML form, invoke the **doc** object's **dynamicXFAForm** method, which returns a boolean result. Note that an Acrobat form does not have an **xfa** object.

Filling in PDF Forms

Completing Form Fields

Completing Form Fields Automatically

In many cases, you may wish to assign a character limit to text fields. For example, when collecting credit card or social security numbers, you may require that the number be split across several text fields. For example, the social security number 555-55-5555 would require three text fields of lengths 3, 2, and 4, respectively.

Spell-Checking Text in Forms

You can spell-check text typed in note comments and form fields, but not in the underlying PDF document.

Importing and Exporting Form Data

Form data can be exported to a separate file, which can then be sent using email or over the Web. When doing this, save either to Forms Data Format (FDF) or XML-based FDF (XFDF). This creates an export file much smaller than the original PDF file.

When importing XFDF data, you will need an XML parser or library to read the XFDF data.

Note that Acrobat forms support the FDF, XFDF, tab-delimited text, and XML formats, and that XML forms support XML and XDP formats.

Saving Form Data as XML or XML Data Package (XDP)

To save your form data in XML format, invoke the **doc** object's **saveAs** method using the conversion ID for XML, as shown in the code below:

```
this.saveAs("myDoc.xml", "com.adobe.acrobat.xml-1-00");
```

To take advantage of XFA functionality, you may save your forms in the XML Data Package format (XDP). This simply requires the usage of the conversion ID for XDP, as shown in the code below:

```
this.saveAs("myDoc.xml", "com.adobe.acrobat.xdp");
```

Emailing Completed Forms

To email a completed form in FDF format, invoke the **doc** object's **mailForm** method, which exports the data to FDF and sends it via email.

To make an interactive email session, pass **true** to the first parameter, which specifies whether a user interface should be used, as shown in the code below:

```
this.mailForm(true);
```

To send the exported FDF data automatically, pass **false** to the first parameter, and specify the **cTO**, **cCc**, **cBcc**, **cSubject**, and **cMsg** fields (all of which are optional), as shown in the code below:

```
this.mailForm(false, "recipient@adobe.com");
```

Global Submit

Suppose you have a document that contains multiple attachments, from which you would like to compile information for submission to a server in XML format. You can create a **Global Submit** button whose **MouseUp** action contains a script that collects the data from each of the attachments and creates a unified collection in XML format.

To do this, you will need to invoke the **doc** object's **openDataObject** method in order to open the attachments, followed by its **submitForm** method to upload the combined XML data to the server.

The following example merges the data from several XML form attachments and submits it to a server:

```
var oParent = event.target;

// Get the list of attachments:
var oDataObjects = oParent.dataObjects;
if (oDataObjects == null)
   app.alert("This form has no attachments!");
else {
   // Create the root node for the global submit:
   var oSubmitData = oParent.xfa.dataSets.createNode(
      "dataGroup",
      "globalSubmitRootNode"
   );

   // Iterate through all the attachments:
   var nChildren = oDataObjects.length;
   for (var iChild = 0; iChild < nChildren; i++) {

      // Open the next attachment:
      var oNextChild = oParent.openDataObject(
         oDataObjects[iChild].name
      );
      // Transfer its data to the XML collection:
      oSubmitData.nodes.append(
         oNextChild.xfa.data.nodes.item(0)
      );

      close the attachment//
      oNextChild.closeDoc();
   }

   // Submit the XML data collection to the server
   oParent.submitForm({
      cURL: "http://theserver.com/cgi-bin/thescript.cgi",
      cSubmitAs: "XML",
      oXML: oSubmitData
   });
}
```

Making Forms Accessible

Making a PDF Form accessible to users who have impaired motor or visual ability requires that the document be structured, which means that PDF tags present in the document ensure that the content is organized according to a logical structure tree. This means that you will have added tags to the document. Once you do this, you may specify alternative text within the tags.

You can make forms accessible through the use of Text-To-Speech engines and tagged annotations containing alternative text.

Text-To-Speech engines can translate structured text in a PDF document into audible sound, and tagged annotations containing alternative text can provide substitute content for graphical representations, which cannot be read by a screen reader. It is useful to consider embedding alternative text in links and bookmarks, as well as specifying the language of the document.

It is not necessary to sacrifice security in order to make a document accessible. Select **Document Properties > Security > Enable Text Access for Screen Reader Devices for the Visually Impaired**.

Text-To-Speech

In order for Text-To-Speech engines to be able to work with your document, it must be structured. You can create structured documents using Adobe FrameMaker® 7.0 or Adobe FrameMaker SGML 6.0 running in **Structured** mode.

To access the Text-To-Speech engine from Acrobat JavaScript, use the **TTS** object, which has methods to render text as digital audio and present it in spoken form to the user.

For example, the following code displays a message stating whether the TTS engine is available:

```
console.println("TTS available: " + tts.available);
```

The next code sample illustrates how to enumerate through all available speakers, queue a greeting into the TTS object for each one, and present the digital audio version of it to the user:

```
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();
}
```

The properties and methods of the **TTS** object are summarized below in Table 6.8 and Table 6.9.

TABLE 6.8   *TTS Properties*

Property	Description
available	Returns **true** if the Text-To-Speech engine is available
numSpeakers	Returns the number of speakers in the engine
pitch	The baseline pitch between 0 and 10
speaker	A speaker with desired tone quality
speechRate	The rate in words per minute
volume	The volume between 0 and 10

Tagging Annotations

TABLE 6.9   *TTS Methods*

Method	Description
getNthSpeakerName	Retrieves Nth speaker in current Text-To-Speech engine
pause	Pauses the audio output
qSilence	Queues a period of silence into the text
qSound	Inserts a sound cue using a .wav file
qText	Inserts text into the queue
reset	Stops playback, flush the queue, reset all Text-To-Speech properties
resume	Resumes playback on a paused **TTS** object
stop	Stops playback and flush the queue
talk	Sends queue contents to Text-To-Speech engine

Tagged files provide the greatest degree of accessibility, and are associated with a logical structure tree that supports the content. Annotations can be dynamically associated with a new structure tree that is separate from the original content of the document, thus supporting accessibility without modifying the original content. The annotation types supported for accessibility are:

> **Text**, **FreeText**, **Line**, **Square**, **Circle**, **Polygon**, **Polyline**, **Highlight**,
>
> **Underline**, **Squiggly**, **Strikeout**, **Stamp**, **Caret**, **Ink**, **Popup**, **FileAttachment**, **Sound**

To add an accessible tag, select **Advanced > Accessibility** and choose **Add Tags to Document**.

Using JavaScript to Secure Forms

As you learned earlier in Signature Properties, you can lock any form fields you deem appropriate once a document has been signed. In addition, you may also encrypt a document.

Acrobat JavaScript provides a number of objects that support security. These are managed by the **security** and **securityHander** objects for building certificates and signatures, as well as the **certificate**, **directory**, **signatureInfo**, and **dirConnection** objects which are used to access the user certificates. (The **certificate** object provides read-only access to an X.509 public key certificate).

These objects, in combination, provide you with the means to digitally sign or encrypt a document. Once you have built a list of authorized recipients, you may then encrypt the document using the **doc** object's **encryptForRecipients** method, save the document to commit the encryption, and email it to them.

For example, you can obtain a list of recipients for which the encrypted document is available, and then encrypt the document:

```
// Invoke the recipients dialog to select which recipients
// will be authorized to view the encrypted document:
var oOptions = {
    bAllowPermGroups: false,
    cNote: "Recipients with email and certificates",
    bRequireEmail: true,
    bUserCert: true
};
var oGroups = security.chooseRecipientsDialog(oOptions);


// Build the mailing list
var numCerts = oGroups[0].userEntities.length;
var cMsg = "Encrypted for these recipients:\n";
var mailList = new Array;
for (var i=0; i<numCerts; i++) {
    var ue = oGroups[0].userEntities[i];
    var oCert = ue.defaultEncryptCert;
    if (oCert == null) oCert = ue.certificates[0];
    cMsg += oCert.subjectCN + ", " + ue.email + "\n";
    var oRDN = oCert.subjectDN;
    if (ue.email) mailList[i] = ue.email;
    else if (oRDN.e) mailList[i] = oRDN.e;
}
// Now encrypt the document
this.encryptForRecipients(oGroups);
```

The **security** object's properties and methods are described below in Table 6.10 and Table 6.11.

TABLE 6.10 **Security Properties**

Property	Description
handlers	Returns an array of security handler names
validateSignaturesOnOpen	User preference to be automatically validated when document opens

TABLE 6.11 **Security Methods**

Method	Description
chooseRecipientsDialog	Opens a dialog to choose a list of recipients
getHandler	Obtains a security handler object
exportToFile	Saves a **Certificate** object to a local disk
importFromFile	Reads in a **Certificate** object from a local disk

7 Review, Markup, and Approval

Introduction

In this chapter you will learn how to make use of Acrobat's ability to facilitate online collaborative reviews for many types of content. At the heart of this process is the set of commenting, markup, and approval tools available to the reviewers, and the tools available to the initiator for managing the review.

You will be able to use Acrobat JavaScript features to customize the review process and how comments are handled, add additional annotations, and configure a SOAP-based online repository.

Chapter Goals

At the end of this chapter, you will be able to:

● Specify the different types of review workflows.

● Initiate a document review.

● Participate in a document review.

● Use the commenting, markup, and approval tools.

● Manage comments.

Contents

Topics
Online Collaboration Essentials
Using Commenting Tools
Managing Comments
Approving Documents Using Stamps (Japanese Workflows)

Online Collaboration Essentials

Introduction

You may initiate several types of review workflows for PDF documents:

● Email the document to all reviewers, and import the returned comments into the original document.

● Set up an automated email-based review.

● Set up an automated browser-based review through the use of a shared server.

● Initiate an email-based approval workflow.

● Initiate an Acrobat JavaScript-based review.

Online Collaboration Essentials Topics
Reviewing Documents with Additional Usage Rights
Emailing PDF Documents
Acrobat JavaScript-based Collaboration Driver

Reviewing Documents with Additional Usage Rights

For email-based reviews, the specification of additional usage rights within a document enables extra capabilities within Adobe Reader. This enables the reviewer to add comments, import and export form-related content, save the document, or apply a digital signature.

For example, when using the **doc** object's **encryptForRecipients** method, you may specify the following permissions for reviewers:

● **allowAll**: permits full and unrestricted access to the entire document.

● **allowAccessibility**: permits content accessed for readers with visual or motor impairments.

● **allowContentExtraction**: permits content copying and extraction.

● **allowChanges**: permits either no changes, or changes to part or all of the document assembly, content, forms, signatures, and notes.

● **allowPrinting**: permits no printing, low-quality printing, or high-quality printing.

The following code allows full and unrestricted access to the entire document for one set of users (**importantUsers**), and allows high quality printing for another set of users (**otherUsers**):

```
var sh = security.getHandler("Adobe.PPKMS");
var dir = sh.directories[0];
var dc = dir.connect();
dc.setOutputFields({oFields:["certificates"]});
var importantUsers = dc.search({oParams:{lastName:"Smith"}});
var otherUsers = dc.search({oParams:{lastName:"Jones"}});
this.encryptForRecipients({
    oGroups:[
        {
            oCerts: importantUsers,
            oPermissions: { allowAll: true }
        },
        {
            oCerts: otherUsers,
            oPermissions: { allowPrinting: "highQuality" }
        }
    ],
    bMetaData: true
});
```

Emailing PDF Documents

In addition to the **Email** options available in the Acrobat menu and toolbar, it is also possible to use Acrobat JavaScript to set up an automated email review workflow. This may be done through the **doc** object's **mailDoc** method. In the code shown below, the document is automatically sent to **recipient@adobe.com**:

```
this.mailDoc(
    false,
    "recipient@adobe.com", "", "",
    "Review",
    "Please review this document and return. Thank you."
);
```

NOTE: For Windows systems, the default mail program must be MAPI-enabled.

Acrobat JavaScript-based Collaboration Driver

Acrobat JavaScript can be used to describe the workflow for a given document review, and can be used in review management. This is done by specifying a state model for the types of annotations a reviewer may use and creating an annotation store on the server for customized comment and review within browser-based workflows. The **Collab** object provides you with control over the possible states **annot** objects may have, and may be used in conjunction with the **SOAP** object to create an annotation store.

There are several methods available within the **Dollab** object that enable you to describe the state model for the review: these include **addStateModel**, **getStateInModel**, **transitionToState**, and **removeStateModel**.

The **addStateModel** method is used to add a new state model to Acrobat describing the possible states for an **annot** object using the model, and the **removeStateModel** method removes the model, though it does not affect previously created **annot** objects. Their usage is shown in the code below:

```
// Add a state model
try{
    var myStates = new Object;
    myStates["initial"] = {cUIName: "Haven't reviewed it"};
    myStates["approved"] = {cUIName: "I approve"};
    myStates["rejected"] = {cUIName: "Forget it"};
    myStates["resubmit"] = {cUIName: "Make some changes"};
    Collab.addStateModel({
        cName: "ReviewStates",
        cUIName: "My Review",
        oStates: myStates,
        cDefault: "initial"
    });

    // Now transition myAnnot to the "approved" state:
    myAnnot.transitionToState("ReviewStates", "approved");
}
catch(e){console.println(e);}

// Now remove the state model
try {Collab.removeStateModel("ReviewStates");}
catch(e){console.println(e);}
```

You may also use the **SOAP** object's **connect**, **request**, and **response** methods to create customized commenting and review within browser-based workflows. You can do this by setting up a SOAP-based annotation store on the server using the **Collab** object's **addAnnotStore** and **setStoreSettings** methods.

The **Collab** object's **addAnnotStore** method requires three parameters:

> **cIntName**: The internal name for the annotation store.
>
> **cDispName**: The display name for the annotation store.
>
> **cStore**: The definition for the new **Collab** store class.

The new **Collab** store class must contain the following definitions for the functions used to add, delete, update, and enumerate through the array of annotations:

- **enumerate**: Communicates with the Web service to request the array of annotations stored on the server. It is used when the PDF document is loaded for online review, or when the user clicks **Upload** or **Send** on the **Commenting Toolbar**.

- **complete**: Passes the annotation data to the collaboration server so that it can be updated.

- **update**: Uploads new and modified annotations to the server.

The class **SDKSampleSOAPAnnotStore**, as shown in the sample code below, is defined in sdkSOAPCollabSample.js in the Acrobat SDK, and contains complete definitions of the three functions described above.

The sample code below provides a standard example of how to use the **SOAP** and **Collab** objects to customize your online collaborative review. Note that all of the reviewers must have a copy of the JavaScript collaboration store code. In Acrobat 7.0, the *Custom* collaboration store type allows you to put the JavaScript on the server. The store type used is **CUSTOM**, and the setting is a URL to the JavaScript file:

```
// Here is the URL for a SOAP HTTP service:
var mySetting = "http://sampleSite/comments.asmx?WSDL";

// Here is the internal name for the collaborative store:
var myType = "mySOAPCollabSample";

// Set the connection settings for the SOAP collab store:
Collab.setStoreSettings(mySetting, myType);

// Set the default collab store:
Collab.defaultStore = myType;
```

```
// Add the collab store to the Acrobat Collab servers:
if (typeof SOAPFileSys == "undefined")
    Collab.addAnnotStore(
        myType,
        "SOAP Sample",
        {
            // Annot store instantiation function is required:
            create: function(doc, user, settings)
            {
                if (settings && settings != "")
                    return new SDKSampleSOAPAnnotStore(
                        doc, user, settings
                    );
                else
                    return null;
            }
        }
    );
```

Using Commenting Tools

The **Commenting** and **Advanced Commenting** toolbars provide reviewers with the tools necessary to create comments, which may be placed in the document in the form of notes, highlighting, and other markup.

Topics
Adding Note Comments
Making Text Edits
Highlighting, Crossing Out, and Underlining Text
Adding and Deleting Custom Stamps
Adding Comments in a Text Box
Adding Attachments
Spell-checking in Comments and Forms
Adding Commenting Preferences
Changing Colors, Icons, and Other Comment Properties
Adding Watermarks
Approval

Adding Note Comments

The Acrobat JavaScript support for note comments is available through the **annot** object's **Text** and **FreeText** types.

Making Text Edits

Text edit comments, also known as markup, are used to indicate text that should either be removed or inserted. Text that should be removed appears to be crossed out, and text that should be inserted appears in a popup window triggered by a caret appearing within the text. The Acrobat JavaScript support for text edits is available through the **annot** object's **Text** and **StrikeOut** types.

Highlighting, Crossing Out, and Underlining Text

The Acrobat JavaScript support for highlighting, crossing out, and underlining text is available through the **annot** object's **HighLight**, **StrikeOut**, and **Underline** types.

Adding and Deleting Custom Stamps

A stamp can be created from a set of predefined stamps, dynamically created using system and identity information, or customized from PDF files or common graphic formats. There are a number of predefined stamps available through the **annot** object's **AP** method.

Adding Comments in a Text Box

You may include a text box comment in a document and control its border, background color, alignment, font, and size characteristic. The Acrobat JavaScript support for text box comments is available through the **annot** object's **Square** and **Line** types, as shown in the example below:

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

Adding Attachments

You may use Acrobat JavaScript to embed sound clips, images, and files within comments. The Acrobat support for sound and file attachments within comments is available through the **annot** object's **Sound** and **FileAttachment** types. Also, you may pass an *Icon Stream Generic Object* as an optional property (**oIcon**) of the **oStates** object literal used in the **Collab** object's **addStateModel** method.

Sound Attachments

To add a sound attachment using Acrobat JavaScript, invoke the **doc** object's **importSound** method to attach the sound file to the PDF. Then create an annotation using the **doc** object's **addAnnot** method, and associate the **Sound** object with the annotation by assigning the name of the **Sound** object to the **annot** object's **soundIcon** property. An example is given below:

```
// Attach the sound file to the PDF file
this.importSound("mySound", "/C/mySound.wav");

// Create the annotation and assign its soundIcon property
this.addAnnot({
    page: 0,
    type: "Sound",
    rect: [0,0,100,100],
    name: "mySoundAnnot",
    soundIcon: "mySound"
});
```

File Attachments

To add a file attachment to a document, invoke one of the **doc** object's **import** methods, such as **importDataObject**. Once the file contents have been imported they can be accessed using such methods as **getDataObject**, and either merged with your document content or saved to a disk location. For example, the following code imports the contents of `myFile.xml` and displays its contents:

```
// Open the file attachment named myFile.xml
this.importDataObject(myData, "/C/myFile.xml");

// Access its contents:
var contents = this.getDataObject("myData");

// Display its contents:
for (var i in contents) console.println(contents[i]);
```

To create an attachment annotation, set the **annot** object's **attachIcon** property. At this point, you may decide how to associate the attachment's name with the icon.

Personalizing Attachments with a Description

To add a description to any annotation, set its **contents** property. In the case of file or sound attachments, the property should be used to describe the contents of the attachment, as shown in the code below:

```
this.addAnnot({
    page: 0,
    type: "Sound",
    rect: [0,0,100,100],
    name: "mySoundAnnot",
    soundIcon: "mySound"
    contents: "This is a sound file."
});
```

Opening and Saving Attachments

You may use Acrobat JavaScript to open any attachment. The general-purpose way to do this is to invoke the **doc** object's **getDataObjectContents** method. The following code opens an attachment and displays its contents in the console:

```
var oFile = this.getDataObjectContents("MyAttachment");
var cFile = util.stringFromStream(oFile);

console.println(cFile);
```

To save and possibly open an attachment, invoke the **doc** object's **exportDataObject** method. The method has an optional parameter called **nLaunch** which may be set to one of three values: **0** (do not open, default), **1** (prompt the user for a save path, and open), **2** (do not prompt the user for a save path, and open). When using documents having extensions other than PDF, include the extension in the file name. The method usage is shown in the examples below:

```
// just save and do not open
this.exportDataObject("MyAttachment");

// save and open
this.exportDataObject({
    cName: "MyComments.xls", nLaunch: 1}
);
```

Saving Modified Files into the Primary Document

Once an attached file has been modified, it may be saved once again. To embed its contents into the primary document, invoke the **doc** object's **getDataObjectContents** method in order to begin accessing its contents, and **setDataObjectContents** in order to save the modified contents, as shown in the code sample below:

```
// Open the file:
this.importDataObject("myFile.txt", "/C/myFile.txt");

// access the FileStream (after which you may modify it):
var oFile = this.getDataObjectContents("myFile.txt");
var cFile = util.stringFromStream(oFile, "utf-8");

// now modify the string cFile
// ...

// Convert the modified string back to a FileStream:
oFile = util.streamFromString(cFile, "utf-8");

// Save the modified FileStream contents:
this.setDataObjectContents("myFile.txt", oFile);
```

Linking Between Files

You may set up links between files by setting the "Go to View" action of a link annotation. In the case of nested PDF documents, intermediate documents automatically open. To use Acrobat JavaScript to create a link, invoke the **doc** object's **addLink** method. To cause the link to open a file, set the **MouseUp** action to open the file.

Deleting Attachments

You may use Acrobat JavaScript to remove a file attachment from a document by invoking the **doc** object's **removeDataObject** method, as shown in the code below:

```
this.removeDataObject("myAttachment");
```

Spell-checking in Comments and Forms

You may check the spelling of any word using the **spell** object's **checkWord** method. This can be applied to any form field or annotation. First retrieve the contents, and submit each word to the method.

Setting Spelling Preferences

To set the dictionary order, first retrieve the array of dictionaries using the **doc** object's **spellDictionaryOrder** property. Then modify the order of the array entries, and assign the array to the same property. An array of currently available dictionaries can be obtained using the **spell** object's **dictionaryNames** property.

To set the language order, perform a similar algorithm using the **doc** object's **spellLanguageOrder** property. An array of currently available dictionaries can be obtained using the **spell** object's **languages** property.

Adding Words to a Dictionary

You may use Acrobat JavaScript to add words to a dictionary by invoking the **spell** object's **addWord** method, as shown in the code sample below:

```
spell.addWord(myDictionary, "myNewWord");
```

Adding Commenting Preferences

To use Acrobat JavaScript to set commenting preferences, create an object literal containing common properties to be applied to your comments. Then for every annotation, pass the object literal to its **annot** object's **setProps** method, as shown in the code sample below:

```
// Create the common properties in an object literal:
var myProps = {
    strokeColor: color.red,
    popupOpen: true,
    arrowBegin: "Diamond",
    arrowEnd: "OpenArrow"
};

// Assign the common properties to a previously created annot:
myAnnot.setProps(myProps);
```

Changing Colors, Icons, and Other Comment Properties

You may use Acrobat JavaScript to change the properties of any type of annotation. To change the background color of a comment, assign a new value to its **fillColor** property. To change the icon, assign a value to its **attachIcon**, **noteIcon**, or **soundIcon** property. All the comment properties are available through the **annot** object, and may be set by invoking its **setProps** method.

