Acrobat JavaScript Scripting Guide

Technical Note #5430

Version: Acrobat 6.0

ADOBE SYSTEMS INCORPORATED
Corporate Headquarters
345 Park Avenue
San Jose, CA 95110-2704
(408) 536-6000
http://partners.adobe.com

May 2003

Contents

Preface

Introduction

Welcome to the Adobe *Acrobat JavaScript Scripting Guide*. This scripting guide is designed to give you an overview of how you can use the Acobe Acrobat 6 Pro JavaScript development environment to develop and enhance Acrobat applications.

The JavaScript language was developed by Netscape Communications so you could more easily create interactive Web pages. Adobe has enhanced JavaScript so you can easily integrate this level of interactivity into your PDF forms. The most common uses for JavaScript in Acrobat forms are formatting data, calculating data, validating data, and assigning an action.

While there are plug-in, document, and field level JavaScripts, we are concerned only with document level and field level scripts here.

- For information on plug-in level scripts, see "Working with JavaScript actions" on page 60.
- Document level scripts are executed with the document open and apply only to this document.
- Field level scripts are associated with a specific form field or fields. This type of script is executed when an event occurs, such as a Mouse Up action.

Audience

The intended audience of this guide includes you if you are an Acrobat solution provider or power user of Acrobat. You are interested in developing solutions that leverage Adobe products. You are resourceful and accustomed to learning new technologies quickly, and with minimal hand holding, if you have access to the necessary knowledge resources. This guide is one of those resources.

Purpose and Scope

The objectives of this guide include:

- Introducing you to the Adobe Acrobat JavaScript functionality that supports developing and deploying Acrobat solutions.
- Providing you with easily understood, detailed information about Acrobat JavaScript scripting features and use.
- Providing 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'll likely find yourself reviewing the content of this guide, as well as exploring additional, more in-depth information that updates to this guide offer in key technical areas.

Assumptions

This guide assumes that you are familiar with the non-scripting elements of the Acrobat 6 user interface that are described in Acrobat's accompanying online help documentation. This guide provides some review but focuses primarily on the user interface elements for writing JavaScript code. You should, in addition, be familiar with the basics of the standard JavaScript scripting language. To work through the exercises in this guide, you must have access to Acrobat 6 Pro.

Organization

This guide is divided into the following chapters and appendix:

- Chapter 1, "Introduction to Acrobat JavaScript"
- Chapter 2, "Acrobat JavaScript Editor and Debugger Console"
- Chapter 3, "Acrobat JavaScript Debugger"
- Chapter 4, "Using Acrobat JavaScript in Forms
- Appendix A, "A Short Acrobat JavaScript FAQ"

How To Use This Guide

Chapters 2 and 3 in this guide include one or more exercises that give you an opportunity to work directly with Acrobat JavaScript. If you plan to do any of the exercises, do the following:

1. Be sure that you have Acrobat Pro 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 a JavaScript exercises directory on your local hard drive. You will use this directory to store the PDF documents and other files used in the exercises.
3. At the beginning of the Debugger chapter, a `.zip` file is identified that contains the files you need to work through the exercises in that chapter. You should extract the contents of these files to your local directory.

NOTE: *Note:* Macintosh users need to have the StuffIt Expander application to extract the `.zip` file contents.

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 Acrobat Solutions Network URL is: http://partners/adobe.com/asn/
	Live links to sections within this document	See Using the SDK.
	Live links to other Acrobat SDK documents	See the *Acrobat Core API Overview*.
	Live links to code items within this and other Acrobat SDK documents	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**

For More Information

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 the Acrobat 6 application.

- JavaScript reference material available on the Netscape Developer Web site (http://devedge.netscape.com/)

  This documentation consists of:
  - *Core JavaScript Guide*
  - *Core JavaScript Reference*

- Acrobat eForms Solution Training

  (http://partners.adobe.com)

- Acrobat Review and Markup Training

  (http://partners.adobe.com)

- *Portable Document Format (PDF) Reference, Version 1.4*

In this document, references to Acrobat SDK documents that appear online (in blue, italics) are live links. However, to activate these links, you must install the documents on your local file system in the same directory structure in which they appear in the SDK. This happens automatically when you install the SDK. If, for some reason, you did not install the entire SDK and you do not have all the documents, please visit Adobe Solutions Network Web site to find the documents you need. Then install them in the appropriate directories. You can use the Acrobat SDK Documentation Roadmap located at the beginning of this document as a guide.