Adding Watermarks

A watermark is an area containing text or graphics appearing underneath or over existing document content when a document is printed. This is often referred to as layered content, and may appear differently in print than it does on the screen. For example, the word "Confidential" could appear as a watermark within a document.

You can add watermarks through Acrobat JavaScript by invoking the **doc** object's **addWatermarkFromFile** or **addWatermarkFromText** method, and you may also create stamp annotations. The **addWatermarkFromFile** method places, into a specified location and at a particular scale factor and rotation, a single page from any document format that Acrobat can convert to PDF (such as JPEG, PNG, TIFF, Word, or AutoCAD).

The Stamping User Interface

You may create an annotation using the **Stamp** type, and invoke the **annot** object's **AP** property to determine the appearance of the stamp in the document. The following appearance options are available for stamp annotations:

- **Approved**
- **AsIs**
- **Confidential**
- **Departmental**
- **Draft**
- **Experimental**
- **Expired**
- **Final**
- **ForComment**
- **ForPublicRelease**
- **NotApproved**
- **NotForPublicRelease**
- **Sold**
- **TopSecret**

For example, the following code adds an "Approved" stamp to the document:

```
var annot = this.addAnnot({
    page: 0,
    type: "Stamp",
    author: "Me",
    name: "myStamp",
    rect: [400,400,550,500],
    contents: "Good work!",
    AP: "Approved"
});
```

Header and Footer Functionality

You may use Acrobat JavaScript to add headers and footers to your documents. For example, you may use the **doc** object's **addWatermarkFromText** method, which has several properties useful for this specific purpose:

● **cText:** The actual text displayed in the header or footer.

● **nTextAlign:** How the text is aligned in the header or footer.

● **vTextAlign:** How the watermark is aligned vertically: a value of **0** aligns it at the top of the page (header), and a value of **2** aligns it at the bottom of the page (footer).

● **nStart:** The starting page for the watermark. A value of **-1** causes the resultant header or footer to appear on every page of the document.

Control of Font, Size, Placement, Rotation, and Opacity

There are several properties to the **doc** object's watermark addition methods useful for controlling font, size, placement, rotation and opacity. These are listed below:

● **cFont:** The font name.

● **cFontSize:** The font size (in points).

● **nTextAlign:** The text alignment.

● **nRotation:** The rotation in degrees.

● **nOpacity:** The opacity from 0.0 to 1.0, where 0 means transparent and 1 means opaque.

Approval

Approval workflows may include an automated process in which a PDF document is automatically sent via email to a recipient for their approval. For example, this may be accomplished through the usage of the **doc** object's **mailDoc** method. The user may then use a standard approval stamp (set through the **annot** object's **AP** property), use a custom stamp, or use a Hanko stamp to create a secure digital signature.

Managing Comments

Topics
Selecting, Moving, and Deleting Comments
Using the Comments List
Exporting and Importing Comments
Comparing Comments in Two PDF Documents
Aggregating Comments for Use in Excel
Extracting Comments in a Batch Process

Selecting, Moving, and Deleting Comments

Just as you can access the **Comments List** in the Acrobat user interface, you may likewise do so through Acrobat JavaScript, using the **doc** object's **syncAnnotScan** and **getAnnots** methods. The **syncAnnotScan** method guarantees that all annotations in the documents are scanned, and the **getAnnots** method returns a list of annotations satisfying specified criteria.

For example, the following code scans all the annotations on page 2 of the document and captures them all in the variable **myAnnotList**:

```
this.syncAnnotScan();
var myAnnotList = this.getAnnots({nPage: 2});
```

To move a comment, use the corresponding **annot** object's **setProps** method to specify a new location or page. To delete the comment, invoke the corresponding **annot** object's **destroy** method. In the code sample below, all the free text comments on page 2 of the document are deleted:

```
for (var i=0; i<myAnnotList.length; i++)
    if (myAnnotList[i].type == "FreeText")
        myAnnotList[i].destroy();
```

Using the Comments List

Once you have acquired the comments list through the **doc** object's **syncAnnotScan** and **getAnnots** methods, you may change their status, appearance, order, and visibility. In addition, you will be able to search for comments having certain characteristics.

Changing the Status of Comments

To change the status of a comment using Acrobat JavaScript, invoke the corresponding **annot** object's **transitionToState** method, as shown in the code below:

```
// Transition myAnnot to the "approved" state:
myAnnot.transitionToState("ReviewStates", "approved");
```

Changing the Appearance of Comments

You may change the appearance of a comment in a variety of ways. If the comment is a stamp annotation, you may change its appearance by setting its **AP** property. In general, the appearance of any comment may be changed by invoking the **annot** object's **setProps** method, as shown in the code below:

```
myAnnot.setProps({
    page: 0,
    points: [[10,40], [200,200]],
    strokeColor: color.red,
    popupOpen: true,
    popupRect: [200,100,400,200],
    arrowBegin: "Diamond",
    arrowEnd: "OpenArrow"
});
```

Marking Comments with Checkmarks

You may use the Acrobat user interface to place checkmarks next to comments, or use Acrobat JavaScript to do the equivalent by changing the status of a comment (see Changing the Status of Comments).

Sorting Comments

If you would like to sort comments using Acrobat JavaScript, you may do so by submitting an optional parameter to the **doc** object's **getAnnots** method. The **nSortBy** parameter may be assigned one of the following values:

● **ANSB_None:** Do not sort.

● **ANSB_Page:** Sort by page number.

● **ANSB_Author:** Sort by author.

● **ANSB_ModDate:** Sort by modification date.

● **ANSB_Type:** Sort by annotation type.

In addition, you may specify that the sorting be performed in reverse order by submitting the optional **bReverse** parameter to the method.

The code sample given below shows how to obtain a list of comments from page 2 of the document, sorted in reverse order by author:

```
this.syncAnnotScan();
var myAnnotList = this.getAnnots({
    nPage: 2,
    nSortBy: ANSB_Author,
    bReverse: true
});
```

Showing and Hiding Comments

To use Acrobat JavaScript to show or hide a comment, set its corresponding **annot** object's **hidden** property. For example, the following code hides **myAnnot**:

```
myAnnot.hidden = true;
```

Finding Comments

There are two ways to use Acrobat JavaScript to find a comment. If you know the name of the comment, you may invoke the **doc** object's **getAnnot** method. Otherwise you may obtain all the comments by invoking the **doc** object's **getAnnots** method, and iterate through the list.

The **getAnnot** method requires two parameters: the page number and the name of the annotation. The following code sample shows how to obtain a comment on page 2 named **knownName**:

```
var comment = this.getAnnot(2, "knownName");
```

Exporting and Importing Comments

To use Acrobat JavaScript to export all the comments in a file, invoke the **doc** object's **exportAsFDF** or **exportAsXFDF** methods. In both cases, set the **bAnnotations** parameter to **true**, as shown in the code sample below, which exports only the comments and nothing else:

```
this.exportAsFDF({bAnnotations: true});
```

To use Acrobat JavaScript to import comments from an FDF or XFDF into a file, invoke the **doc** object's **importAnFDF** or **importAnXFDF** methods.

Comparing Comments in Two PDF Documents

While the Acrobat user interface provides you with a menu choice for comparing two documents, it is possible to customize your comparisons using Acrobat JavaScript. To gain access to multiple documents, invoke the **app** object's **openDoc** method for each document you would like to analyze. Each **doc** object exposes the contents of each document, such as an array of annotations. You may then compare and report any information using customized algorithms. For example, the code below reports how many annotations exist in the two documents:

```
var doc2 = app.openDoc("/C/secondDoc.pdf");
var annotsDoc1 = this.getAnnots();
var annotsDoc2 = doc2.getAnnots();
console.println("Doc 1: " + annotsDoc1.length + " annots.");
console.println("Doc 2: " + annotsDoc2.length + " annots.");
```

Aggregating Comments for Use in Excel

The **doc** object's **exportDataObject** method may be used to create a tab-delimited text file, which can then be used in Excel. To use Acrobat JavaScript to aggregate comments for use in Excel, collect all the comments using the **doc** object's **getAnnots** method, iterate through them and save them into a tab-delimited string, create a text file attachment object using the **doc** object's **createDataObject** method, and pass the string to the **cValue** parameter in the **exportDataObject** method.

Extracting Comments in a Batch Process

In a batch process, you may open any number of **doc** objects using the **app** object's **openDoc** method. For each open document, you may invoke its corresponding **doc** object's **getAnnots** method to collect the comments in that file. If you would like to put all the comments together in one file, you may do so by creating a new document and saving the various arrays of comments into that new file.

Approving Documents Using Stamps (Japanese Workflows)

Approval workflows are similar to other email-based collaborative reviews, and provide you with the ability to set the order in which participants are contacted. This means that, based on the approval issued by a participant, the document may be mailed to the next participant, and an email may be sent to the initiator.

Topics
Setting up a Hanko Approval Workflow
Participating in a Hanko Approval Workflow

Setting up a Hanko Approval Workflow

A registered Hanko is a stamp used in Japanese document workflows, and may be used to sign official contracts. Every registered hanko is unique and is considered a legal form of identification.

A personal Hanko is not registered, and is used for more common types of signatures, such as those used in meeting notes or budget proposals. Everyone in an organization who is involved in a document review must add their Hanko to the document in order for it to gain final approval.

Adding Instructions

Acrobat provides an assistant to help you set up an approval workflow. You may customize your workflow as well, by adding form fields to the document containing recipient lists to be chosen by the participant. This way, in case there are multiple directions for a given branch in the workflow, the participant may invoke automated functions that send the document to the correct participants, as well as an email to the initiator containing a record of activity.

Sending a Document by Email for Approval

You may use Acrobat JavaScript to automate various steps within the workflow by sending the document and other information by email by via the `doc` object's `mailDoc` method.

Participating in a Hanko Approval Workflow

A participant receives an email with instructions for opening the document and completing their portion of the approval process. As noted above, this can be customized and automated through the use of form fields if the workflow is complex.

Applying a Hanko or Inkan Stamp

A Hanko stamp is a commenting tool used in approval workflows, and an Inkan stamp is a unique image that can represent an individual's identity and may be used in place of a signature. Both are created, customized, and managed through the Acrobat user interface.

In order to use a Hanko or Inkan stamp, you will need to create a custom stamp and add digital signature information. Once the stamp has been created, you may apply it in your workflows.

Using Other Stamps and Comments

You may use Acrobat JavaScript to create other stamps by creating a stamp annotation and setting its corresponding `annot` object's `AP` property. You can obtain a list of stamp names available in the document by accessing the `doc` object's `icons` property.

Installing and Customizing Hanko Stamps

Creating custom Hanko stamp information involves the combination of user information and a digital signature. Once you have set this up, it may be saved in a PDF file which is stored in the `Stamps` folder.

Creating Custom Inkan Stamps

To create an Inkan stamp, add your name, title, department, and company, choose a layout, and provide a name to use for the stamp. You may also import a PDF form to add customized features and additional fields containing personal information. In addition, it is possible to add secure digital signature information to an Inkan stamp.

Deleting Custom Stamps

You may delete any Hanko and Inkan stamps that you created, though it is not possible to delete any of the predefined stamps in the **Stamps** palette.

8 Working with Digital Media in PDF Documents

Introduction

In this chapter you will learn how to use Acrobat JavaScript to extend Acrobat's ability to integrate digital media into PDF documents. You will learn how to set up, control, and customize properties and preferences for media players and monitors, how to integrate movie and sound clips into your documents, and how to add, edit, and control the settings for their renditions.

Chapter Goals

At the end of this chapter, you will be able to:

● Customize the settings, renditions, and events associated with media players.

● Access and control the properties for all monitors connected to the system.

● Add movie and sound clips.

● Add and edit renditions.

● Control rendition settings.

● Set multimedia preferences that apply throughout a document.

Contents

Topics
Media Players: Control, Settings, Renditions, and Events
Monitors
Integrating Media into Documents
Setting Multimedia Preferences

Media Players: Control, Settings, Renditions, and Events

Introduction

There are several objects provided in Acrobat JavaScript that provide you with the means to customize the control, settings, renditions, and events related to media players. These are shown below in Table 8.1.

TABLE 8.1   *Media Player Objects*

Object	Description
`App.media`	Primary object for media control, settings, and renditions
`MediaOffset`	Time or frame position within a media clip
`Event`	A multimedia event fired by a `Rendition` object
`Events`	A collection of multimedia event objects
`MediaPlayer`	An instance of a multimedia player
`Marker`	A location representing a frame or time value in a media clip
`Markers`	Aall the markers in the currently loaded media clip
`MediaReject`	Contains error information when a `Rendition` object is rejected
`MediaSelection`	A media selection object used to create the `MediaSettings` object
`MediaSettings`	An object containing settings used to create a `MediaPlayer` object
`Monitor`	A display monitor used for playback
`Monitors`	An array of display monitors connected to the user's system
`PlayerInfo`	An available media player
`PlayerInfoList`	An array of `PlayerInfo` objects
`Rendition`	Contains information needed to play a media clip
`ScreenAnnot`	A display area used for media playback

Media Players Topics

Accessing a List of Active Players

Specifying Playback Settings

Accessing a List of Active Players

To obtain a list of available players, call the **app.media** object's **getPlayers** method, which accepts an optional parameter specifying the MIME type and returns a **PlayerInfoList** object. The **PlayerInfoList** object is an array of **PlayerInfo** objects that can be filtered using its **select** method.

The following code sample shows how to obtain a list of all available players:

```
var mp = app.media.getPlayers();
```

The following code sample shows how to obtain a list of all available MP3 players and prints them out to the console:

```
var mp = app.media.getPlayers("audio/MP3");
for (var i = 0; i < mp.length; i++) {
    console.println("\nmp[" + i + "] Properties");
    for (var p in mp[i])
        console.println(p + ": " + mp[i][p]);
}
```

To filter the list of players using the **PlayerInfoList** object's **select** method, you may supply an optional **object** parameter which may contain any combination of **id**, **name**, and **version** properties, each of which may be either a string or a regular expression. For example, the following code obtains the QuickTime media player:

```
var mp = app.media.getPlayers().select({id: /quicktime/i});
```

In addition, the **doc.media** object's **getOpenPlayers** method returns an array of all currently open **MediaPlayer** objects. With this array, you can stop or close all players, and manipulate any subset of the open players. The following example stops all running players in the document:

```
var players = doc.media.getOpenPlayers(oDoc);
for (var i in players)
    players[i].stop();
```

Specifying Playback Settings

You can use Acrobat JavaScript to obtain and adjust the media settings offered by a player. To do this, invoke the **Rendition** object's **getPlaySettings** method, which returns a **MediaSettings** object, as shown in the code below:

```
var settings = myRendition.getPlaySettings();
```

In additon to the **App.media** properties and methods, a **MediaSettings** object, which is used to create a **MediaPlayer** object, contains many properties related to the functional capabilities of players. These are described below in Table 8.2:

TABLE 8.2   MediaSettings Object Properties

Property	Description
autoPlay	Determines whether to play media clip automatically when the player is opened
baseURL	Used to resolve any relative URLs used in the media clip
bgColor	Specifies the background color for the media player window
bgOpacity	Specifies the background opacity for the media player window
endAt	Defines the ending time or frame for playback
data	The contents of the media clip (**MediaData** object)
duration	The number of seconds required for playback
floating	An object containing the location and size properties of a floating window used for playback
layout	A value indicating whether and how the content should be resized to fit the window
monitor	Defines the rectangle containing the display monitor used for playback
monitorType	The category of display monitor used for playback (such as primary, secondary, best color depth, etc)
page	The document page number used in case a docked media player is used
palindrome	Indicates that the media can play from beginning to end, and then in reverse from the end to the beginning
players	The list of available players for this rendition
rate	The playback speed

TABLE 8.2 *MediaSettings Object Properties*

Property	Description
repeat	The number of times the playback repeats
showUI	Indicates whether the media player controls will be visible
startAt	Defines the starting time or frame for playback
visible	Indicates whether the media player will be visible
volume	The playback volume
windowType	An enumeration obtained from **App.media.WindowType** indicating whether the media player window will be docked or floating

The example below illustrates how to customize the number of repetitions for playback:

```
// obtain the MediaSettings object, and store its repeat value
var nRepeat = event.action.rendition.getPlaySettings().repeat;

if (nRepeat == 1)
    nRepeat = 2;
else
    nRepeat = 1;

// set the new repeat value when opening the media player
var args = { settings: {repeat: nRepeat} };
app.media.openPlayer(args);
```

The example below illustrates how to play a docked media clip in a Screen annotation:

```
app.media.openPlayer({
    rendition: this.media.getRendition("myClip"),
    annot: this.media.getAnnot({
        nPage: 0,
        cAnnotTitle: "myScreen"
    }),
    settings: {
        windowType: app.media.WindowType.docked
    }
});
```

The next example illustrates how to play back the alternate text of a rendition in a floating window:

```
// set up the alternate text
var rendition = this.media.getRendition("myClip");
var settings = rendition.getPlaySettings();
var args = {
    settings: settings,
    showAltText: true,
    showEmptyAltText: true
};

var selection = rendition.select();
settings = app.media.getAltTextSettings(args, selection);
settings.data = "A.C. Robat");

// set up the floating window
settings.windowType = app.media.windowType.floating;
settings.floating = {
    canResize: app.media.canResize.keepRatio,
    hasClose: true,
    width: 400,
    height: 100
};

// play the alternate text in the floating window
args = {
    rendition: rendition,
    annot: this.media.getAnnot({
        nPage: 0,
        cAnnotTitle: "myScreen"
    }),
    settings: settings
};
app.media.openPlayer(args);
```

Monitors

The Acrobat JavaScript **Monitors** object is a read-only array of **Monitor** objects, each of which represents a display monitor connected to the user's system. It is available as a property of the **app** object, and you may write customized Acrobat JavaScript code to iterate through this array to obtain information about the available monitors and select one for a full-screen or popup media player.

It is possible to apply filtering criteria to select a monitor. For example, you can select the monitor with the best color, or if there are multiple instances, additionally select the monitor with the greatest color depth. These criteria are methods of the **Monitor** object, and are listed below in Table 8.3.

TABLE 8.3   *Monitors Filter Criteria Methods*

Method	Description
bestColor	Returns the monitors with the greatest color depth
bestFit	Returns the smallest monitors with minimum specified dimensions
desktop	Creates a new monitor representing the entire virtual desktop
document	Returns the monitors containing the greatest amount of the document
filter	Returns the monitors having the highest rank according to a ranking function supplied as a parameter
largest	Returns the monitors with the greatest area in pixels
leastOverlap	Returns the monitors overlapping the least with a given rectangle
mostOverlap	Returns the monitors overlapping the most with a given rectangle
nonDocument	Returns the monitors displaying the least amount (or none) of the document
primary	Rreturns the primary monitor
secondary	Returns all monitors except for the primary one
select	Returns monitors filtered by monitor type
tallest	Returns the monitors with the greatest height in pixels
widest	Returns the monitors with the greatest width in pixels

In addition to the capabilities within the **Monitors** object, the **Monitor** object provides the properties shown below in Table 8.4:

TABLE 8.4   *Monitor Object Properties*

Property	Description
colorDepth	The color depth of the monitor in bits per pixel
isPrimary	Returns **true** if the monitor is the primary one
rect	The boundaries of the monitor in virtual desktop coordinates
workRect	The monitor's workspace boundaries in virtual desktop coordinates

The example below illustrates how to obtain the primary monitor and check its color depth:

```
var monitors = app.monitors.primary();
if (monitors.length > 0)
    console.println("Color depth: " + monitors[0].colorDepth);
```

The next example llustrates how to obtain the monitor with the greatest color depth, with a minimum specified depth of 32:

```
var monitors = app.monitors.bestColor(32);
if (monitors.length > 0)
    console.println("Found the best color depth over 32!");
```

The next example llustrates how to obtain the monitor with the greatest width in pixels, and determines whether it is the primary monitor:

```
var monitors = app.monitors.widest();
var isIsNot = (monitors[0].isPrimary) ? " is " : " is not ";
console.println("Widest monitor" + isIsNot + "the primary.");
```

Integrating Media into Documents

You may use Acrobat JavaScript to integrate media into documents, which may be played in either *Screen Annot* objects or floating windows. The **doc.media** object is useful for accessing Screen Annot objects in which the media clips can be played.

Topics
Adding Movie Clips
Adding Sound Clips
Adding and Editing Renditions
Setting Multimedia Preferences

Adding Movie Clips

It is possible to embed a movie in a document or create a link to one. To add a movie clip to a document, you must first obtain a media player capable of playing it. Do this by invoking the **app.media** object's **getPlayers** method, which accepts an optional parameter specifying the MIME type. For example, the code below shows how to obtain a player that can play a movie:

```
// create the media player object
var player = app.media.createPlayer();

// filter for those that can play a movie (specify MIME type)
player.settings.players = app.media.getPlayers("video/x-mpg");

// choose which file to play
player.settings.data = "myClip.mpg";

// open the player
player.open();
```

Adding Sound Clips

The procedure for adding a sound clip is similar to that for movie clips. Specify the MIME type and the sound data to be played. In the example below, a sound clip is loaded from a URL and played in a floating window:

```
var myURLClip = "http://myWebSite.com/mySoundClip.mp3";
var args = {
    URL: myURLClip,
    mimeType: "audio/mp3",
    doc: this,
    settings: {
        players: app.media.getPlayers("audio/mp3"),
        windowType: app.media.windowType.floating,
        data: app.media.getURLData(myURLClip, "audio/mp3"),
        floating: {height: 400, width: 600}
    },
};
app.media.openPlayer(args);
```

Adding and Editing Renditions

A **rendition** object contains information needed to play a media clip, including embedded media data (or a URL), and playback settings, and corresponds to the **Rendition** in the Acrobat user interface. When you add a movie or sound clip to your document, a default rendition is listed in the **Multimedia Properties** dialog box and is assigned to a **Mouse Up** action. In case the rendition cannot be played, you may add other renditions or edit the existing ones.

If you add alternate versions of the media clip, these become new renditions that can serve as alternates in case the default choice cannot be played. It is then possible to invoke the **rendition** object's **select** method to obtain the available media players for each rendition.

There are several types of settings that can be specified for a given rendition: media settings, playback settings, playback location, system requirements, and playback requirements. You can use Acrobat JavaScript to customize some of these settings through the **rendition** object. There are several properties to which you have read-only access when editing a rendition. These are listed below in Table 8.5.

TABLE 8.5   *Rendition Object Properties*

Property	Description
altText	The alternate text string for the rendition
doc	The document that contains the rendition
fileName	Returns the filename or URL of an external media clip
type	A **MediaRendition** object or a rendition list
uiName	The name of the rendition

In addition to these properties, you may invoke the **rendition** object's **getPlaySettings** method, which returns a **MediaSettings** object. As you learned earlier in Specifying Playback Settings, you can adjust the settings through this object. You may also invoke its **testCriteria** method, with which you can test the rendition against any criteria specified in the PDF file, such as minimum bandwidth.

Setting Multimedia Preferences

In general, you may choose which media player should be used to play a given clip, determine whether the Player Finder dialog box is displayed, and set accessibility options for impaired users (these include subtitles, dubbed audio, or supplemental text captions).

In addition, you may use Acrobat JavaScript to access or customize multimedia preferences. For example, the **doc.media** object's **canPlay** property may be used to indicate whether multimedia playback is allowed for the document. The **MediaSettings** object's **bgColor** property can be used to specify the background color for the media player window. Examples of each are given below:

```
var canPlay = doc.media.canPlay;


if (canPlay.no) {
    // determine whether security settings prohibit playback:
    if (canPlay.no.security) {
        if (canPlay.canShowUI)
            app.alert("Security prohibits playback.");
        else
            console.println("Security prohibits playback.");
    }
}

// Set the background color to red:
settings.bgColor = ["RGB", 1, 0, 0];
```

9 Acrobat Templates

Introduction

This chapter will help you gain a greater depth of understanding of the purpose and usage of templates. You will understand the role of templates in PDF form structures, and the options and implications related to their usage. Finally, you will learn the details related to the proper usage of the parameters defined for the **template** object's methods.

Chapter Goals

At the end of this chapter, you will be able to:

- Understand the role of templates within the architecture of PDF forms.
- Understand the implications of spawning templates.
- Understand the details of how templates handle form fields in dynamic page generation.
- Specify the proper syntax and usage of all the parameters available in the **template** object's methods.
- Create templates and use them to dynamically create content in your documents.

Contents

Topics
The Role of Templates in PDF Form Architecture
Spawning Templates

The Role of Templates in PDF Form Architecture

Introduction

Acrobat JavaScript defines a **template** object that supports interactive form architectures. In this context, a **template** is a named page within a PDF document that provides a convenient format within which to automatically generate and manipulate a large number of form fields. These pages contain visibility settings, and can be used to spawn new pages containing identical sets of form controls to those defined within the template.

As you learned earlier, it is possible to use templates to dynamically generate new content within a document. Templates can be used to assure that your content is reusable, and can be used for replicating the logic you previously created.

A template is used to reproduce the logic on a given page at any new location in the document. This logic may include form fields such as text fields and buttons, digital signatures, and embedded logic such as JavaScripts that email form data to another user. To create a template based on a page, invoke the **doc** object's **createTemplate** method, in which you will name your template and specify the page from which it will be created. The code below creates a template called **myTemplate** based on page 5 of the current document:

```
this.createTemplate({cName: "myTemplate", nPage: 5});
```

There are two steps required to generate pages based on a template contained in the document:

1. Select a template from the **doc** object's **templates** property, which is an array of **template** objects.

2. Spawn a page invoking the **template** object's **spawn** method.

The following code adds a new page at the end of the current document that is based on the first template contained in the document:

```
var myTemplateArray = this.templates;
var myTemplate = myTemplateArray[0];
myTemplate.spawn(this.numPages, false, false);
```

Spawning Templates

Dynamic Form Field Generation

When spawning templates, you may specify whether the form fields are renamed on the new page or retain the same names as those specified in the template. This is done through the optional **bRename** parameter. If you set the parameter's value to **true**, the form fields on each spawned page have unique names, and values entered into any of those fields do not affect values in their counterparts on other pages. This would be useful, for example, in forms containing expense report items. If you set the parameter's value to **false**, the form fields on each spawned page have the same name as their counterparts on all the other spawned pages. This might be useful if you would like, for example, to duplicate a button or display the date on every page, since entering it once results in its duplication throughout all the spawned pages.

Suppose the **bRename** parameter is **true** and the field name on the template is **myField**. If the template is named **myTemplate** and is spawned onto page 4, the new corresponding field name is **P.4.myTemplate.myField**. The page number embedded in the new field guarantees its uniqueness.

Dynamic Page Generation

When templates are used to spawn new pages, those pages contain an identical set of form fields to those defined in the template. Depending on the parameters used, this process may result in a size inflation problem. This is because there are two ways to specify page generation: one option is to repeatedly spawn the same page which results in the duplication of **XObject** objects (external graphics objects), and the other is to generate page contents as **XObject** objects, which only requires that those objects be repositioned.

The **nPage** parameter is used to specify the zero-based index of the page number used to create the page. If the **bOverlay** value is set to **true**, the new page overlays onto the page number specified in the **nPage** parameter. If the **bOverlay** value is set to **false**, the new page is inserted as a new page before the specified page. To append a page at the end of the document, set the **bOverlay** value to false and the **nPage** parameter to the total number of pages in the document.

Template Syntax and Usage

In this first example, all the templates will be spawned once, the field names will not be unique in each resultant page (**bRename** will be **false**), and the resultant pages will be appended to the end of the document (**bOverlay** will be **false**). The **oXObject** parameter will be used to prevent size inflation:

```
// Obtain the collection of templates:
var t = this.templates;

// Spawn each template once as a page appended at the end:
for (var i = 0; i < t.length; i++)
    t[i].spawn(this.numPages, false, false);
```

In this next example, the same template will be spawned 10 times, will overlay onto pages 0 through 9 (**bOverlay** will be **true**), and the field names will be unique on each page (**bRename** will be **true**):

```
// Obtain the template:
var t = this.templates;
var T = t[0];

// Prevent file size inflation by using the XObject. Do this by
// spawning once, saving the result (an XObject), and passing
// the resultant XObject to the oXObject parameter in
// the subsequent calls to the spawn method:
var XO = T.spawn(0, true, true);
for (var i = 1; i < 10; i++)
    T.spawn(i, true, true, XO);
```

In this next example, we will retrieve the template named **myTemplate**, overlay it onto pages 5 through 10 (**bOverlay** will be **true**), and use the same field names on each page (**bRename** will be **false**):

```
// Obtain the template name "myTemplate":
var t = this.getTemplate("myTemplate");

// Prevent file size inflation:
var XO = t.spawn(5, true, false);

// Spawn the remaining pages:
for (var i = 6; i <= 10; i++)
    t.spawn(i, true, false, XO);
```

10   Modifying the User Interface

Introduction

This chapter will provide you with an in-depth understanding of the ways in which you may present and modify the user interface. You will learn how to use Acrobat JavaScript to access the Adobe Dialog Manager (ADM), customize navigation in PDF documents, customize PDF layers, and manage print production.

Chapter Goals

At the end of this chapter, you will be able to:

● Create modal dialogs through Acrobat JavaScript access to ADM.

● Add and customize navigation in PDF documents.

● Customize the properties and behavior or PDF layers.

● Understand how to manage print production of PDF documents.

Contents

Topics
Using Adobe Dialog Manager (ADM) in Acrobat JavaScript
Adding Navigation to PDF Documents
Working with PDF Layers

Using Adobe Dialog Manager (ADM) in Acrobat JavaScript

Introduction

The Adobe Dialog Manager (ADM) is a cross-platform API for implementing dialog-based user interfaces in a uniform way for Adobe applications such as Acrobat, Photoshop, Illustrator, and After Effects. Acrobat JavaScript provides a convenient interface to ADM through which you may implement modal dialogs containing list objects and other controls normally included in graphical user interfaces, including buttons, text, text boxes, check boxes, radio buttons, progress bars, scroll bars, and sliders. These are summarized below in Table 10.1.

TABLE 10.1   *ADM Dialog Elements*

Element Type	Description
`button`	push button
`check_box`	check box
`radio`	radio button
`list_box`	list box
`hier_list_box`	hierarchical list box
`static_text`	static text box
`edit_text`	editable text box
`popup`	popup control
`ok`	OK button
`ok_cancel`	OK and Cancel button
`ok_cancel_other`	OK, Cancel, and Other button
`view`	container for a set of controls
`cluster`	frame for a set of controls
`gap`	place holder

ADM Object Hierarchy

An ADM user interface consists of dialog objects and the items they contain. All items have properties and events that determine their default behavior and allow them to be modified or extended. Thus, when an ADM dialog object is created, it contains a series of methods and utilities. There are three categories of ADM dialog items:

● **ADM Hierarchy List**
  – Hierarchy List Box

● **ADM List**
  – List Box
  – Popup List
  – Popup Menu
  – Scrolling Popup List
  – Spin Edit Popup
  – Spin Edit Scrolling Popup
  – Text Edit Popup
  – Text Edit Scrolling Popup

● **non-list items**
  – Frame
  – Picture Push Button
  – Picture Radio Button
  – Picture Static
  – Picture Check Box
  – Resize
  – Scrollbar
  – Slider
  – Spin Edit
  – Text Check Box
  – Text Edit
  – Text Edit Read-only
  – Text Edit MultiLine
  – Text Edit MultiLine Read-only
  – Text Push Button
  – Text Radio Button
  – Text Static
  – Text Static MultiLine
  – Progress Bar
  – Chasing Arrows
  – Dial
  – ItemGroup
  – Popup Control

- – Popup Control Button
- – Popup Spin Edit Control
- – Password Text Edit

Access to ADM through JavaScript

The **app** object's **execDialog** method is the Acrobat JavaScript method used to execute ADM dialogs. The method requires a single parameter: an object literal known as a *dialog descriptor*, which contains properties and method handlers for all the dialog elements.

An instance of the **Dialog** object is passed to the method handlers. The **Dialog** object has two important methods: the **load** method is used to initialize dialog elements, and the **store** method retrieves values stored in dialog elements.

The dialog handlers are described below in Table 10.2:

TABLE 10.2   *Dialog Handlers*

Handler	Description
initialize	Method that runs when the dialog is created
validate	Method called when validating field values
commit	Method called when OK button is selected
destroy	Method that runs when the dialog is destroyed
itemID	Method called when an element's **itemID** property is modified

A dialog descriptor is a tree consisting of a root and child nodes. The root is an object literal containing the properties shown below in Table 10.4, and each child node is a dialog element containing the properties shown below in Table 10.5. In addition to the standard properties described in Table 10.5 for dialog elements, there are sets of properties available for the **static_text**, **edit_text**, **ok**, **ok_cancel**, and **ok_cancel_other** elements. These are listed below in Table 10.3:

TABLE 10.3   *Dialog Descriptor Element Properties*

Element	Properties
`static_text`	`multiline`
`edit_text`	`multiline`, `readonly`, `password`, `numeric`, `PopupEdit`, `SpinEdit`
`ok`	`ok_name`, `cancel_name`, `other_name`
`ok_cancel`	`ok_name`, `cancel_name`, `other_name`
`ok_cancel_other`	`ok_name`, `cancel_name`, `other_name`

TABLE 10.4   *Root Node of Dialog Descriptor*

Property	Description
`name`	Title bar
`target_id`	`itemID` value for the active dialog item when the dialog is created
`first_tab`	`itemID` value for the dialog item first in tab order
`width`	Width of the dialog in pixels
`height`	Height of the dialog in pixels
`char_width`	Width of the dialog in characters
`char_height`	Height of the dialog in characters
`align_children`	Alignment for all descendants
`elements`	Array of elements contained in the dialog

TABLE 10.5   *Dialog Element Object Literal*

Property	Description
`name`	Displayed name of dialog element
`item_id`	`itemID` value for the dialog
`type`	Dialog element type
`next_tab`	`itemID` value for the next dialog item in the tab order
`width`	Width of the element in pixels
`height`	Height of the element in pixels
`char_width`	Width of the element in characters
`char_height`	Height of the element in characters
`font`	The element's font: **default**, **dialog**, or **palette**
`bold`	Bold font
`italic`	Italic font
`alignment`	Alignment for the element
`align_children`	Alignment for all descendents
`elements`	Array of elements contained in this element

The following example creates the modal dialog as shown below in Figure 10.1:

FIGURE 10.1   ADM Dialog with Item Cluster
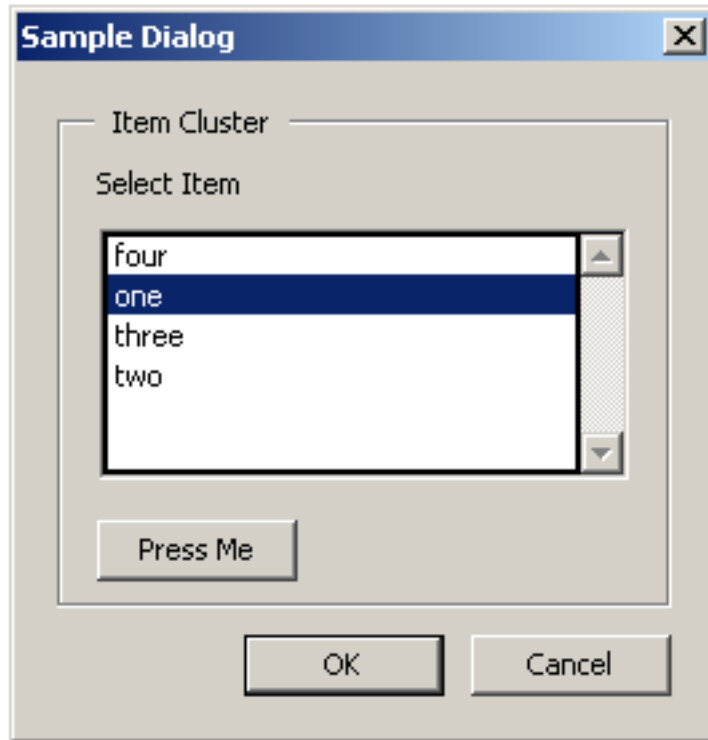
To create a dialog such as the one above in Figure 10.1, first create an object literal containing a dialog descriptor, and pass it as a parameter to the **app** object's **execDialog** method. The object literal may be comprised of one or more of the handlers described in Table 10.2, a root dialog element, and a hierarchy of dialog items as children of the root node.

We will begin by creating the handlers for the dialog descriptor. The **initialize** handler is called as the dialog is created, and the **commit** handler is called when **OK** is selected. The **initialize** handler sets up the list tems, and the **commit** handler processes those items.

The following **initialize** handler initializes the items used in the **list_box**. Note that it receives the **dialog** object as a parameter:

```
// Event handler to initialize the list_box
initialize: function(dialog)
{
    // Create a list_box named sub1 and set up 4 choices:
    dialog.load({
    "sub1":
    {
        "one": -1,
        "two": -2,
        "three": +3,
        "four": -4
    }
    });
}
```

The following **commit** handler is called when **OK** is clicked, retrieves the values of the items used in the **list_box**, and can take action based on those values. Note that it receives the **dialog** object as a parameter:

```
commit: function(dialog)
{
    // Retrieve the values stored in list_box sub1:
    var elements = dialog.store()["sub1"];

    // Iterate through items and take actions as needed
    for (var e in elements)
    {
        // perform action
    }
}
```

It is also possible to write event handlers for specific items selected in the dialog. The following handler is called when the button labeled **butn** is clicked:

```
"butn": function(dialog)
{
    app.alert("butn was clicked.");
}
```

Finally, to create the root node, we must create the **description** object literal, which is the **dialog** descriptor. This is shown in the complete sample code shown below:

```
var myDialog =
{
    initialize: function(dialog)
    {
        // Set up and store the values in list_box sub1:
        dialog.load({
        "sub1":
        {
            // Note: positive value represents the selected item
            "one": -1,
            "two": -2,
            "three": +3, // currently selected item
            "four": -4
        }
        });
    },
    commit: function(dialog)
    {
        // Retrieve the values stored in list_box sub1:
        var elements = dialog.store();

        // Iterate through items and take actions as needed
        for (var e in elements["sub1"])

            // If the value is positive, it was selected:
            if (elements["sub1"][e] > 0)
            {
                // display the list value selected:
                app.alert("You chose:\n" + e);

                // call a related function for the selection
                rc = elements["sub1"][e];
                ListHandler(rc);
            }
    },

    // event handler for butn button clicks
    "butn": function(dialog)
    {
        app.alert("JavaScript ADM Dialog in Acrobat 7.");
    },
```

```
// Dialog object descriptor (root node)
description:
{
    name: "Sample Dialog",
    elements:
    [
    {
        type: "view",
        align_children: "align_left",
        elements:
        [
            {
                type: "cluster",
                name: "Item Cluster",
                elements:
                [
                    {
                        type: "static_text",
                        name: "Select Item",
                        font: "default"
                    },
                    {
                        type: "list_box",
                        item_id: "sub1",
                        width: 200,
                        height: 100
                    },
                    {
                        type: "button",
                        item_id: "butn",
                        name: "Press Me"
                    }
                ]
            },
            {
                type: "ok_cancel"
            }
        ]
    }
    ]
};
```

```
// Function to handle the user's list selection:
function ListHandler(rc)
{
    switch (rc) {
        case 1:
            app.alert("ListHandler response for 1 - one");
            break;
        case 2:
            app.alert("ListHandler response for 2 - two");
            break;
        case 3:
            app.alert("ListHandler response for 3 - three");
            break;
        case 4:
            app.alert("ListHandler response for 4 - four");
            break;
        default:
            app.alert("Invalid selection");
            break;
    }
}

rc = app.execDialog(myDialog);
```

Adding Navigation to PDF Documents

Acrobat JavaScript provides a number of constructs that enable you to add and customize navigation features within PDF documents. These features make it convenient for the user to see and visit areas of interest within the document, and you may associate a variety of actions with navigation events. In addition, you may customize the appearance of your form fields and pages, manipulate multiple documents, add and delete pages, and add headers, footers, watermarks, backgrounds, and buttons.

Topics
Thumbnails
Bookmarks
Links
Using Actions for Special Effects
Highlighting Form Fields and Navigational Components
Setting Up a Presentation
Numbering Pages
Creating Buttons

Thumbnails

Creating Page Thumbnails

The **doc** object provides methods for adding and removing thumbnails in a document. To add a set of thumbnails, invoke the **dialog** object's **addThumbnails** method, which creates thumbnails for a specified set of pages in the document. It accepts two optional parameters: **nStart** and **nEnd** represent the beginning and end of an inclusive range of page numbers.

For example, to add thumbnails for pages 2 through 5, use the following command:

```
this.addThumbnails({nStart: 2, nEnd: 5});
```

To add a thumbnail for just one page, just provide a value for **nStart**. The following example adds a thumbnail for page 7:

```
this.addThumbnails({nStart: 7});
```

To add thumbnails from page 0 to a specified page, just provide a value for **nEnd**. The following example adds thumbnails for pages 0-7:

```
this.addThumbnails({nEnd: 7});
```

To add thumbnails for all the pages in the document, omit both parameters:

```
this.addThumbnails();
```

To remove a set of thumbnails, invoke the **doc** object's **removeThumbnails** method, which accepts the same parameters as the **addThumbnails** method. For example, to remove pages 2 to 5, use the following code:

```
this.removeThumbnails({nStart: 2, nEnd: 5});
```

Adding Page Actions with Page Thumbnails

You may associate a **Page Open** event with a page thumbnail. The most straightforward way of doing this is to specify a **Page Open** action in the **Page Properties** dialog.

To customize a page action with Acrobat JavaScript, invoke the **doc** object's **setPageAction** method for the page to be opened. In the following example, a greeting is displayed when the user clicks on the thumbnail for page 2:

```
this.setPageAction(2, "Open", "app.alert('Hello');");
```

The advantage of this approach is that you can dynamically build JavaScript strings to be used in the method call.

Bookmarks

You can use Acrobat JavaScript to customize the appearance and behavior of the bookmarks that appear in the **Bookmarks** navigation panel. Every PDF document has an object known as **bookmarkRoot**, which is the root of the bookmark tree for the document. It is possible to recursively add and modify levels of bookmarks underneath the root. Each node is a **bookmark** object which can have any number of children.

Acrobat JavaScript makes the **bookmarkRoot** object available as a property of the **doc** object. This root node contains a property called **children**, which is an array of **bookmark** objects. The **bookmark** object has the properties shown below in Table 10.6, and the methods shown below in Table 10.7:

TABLE 10.6   *Bookmark Properties*

Property	Description
children	Returns the array of child objects for the current node
color	Specifies the color for the bookmark
doc	The document object for the bookmark
name	The text string appearing in the navigational panel
open	Determines if children are shown
parent	The parent bookmark
style	Font style

TABLE 10.7   *Bookmark Methods*

Method	Description
createChild	Creates a new child bookmark
execute	Executes the **Mouse Up** action for the bookmark
insertChild	Inserts a bookmark as a new child for this bookmark (this may be used to move existing bookmarks)
remove	Removes the bookmark and all its children
setAction	Sets a **Mouse Up** action for the bookmark

Creating Bookmarks

To create a bookmark, it is necessary to navigate through the bookmark tree and identify the parent of the new node. Begin by accessing the **bookmarkRoot**, which is a property of the current document representing the top node in the bookmark tree:

```
var myRoot = this.bookmarkRoot;
```

Assume there are no bookmarks in the document. To create a new bookmark, invoke the **bookmarkRoot** object's **createChild** method to which you may submit the following parameters: **cName** (the name to appear in the navigational panel), **cExpr** (an optional JavaScript to be executed when the bookmark is clicked), and **nIndex** (an optional zero-based index into the **children** array).

The following code creates a bookmark that displays a greeting when clicked. Note that the omission of the **nIndex** value means that it is placed at position 0 in the **children** array:

```
myRoot.createChild("myBookmark", "app.alert('Hello!');");
```

The following code adds a bookmark called **grandChild** as a child of **myBookmark**, :

```
var current = myRoot.children[0];
current.createChild("grandChild");
```

Suppose that you would like to move **grandChild** so that it becomes a child of the root. Invoke the root bookmark's **insertChild** method, and provide a reference to **grandChild** as a parameter:

```
var grandChild = myRoot.children[0].children[0];
myRoot.insertChild(grandChild, 1);
```

Managing Bookmarks

You can use Acrobat JavaScript to change the **name**, **color**, and **style** properties of a bookmark. Note that the **style** property is an integer: **0** means normal, **1** means italic, **2** means bold, and **3** means bold-italic. The code below changes the name to **New Name**, the color to red, and the font style to bold:

```
var myRoot = this.bookmarkRoot;
var myChild = myRoot.children[0];
myChild.name = "New Name";
myChild.color = color.red;
myChild.style = 2;
```

In addition to adding new or existing bookmarks as you learned in Creating Bookmarks, you may also delete a bookmark and its children by invoking its **remove** method. The following line of code removes all bookmarks from the document:

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

Creating a Bookmark Hierarchy

Because of the tree structure associated with bookmarks, it is possible to construct a hierarchy of bookmarks; a child of a bookmark represents a subsection of the section represented by that bookmark. To create a hierarchy, first add bookmarks to the root, then to the children of the root, and recursively to their children.

The following code creates bookmarks **A**, **B**, **C**. Each section has 3 children. Child **A** has children **A0**, **A1**, and **A2**. Child **B** has children **B0**, **B1**, and **B2**. Child **C** has children **C0**, **C1**, and **C2**:

```
var myRoot = this.bookmarkRoot;
myRoot.createChild("A");
myRoot.createChild({cName: "B", nIndex: 1});
myRoot.createChild({cName: "C", nIndex: 2});
for (var i = 0; i < myRoot.children.length; i++) {
    var child = myRoot.children[i];
    for (var j = 0; j < 3; j++) {
        var name = child.name + j;
        child.createChild({cName: name, nIndex: j});
    }
}
```

To print out the hierarchy to the console, you can keep track of levels as shown in the following code. Note its recursive nature:

```
function DumpBookmark(bm, nLevel){
    // build indents to illustrate the level
    var s = "";
    for (var i = 0; i < nLevel; i++) s += " ";

    // print out the bookmark's name:
    console.println(s + "+-" + bm.name);

    // recursively print out the bookmark's children:
    if (bm.children != null)
        for (var i = 0; i < bm.children.length; i++)
            DumpBookmark(bm.children[i], nLevel+1);
}

// open the console to begin:
console.clear(); console.show();

// recursively print out the bookmark tree
DumpBookmark(this.bookmarkRoot, 0);
```

Links

Acrobat JavaScript provides support for the addition, customization, or removal of links within PDF documents. These links may be used to access URLs, file attachments, or destinations within the document.

The **doc** object contains methods for adding, retrieving, and removing links. These include the methods listed below in Table 10.8. This is used in conjunction with the **link** object, which contains properties as well as a **setAction** method for customizing the appearance and behavior of a given link. Its properties are listed below in Table 10.9.

In addition, the **app** object contains a property called **openInPlace**, which can be used to specify whether cross-document links are opened in the same window or in a new one.

TABLE 10.8   *Doc Object Link Methods*

Method	Description
addLink	Adds a new link to a page
addWeblinks	Converts text instances to Web links with URL actions
getLinks	Retrieves the links within a specified area on a page
getURL	Opens a web page
gotoNamedDest	Goes to a named destination within the document
removeLinks	Removes the links within a specifed area on a page
removeWeblinks	Removes Web links created with the Acrobat user interface

TABLE 10.9   *Link Properties*

Property	Description
borderColor	The border color of the bounding rectangle
borderWidth	The border width of the surrounding rectangle
highlightMode	The visual effect when the user clicks the link
rect	The rotated user space coordinates of the link

Creating Links

If a PDF document contains text beginning with **http://** such as **http://myURL.com**, you may convert all such instances to links with URL actions by invoking the **doc** object's **addWeblinks** method. The method returns an integer representing the number of text instances converted, as shown in the code below:

```
var numberOfLinks = this.addWeblinks();
console.println("Converted " + numberOfLinks + " links.");
```

To add a single link to a PDF document, first invoke the **doc** object's **addLink** method, and then customize the returned **link** object's properties. The **addLink** method requires two parameters: the page number and the coordinates, in rotated user space, of the bounding rectangle. In the following example, navigational links are added to the lower left and right corners of each page in the document. The left link opens the previous page, and the right link opens the next page:

```
var linkWidth = 36, linkHeight = 18;
for (var i = 0; i < this.numPages; i++)
{
    // Create the coordinates for the left link:
    var lRect = [0, linkHeight, linkWidth, 0];

    // Create the coordinates for the right link:
    var cropBox = this.getPageBox("Crop", i);
    var offset = cropBox[2] - cropBox[0] - linkWidth;
    var rRect = [offset, linkHeight, linkWidth + offset, 0];

    // Create the Link objects:
    var leftLink = this.addLink(i, lRect);
    var rightLink = this.addLink(i, rRect);

    // Calculate the previous and next page numbers:
    var nextPage = (i + 1) % this.numPages;
    var prevPage = i - 1;
    if (prevPage < 0) prevPage = this.numPages - 1;

    // Set the link actions to go to the pages:
    leftLink.setAction("this.pageNum = " + prevPage);
    rightLink.setAction("this.pageNum = " + nextPage);

    // Customize the link appearance:
    leftLink.borderColor = color.red;
    leftLink.borderWidth = 1;
    rightLink.borderColor = color.red;
    rightLink.borderWidth = 1;
}
```

Defining the Appearance of a Link

The previous example contained code that set the appearance of the bounding rectangle for the links through their **borderColor** and **borderWidth** properties. You may also specify how the link will appear when it is clicked by setting its **highlightMode** property to one of four values: **None**, **Outline**, **Invert** (the default), or **Push**.

For example, the following code sets the border color to blue, the border thickness to 2, and the highlight mode to **Outline** for **myLink**:

```
myLink.borderColor = color.blue;
myLink.borderWidth = 2;
myLink.highlightMode = "Outline";
```

Editing Links

In addition to adding links and modifying their appearance, you may also remove links from a document. To remove a known link object from a given page, retrieve its bounding rectangle coordinates and invoke the **doc** object's **removeLinks** method. In the following example, **myLink** is removed from page 2 of the document:

```
var linkRect = myLink.rect;
this.removeLinks(2, linkRect);
```

To remove all links from the document, simply use the crop box for each page, as shown in the code below:

```
for (var page = 0; page < this.numPages; page++)
{
    var box = this.getPageBox("Crop", page);
    this.removeLinks(page, box);
}
```

Creating Links from URLs

To open a web page for a given link, invoke the **link** object's **setAction** method, and pass in a script containing a call to the **doc** object's **getURL** method.

For example, suppose you have created a **link** object named **myLink**. The following code opens **http://myWebPage.com**:

```
myLink.setAction("this.getURL('http://myWebPage.com')");
```

Linking to File Attachments

To open a file attachment, embed a JavaScript in the call to the **link** object's **setAction** method. The script contains a call to the **app** object's **openDoc** method.

The following example opens myDoc.pdf when **myLink** is clicked:

```
myLink.setAction("app.openDoc('/C/myDoc.pdf');");
```

Removing Web Links

To remove Web links that were authored in Acrobat, invoke the **doc** object's **removeWeblinks** method. It accepts two optional parameters: **nStart** and **nEnd** represent the beginning and end of an inclusive range of page numbers. The following examples illustrate how to remove Web links from different page ranges in the document:

```
// remove the Web links from pages 2 through 5:
this.removeWeblinks({nStart: 2, nEnd: 5});

// remove the Web links from page 7
this.removeWeblinks({nStart: 7});

// remove the Web links from pages 0 through 7:
this.removeWeblinks({nEnd: 7});

// remove all the Web links in the document:
this.removeWeblinks();
```

Using Destinations

To go to a named destination within a document, embed a JavaScript in the call to the **link** object's **setAction** method. The script contains a call to the **doc** object's **gotoNamedDest** method.

The following example goes to the destination named as **Chapter5** in the current document when **myLink** is clicked:

```
myLink.setAction("this.gotoNamedDest('Chapter5');");
```

Using Actions for Special Effects

Thumbnails, bookmarks, links, and other objects have actions associated with them, and you may use Acrobat JavaScript to customize their behavior. For example, you can use them to display messages, jump to destinations in the same document or any other, open attachments, open Web pages, execute menu commands, or perform a variety of other tasks.

Adding Actions

As you learned earlier, you may associate a thumbnail with a **Page Open** event, and associate bookmarks and links with **Mouse Up** events.

You may use Acrobat JavaScript to customize the actions associated with a thumbnail by invoking the **doc** object's **setPageAction** method. To customize the actions associated with bookmarks and links, create a string containing Acrobat JavaScript code and pass it to the object's **setAction** method. In the examples shown below, a greeting is displayed when a thumbnail, bookmark, and link are clicked:

```
// Open action for thumbnail:
this.setPageAction(2, "Open", "app.alert('Hello!');");

// MouseUp actions for bookmark and link:
myBookmark.setAction("app.alert('Hello!');");
myLink.setAction("app.alert('Hello!');");
```

Action Types

The actions applied to navigational components represent a small portion of the possible events that may be customized. A complete list of event types and the actions with which they are associated is shown below in Table 10.10:

TABLE 10.10   Action Types and Associated Events

Action Type	Event Names
`App`	`Init`
`Batch`	`Exec`
`Bookmark`	`Mouse Up`
`Console`	`Exec`
`Doc`	`DidPrint`, `DidSave`, `Open`, `WillClose`, `WillPrint`, `WillSave`
`External`	`Exec`
`Field`	`Blur`, `Calculate`, `Focus`, `Format`, `Keystroke`, `Mouse Down`, `Mouse Enter`, `Mouse Exit`, `Mouse Up`, `Validate`
`Link`	`Mouse Up`
`Menu`	`Exec`
`Page`	`Open`, `Close`
`Screen`	`InView`, `OutView`, `Open`, `Close`, `Focus`, `Blur`, `Mouse Up`, `Mouse Down`, `Mouse Enter`, `Mouse Exit`

Type of Triggers

The event names shown above in Table 10.10 represent the types of triggers that initiate the actions associated with events. They are described below in Table 10.11.

TABLE 10.11 Trigger Descriptions

Trigger	Description
`Init`	Acrobat or Adobe Reader starts
`Exec`	Batch sequence starts
`DidPrint`	After document has printed
`DidSave`	After document has saved
`Open`	When the document is opened
`Close`	New page is opened or document is closed
`WillClose`	Just before the document is closed
`WillPrint`	Just before the document is printed
`WillSave`	Just before the document is saved
`Blur`	Just as the field loses focus
`Calculate`	A calculation is required for a field
`Focus`	After **Mouse Down** and before **Mouse Up**
`Format`	After dependent `Calculate` events occurs
`Keystroke`	Keystroke in textbox or combobox, or item is selected in combobox or listbox
Mouse Down	Mouse button is down
Mouse Up	Mouse button has been released
Mouse Enter	Mouse enters a field or screen rectangle
Mouse Exit	Mouse exits a field or screen rectangle
`Validate`	After value has been committed to a field
`InView`	New page comes into view
`OutView`	Old page leaves view

Highlighting Form Fields and Navigational Components

You can use Acrobat JavaScript to customize the actions associated with buttons, links, and bookmarks so that they change their appearance after the user has clicked on them.

For a button, which is a field, you can invoke its **highlight** property, which allows you to specify how the button appears once it has been clicked. There are four choices, as shown below in Table 10.12:

TABLE 10.12   Button Appearance

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

For example, the following code makes the button appear pushed when clicked:

```
// set the highlight mode to push
var f = this.getField("myButton");
f.highlight = highlight.p;
```

As you learned earlier, the **link** object also has a **highlight** property.

There are other ways in which you can creatively address the issue of highlighting. For example, you can change the background color of the button when clicked, by including a line of code in the script passed into its **setAction** method. In the following example, the button displays a greeting and changes its background color to blue when the mouse enters its area:

```
var script = "app.alert('Hello!');";
script += "var myButton = this.getField('myButton');";
script += "myButton.fillColor = color.blue;";
f.setAction("MouseEnter", script);
```

This idea can be applied to the **bookMark** object's **color** property, as well as the **link** object's **borderColor** property. In both cases, similar code to that shown in the example above can be used in the scripts passed into their **setAction** methods.

For **bookMark** objects, you may additionally consider changing the text or font style through its **name** and **style** properties. For example, the following code adds the word **VISITED** to **myBookmark** and changes the font style to bold:

```
myBookmark.name += " - VISITED");
myBookmark.style = 2;
```

Setting Up a Presentation

There are two viewing modes for Acrobat and Adobe Reader: full screen mode and regular viewing mode. Full screen mode is often appropriate for presentations, since PDF pages can fill the entire screen with the menu bar, toolbar, and window controls hidden.

It is possible to use Acrobat JavaScript to customize the viewing mode when setting up presentations. The **app** object's **fullScreen** property, as well as the **app.media** object's **windowType** property may be used to set the viewing mode.

Defining the Initial View in Full Screen View

To cause Acrobat and Adobe Reader to display in full screen mode, you may include the following statement in a document-level script triggered when the document is opened, or in an application-level script triggered when the application is first started:

```
app.fullscreen = true;
```

Defining an Initial View

In addition to specifying whether the full screen or regular viewing mode will be used, you may also use Acrobat JavaScript to set up the document view. You can customize the initial view in terms of magnification, page layout, application and document viewing dimensions, the initial page to which the document opens, and whether parts of the user interface will be visible.

The **doc** object's **layout** property allows you to specify page layout by assigning one of the following values:

- **SinglePage**
- **OneColumn**
- **TwoColumnLeft**
- **TwoColumnRight**
- **TwoPageLeft**
- **TwoPageRight**

To set up the dimensions of the various view windows, assign a **rect** value to one of the following **doc** object properties:

- **innerAppWindowRect**: the inner application window (excludes title bar, border, etc)
- **innerDocWindowRect**: the inner document window
- **outerAppWindowRect**: the outer application window
- **outerDocWindowRect**: the outer document window
- **pageWindowRect**: the page view window for the document content

To set up the magnification, assign a value to the **doc** object's **zoom** property. For example, the following code sets up a magnification of 125%:

```
this.zoom = 125;
```

You can also set the zoom type by assigning one of the settings, shown below in Table 10.13, to the **doc** object's **zoomtype** property:

TABLE 10.13   ZoomType Settings

Zoom Type	Property Value
`NoVary`	`zoomtype.none`
`FitPage`	`zoomtype.fitP`
`FitWidth`	`zoomtype.fitW`
`FitHeight`	`zoomtype.fitH`
`FitVisibleWidth`	`zoomtype.fitV`
`Preferred`	`zoomtype.pref`
`ReflowWidth`	`zoomtype.refW`

The following example sets the zoom type of the document to fit the width:

```
this.zoomType = zoomtype.fitW;
```

To specify the page to which the document initially opens, set the **doc** object's **pageNum** property. If the following code is included in the script used in the document **Open** event, the document automatically opens to page 30:

```
this.pageNum = 30;
```

Finally, you may choose whether menu items and toolbar buttons will be visible by invoking the following methods of the **app** object:

- **hideMenuItem**: removes a specific menu item
- **hideToolbarButton**: removes a specific toolbar button

For example, if the following code is placed in a folder-level script, the "Hand" icon is removed when Acrobat or Adobe Reader is started:

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

Adding Page Transitions

You may use Acrobat JavaScript to customize how page transitions occur for any pages within a document. This is accomplished through the **doc** object's **setPageTransitions** and **getPageTransitions** methods.

The **setPageTransitions** method accepts three parameters:

- **nStart**: the zero-based index of the beginning page
- **nEnd**: the zero-based index of the last page
- **aTrans**: a page transition array containing three values:
    - **nDuration**: the time a page is displayed before automatically changing
    - **cTransition**: the name of the transition to be applied
    - **nTransDuration**: the duration in seconds of the transition effect

The name of the transition to be applied can be chosen from a comprehensive list made available through the **fullscreen** object's **transitions** property. To obtain the list, type the following code into the **Console**:

```
console.println("[" + app.fs.transitions + "]");
```

In addition, you may set up a default page transition through the **fullscreen** object's **defaultTransition** property.

In the following example, page transitions are applied to pages 2-5. Each page displays for 10 seconds, and then an automatic transition occurs for one second:

```
this.setPageTransitions({
    nStart: 2,
    nEnd: 5,
    aTrans: {
        nDuration: 10,
        cTransition: "WipeLeft",
        nTransDuration: 1
    }
});

// Set the viewing mode to full screen
app.fullScreen = true;
```

Numbering Pages

You may use Acrobat JavaScript to customize the page numbering schemes used throughout a document. There are three numbering formats:

● decimal (often used for normal page ranges)

● roman (often used for front matter such as a preface)

● alphabetic (often used for back matter such as appendices)

The **doc** object's **getPageLabel** and **setPageLabels** methods can be used to control and customize the appearance of numbering schemes within a PDF document.

The **getPageLabel** method accepts the zero-based page index and returns a string containing the label for a given page.

The **setPageLabels** method accepts two parameters: **nPage** is the zero-based index for the page to be labeled, and **aLabel** is an array of three values representing the numbering scheme. If **aLabel** is not supplied, the method removes page numbering for the specified page and any others up to the next specified label.

The **aLabel** array contains three required values:

● **cStyle**: the style of page numbering as shown below in

● **cPrefox**: the string used to prefix the numeric portion of the page label

● **nStart**: the ordinal with which to start numbering the pages

TABLE 10.14   *Page Numbering Style Values*

cStyle value	Description
D	Decimal numbering
R	Upper case Roman numbering
r	Lower case Roman numbering
A	Upper case alphabetic numbering
a	Lower case alphabetic numbering

For example, the code shown below labels 10 pages within a document using the following scheme: **i, ii, ii, 1, 2, 3, 4, 5, Appendix-A, Appendix-B**:

```
// Pages 0-2 will have lower case roman numerals i, ii, iii:
this.setPageLabels(0, ["r", "", 1]);

// Pages 3-7 will have decimal numbering 1-5:
this.setPageLabels(3, ["D", "", 1]);

// Pages 8-9 will have alphabetic numbering:
this.setPageLabels(8, ["A", "Appendix-", 1]);

// The page labels will be printed to the console:
var labels = this.getPageLabel(0);
for (var i=1; i<this.numPages; i++)
    labels += ", " + this.getPageLabel(i);
console.println(labels);
```

It is also possible to remove a page label by omitting the **aLabel** parameter, as shown in the code below (which assumes the existence of the labels in the previous example:

```
// The labels for pages 3-7 will be removed:
this.setPageLabels(3);
```

Creating Buttons

Though buttons are normally considered form fields, you can add them to any document. A button may be used for a variety of purposes, such as opening files, playing sound or movie clips, or submitting data to a web server. As you learned earlier, you can place text and images on a button, making it a user-friendly interactive portion of your document. To show or hide portions of graphic buttons, use the **Mouse Enter** and **Mouse Exit** events or other types of control mechanisms to manage the usage of the button field's **buttonSetIcon** method.

For example, the following code shows one icon when the mouse enters the button field, and a different icon when the mouse exits:

```
var script = "var f = this.getField('myButton');";
script += "f.buttonSetIcon(this.getIcon('oneIcon');";
myButton.setAction("MouseEnter",script);

script = "var f = this.getField('myButton');";
script += "f.buttonSetIcon(this.getIcon('otherIcon');";
myButton.setAction("MouseExit",script);
```

Working with PDF Layers

About PDF Layers

PDF layers are components of content that may occupy the same space as other components. Multiple components may be visible or invisible depending on their settings, and may be used to support the display, navigation, and printing of layered PDF content by various applications. It is possible to edit the properties of layers, lock layers, add navigation to them, merge or flatten layers, and combine PDF layered documents. PDF layers are supported through the usage of Acrobat JavaScript Optional Content Group (OCG) objects.

To obtain an array of the **OCG** objects for a given page in the document, invoke the **doc** object's **getOCGs** method. The following code obtains the array of OCG objects contained on page 3 of the document:

```
var ocgArray = this.getOCGs(3);
```

Navigating with Layers

Since information can be stored in different layers of a PDF document, navigational controls can be customized within different layers, whose visibility settings may be dynamically customized so that they are tied to context and user interaction. For example, if the user selects a given option, a set of navigational links belonging to a corresponding optional content group may be shown.

In addition, it is possible to determine the order in which layers are displayed in the user interface by invoking the **doc** object's **getOCGOrder** and **setOCGOrder** methods. In the following example, the display order of all the layers is reversed:

```
var ocgOrder = this.getOCGOrder();
var newOrder = new Array();
for (var i=0; i<ocgOrder.length; i++)
    newOrder[i] = ocgOrder[ocgOrder.length - i - 1];
this.setOCGOrder(newOrder);
```

Editing the Properties of PDF Layers

The **OCG** object provides properties that can be used to determine whether the object's default state should be on or off, whether its intent should be for viewing or design purposes, whether it should be locked, the text string seen in the user interface, and the current state. The properties are shown below in Table 10.15:

TABLE 10.15   OCG Properties

Property	Description
initState	Determines whether the **OCG** object is on or off by default
intent	The intent of the **OCG** object (View or Design)
locked	Whether the on/off state can be toggled through the user interface
name	The text string seen in the user interface for the **OCG** object
state	The current on/off state of the **OCG** object

The **initState** property can be used to set the default state for an optional content group. In the following example, **myLayer** is set to **on** by default:

```
myLayer.initState = true;
```

The **intent** property, which is an array of values, can be used to define the intent of a particular optional content group. There are two possible values used in the array: **View** and **Design**. A **Design** layer is created for informational purposes only, and does not affect the visibility of content. Its purpose is to represent a document designer's structural organization of artwork. The **View** layer is intended for interactive use by document consumers. If **View** is used, the visibility of the layer is affected. In the following example, the intent of all the **OCG** objects in the document is set to both values:

```
var ocgs = this.getOCGs();
for (var i=0; i<ocgs.length; i++)
    ocgs[i].intent = ["View", "Design"];
```

The **locked** property is used to determine whether a given layer can be toggled through the user interface. In the following example, **myLayer** is locked, meaning that it cannot be toggled through the user interface:

```
myLayer.locked = true;
```

The **state** property represents the current on/off state for a given OCG. In the following example, all the OCGs are turned on:

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

The **name** property represents the text string seen in the user interface that is used to identify layers. In the following example, the **Watermark** OCG is toggled:

```
var ocgs = this.getOCGs();
for (var i=0; i<ocgs.length; i++)
    if (ocgs[i].name == "Watermark")
        ocgs[i].state = !ocgs[i].state;
```

Merging Layers

You can use Acrobat JavaScript to merge layers in a PDF document. A merged layer (the source layer) will acquire the properties of the layer into which they are merged (the target layer). In the following example, **sourceLayer** is merged into **targetLayer**:

```
sourceLayer.initState = targetLayer.initState;
sourceLayer.intent = targetLayer.intent;
sourceLayer.locked = targetLayer.locked;
sourceLayer.name = targetLayer.name;
sourceLayer.state = targetLayer.state;
```

Flattening PDF Layers

Flattening layers means that they will be consolidated. This operation is done through the **Layers** tab in the user interface.

Combining PDF Layered Documents

Multiple PDF documents containing layers can be combined while preserving the layered information. When doing this, it may be necessary to rearrange the order of the merged OCG arrays. To accomplish this, obtain the OCG order arrays for each document, and merge them into a new array in the merged document. In the example shown below, `source1.pdf` and `source2.pdf` are combined into `target.pdf`. It is assumed that there are no common layers between the two documents:

```
// Open the source documents:
var source1 = app.openDoc("/C/source1.pdf");
var source2 = app.openDoc("/C/source2.pdf");

// Obtain the OCG order array for each source document:
var mergedOCGArray = new Array();
var source1OCGArray = source1.getOCGOrder();
var source2OCGArray = source2.getOCGOrder();

// Merge the OCG order arrays into the target array:
for (var i=0; i<source1OCGArray.length; i++)
    mergedOCGArray[i] = source1OCGArray[i];
var offset = source1OCGArray.length;
for (var j=0; j<source2OCGArray.length; j++)
    mergedOCGArray[offset + j] = source2OCGArray[j];

// Create the target document:
var target = app.newDoc("/C/target.pdf");

// Insert source1.pdf:
target.insertPages({
    nPage : -1,
    cPath : "/c/source1.pdf",
});

// Insert source2.pdf:
target.insertPages({
    nPage : target.numPages,
    cPath : "/c/source2.pdf",
});

// Set the OCG order array in the target document
target.setOCGOrder(mergedOCGArray);

// Save the target document:
target.saveAs({
    "/c/target.pdf");
});

// Close the target document without notifying the user:
target.closeDoc(true);
```

11 Search and Index Essentials

Introduction

This chapter will enable you to customize and extend searching operations for PDF document content and metadata, as well as indexing operations. The principal Acrobat JavaScript objects used in searching and indexing are the **search**, **catalog**, and **index** objects. You will learn how to use these objects to accomplish the goals listed below.

Chapter Goals

At the end of this chapter, you will be able to:

● Use Acrobat JavaScript to search for text in one or more PDF documents.

● Customize searching to perform advanced queries.

● Customize indexing of PDF documents.

● Use JavaScript to read and search XMP metadata.

Contents

Topics
Searching for Text in PDF Documents
Indexing Multiple PDF Documents
Searching Metadata

Searching for Text in PDF Documents

Introduction

Acrobat JavaScript provides a static **search** object, which provides powerful searching capabilities that may be applied to PDF documents and indexes. Its properties and methods are described below in Table 11.1 and Table 11.2.

TABLE 11.1   *Search Properties*

Property	Description
attachments	Searches PDF attachments along with base document
available	Determines if searching is possible
docInfo	Searches document metadata information
docText	Searches document text
docXMP	Searches document XMP metadata
bookmarks	Searches document bookmarks
ignoreAccents	Ignores accents and diactrics in search
ignoreAsianCharacterWidth	Matches Kana characters in query
indexes	Obtains all accessible **index** objects
jpegExif	Searches EXIF data in associated JPEG images
markup	Searches annotations
matchCase	Determines whether query is case-sensitive
matchWholeWord	Finds only occurrences of complete words
maxDocs	Maximum number of documents returned
proximity	Uses proximity in results ranking for **AND** clauses
proximityRange	Range of proximity search in number of words
refine	Uses previous results in query
stem	Uses word stemming in searches

TABLE 11.1   *Search Properties*

Property	Description
`wordMatching`	Determines how words will be matched (phrase, all words, any words, boolean query)

TABLE 11.2   *Search Methods*

Method	Description
`addIndex`	Adds an index to the list of searchable indexes
`getIndexForPath`	Searches the index list according to a specified path
`query`	Searches the document or index for specified text
`removeIndex`	Removes an index from the list of searchable indexes

Finding Words in an PDF Document

The **search** object's **query** method is used to search for text within a PDF document. It accepts three parameters:

● **cQuery**: the text for which to search
● **cWhere**: where to search for the text:
  – **ActiveDoc**: search within the active document
  – **Folder**: search within a specified folder
  – **Index**: search within a specified index
  – **ActiveIndexes**: search within the active set of available indexes (the default)
● **cDIPath**: path to folder or index used in search

Performing a Simple Search of a Document

The simplest type of search is applied to the text within the PDF document. For example, the following code performs a case-insensitive search for the word **Acrobat** within the current document:

```
search.query("Acrobat");
```

Using Advanced Search Options

You can set the **search** object's properties to use advanced searching options, which can be used to determine how to match search strings, and whether to use proximity or stemming.

To determine how the words in the search string will be matched, set the **search** object's **wordMatching** property to one of the following values:

- **MatchPhrase**: match the exact phrase

- **MatchAllWords**: match all the words without regard to the order in which they appear

- **MatchAnyWord**: match any of the words in the search string

- **BooleanQuery**: perform a boolean query for multiple-document searches (the default)

For example, the following code matches the phrases **"My Search"** or **"Search My"**:

```
search.wordMatching = "MatchAllWords";
search.query("My Search");
```

To determine whether proximity is used in searches involving multiple documents or index definition files, set the **search** object's **wordMatching** property to **MatchAllWords** and set its **proximity** property to **true**. In the example below, all instances of the words **My** and **Search** that are not separated by more than 900 words will be listed in the search:

```
search.wordMatching = "MatchAllWords";
search.proximity = true;
search.query("My Search");
```

To use stemming in the search, set the **search** object's **stem** property to **true**. For example, the following search lists words that begin with "run", such as "running" or "runs":

```
search.stem = true;
search.query("run");
```

Searching Across Multiple PDF Documents

Searching all PDF Files in a Specific Location

To search all the PDF files within a folder, set the **cWhere** parameter in the **search** object's **query** method to **Folder**. In the following example, all documents in /C/MyFolder will be searched for the word "Acrobat":

```
search.query("Acrobat", "Folder", "/C/MyFolder");
```

Using Advanced Search Options for Multiple Document Searches

In addition to the advanced options for matching phrases, using stemming, and using proximity, it is also possible to specify whether the search should be case-sensitive, whether to match whole words, set the maximum number of documents to be returned as part of a query, and whether to refine the results of the previous query.

To specify that a search should be case sensitive, set the **search** object's **matchCase** property to **true**. For example, the following code matches "Acrobat" but not "acrobat":

```
search.matchCase = true;
search.query("Acrobat", "Folder", "/C/MyFolder");
```

To specify that the search should only identify occurrences of complete words, set the **search** object's **matchWholeWord** property to **true**. For example, the following code matches "stick", but not "tick" or "sticky":

```
search.matchWholeWord = true;
search.query("stick", "Folder", "/C/MyFolder");
```

To set the maximum number of documents to be returned as part of a query, set the **search** object's **maxDocs** property to the desired number (the default is 100). For example, the following code limits the number of documents to be searched to 5:

```
search.maxDocs = 5;
```

To refine the results of the previous query, set the **search** object's **refine** property to **true**, as shown in the following code:

```
search.refine = true;
```

Searching PDF Index Files

A PDF index file often covers multiple PDF files, and the time required to search an index is much less than that required to search each of the corresponding individual PDF files.

To search a PDF index, set the **cWhere** parameter in the **search** object's **query** method to **Index**. In the following example, **myIndex** is searched for the word "Acrobat":

```
search.query("Acrobat", "Index", "/C/MyIndex.pdx");
```

Using Boolean Queries in Multiple Document Searches

Boolean queries may be applied to multiple PDF documents using the following operations:

- **AND**
- **OR**
- **^** (exclusive or)
- **NOT**

For example, the phrase **"Paris AND France"** used in a search would return all documents containing both the words **Paris** and **France**.

The phrase **"Paris OR France"** used in a search would return all documents containing one or both of the words **Paris** and **France**.

The phrase **"Paris ^ France"** used in a search would return all documents containing exactly one (not both) of the words **Paris** and **France**.

The phrase **"Paris NOT France"** used in a search would return all documents containing **Paris** that do not contain the word **France**.

In addition, parentheses may be used. For example, the phrase **"one AND (two OR three)"** would be equivalent to performing two searches: one using the statement **"one AND two "**, followed by another using the statement **"one AND three"**.

To specify that a boolean query will be used, be sure that the **search** object's **wordMatching** property is set to **BooleanQuery** (which is the default).

Indexing Multiple PDF Documents

It is possible to extend and customize indexes for multiple PDF documents using the Acrobat JavaScript **catalog**, **catalogJob**, and **index** objects. These objects may be used to build, retrieve, or remove indexes. The **index** object represents a **catalog**-generated index, contains a **build** method that is used to create an index (and returns a **catalogJob** object containing information about the index), and has the properties shown below in Table 11.3:

TABLE 11.3   Index Properties

Property	Description
available	Indicates whether an index is available
name	The name of the index
path	The device-independent path of the index
selected	Indicates whether the index will participate in the search

The **catalog** object may be used to manage indexing jobs and retrieve indexes. It contains a **getIndex** method for retrieving an index, a **remove** method for removing a pending indexing job, and properties containing information about indexing jobs.

Creating, Updating, or Rebuilding Indexes

To determine which indexes are available, use the **search** object's **indexes** property, which contains an array of **index** objects. For each object in the array, you can determine its name by using its **name** property. In the code below, the names and paths of all available selected indexes are printed to the console:

```
var arr = search.indexes;
for (var i=0; i<arr.length; i++)
{
    if (arr[i].selected)
    {
        var str = "Index[" + i + "] = " + arr[i].name;
        str += "\nPath = " + arr[i].path;
        console.println(str);
    }
}
```

To build an index, first invoke the **catalog** object's **getIndex** method to retrieve the **index** object. This method accepts a parameter containing the path of the **index** object. Then invoke the **index** object's **build** method, which returns a **catalogJob** object. The method accepts two parameters:

● **cExpr**: an Acrobat JavaScript expression executed once the build operation is complete

● **bRebuildAll**: indicates whether to perform a clean build in which the existing index is first deleted and then completely built

Finally, the returned **catalogJob** object contains three properties providing useful information about the indexing job:

● **path**: the device-independent path of the index

● **type**: the type of indexing operation (**Build**, **Rebuild**, or **Delete**)

● **status**: the status of the indexing operation (**Pending**, **Processing**, **Completed**, or **CompletedWithErrors**)

In the code shown below, the index **myIndex** is completely rebuilt, after which its status is reported:

```
// Retrieve the Index object
var idx = catalog.getIndex("/C/myIndex.pdx");

// Build the Index
var job = idx.build("app.alert('Index build');", true);

// Confirm the path of the rebuilt index:
console.println("Path of rebuilt index: " + job.path);

// Confirm that the index was rebuilt:
console.println("Type of operation: " + job.type);

// Report the job status
console.println("Status: " + job.status);
```

Searching Metadata

Using Acrobat JavaScript to Read and Search XMP Metadata

PDF documents contain document metadata in XML format, which includes information such as the document title, subject, author's name, keywords, copyright information, date modified, file size, and file name and location path.

To use Acrobat JavaScript to search a document's XMP metadata, set the **search** object's **docXMP** property to **true**, as shown in the following code:

```
search.docXMP = true;
```

12 Security

Introduction

This chapter will introduce you to the various security options available through Acrobat JavaScript. You will understand how to customize security in PDF documents by applying passwords and digital signatures, certifying documents, encrypting files, adding security to attachments, managing digital IDs and certificates, and customizing security policies.

Chapter Goals

At the end of this chapter, you will be able to:

- Understand the Acrobat JavaScript security model supporting PDF documents.
- Use Acrobat JavaScript to add, remove, and validate digital signatures in a PDF document.
- Use Acrobat JavaScript to apply passwords, security options, usage rights, and encryption to PDF documents and attachments.
- Use Acrobat JavaScript to create, use, and manage digital IDs and certificates.

Contents

Topics
Security Essentials
Digitally Signing PDF Documents
Adding Security to PDF Documents
Digital IDs and Certification Methods

Security Essentials

Acrobat JavaScript provides a number of objects that support security. These are managed by the **security**, **securityPolicy**, and **securityHandler** objects for managing certificates, security policies, and signatures. The **certificate**, **directory**, **signatureInfo**, and **dirConnection** objects are used to manage digital signatures and access the user certificates.

Methods for Adding Security to PDF Documents

The general procedures for applying various types of security to a PDF document are described below. Details and examples are provided in the later sections of this chapter.

Passwords and Restrictions

The basic way to protect a document from unauthorized access is to encrypt it for a list of authorized recipients using the **doc** object's **encryptForRecipients** method. This essentially requires that the authorized recipients use a private key or credential to gain access to it. Restrictions may be applied so that the recipients' access to the document may be controlled.

Certifying Documents

The author signature for a document is what makes modification detection and prevention (mdp) possible. When this type of signature is applied, it is possible to certify the document, which means that you will specify information about its contents and the types of changes that are allowed in order for the document to remain certified.

To apply an author signature to a document, create an author signature field using the **doc** object's **addField** method. Then sign the field using the **field** object's **signatureSign** method, in which you will provide parameters containing the security handler, a **signatureInfo** object containing an **mdp** property value other than **allowAll**, and a legal attest explaining why certain legal warnings are embedded in the document. The **signatureInfo** object has properties common to all security handlers. These properties are described below in Table 12.1:

TABLE 12.1   *SignatureInfo Properties*

Property	Description
buildInfo	Software build and version for the signature
date	Date and time of the signature
handlerName	Security handler name specified in the **Filter** attribute in the signature dictionary
handlerUserName	Security handler name specified by **handlerName**
handlerUIName	Security handler name specified by **handlerName**

TABLE 12.1   *SignatureInfo Properties*

Property	Description
`location`	Physical location or hostname
`mdp`	Modification detection and prevention setting (`allowNone`, `allowAll`, `default`, `defaultAndComments`)
`name`	Name of the user
`numFieldsAltered`	Number of fields altered since the previous signature
`numFieldsFilledIn`	Number of fields filled in since the previous signature
`numPagesAltered`	Number of pages altered since the previous signature
`numRevisions`	The number of revisions in the document
`reason`	Reason for signing
`revision`	Signature revision
`status`	Validity status (**4** represents a completely valid signature)
`statusText`	String representation of signature status
`subFilter`	Formats used for public key signatures
`verifyHandlerName`	Security handler used to validate signature
`verifyHandlerUIName`	Handler specified by `verifyHandlerName`

Encrypting Files Using Certificates

When you invoke the **doc** object's **encryptForRecipients** method, it encrypts the document using the public key certificates of each recipient. The groups of recipients are specified in the **oGroups** parameter, which is an array of **Group** objects, each of which contains two properties: **permissions** and **userEntities**. The **userEntities** property is an array of **UserEntity** objects (described below in Table 12.2), each of which describes a user and their associated certificates, and is returned by a call to the **dirConnection** object's **search** method. The associated certificates are represented in a property containing an array of **Certificate** objects (described below in Table 12.3), each of which contains read-only access to the properties of an X.509 public key certificate.

To obtain a group of recipients (the **oGroups** parameter mentioned above), you may invoke the **security** object's **chooseRecipientsDialog** method, which opens a dialog box prompting the user to choose a list of recipients.

TABLE 12.2   *UserEntity Object Properties*

Property	Description
firstName	the first name of the user
lastName	the last name of the user
fullName	the full name of the user
certificates	array of **Certificate** objects for the user
defaultEncryptCert	the preferred **Certificate**

TABLE 12.3   *Certificate Object Properties*

Property	Description
binary	the raw bytes of the certificate
issuerDN	the distinguished name of the user
keyUsage	the value of the certificate key usage extension
MD5Hash	the MD5 digest of the certificate
SHA1Hash	the SHA1 digest of the certificate
serialNumber	a unique identifier for the certificate
subjectCN	the common name of the signer
subjectDN	the distinguished name of the signer
usage	purposes: end-user signing or encryption
ubrights	an application **Rights** object

Security Policies

Security policies are common specifications that include the type of encryption, the permission settings, and the password or public key to be used. You may create folder-level scripts containing objects that reflect these policies. Security policies may be customized through the use of **securityPolicy** objects, which may be accessed and managed by the **security** object's **getSecurityPolicies** and **chooseSecurityPolicy** methods, as well as the **doc** object's **encryptUsingPolicy** method.

Secure Forms

You can lock form fields by creating a script containing a call to the **signature** field's **setLock** method, and passing that script as the second parameter to the **signature** field's **setAction** method.

In addition, you may sign an embedded FDF data object by invoking its **signatureSign** method, and subsequently validate the signature by invoking its **signatureValidate** method.

Digitally Signing PDF Documents

A digital signature contains identifying information about the person signing the document. When applying an author signature (the first time a signature is applied to a document), it is also possible to certify the document. This involves providing a legal attest with regard to the document's contents and specifying the types of changes allowed for the document in order for it to remain certified.

Signing a PDF Document

To sign a document, create a signature field, choose a security handler, and invoke the field's **signatureSign** method, which accepts the following parameters:

● **oSig**: the security handler object

● **oInfo**: a **signatureInfo** object

● **cDIPath**: the device-independent path to which the file will subsequently be saved

● **bUI**: whether the security handler will display a user interface when signing

● **cLegalAttest**: a string explaining legal warnings (for author signatures only)

The creation and usage of these parameters are explained below in the following sections: The Security Handler Object, The SignatureInfo Object, and Applying the Signature.

The Security Handler Object

To obtain a security handler (the **oSig** parameter), invoke the **security** object's **getHandler** method, which creates a new security handler engine, and accepts the following parameters:

● **cName**: the name of the security handler (contained in the **security** object's **handlers** property)

● **bUIEngine**: the existing engine associated with the Acrobat user interface

The following code illustrates how to set up signature validation whenever the document is opened, lists all available security handlers, and selects the **Adobe.PPKLite** engine associated with the Acrobat user interface:

```
// Validate signatures when the document is opened:
security.validateSignaturesOnOpen = true;

// List all the available signature handlers
for (var i=0; i<security.handlers.length; i++)
    console.println(security.handlers[i]);

// Select the Adobe.PPKLite engine with Acrobat user interface:
var ppklite = security.getHandler("Adobe.PPKLite", true);
```

After obtaining the security handler, invoke the **securityHandler** object's **login** method, which makes it possible to access and select your digital ID, as shown in the following code:

```
var oParams = {
    password: "myPassword",
    cDIPath: "/C/signatures/myName.pfx" // digital signature profile
};
ppklite.login(oParams);
```

The SignatureInfo Object

To create the **oInfo** parameter for the signature field's **signatureSign** method, create a generic object containing the properties as described above in Table 12.1. An example of its usage when creating an author signature is given below:

```
var myInfo = {
    password: "myPassword",
    location: "San Jose, CA",
    reason: "I am approving this document",
    contactInfo: "userName@adobe.com",
    appearance: "Fancy",
    mdp: "allowNone" // an mdp value is needed for author signatures
};
```

Applying the Signature

Now that the security handler and signature information have been created, you may invoke the signature field's **signatureSign** method, as shown in the code below:

```
// Obtain the signature field object:
var f = this.getField("myAuthorSignatureField");

// Sign the field:
f.signatureSign(
    oSig: ppklite,
    oInfo: myInfo,
    cDIPath: "/C/mySignedFile.pdf",
    bUI: true,
    cLegalAttest: "Fonts are not embedded to reduce file size"
); //end of signature
```

Creating a New Signature Appearance

You may create a new signature appearance through the Acrobat user interface, access signature appearances through the **appearances** property of the **securityHandler** object, and use the signature field's **signatureSetSeedValue** method, which accepts a **SeedValue** object used to control signature properties.

One of the **SeedValue** properties is called **subFilter**, which is an array of acceptable formats to use for the signature. In the following example, the signing handler is set to **PPKMS** and the format is set to **adbe.pksc7.sha1**:

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

f.signatureSetSeedValue({
    filter: "Adobe.PPKMS",
    subFilter: ["adbe.pksc7.sha1"],
    flags: 0x03
});
```

Clearing a Digital Signature from a Signature Field

To clear a signature, invoke the **doc** object's **resetForm** method. In the example below, **mySignatureField** is cleared:

```
var f = this.getField("mySignatureField");
this.resetForm(f);
```

Getting Signature Information from Another User

You may maintain a list of trusted user identities by adding the certificates contained within FDF files sent to you by other users. You may also obtain signature information from an FDF file by invoking the **FDF** object's **signatureValidate** method, which returns a **signatureInfo** object, as shown in the example below:

```
// Open the FDF file sent to you by the other user:
var fdf = app.openFDF("/C/myDoc.fdf");

// Obtain the security handler:
var engine = security.getHandler("Adobe.PPKLite");

// Check to see if the FDF has been signed:
if (fdf.isSigned)
{
    // Obtain the other user's signature info:
    sigInfo = fdf.signatureValidate({
       oSig: engine,
       bUI: true
    });

    // Display the signature status and description:
    console.println("Signature Status: " + sigInfo.status);
    console.println("Description: " + sigInfo.statusText);
}
else
    console.println("This FDF was not signed.");
```

Removing Signatures

To remove a signature field, invoke the **doc** object's **removeField** method. In the example below, **mySignatureField** is removed:

```
var f = this.getField("mySignatureField");
this.removeField(f);
```

Certifying a Document

When applying an author signature to certify a document, check **trustFlags**, which is a read-only property of the **signatureInfo** object. If its value is **2**, the signer is trusted for certifying documents.

Validating Signatures

To validate a signature, invoke the signature field's **signatureValidate** method, which returns one of the following integer validity status values:

- **-1**: not a signature field
- **0**: signature is blank
- **1**: unknown status
- **2**: signature is invalid
- **3**: signature is valid, identity of signer could not be verified
- **4**: signature and identity of signer are both valid

The method accepts two parameters:

- **oSig**: the security handler used to validate the signature (a **securityHandler** or **SignatureParameters** object)
- **bUI**: determines whether the user interface is shown when validating the data file

A **SignatureParameters** object contains two properties:

- **oSecHdlr**: the security handler object
- **bAltSecHdlr**: determines whether an alternate security handler may be used

In the following example, **mySignatureField** is analyzed for validity:

```
// Obtain the signature field:
var f = this.getField("mySignatureField");

// Validate the signature field:
var status = f.signatureValidate();

// Obtain the signature information
var sigInfo = f.signatureInfo();

// Check the status returned from the validation:
if (status < 3)
    var msg = "Signature is not valid: " + sigInfo.statusText;
else
    var msg = "Signature is valid: " + sigInfo.statusText;

// Display the status message:
app.alert(msg);;
```

Using Approval Stamps

You may apply a stamp annotation to a document that indicates whether approval is indicated. To do this, set the annotation's **AP** property to either **Approved** or **NotApproved**, as shown in the example below:

```
var annot = this.addAnnot({
    page: 0,
    type: "Stamp",
    author: "A.C. Robat",
    rect: [400, 400, 550, 500],
    contents: "This is good."
    AP: "Approved"
});
```

Setting Digital Signature Preferences

When applying digital signatures, you may specify the appearance and the default signing method. In addition, you may set the **security** object's **validateSignaturesOnOpen** property to verify signatures whenever the document is opened, and set up policies that examine the signature information. Signature information may be obtained by invoking the signature field's **signatureInfo** method. At this point you can customize the behavior based on the information found within the **signatureInfo** object.

The following example illustrates how to access the signature information:

```
// Obtain the signature field:
var f = this.getField("mySignatureField");

// Validate the signature field:
var status = f.signatureValidate();

// Obtain the signature information
var sigInfo = f.signatureInfo();

console.println("Name: " + sigInfo.name);
console.println("Reason: " + sigInfo.reason);
console.println("Date: " + sigInfo.date);
console.println("Contact Info: " + sigInfo.contactInfo);
```

Adding Security to PDF Documents

Adding Passwords and Setting Security Options

Since the **Standard** security handler, used for password encryption of documents, is not JavaScript-enabled, the most direct way to add passwords is through the creation of user or master passwords in the Acrobat user interface.

As you learned earlier in Encrypting Files Using Certificates, you may encrypt a document for a number of recipients using certificates, and can set security policies through the

application of an author signature accompanied by the desired modification, detection, and prevention settings shown above in Table 12.1.

Adding Usage Rights to a Document

You may decide which usage rights will be permitted for a set of users. You may specify either full, unrestricted access to the document, or rights that address accessibility, content extraction, allowing changes, and printing. You may use Acrobat JavaScript to customize these rights when encrypting a document for a list of recipients. For more information, see Rights-Enabled PDF Files.

Encrypting PDF Files for a List of Recipients

As you have seen throughout this chapter, the **doc** object's **encryptForRecipients** method is the primary means of encrypting PDF files for a list of recipients using Acrobat JavaScript. In the previous example, the certificates used were gathered by connecting to a **directory**, which is a repository of user information. The **directory** object contains an **info** property with which it is possible to create and activate a new directory, and is accessible either through the **directories** property or the **newDirectory** method of the **securityHandler** object.

The **info** object is a **DirectoryInformation** object, which may contain standard properties related to the name of the directory, as well as additional properties specific to a particular directory handler (these may include server and port information).

To create a new directory, create a **DirectoryInformation** object, obtain a **securityHandler** object and invoke its **newDirectory** method, and assign the **DirectoryInformation** object to the new directory's **info** property. An example of this is given below:

```
// Create and activate a new directory:
var newDirInfo = {
    dirStdEntryID: "dir0",
    dirStdEntryName: "Employee LDAP Directory",
    dirStdEntryPrefDirHandlerID: "Adobe.PPKMS.ADSI",
    dirStdEntryDirType: "LDAP",
    server: "ldap0.acme.com",
    port: 389
};

// Obtain the security handler:
var sh = security.getHandler("Adobe.PPKMS");

// Create the new directory object:
var newDir = sh.newDirectory();

// Store the directory information in the new directory:
newDir.info = newDirInfo;
```

In order to obtain certificates from a directory, you must first connect to it using the **directory** object's **connect** method, which returns a **DirConnection** object. An example is given below:

```
// Obtain the security handler:
var sh = security.getHandler("Adobe.PPKMS");
var dc = sh.directories[0].connect();
```

It is then possible to use the **DirConnection** object to search for certificates. You can specify the list of attributes to be used for the search by invoking the **DirConnection** object's **setOutputFields** method, which accepts two parameters:

● **oFields**: an array of attributes to be used in the search

● **bCustom**: whether the attributes are standard output attribute names

For example, the following code specifies standard output attributes (**certificates** and **email**):

```
dc.setOutputFields({oFields: ["certificates", "email"]});
```

To perform the search, invoke the **DirConnection** object's **search** method, which accepts the following parameters:

● **oParams**: an array of key-value pairs consisting of search attribute names and their corresponding strings

● **cGroupName**: the name of the group to which to restrict the search

● **bCustom**: whether **oParams** contains standard attribute names

● **bUI**: whether a user interface is used to collect the search parameters

In the following example, the directory is searched for certificates for the user whose last name is "Smith", and displays the user's email address:

```
var retval = dc.search({oParams: {lastName: "Smith"}});
if (retval.length > 0)
    console.println(retval[0].email);
```

When you invoke the object's **encryptForRecipients** method, the **oGroups** parameter is an array of **Group** objects, each of which contains a **permissions** property. The **permissions** property is an object containing the properties described below in Table 12.4:

TABLE 12.4   *Permissions Object*

Property	Description
`allowAll`	Full, unrestricted access
`allowAccessibility`	Content access for the visually impaired
`allowContentExtraction`	content copying and extraction
`allowChanges`	Allowed changes (**none**, **documentAssembly**, **fillAndSign**, **editNotesFillAndSign**, **all**)
`allowPrinting`	Printing security level (**none**, **lowQuality**, **highQuality**)

The following code allows full and unrestricted access to the entire document for one set of users (**importantUsers**), and allows high quality printing for another set of users (**otherUsers**):

```
// Obtain the security handler:
var sh = security.getHandler("Adobe.PPKMS");

// Connect to the directory containing the user certificates:
var dir = sh.directories[0];
var dc = dir.connect();

// Search the directory for certificates:
dc.setOutputFields({oFields:["certificates"]});
var importantUsers = dc.search({oParams:{lastName:"Smith"}});
var otherUsers = dc.search({oParams:{lastName:"Jones"}});

// Allow important users full, unrestricted access:
var importantGroup = {
    oCerts: importantUsers,
    oPermissions: {allowAll: true}
};

// Allow other users high quality printing:
var otherGroup = {
    oCerts: otherUsers,
    oPermissions: {allowPrinting: "highQuality"}
};

// Encrypt the document for the intended recipients:
this.encryptForRecipients({
    oGroups:[importantGroup, otherGroup],
    bMetaData: true
});
```

Encrypting PDF Files Using Security Policies

It is possible to define a security policy for a PDF document. The policy can contain a list of people who can open the document, restrictions limiting their ability to modify, print, or copy the document, and an expiration date for the document after which it cannot be opened.

There are two kinds of security policies: a custom policy is one created by a user and is stored on a local computer, and a corporate policy is developed by an organization and stored on a policy server such as LiveCycle^TM Policy Server^®).

There are three types of custom policies. You may create policies for password security, certificate security, and policies used on LiveCycle Policy Server.

Acrobat JavaScript defines a **securityPolicy** object that contains the following properties:

● **id**: a machine-readable policy id string

● **name**: the policy name

● **description**: the policy description

● **lastModified**: the date when the policy was last modified

● **handler**: the handler that implements the policy (**Adobe.APS**, **Adobe.PubSec**, and **Adobe.Standard**)

● **target**: the target data covered by the policy (**document** or **attachments**)

To obtain a list of the security policies currently available, invoke the **security** object's **getSecurityPolicies** method, which accepts two parameters:

● **oOptions**: a **securityPolicyOptions** object containing parameters used to filter the list

● **bUI**: determines whether the user interface will be displayed (affects **bCheckOnline** in the **oOptions** parameter)

The **securityPolicyOptions** object is a generic object used to filter the list of security policies that will be returned by the method, and contains the following properties:

● **bCheckOnline**: determines whether to check online for updated security policies

● **bFavorites**: determines whether to return policies which are favorites

● **cFilter**: returns policies using the specified **PDCrypt** filter (**Adobe.APS**, **Adobe.PubSec**, and **Adobe.Standard**)

● **cTarget**: returns policies using the specified **target** (**document** or **attachments**)

The following example illustrates how to request and display a list of favorite security policies:

```
// Set up the filtering options (SecurityOptionsPolicy object):
var options = {
    bFavorites: true,
    cFilter: "Adobe.PubSec"
};

// Obtain the filtered list of security policies:
var policyArray = security.getSecurityPolicies(options);

// Display the list of security policies by name:
for (var i=0; i<policyArray.length; i++)
    console.println(policyArray[i].name);
```

To encrypt a PDF file using a security policy, you must first choose a security policy by invoking the **security** object's **chooseSecurityPolicy** method, and then encrypt the file by invoking the **doc** object's **encryptUsingPolicy** method.

The **security** object's **chooseSecurityPolicy** method opens a dialog that permits the user to choose from a list of security policies, filtered according to a **securityPolicyOptions** object.

The **doc** object's **encryptUsingPolicy** method accepts three parameters:

● **cPolicyID**: the policy id to use when encrypting the document

● **oRecipients**: a **Group** object containing a list of recipients

● **bIgnoreUnknowns**: determines whether unknown or incomplete recipients will cause an exception to be thrown

In the following example, a newly created document is encrypted for a list of recipients, using the **encryptUsingPolicy** method, by choosing and applying a security policy:

```javascript
// Create the new document
var myDoc = app.newDoc();

// Set up the filtering options (SecurityOptionsPolicy object):
var options = {
    bCheckOnline: true,
    cFilter: "Adobe.APS"
};

// Choose the security policy:
var policy = security.chooseSecurityPolicy(options);

// Choose the list of recipients
var recipients = {
    userEntities: [
        {email: "user1@adobe.com"},
        {email: "user2@adobe.com"},
        {email: "user3@adobe.com"}
    ]
};

// Encrypt the document using the security policy:
var policyID = myDoc.encryptUsingPolicy({
    cPolicyId: "adobe_secure_for_recipients",
    oRecipients: [recipients]
});

// Display the policy ID:
console.println("Policy ID = " + policyID);
```

Adding Security to Document Attachments

You may add security to a document by encrypting its attachments and enclosing them in an *eEnvelope*. To do this with Acrobat JavaScript, invoke the **doc** object's **addRecipientListCryptFilter** method, which is used to encrypt data objects and accepts two parameters:

● **oCryptFilter**: the name of the encryption filter

● **oGroup**: an array of **Group** objects representing the intended recipients

Thus, an eEnvelope is a PDF file that contains encrypted attachments. The name of the crypt filter, which represents the recipient list, is defined and used when importing the attachment. An example is given below:

```
// Create instructions to be used in the recipient dialog:
var note = "Select the recipients. Each must have ";
note += "an email address and a certificate.";

// Specify the remaining options used in the recipient dialog:
var options = {
    bAllowPermGroups: false,
    cNote: note,
    bRequireEmail: true
};

// Obtain the intended recipient Group objects:
var groupArray = security.chooseRecipientsDialog(options);

// Open the eEnvelope document:
var env = app.openDoc("/C/myeEnvelope.pdf");

// Set up the crypt filter:
env.addRecipientListCryptFilter("myFilter", groupArray);

// Attach the current document to the eEnvelope:
env.importDataObject("secureMail0", this.path, "myFilter");

// Save the eEnvelope:
env.saveAs("/C/outmail.pdf");
```

Digital IDs and Certification Methods

It is possible to customize and extend the management and usage of digital IDs using Acrobat JavaScript. In addition, it is possible to share digital ID certificates, build a list of trusted identities, and analyze the information contained within certificates.

Digital IDs

As you learned earlier, a digital ID is represented with a **signatureInfo** object, which contains properties of the digital signature common to all handlers, in addition to other properties defined by public key security handlers. These additional properties are described below in Table 12.5:

TABLE 12.5   *SignatureInfo Public Key Security Handler Properties*

Property	Description
appearance	User-configured appearance name
certificates	Chain of certificates from signer to certificate authority
contactInfo	User-specified contact information for determining trust
byteRange	Bytes covered by this signature
docValidity	Validity status of the document byte range digest
idPrivValidity	Validity of the identity of the signer
idValidity	Numerical validity of the identity of the signeN
objValidity	Validity status of the object digest
trustFlags	What the signer is trusted for
password	Password used to access private key for signing

About Digital ID Providers

A digital ID provider is a trusted 3rd party, often called a *certificate authority* or *signature handler*, that verifies the identity, issues the private key, and protects the public key. The **certificates** property of the **signatureInfo** object contains an array of certificates that reflects the hierarchy leading from the signer's certificate to that issued by the certificate authority. Thus, you may inspect the details of the certificate issued by the digital ID provider, such as its **usage** property.

For example, the following code encrypts the current document for everyone in the address book. It does this by creating a collection of certificates suitable for encrypting documents, which are filtered from the overall collection. This is accomplished by examining all the certificates in the address book and excluding those entries containing sign-only certificates, CA certificates, no certificates, or certificates otherwise unsuitable for encryption:

```
// Obtain the security handler:
var eng = security.getHandler("Adobe.AAB");

// Connect to the directory containing the certificates:
var dc = eng.directories[0].connect();

// Obtain the list of all recipients in the directory:
var rcp = dc.search();

// Create the filtered recipient list:
var fRcp = new Array();

// Populate the filtered recipient list:
for (var i=0; i<rcp.length; i++) {
    if (rcp[i].defaultEncryptCert &&
        rcp[i].defaultEncryptCert.usage.endUserEncryption)
        fRcp[fRcp.length] = rcp[i];
    if (rcp[i].certificates) {
        for (var j=0; j<rcpp[i].certificates.length; j++)
            if (rcp[i].certificates[j].usage.endUserEncryption)
                fRcp[fRcp.length] = rcp[i];
    }
}

// Now encrypt for the filtered recipient list:
this.encryptForRecipients({[userEntities: fRcp]});
```

Creating a Digital ID (Default Certificate Security)

If you would like to create a certificate for a new user, invoke the **securityHandler** object's **newUser** method, which supports enrollment with the **Adobe.PPKLite** and **Adobe.PPKMS** security handlers by creating a new self-sign credential, and prevents the user from overwriting the file. It accepts the following parameters:

- **cPassword**: the password needed to access the digital ID file
- **cDIPath**: the location of the digital ID file
- **oRDN**: the *relative distinguished name* (represented as an **RDN** object) containing the issuer or subject name for the certificate
- **oCPS**: the certificate policy information, which is a generic object containing the following properties:
  - **oid**: the certificate policy object identifier
  - **url**: URL pointing to detailed policy information
  - **notice**: shortened version of detailed policy information
- **bUI**: determines whether to use the user interface to enroll the new user

The relative distinguished name is a generic object containing the properties shown below in Table 12.6:

TABLE 12.6 **RDN Object**

Property	Description
c	Country or region
cn	Common name
o	Organization name
ou	Organization unit
e	Email address

An example is given below:

```
// Obtain the security handler:
var ppklite = security.getHandler("Adobe.PPKLite");

// Create the relative distinguished name:
var newRDN = {
    cn: "newUser",
    c: "US"
};

// Create the certificate policy information:
var newCPS = {
    oid: "1.2.3.4.5",
    url: "http://newca.com/newCPS.html",
    notice: "This is a self-generated certificate"
};

// Create the new user's certificate:
security.newUser({
    cPassword: "newUserPassword",
    cDIPath: "/C/newUser.pfx",
    oRDN: newRDN,
    oCPS: newCPS,
    bUI: false
});
```

The **securityHandler** object has a **DigitalIDs** property that contains the certificates associated with the currently selected digital IDs for the security handler. The **DigitalIDs** property is a generic object containing the following properties:

● **oEndUserSignCert**: the certificate used when signing

● **oEndUserCryptCert**: the certificate used when encrypting

● **certs**: an array of certificates corresponding to all the digital IDs

● **stores**: an array of strings (one for every **certificate** object) indicating where the digital IDs are stored

You may use the **security** object's **exportToFile** method to save a certificate file to disk. In the following example, the signing certificate is written to disk:

```
// Obtain the security handler:
var sh = security.getHandler("Adobe.PPKMS");

// Obtain the certificates:
var ids = sh.DigitalIDs;

// Write the signing certificate to disk:
security.exportToFile(ids.oEndUserSignCert, "/C/mySignCert.cer");
```

Using Digital IDs (Default Certificate Security)

As you learned earlier, you may obtain signature information from a signature field by invoking its **signatureInfo** method. In addition to this, you may also use an existing certificate to create a digital ID. To do this, obtain the certificate from an existing, signed field and create the relative distinguished name using the information it contains, as shown in the following example:

```
// Obtain the security handler:
var ppklite = security.getHandler("Adobe.PPKLite");

// Obtain the signature field:
var f = this.getField("existingSignature");

// Validate the signature:
f.signatureValidate();

// Obtain the signature information:
var sigInfo = f.signatureInfo();

// Obtain the certificates and distinguished name information
var certs = sigInfo.certificates;
var rdn = certs[0].subjectDN;

// Now create the digital signature:
ppklite.newUser({
    cPassword: "newUserPassword",
    cDIPath: "/C/newUser.pfx",
    oRDN: rdn,
});
```

Managing Digital IDs (Windows Certificate Security)

A `directory` object is a repository of user information, including public key certificates. On Windows, the `Adobe.PPKMS` security handler provides access, through the **Microsoft Active Directory Script Interface** (**ADSI**), to the directories created by the user. These are created sequentially with the names `Adobe.PPKMS.ADSI.dir0`, `Adobe.PPKMS.ADSI.dir1`, etc. In this case, the `Adobe.PPKMS.ADSI` directory handler includes the directory information object properties shown below in Table 12.7:

TABLE 12.7   *Adobe.PPKMS.ADSI Directory Handler Object Properties*

Property	Description
`server`	The server hosting the data
`port`	The port number (standard LDAP port is 389)
`searchBase`	Used to narrow the search to a section of the directory
`maxNumEntries`	Maximum number of entries retrieved from search
`timeout`	Maximum time allowed for search

For example, the following code displays information for an existing directory:

```
// Obtain the security handler:
var ppkms = security.getHandler("Adobe.PPKMS");

// Obtain the directory information object:
var info = ppkms.directories[0].info;

// Display some of the directory information:
console.println("Directory: " + info.dirStdEntryName);
console.println("Address: " + info.server + ":" + info.port);
```

Managing Digital ID Certificates

Sharing Digital ID Certificates

You may share a self-signed digital ID certificate by exporting it as an FDF file. To do this, sign the FDF file by invoking the **FDF** object's **signatureSign** method, which works similarly to that of the **doc** object. Its usage is illustrated in the example below:

```
// Obtain the security handler:
var eng = security.getHandler("Adobe.PPKLite");

// Access the digital ID:
eng.login("myPassword", "/C/myID.pfx");

// Open the FDF:
var myFDF = app.openFDF("/C/myFDF.fdf");

// Sign the FDF:
if (!myFDF.isSigned) {
    // Sign the FDF
    myFDF.signatureSign({
        oSig: eng,
        nUI: 1,
        cUISignTitle: "Sign Embedded File FDF",
        cUISelectMsg: "Please select a digital ID"
    });

    // Save the FDF
    myFDF.save("/C/myFDF.fdf");
}
```

Building a List of Trusted Identities

The trust level associated with a digital ID is stored in the **trustFlags** property defined in the **signatureInfo** object's public key security handler properties. The bits in this number indicate the level of trust associated with the signer, and are valid only when the **status** property has a value of **4**. These trust settings are derived from those in the recipient's trust database, such as the *Acrobat Address Book* (**Adobe.AAB**). The following bit assignments are described below:

● **1**: trusted for signatures

● **2**: trusted for certifying documents

● **3**: trusted for dynamic content such as multimedia

● **4**: Adobe internal use

● **5**: the JavaScript in the PDF file is trusted to operate outside the normal PDF restrictions

Checking Information on Certificates

As you learned earlier, you may obtain a certificates through the **certificates** property of a **signatureInfo** object, which is returned by a call to the signature field's **signatureInfo** method. The certificate properties are described above in Table 12.3, and the relative distinguished name properties are defined in Table 12.6.

In the following example, the signer's common name, the certificate's serial number, and the distinguished name information are displayed:

```
// Obtain the signature field:
var f = this.getField("mySignatureField");

// Validate the signature field:
var status = f.signatureValidate();

// Obtain the signature information
var sigInfo = f.signatureInfo();

// Obtain the certificate:
var cert = sigInfo.certificates[0];

console.println("signer's common name: " + cert.subjectCN);
console.println("serial number: " + cert.serialNumber);

// Distinguished name information:
console.println("distinguished common name: " + cert.subjectDN.cn);
console.println("distinguished organization: " + cert.subjectDN.o);
```

Tokenized Acrobat JavaScript Security Model

In order to maintain security for cases in which Acrobat JavaScript connects to data sources external to the current document without user interaction (excluding console and batch sequences), a token-based security scheme is used to match and validate JavaScript *resources* and operations (a *resource* is defined as the combination of a protocol and a host). Each JavaScript is provisionally granted a token, which is matched against the resource.

The methods and properties shown below in Table 12.8 use this token-based security model:

TABLE 12.8   Methods and Properties Using Tokenized Security Model

Method/Property	Description
`doc.submitForm`	Get/send
`doc.getURL`	Send
`doc.mailDoc`	Send when **bUI** is **false**
`doc.mailForm`	Send when **bUI** is **false**
`doc.save`	May mark doc with data sources
`doc.saveAs`	May mark doc with data sources
`soap.connect`	Get/send
`soap.request`	Get/send
`soap.response`	Get/send
`ADBC.newConnection`	Get/send
`app.activeDocs`	Get
`app.openDoc`	Get
`app.global`	Get

13 Rights-Enabled PDF Files

Introduction

When creating a PDF document, it is possible to create certified documents by assigning special rights to it that enable users of Adobe Reader to fill in forms, participate in online reviews, and attach files. LiveCycle Reader Extensions® may be used to activate additional functionality within Adobe Reader for a particular document, thereby enabling the user to perform such operations as save, fill in, annotate, sign, certify, authenticate, and submit the document, thus streamlining collaboration for document reviews and automating data capture with electronic forms. In addition, users of Acrobat Professional can Reader-enable collaboration.

Chapter Goals

At the end of this chapter, you will be able to:

- List the additional usage rights that can be enabled in a PDF document.
- Understand the roles of Adobe Document Server and LiveCycle Reader Extensions.
- Understand how Reader-enabled PDF documents affect the usage of Acrobat JavaScript.
- Understand how to enable both file-based and server-based collaboration.
- Understand how the security model is related to additional usage rights.

Contents

Topics
Additional Usage Rights
LiveCycle Reader Extensions
Writing Acrobat JavaScript for Reader
Enabling Collaboration

Additional Usage Rights

Livecycle Reader Extensions sets permissions within a PDF document that would otherwise be disabled through the usage of a certificate issued by the Adobe Reader-Extension Root Certificate Authority. Permissions for Reader-enabled PDF files are determined by the combination of the user rights settings in the PDF file and and a special Object Identifier (OID), embedded within the certificate, specifying the additional permissions to be made available. The permissions are listed below:

● Form: fill-in and document full-save

● Form: import and export

● Form: add and delete

● Form: submit standalone

● Form: spawn template, spawn page from template, and delete spawned pages.

NOTE: Changing page visibility using `template.hidden` is not allowed through Acrobat JavaScript.

● Signature: modify

● Annotation: create, delete, modify, and copy

● Annotation: import and export

● Form: barcode plain text

● Annotation: online

● Form: online

NOTE: Acrobat JavaScript allows SOAP access in forms. For Adobe Reader 6.0, SOAP access is allowed in Acroforms. For Adobe Reader 6.02, OleDb database access is allowed on Windows for static XML forms. For Adobe Reader 7.0.5, SOAP access is allowed for static and dynamic XML forms.

● Embedded File: create, delete, modify, copy, and import.

NOTE: The ability to manipulate embedded data objects is available in Adobe Reader 6.0 and later.

LiveCycle Reader Extensions

During the design process, a PDF document may be created through the usage of PDF generation products of LiveCycle or other products. Then the document creator may assign appropriate usage rights using the LiveCycle Reader Extensions. The PDF document is made available on the Web, and users may complete the form on the web site, or save it locally and subsequently complete and annotate it, digitally sign it, add attachments, and return it.

In effect, LiveCycle Reader Extensions will enable functionality within Adobe Reader for a given document that is not normally available. After the user has finished working with the document, those functions will be disabled until the user receives another rights-enabled PDF file.

One major advantage of LiveCycle Reader Extensions is that the client application (Adobe Reader) is on the user's desktop, which means that there is no cost to the end user.

Writing Acrobat JavaScript for Reader

It is possible to access or assign additional usage rights within a PDF file by using the LiveCycle Reader Extensions API or its Web user interface.

NOTE: For rights-enabled documents, certain editing features normally available within the Acrobat Standard and Professional products will be disabled. This will ensure that the user does not inadvertantly invalidate the additional usage rights in a document under managed review before passing the document on to an Adobe Reader user for further commenting.

The methods shown below in Table 13.1 will be disabled by LiveCycle Reader Extensions:

TABLE 13.1   ***Features Disabled by Livecycle Reader Extensions***

Features	Methods
delete, add, replace, and move pages	`doc.deletePages`, `doc.newPage`, `doc.replacePages`, `doc.movePage`, `doc.insertPages` `doc.spawnPageFromTemplate`
modify page content	`doc.addWatermarkFromText`, `doc.addWatermarkFromFile`, `doc.flattenPages`, `doc.deleteSound`
add or modify JavaScripts	`doc.setAction`, `doc.setPageAction`, `field.setAction`, `link.setAction`, `ocg.setAction`, `bookmark.setAction`, `this.importAnFDF`
invoke web services	`SOAP.connect` `SOAP.request` `SOAP.response`

In Acrobat Standard and Professional, the following controls will be disabled for rights-enabled documents:

● Menu items under **Advanced > JavaScript** that allow the addition or modification of JavaScripts (except for the **Debugger**).

● Menu items under **Advanced > Forms** that allow the creation, modification, or deletion of form fields (except the **Import** and **Export** menu items).

● Certain operations within **Document Properties > Security Panel** will be marked as **Not Allowed**.

In addition, since the following menu operations will be affected in Acrobat Standard and Professional, so will their corresponding JavaScript methods, indicated below in Table 13.2:

TABLE 13.2　*Controls Affected by Livecycle Reader Extensions in Acrobat Standard and Professional*

Menu Operation	Equivalent JavaScript Method
File > **Reduce File Size**	`app.execMenuItem`
Document > **Pages** > **Insert**	`doc.insertPage`
Document > **Pages** > **Replace**	`doc.replacePages`
Document > **Pages** > **Delete**	`doc.deletePages`
Document > **Pages** > **Crop**	`doc.setPageBoxes`
Document > **Pages** > **Rotate**	`doc.setPageRotations`
Document > **Pages** > **Set Page Transitions**	`app.defaultTransitions` (read-only), `doc.setPageTransitions`
Document > **Pages** > **Number Pages**	`doc.setPageLabels`
Document > **Add Watermark & Background**	`doc.addWatermarkFromText`, `doc.addWatermarkFromFile`
Edit > **Add Bookmark**	`bookmark.createChild`

If a document is rights-enabled but commenting is not allowed, then the JavaScript methods shown below in Table 13.3 will be disabled:

TABLE 13.3   ***When Commenting is Not Allowed in Reader-Enabled Documents***

Feature	Method
Add a comment	`doc.addAnnot`
Import comments	`doc.importAnFDF`
Export comments	`doc.exportAsFDF` (when `bAnnotations` is set to `true`)

If a document is rights-enabled but file attachments are not allowed, then the following JavaScript methods will be disabled:

- `doc.createDataObject`
- `doc.importDataObject`
- `doc.setDataObjectContents`
- `doc.removeDataObject`

If a document is rights-enabled but digital signatures are not allowed, then the following JavaScript methods will be disabled:

- `doc.getField` (for signature fields)
- `doc.addField` (when `cFieldType = "signature"`)
- `field.removeField` (when `cFieldType = "signature"`)
- `field.signatureSign`

Enabling Collaboration

By using Really Simple Syndication (RSS), collaboration servers can provide customized XML-based user interfaces directly inside of Acrobat itself, thus providing a more dynamic and personalized tool, and providing JavaScript developers a means to extend collaboration, including full user interfaces.

In addition, it is now straightforward to migrate comments from one document to another, carry comments across multiple versions of a document, and anchor comments to content so that the annotations remain in the right place even if the content changes.

The advantages of this are that it is possible to automate discovery of collaboration servers, initiation workflows, and RSS feeds which may be used to populate content inside Adobe Reader.

It is significant to note that users of Acrobat Professional can enable both file-based and online collaboration, thus enabling them to invite users of Adobe Reader to participate in the review process.

The following JavaScript methods will be enabled in Adobe Reader when collaboration is enabled:

- `doc.addAnnot`
- `doc.importAnFDF`
- `doc.exportAnFDF`

When collaboration is not enabled, it is still possible for annotations to appear in a browser by embedding the following statement in the FDF file:

```
Collab.showAnnotToolsWhenNoCollab = true;
```

A complete example of an FDF file that makes it possible for annotations to appear in a browser is shown below:

```
%FDF-1.2
%âãÏÓ
1 0 obj
<<
/FDF
<<
/F (file:///C:/ReaderAnnots/seafood_wallet_re.pdf)
/JavaScript
<<
/AfterPermsReady 2 0 R
>>
>>
>>
endobj
2 0 obj
<<
>>
stream
app.alert("DocumentOpen Script Start");
Collab.showAnnotToolsWhenNoCollab = true;
endstream
endobj
trailer
<<
/Root 1 0 R
>>%%EOF
```

14 Interacting with Databases

Introduction

It is possible to use Acrobat JavaScript to interact with databases through an Open Database Connectivity (ODBC) connection. This means that you may use Acrobat JavaScript objects to connect to a database, retrieve tables, and execute queries. The object model associated with database interaction is centered on the **ADBC** object, which provides an interface to the ODBC connection. The **ADBC** object interacts with other objects to facilitate database connectivity and interaction: **DataSourceInfo**, **connection**, **statement**, **Column**, **ColumnInfo**, **row**, and **TableInfo**. These objects may be used in document-level scripts to execute database queries.

Chapter Goals

At the end of this chapter, you will be able to:

- Obtain a list of accessible databases.
- Connect to a database.
- Execute an SQL query.
- Access table, row, and column properties and data.

Contents

Topics
Introduction to ADBC
Establishing an ADBC Connection
Executing SQL Statements

Introduction to ADBC

Acrobat JavaScript provides an ODBC-compliant object model called Acrobat Database Connectivity (ADBC), which can be used in document-level scripts to connect to a database for the purposes of inserting new information, updating existing information, and deleting database entries. ADBC provides a simplified interface to ODBC, which it uses to establish a connection to a database and access its data, and supports the usage of SQL statements for data access, update, deletion, and retrieval.

Thus, a necessary requirement to the usage of ADBC is that ODBC must be installed on a client machine running a Microsoft Windows operating system. In addition, ADBC does not provide security measures with respect to database access; it is assumed that the database administrator will establish and maintain the security of all data.

The Acrobat JavaScript **ADBC** object thus provides methods through which you can obtain a list of accessible databases and form a connection with one of them. These methods are called **getDataSourceList** and **newConnection**. In addition, the **ADBC** object provides a number of properties corresponding to all supported SQL and Acrobat JavaScript data types, which include representations of numeric, character, time, and date formats.

Establishing an ADBC Connection

There are normally two steps required to establish an ADBC connection. First, obtain a list of accessible databases by invoking the **ADBC** object's **getDataSourceList** method. Then establish the connection by passing the Data Source Name (DSN) of one of the databases in the list to the **ADBC** object's **newConnection** method.

The **getDataSourceList** method returns an array of **DataSourceInfo** generic objects, each of which contains the following properties:

● **name**: a string identifying the database

● **description**: a description containing specific information about the database

In the following example, a list of all available databases is retrieved, and the DSN of the **DataSourceInfo** object representing **Q32000Data** is identified and stored in variable **DB**:

```
// Obtain a list of accessible databases:
var databaseList = ADBC.getDataSourceList();

// Search the DataSourceInfo objects for the "Q32000Data" database:
if (databaseList != null) {
    var DB = "";
    for (var i=0; i<databaseList.length; i++)
        if (databaseList[i].name == "Q32000Data") {
            DB = databaseList[i].name;
            break;
        }
}
```

To establish the database connection, invoke the **ADBC** object's **newConnection** method, which accepts the following parameters:

● **cDSN**: the Data Source Name (DSN) of the database

● **cUID**: the user ID

● **cPWD**: the password

The **newConnection** method returns a **connection** object, which encapsulates the connection by providing methods which allow you to create a **statement** object, obtain information about the list of tables in the database or columns within a table, and close the connection.

In the following example, a connection is established with the **Q32000Data** database:

```
if (DB != "") {
    // Connect to the database and obtain a Connection object:
    var myConnection = ADBC.newConnection(DB.name);
}
```

The **connection** object provides the methods shown below in Table 14.1:

TABLE 14.1   *Connection Object*

Method	Description
close	Closes the database connection
newStatement	Creates a **statement** object used to execute SQL statements
getTableList	Retrieves information about the tables within the database
getColumnList	Retrieves information about the various columns within a table

The **connection** object's **getTableList** method returns an array of **TableInfo** generic objects, each of which corresponds to a table within the database and contains the following properties:

- **name**: the table name

- **description**: a description of database-dependent information about the table

In the following example, the name and description of every table in the database is printed to the console:

```
// Obtain the array of TableInfo objects representing the database tables:
var tableArray = myConnection.getTableList();

// Print the name and description of each table to the console:
for (var i=0; i<tableArray.length; i++) {
    console.println("Table Name: " + tableArray[i].name);
    console.println("Table Description: " + tableArray[i].description);
}
```

The **connection** object's **getColumnList** method accepts a parameter containing the name of one of the database tables, and returns an array of **ColumnInfo** generic objects, each of which corresponds to a column within the table and contains the following properties:

- **name**: the name of the column

- **description**: a description of database-dependent information about the column

- **type**: a numeric identifier representing an **ADBC** SQL type

- **typeName**: a database-dependent string representing the data type

In the following example, a complete description of every column in the **Q32000Data** database table called **Sales** is printed to the console:

```
// Obtain the array of ColumnInfo objects representing the Sales table:
var columnArray = myConnection.getColumnList("Sales");

// Print a complete description of each column to the console:
for (var i=0; i<columnArray.length; i++) {
    console.println("Column Name: " + columnArray[i].name);
    console.println("Column Description: " + columnArray[i].description);
    console.println("Column SQL Type: " + columnArray[i].type);
    console.println("Column Type Name: " + columnArray[i].typeName);
}
```

Executing SQL Statements

To execute SQL statements, first create a **statement** object by invoking the **connection** object's **newStatement** method. The newly created **statement** object can be used to access any row or column within the database table and execute SQL commands.

In the following example, a **statement** object is created for the **Q32000Data** database created in the previous sections:

```
myStatement = myConnection.newStatement();
```

The **statement** object provides the methods shown below in Table 14.2:

TABLE 14.2   Statement Object

Method	Description
execute	Executes an SQL statement
getColumn	Obtains a column within the table
getColumnArray	Obtains an array of columns within the table
getRow	Obtains the current row in the table
nextRow	Iterates to the next row in the table

In addition to the methods shown above in Table 14.2, the **statement** object provides two useful properties:

- **columnCount**: the number of columns in each row of results returned by a query

- **rowCount**: the number of rows affected by an update

To execute an SQL statement, invoke the **statement** object's **execute** method, which accepts a string parameter containing the SQL statement. Note that any names containing spaces must be surrounded by escaped quotation marks, as shown in the following example:

```
// Create the SQL statement:
var SQLStatement = 'Select * from \"Client Data\"';

// Execute the SQL statement:
myStatement.execute(SQLStatement);
```

There are two steps required to obtain a row of data. First, invoke the **statement** object's **nextRow** method; this makes it possible to retrieve the row's information. Then, invoke the **statement** object's **getRow** method, which returns a **Row** generic object representing the current row.

In the example shown below, the first row of information will be displayed in the console. Note that the syntax is simplified in this case because there are no spaces in the column names:

```
// Create the SQL statement:
var st = 'Select firstName, lastName, ssn from \"Employee Info\"';

// Execute the SQL statement:
myStatement.execute(st);

// Make the next row (the first row in this case) available:
myStatement.nextRow();

// Obtain the information contained in the first row (a Row object):
var firstRow = myStatement.getRow();

// Display the information retrieved:
console.println("First name: " + firstRow.firstName.value);
console.println("Last name: " + firstRow.lastName.value);
console.println("Social Security Number: " + firstRow.ssn.value);
```

If the column names contain spaces, the syntax can be modified as shown below:

```
// Create the SQL statement:
var st = 'Select \"First Name\", \"Last Name\" from \"Employee Info\"';

// Execute the SQL statement:
myStatement.execute(st);

// Make the next row (the first row in this case) available:
myStatement.nextRow();

// Obtain the information contained in the first row (a Row object):
var firstRow = myStatement.getRow();

// Display the information retrieved:
console.println("First name: " + firstRow["First Name"].value);
console.println("Last name: " + firstRow["Last Name"].value);
```

15 SOAP and Web Services

Introduction

The basis for most current models of web services is the Extensible Markup Language (XML), which facilitates interoperability by permitting the exchange of information between applications and platforms in a text-based manner. The Simple Object Access Protocol (SOAP) is an XML-based messaging protocol used as a medium for information and instruction exchange between web services. Information sent using the SOAP standard is placed in a file called a SOAP envelope. The Web Services Description Language (WSDL) is an XML-based standard for describing web services and their capabilities. Acrobat's support for these standards makes it possible for PDF documents to locate services, understand their capabilities, and interact with them.

In order to guarantee interoperability between various platforms and applications, it is necessary to exchange information that adheres to a standardized format for data types. To address this need, the XML Schema Definition Language (XSD) provides a standard set of definitions for the various data types that can be exchanged in SOAP envelopes.

Acrobat 7.0 provides support for the SOAP 1.1 and 1.2 standards in order to enable PDF forms to communicate with web services. This support has made it possible to include both SOAP header and body information, send binary data more efficiently, use document/literal encoding, and facilitate authenticated or encrypted communications. In addition, it also provides the ability to locate network services using DNS Service Discovery. All of this makes it possible to integrate PDF files into existing workflows by binding XML forms to schemas, databases, and web services. These workflows include the ability to share comments remotely or invoke web services through form field actions.

Chapter Goals

At the end of this chapter, you will be able to:

- Understand the support available for the SOAP standards.
- Connect to a web service.
- Exchange information with a web service.
- Search and manage XML-based information.
- Use DNS Service Discovery to locate network services.
- Understand how to apply Acrobat's SOAP support to various workflows.

Contents

Topics
Using SOAP and Web Services
DNS Service Discovery
Managing XML-based Information
Workflow Applications

Using SOAP and Web Services

Acrobat JavaScript provides a **SOAP** object that encapsulates the support for web services and service discovery. Through the usage of this object, you can extend and customize your workflows by engaging in XML-based communication with remote servers.

The **SOAP** object has a **wireDump** property that sends all XML requests and responses to the JavaScript Console for debugging purposes. In addition, the **SOAP** object provides the methods described below in Table 15.1:

TABLE 15.1   *SOAP Object*

Method	Description
connect	Obtains a WSDL proxy object used to invoke a web service
queryServices	Locates network services using DNS Service Discovery
resolveService	Binds a service name to a network address and port
request	The principal method used to invoke a web service
response	A callback method used in asynchronous web method calls
streamDecode	Decodes a Base64 or Hex stream object
streamEncode	Applies Base64 or Hex encoding to a stream object
streamFromString	Converts a string to a stream object
stringFromStream	Converts a stream object to a string

SOAP and Web Services Topics

Using a WSDL Proxy to Invoke a Web Service

When connecting to a web service, your Acrobat JavaScript code may use a WSDL proxy object to generate a SOAP envelope. The envelope contains a request for the server to run a web method on behalf of the client. To obtain the WSDL proxy object, invoke the **SOAP** object's `connect` method, which accepts a single parameter containing the HTTP or HTTPS URL of the WSDL document.

The returned proxy object contains the web methods available in the web service described by the WSDL document. If the web service uses SOAP/RPC encoding, the parameters in the proxy object's methods will match the order specified in the WSDL document. If the web service uses document/literal encoding, the methods will accept a single parameter describing the request message.

In the example shown below, a connection to a web service will be established, and its RPC-encoded web methods will be invoked. Assume that `myURL` contains the address of the WSDL document:

```
// Obtain the WSDL proxy object:
var myProxy = SOAP.connect(myURL);

// Invoke the echoString web service, which requires a string parameter:
var testString = "This is a test string.";
var result = myProxy.echoString(testString);

// Display the response in the console:
console.println("Result is " + result);

// Invoke the echoInteger web service, which requires an integer parameter:
// Since JavaScript does not support XSD-compliant integer values,
```

```
// We will create an integer object compliant with the XSD standard:
var myIntegerObject = {
    soapType: "xsd:int",
    soapValue: "10"
};
var result = myProxy.echoInteger(myIntegerObject);

// Display the response in the console:
console.println("Result is " + result);
```

Note that each call to a web method generates a SOAP envelope that is delivered to the web service, and that the return value is extracted from the corresponding envelope returned by the web service. Also, since XML relies on text, there is no problem sending a string to the web service. In the case of integers, however, it is necessary to create an XSD-compliant object to represent the integer value. Acrobat JavaScript does support some of the standard data types specified in the XSD. These are shown below in Table 15.2:

TABLE 15.2   *XSD-Compliant Data Types Supported in JavaScript*

JavaScript Type	Equivalent XSD-Compliant Type
`String`	`xsd:string`
`Number`	`xsd:float`
`Date`	`xsd:dateTime`
`Boolean`	`xsd:boolean`
`ReadStream`	`SOAP-ENC:base64`
`Array`	`SOAP-ENC:Array`

Synchronous and Asynchronous Information Exchange

The **SOAP** object's **request** method may be used to establish either synchronous or asynchronous communication with a web service, and provides extensive support for SOAP header information, firewall support, the type of encoding used, namespace qualified names, compressed or uncompressed attachments, the SOAP protocol version to be used, authentication schemes, response style, and exception handling.

The **request** method accepts the parameters shown below in Table 15.3:

TABLE 15.3   Request Method

Parameter	Description
cURL	URL for SOAP HTTP endpoint
oRequest	Object containing RPC information
oAsync	Object used for asynchronous method invocation
cAction	SOAPAction header used by firewalls and servers to filter requests
bEncoded	Indicates whether SOAP encoding is used
cNamespace	Message schema namespace when SOAP encoding is not used
oReqHeader	SOAP header to be included with request
oRespHeader	SOAP header to be included with response
cVersion	SOAP protocol version to be used (1.1 or 1.2)
oAuthenticate	Authentication scheme
cResponseStyle	The type and structure of the response information (JS, XML, Message)

Establishing a Synchronous Connection

The **SOAP** object's **request** method may be used to establish a synchronous connection with a web service. To establish the connection and invoke the web methods, it is necessary to provide the **cURL**, **oRequest**, and **cAction** parameters. The example below demonstrates how to invoke the same web services used in the previous example.

Similar to the parameter used in the **connect** method, the **cURL** parameter contains the URL for the WSDL document. For the purposes of our example, assume that **myURL** represents the WSDL document location.

The **oRequest** parameter is a fully qualified object literal specifying both the web method name and its parameters, in which the namespace is separated from the method name by a colon. It may also contain the following properties:

- **soapType**: the SOAP type used for the value

- **soapValue**: the SOAP value used when generating the message

- **soapName**: the element name used when generating the SOAP message

- **soapAttributes**: an object containing the XML attributes in the request node

- **soapQName**: the namespace-qualified name of the request node

- **soapAttachment**: determines whether **soapValue** is encoded as an attachment according to the SwA/MTOM® specification. In this case, **soapValue** will be a stream.

Assume that the namespace is **http://mydomain/methods**, the method name is **echoString**, and it accepts a string parameter called **inputString**. The following code represents the **oRequest** parameter:

```
var echoStringRequest = {
    "http://mydomain/methods:echoString {
        inputString: "This is a test."
    }
};
```

The **cAction** parameter contains header information described by the WSDL document that is used by firewalls to filter SOAP requests. In our example, we will supply the location of the WSDL document:

```
var mySOAPAction = "http://mydomain/methods";
```

We may now invoke the **echoString** web method:

```
var response = SOAP.request ({
    cURL: myURL,
    oRequest: echoStringRequest,
    cAction: mySOAPAction
});
```

In the case of synchronous requests such as this one, the value returned by the **request** method (**response** in this example) is an object containing the result of the web method, which will be one of the Acrobat JavaScript types corresponding to XSD types. The default format for the response value is an object describing the SOAP body (or header) of the returned message.

NOTE: In the case of base64 or hex encoding of binary information, the type returned will be a **readStream** object.

We may now obtain the returned value by using syntax that corresponds to the SOAP body sent back by the web method:

```
var responseString = "http://mydomain/methods:echoStringResponse";
var result = response[responseString]["return"];

// Display the response in the console:
console.println("Result is " + result);
```

Similarly, we can invoke the **echoInteger** method. To do this, we will use the same value for the **cURL** and **cAction** parameters, and develop an **oRequest** parameter like the one we used for the **echoString** method. In this case, however, we must supply an XSD-compliant object to represent the integer value:

```
var myIntegerObject = {
    soapType: "xsd:int",
    soapValue: "10"
};

var echoIntegerRequest = {
    "http://mydomain/methods:echoInteger {
        inputInteger: myIntegerObject
    }
};
```

Now we may invoke the **echoInteger** web method and display its results:

```
var response = SOAP.request ({
    cURL: myURL,
    oRequest: echoIntegerRequest,
    cAction: mySOAPAction
});

var responseString = "http://mydomain/methods:echoIntegerResponse";
var result = response[responseString]["return"];

// Display the response in the console:
console.println("Result is " + result);
```

Asynchronous Web Service Calls

The **SOAP** object's **request** method may be used in conjunction with the **response** method to establish asynchronous communication with a web service. In this case, the **request** method calls a method on the notification object, and does not return a value.

Asynchronous communication is made possible by assigning a value to the **request** method's **aSync** parameter, which is an object literal that must contain a function called **response** that accepts two parameters: **oResult** (the result object) and **cURI** (the URI of the requested HTTP endpoint).

In the example below, the **aSync** parameter named **mySync** contains an attribute called **isDone**, which is used to monitor the status of the web service call, and an attribute called **val** which will contain the result of the web service call. When the **response** function is called by the web service, it sets **isDone** to **true** indicating that the asynchronous call has been completed:

```
// Create the aSync parameter:
var mySync = {
    isDone: false,
    val: null,

    // Generates the result of the web method:
    result: function(cMethod)
    {
        this.isDone = false;
        var name = "http://mydomain/methods/:" + cMethod + "Response";

        if (typeof this.val[name] == "undefined")
            return null;
        else
            return this.val[name]["return"];
    },

    // The method called by the web service after completion:
    response: function(oResult, cURI)
    {
        this.val = oResult;
        this.isDone = true;
    },

    // While the web service is not done, do something else:
    wait: function()
    {
        while (!this.isDone) doSomethingElse();
    }
};
```

```
// Invoke the web service:
SOAP.request({
    cURL: myURL,
    oRequest: echoIntegerRequest,
    oAsync: mySync,
    cAction: mySOAPAction
});


// The response callback function could contain the following code:

// Handle the asynchronous response:
var result = mySync.result("echoInteger");

// Display the response in the console:
console.println("Result is " + result);
```

Using Document/Literal Encoding

You may use document/literal encoding in your SOAP messages by assigning values to the following parameters of the **request** method:

● **bEncoded**: Assign a value of **false** to this parameter.

● **cNamespace**: Specify a namespace for the message schema.

● **cResponseStyle**: Assign **SOAPMessageStyle.JS**, **SOAPMessageStyle.XML**, or **SOAPMessageStyle.Message** value to this parameter.

Once this is done, fill the **oRequest** parameter with an object literal containing the data. An example is given below:

```
// Set up two integer values to be added in a web method call:
var aInt = {soapType: "xsd:int", soapValue: "10"};
var bInt = {soapType: "xsd:int", soapValue: "4"};

// Set up the document literal:
var req = {};
req["Add"] = {a: aInt, b: bInt};

// Invoke the web service:
var response = SOAP.request({
    cURL: myURL,
    oRequest: req,
    cAction: mySOAPAction,
    bEncoded: false,
    cNamespace: myNamespace,
    cResponseStyle: SOAPMessageStyle.Message
});
// Display the response to the console:
var value = response[0].soapValue[0].soapValue;
console.println("ADD(10 + 4) = " + value);
```

Exchanging File Attachments and Binary Data

As you learned earlier, the **oRequest** parameter provides alternative options for sending binary-encoded data. This may be useful for sending information such as serialized object data or embedded images. You may embed binary information in text-based format in the SOAP envelope by using base64 encoding, or take advantage of the Soap With Attachments® (SwA) standard or the Message Transmission Optimization Mechanism® (MTOM) to send the binary data in a more efficient format. Both SwA and MTOM can significantly reduce network transmission time, file size, and XML parsing time.

SwA can be used by setting the **oRequest** parameter's **soapAttachment** value to **true**, as shown in the example below. Assume **myStream** is a **readStream** object containing binary data:

```
// Use the SwA standard:
var SwARequest = {
    "http://mydomain/methods:echoAttachment": {
        dh:
        {
            soapAttachment: true,
            soapValue: myStream
        }
    }
};

var response = SOAP.request ({
    cURL: myURL,
    oRequest: SwARequest
});

var responseString = "http://mydomain/methods:echoAttachmentResponse";
var result = response[responseString]["return"];
```

MTOM is used by additionally setting the **request** method's **bEncoded** parameter to **false** and the **cNamespace** parameter to an appropriate value. This is illustrated in the following code, which creates an **oRequest** object:

```
// Use the MTOM standard:
var MTOMRequest = {
    "echoAttachmentDL": {
        dh:
        {
            inclusion:
            {
                soapAttachment: true,
                soapValue: myStream
            }
        }
    }
};

var response = SOAP.request({
    cURL: myURL,
    oRequest: MTOMRequest,
    bEncoded: false,
    cNamespace: myNamespace
});
```

Converting Between String and ReadStream Information

The **SOAP** object's **streamFromString** and **stringFromStream** methods are useful for converting between formats. The **streamFromString** method is useful for submitting data in a web service call, and the **stringFromStream** method is useful for examining the contents of a response returned by a web service call. An example is shown below:

```
// Create a ReadStream object from an XML string:
var myStream = streamFromString("<mom name = 'Mary'></mom>");

// Place the information in an attachment:
this.setDataObjectContents("org.xml", myStream);

// Convert the ReadStream object back to a string and display in console:
console.println(stringFromStream(myStream));
```

Accessing SOAP Version Information

Acrobat 7.0 provides improved support for SOAP Version 1.1 and support for Version 1.2. To encode the message using a specific version, assign one of the following values to the **request** method's **cVersion** parameter: **SOAPVersion.version_1_1** (SOAP Version 1.1) or **SOAPVersion.version_1_2** (SOAP Version 1.2). Its usage is shown in the following example:

```
var response = SOAP.request ({
    cURL: myURL,
    oRequest: myRequest,
    cVersion: SOAPVersion.version_1_2
});
```

Accessing SOAP Header Information

You may send SOAP header information to the web service using the **request** method's **oReqHeader** parameter, and access the returned header information using the **oRespHeader** parameter. The **oReqHeader** is identical to the **oRequest** object, with the addition of two attributes:

● **soapActor**: the SOAP actor that should interpret the header

● **soapMustUnderstand**: determines whether the SOAP actor must understand the header contents

Their usage is shown in the following example:

```
// Set up the namespace to be used:
var myNamespace = "http://mydomain/methods/:";

// Create the oReqHeader parameter:
var sendHeader = {};
sendHeader[myNamespace + "testSession"] = {
    soapType: "xsd:string",
    soapValue: "Header Test String"
};

// Create the intially empty oRespHeader parameter:
var responseHeader = {};
responseHeader[myNamespace + "echoHeader"] = {};

// Exchange header information with the web service:
var response = SOAP.request({
    cURL: myURL,
    oRequest: {},
    cAction: "http://mydomain/methods",
    oReqHeader: sendHeader,
    oRespHeader: responseHeader
});
```

Authentication

You may use the **request** method's **oAuthenticate** parameter to specify how to handle HTTP authentication or provide credentials used in Web Service Security (WS-Security). Normally, if authentication is required, an interface will handle HTTP authentication challenges for BASIC and DIGEST authentication using the SOAP header, thus making it possible to engage in encrypted or authenticated communication with the web service. This parameter helps to automate the authentication process.

The **oAuthenticate** parameter contains two properties:

● **Username**: A string containing the username

● **Password**: A string containing the authentication credential

Its usage is shown in the following example:

```
// Create the oAuthenticate object:
var myAuthentication = {
    Username: "myUsername",
    Password: "myPassword"
};

// Invoke the web service using the username and password:
var response = SOAP.request ({
    cURL: myURL,
    oRequest: echoStringRequest,
    cAction: mySOAPAction
    oAuthenticate: myAuthentication
});
```

Error Handling

The **SOAP** object provides extensive error handling capabilities within its methods. In addition to the standard Acrobat JavaScript exceptions, the **SOAP** object also provides **SOAPError** and **NetworkError** exceptions.

A **SOAPError** exception is thrown when the SOAP endpoint returns a SOAPFault. The **SOAPError** exception object will include information about the SOAP Fault code, the SOAP Actor, and the details associated with the fault. The **SOAP** object's **connect** and **request** methods support this exception type.

A **NetworkError** exception is thrown when there is a problem with the underlying HTTP transport layer or in obtaining a network connection. The **NetworkError** exception object will contain a status code indicating the nature of the problem. The **SOAP** object's **connect**, **request**, and **response** methods support this exception type.

DNS Service Discovery

Suppose the exact URL for a given service is not known, but that it is available locally because it has been published using *DNS Service Discovery* (DNS-SD). You may use the **SOAP** object's `queryServices` and `resolveService` methods to locate the service on the network and bind to it for communications.

The `queryServices` method can locate services that have been registered using Multicast DNS (mDNS) for location on a local network link, or through unicast DNS for location within an enterprise. The method performs an asynchronous search, and the resultant service names can be subsequently bound to a network location or URL through the **SOAP** object's `resolveService` method.

The `queryServices` method accepts the following parameters:

● `cType`: The DNS SRV service name (such as "http" or "ftp")

● `oAsync`: A notification object used when services are located or removed (implements `addServices` and `removeServices` methods). The notification methods accept a parameter containing the following properties:
  – `name`: the unicode display name of the service
  – `domain`: the DNS domain for the service
  – `type`: the DNS SRV service name (identical to `cType`)

● `aDomains`: An array of domains for the query. The valid domains are `ServiceDiscovery.local` (searches the local networking link using mDNS) and `ServiceDiscovery.default` (searches the default DNS domain using unicast DNS).

An example of its usage is shown below:

```
// Create the oAsync notification object:
var myNotifications = {
    // This method is called whenever a service is added:
    addServices: function(services)
    {
        for (var i=0; i<services.length; i++) {
            var str = "ADD: ";
            str += services[i].name;
            str += " in domain ";
            str += services[i].domain;
            console.println(str);
        }
    }

    // This method is called whenever a service is removed:
    removeServices: function(servces)
    {
            var str = "DEL: ";
            str += services[i].name;
            str += " in domain ";
            str += services[i].domain;
            console.println(str);
    }
};


// Perform the service discovery:
SOAP.requestServices({
    cType: "http",
    oAsync: myNotifications,
    aDomains: [ServiceDiscovery.local, ServiceDiscover.default]
});
```

Once a service has been discovered, it can be bound through the **SOAP** object's **resolveService** method to a network address and port so that a connection can be established. The **resolveService** method accepts the following parameters:

● **cType**: the DNS SRV service name (such as "http" or "ftp").

● **cDomain**: the domain in which the service is located.

● **cService**: the service name to be resolved.

● **oAsync**: a notification object used when the service is resolved. It implements a **resolve** method that accepts parameters **nStatus** (**0** if successful) and **oInfo** (used if successful, contains a **serviceInfo** object). The **serviceInfo** object contains the following properties:

  – **target**: the IP address or DNS name of the machine providing the service.

  – **port**: the port on the machine.

  – **info**: an object with name/value pairs supplied by the service.

Its usage is illustrated in the following example:

```
// Create the oAsync notification object:
var myNotification = {
    // This method is called when the service is bound:
    resolve: function(nStatus, oInfo)
    {
        // Print the location if the service was bound:
        if (nStatus == 0){
            var str = "RESOLVE: http://";
            str += oInfo.target;
            str += ":";
            str += oInfo.port;
            str += "/";
            str += oInfo.info.path;
            console.println(str);
        }

        // Display the error code if the service was not bound:
        else
            console.println("ERROR: " + nStatus);
    }
};

// Attempt to bind to the service:
SOAP.resolveService({
    cType: "http",
    cDomain: "local.",
    cService: "My Web Server",
    oAsync: myNotification
});
```

Managing XML-based Information

Acrobat JavaScript provides support for XML-based information generated within your workflows by providing an **XMLData** object, which represents an XML document tree that may be manipulated via the XFA Data DOM. (For example, it is possible to apply an XSL transformation (XSLT) to a node and its children using the **XFA** object). The **XMLData** object provides two methods for manipulating XML documents:

● **parse**: Creates an object representing an XML document tree.

● **applyXPath**: Permits the manipulation and query of an XML document via XPath expressions.

You may convert a string containing XML information into a document tree using the **XML** object's **parse** method, and then manipulate and query the tree using its **applyXPath** method.

The **XML** object's **parse** method accepts two parameters:

- **param1**: A string containing the text in the XML document.

- **param2**: A boolean that determines whether the root node should be ignored.

Its usage is illustrated below:

```
// XML string to be converted into a document tree:
myXML = "<family name = 'Robat'>\
          <mom id = 'm1' name = 'Mary' gender = 'F'>\
             <child> m2 </child>\
             <personal>\
                 <income>75000</income>\
             </personal>\
          </mom>\
          <son id = 'm2' name = 'Bob' gender = 'M'>\
             <parent> m1 </parent>\
             <personal>\
                 <income>35000</income>\
             </personal>\
          </son>\
       </family>";

// Generate the document tree:
var myTree = XML.parse(myXML, false);

// Print mom's name:
console.println("Mom: " + myXML.family.mom.value);

// Change son's income:
myXML,family.son.personal.income.value = 40000;
```

The **XML** object's **applyXPath** method accepts two parameters:

- **oXML**: An object representing the XML document tree.

- **cXPath**: A string containing an XPath query to be performed on the document tree.

In the following example, assume that **myXML** is the document tree obtained in the previous example:

```
// Obtain mom's information:
var momInfo = XML.applyXPath(myXML, "//family/mom");

// Save the information to a string:
momInfo.saveXML('pretty');

// Give mom a raise:
momInfo.personal.income.value = "80000";
```

Workflow Applications

One major impact that Acrobat JavaScript support for SOAP has is on collaboration workflows. A SOAP-based collaboration server can be used to share comments remotely via a Web-based comment repository. Through the DNS Service Discovery support, it is possible to enable dynamic discovery of collaboration servers, initiation workflows, and RSS feeds that can provide customized user interfaces, via XML, directly inside of Acrobat 7.0.

In addition, it is possible to deploy Web-based JavaScripts that always maintain the most updated information and processes, and to connect to those scripts via form field actions that invoke web services.

In the following example, a form is submitted to a server using a SOAP-based invocation:

```
// Populate the content object with form and SOAP information:
var location = "http://adobe.com/Acrobat/Form/submit/"
var formtype = location + "encodedTypes:FormData";
var content = new Array();
for(var i = 0; i < document.numFields; i++) {
        var name = document.getNthFieldName(i);
        var field = document.getField(name);
        content[i] = new Object();
        content[i].soapType = formtype;
        content[i].name = field.name;
        content[i].val = field.value;
}

// Send the form to the server:
    SOAP.request({
        cURL: cURL,
        oRequest: {
            location + ":submitForm":
            {
                content: content
            }
        },
        cAction: location + "submitForm"
    }
```

A **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, and *field* level, and *batch* level. These restrict the type of processing that can occur and are loaded at different times.

How should I name my Acrobat form fields?

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

Acrobat form field names can be more useful by creating a hierarchal structure. For example, if we change **FirstName** to **Name.First** and **LastName** to **Name.Last** we form a tree of fields. The period (".") separator is 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 to which 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 form applications when there are many fields in the document.

Using our original flat names **FirstName**, **MiddleName** and **LastName**, you can change the background color of these fields to yellow (e.g. to indicate missing data or emphasize an important point). In this case, two lines of JavaScript code would be needed 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;
```

---

1. Frequently Asked Questions

With the hierarchy of **Name.First**, **Name.Middle** and **Name.Last** suggested above, you 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, you can only change appearance-related properties of the parent field, and those changes will propagate to all its children. However, 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** methods that process dates. JavaScript **Date** objects actually contain both a date and a 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.

NOTE: When using the **Date** object, do not use the **getYear()** method, which returns the current year minus 1900. Instead use the **getFullYear()** method which always returns a four digit year. For example,

```
var d = new Date()
d.getFullYear();
```

Converting from a Date to a String

Acrobat 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's ability to handle dates in many formats, the **Date** object does not always handle these conversions correctly.

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

```
/* Example of util.printd */
var d = new Date(); // Create a Date object containing the current date
/* Create some strings from the Date object with various formats with
** util.printd */
var 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: 01/15/05
Format yy/mm/dd looks like: 05/1/15
Format mmmm dd, yyyy looks like: January 15, 2005
Format dd-mmm-yyyy looks like: 15-Jan-2005
```

NOTE: 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, use the **Util** object's **scand** method. 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, in milliseconds. */
/* 1000 ms/sec, 60 sec/min, 60 min/hour, 24 hours/day, 13 days */
num += 1000 * 60 * 60 * 24 * 13;
/* Create our new date, 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 restricted Acrobat JavaScript methods available to users?

Many of the methods in Acrobat JavaScript are restricted for security reasons, and their execution is only allowed during batch, console or menu events. The *Acrobat JavaScript Scripting Reference* identifies these methods by a padlock symbol in the quick bar. This restriction is a limitation when enterprise customers try to develop solutions that require these methods and know that their environment is secure.

Three requirements must be met to make restricted Acrobat JavaScript methods available to users.

● You must obtain a digital ID.

● You must sign the PDF document containing the restricted JavaScript methods using the digital ID.

For details on where you can obtain digital IDs and the procedures for using them to sign documents, see Acrobat Help.

● The recipient should trust the signer for certified documents and JavaScript.

For details, see Acrobat Help.

All trusted certificates can be accessed by selecting Certificates from **Advanced > Manage Digital IDs > Trusted Identities** in Acrobat's main menu.

How can I make my documents accessible?

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 following is a set of guidelines to follow in order to make a form minimally accessible.

Document Metadata

The metadata for a document can be specified using **File > Document Properties > Description** or **Advanced > Document Metadata**.

When a document is opened, saved, printed, or closed, the document title is spoken to the user. If the title has not been specified in the **Document Metadata**, then the file name is used. Often, file names are abbreviated or changed, so it is advised 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 2004".

In addition, third-party screen readers usually read the title in the window title bar. You can specify what appears in the window title bar by using **File > Document Properties > Initial View** and in the Window Options, choose to **Show** either the **File Name** or **Document Title**.

Filling all of the additional metadata associated with a document (**Author**, **Subject**, **Keywords**) also makes it more easily searchable using **Acrobat Search** and Internet search engines.

Short Description

Every field that is not hidden must contain a user-friendly name (tooltip). 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`". The name should not depend on the surrounding context. For instance, if both the main section and spouse section of a document contain a "`First Name`" field, the field in the spouse section might be named "`Spouse's First Name`". This description is also displayed as a tooltip when the user positions the mouse over the field.

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.

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.

Reading Order

The reading order of a document is determined by the Tags tree. In order for a form to be used effectively by a visually impaired user, the content and fields of a page must be included in the Tags tree. The Tags tree can also indicate the tab order for the fields on a page.

How can I define global variables 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 variable 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 Global Variables Persistent

Global data does not persist across user sessions unless you specifically make your global varaibles persistent. The predefined **Global** object has a method designed to do this. To make a variable named '**myVariable**' persist across sessions, use the following syntax:

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

In future sessions, the variable will still exist with its previous value intact.

How can I hide an Acrobat form field based on the value of another?

Use the **display** method of the **Global** 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 an Acrobat form field value in another open form?

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 may 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 is 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 in Acrobat forms?

Create a Custom Keystroke Filter script (see the Format tab in the **Properties** dialog for any text field or combo box) 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 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** defined in the **App** object. 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 clicks **Cancel**, 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 **doc** object. 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 determine if the mouse has entered/left a certain area on an Acrobat form?

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.

How can I disallow changes in scripts contained in my document?

Go to **File > Document Properties** and select the **Security** tab. Set up either password or certificate security for the document by clicking **Security Method** and choosing either **Password Security** or **Certificate Security**. In the **Permissions** area of the dialog box that pops up, click **Changes Allowed** and select any of the options except **Any except extracting pages**. You can verify that changes to scripts have been disabled by returning to the **Security** tab. In the **Document Restrictions Summary** portion, **Changing the Document** should be set to **Not Allowed**.

How can I hide scripts contained in my document?

Go to **File > Document Properties** and select the **Security** tab. Set up either password or certificate security for the document by clicking **Security Method** and choosing either **Password Security** or **Certificate Security**. In the **Permissions** area of the dialog box that pops up, ensure that **Enable copying of text, images, and other content** is unchecked. You can verify that changes to scripts have been disabled by returning to the **Security** tab. In the **Document Restrictions Summary** portion, **Changing the Document** should be set to **Not Allowed**.

Index

A

Acrobat
  plug-in 14
Acrobat Database Connectivity 24, 236
Acrobat forms 87
  Acrobat form field 89
action types 179
ADBC 24, 235, 236
  getDataSourceList 236
  newConnection 236, 237
adding security to attachments 217
additional usage rights 124, 227
ADM 26, 159, 160
ADO 27, 114
Adobe Dialog Manager 26, 159
Adobe LiveCycle Designer 87, 108, 109
Adobe LiveCycle Policy Server 214
Adobe Reader-Extension Root Certificate
     Authority 228
Adobe.PPKLite 220
Adobe.PPKMS 220, 223
Adobe.PPKMS.ADSI 223
ADSI 223
Advanced Editing Toolbar 53
advanced search options 193
annot
  AP 129, 134, 137
  attachIcon 130
  destroy 136
  hidden 138
  setProps 133, 136, 137
  soundIcon 130
  transitionToState 137
annotations 20
app 22
  execDialog 162
  fullscreen 182
  hideMenuItem 183
  hideToolbarButton 183

newDoc 72
openDoc 69, 72, 139
openInPlace 175
printColorProfiles 79, 82
printerNames 79
app.media 145, 151, 182
  getPlayers 145, 151
  windowType 182
Approval 135, 140
approval stamp 209
Array 97
  sort 97
asynchronous communication 248

B

base64 encoding 250
BASIC 253
batch 139
binary data 250
bookmark
  color 181
  hierarchy 173
boolean queries 196
breakpoints 58
  conditional 59
  define 60
button 186
  buttonAlignX 95
  buttonAlignY 95
  buttonScaleHow 95
  buttonScaleWhen 95
  buttonSetIcon 186
  highlight 94, 181
  setIcon 95

C

calculation 101, 106
call stack 55

Index

Acrobat JavaScript Scripting Guide