1 Introduction to Acrobat JavaScript

Introduction

This chapter introduces Adobe Acrobat JavaScript. The JavaScript development environment for Acrobat 6, which includes new debugging capabilities, is only supported on the Acrobat Pro "flavor" of Acrobat 6.

This guide shows a variety of ways that you can use Acrobat JavaScript to implement solutions to your customer or company's needs. For further information and examples, see the Web sites and other resources listed in For More Information on page 10.

Chapter Goals

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

- Describe the difference between Acrobat JavaScript and standard, or HTML JavaScript.
- List what you can do with Acrobat JavaScript.
- Describe the Doc Object containment hierarchy and briefly state for what purpose the **`App`**, **`Doc`**, **`Console`**, **`Global`**, **`Util`**, **`Connection`**, and **`Statement`** objects are used.

Contents

Overview

Most people know Adobe 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 an Adobe Portable Document Format (PDF) document so that it contains fields to capture user-entered data as well as buttons to initiate user actions. This type of

PDF document is referred to as an *eForm*. eForms are integral to an automated paper document processing solution for a company. In this scenario, eForms can replace existing paper forms, allowing employees within a company to fill out forms and submit them via PDF files.

Acrobat also contains functionality to support *online team review*. Documents that are ready for review are converted to Adobe PDF. When a reviewer views an Adobe 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 additional functionality beyond what Acrobat provides, and 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 field or button on 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 the Acrobat application. 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 languge like C or C++ language. Now, much of that same functionality is available through Acrobat JavaScript extensions. You will find that writing a JavaScript script to perform a task such as adding a menu to Acrobat's user interface much easier to do than writing a plug-in.

What Is Acrobat JavaScript?

Acrobat JavaScript is based on the core of JavaScript version 1.5 of ISO-16262, formerly known as ECMAScript. JavaScript is an object-oriented scripting language developed by Netscape Communications. It 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 programming language. These Acrobat-specific objects enable a developer to manipulate a PDF file, allowing the PDF file to communicate with a database, modify its appearance, and so on. Because the Acrobat-specific objects are added on top of core JavaScript, you still have access to standard classes like **`Math`**, **`String`**, **`Date`**, **`Array`**, and **`RegExp`**.

PDF documents have great versatility since they can be displayed in a Web browser via an Acrobat plug-in. In this situation, you need to be aware that there are 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 objects such as **`Window`**. Acrobat JavaScript cannot access this particular object but it can manipulate PDF-specific objects such as **`Doc`** and **`annot`**.

What Can You Do with Acrobat JavaScript?

Acrobat JavaScript enables you to do a wide variety of things within a PDF document. The Adobe Solutions Network (ASN) has an Adobe Acrobat JavaScript Training course that covers a wide variety of the possibilities available within Acrobat JavaScript. This course material is in a PDF file available at:

`http://partners.adobe.com/asn/developer/training/acrobat/javascript/main.html`

The ASN course includes:

- Performing calculations

  You can create fields on a PDF document that collect numeric data. Like a spreadsheet, you can specify calculations that can be performed on several field values to yield a desired result. See the ASN JavaScript training module on "JavaScript Basics" for more information.

- Responding to user actions

  When a user interacts with a PDF document, mouse clicks, text entry, entering or exiting fields are all possible actions that can occur. You can tie JavaScript code, or scripts, to a particular action or event. Acrobat detects when the action occurs and calls the associated script. See the ASN JavaScript training module on "JavaScript Basics" and the ASN JavaScript training module on "Location Matters" for more information.

- Validating user data

  You can check whether or not the data entered by the user on an eForm is valid. You can check whether a given date is legitimate for the application, whether a particular value makes sense given other information the user has provided on the form, and so on. See "Performing Validations and Calculations" in the ASN JavaScript training module on "JavaScript Basics" for more information.

- Modifying the Acrobat application

  Acrobat gives you the ability to alter its menus and tool bars. Use JavaScript to add custom menu items, hide or display tool bars and menus. See the ASN JavaScript training module on "Location Matters" for details.

- Controlling the behavior of the document

  You can control what happens when a document first opens by using document-level scripts. Document-level scripts can cause changes that are visible to the user, such as setting up viewing parameters for a PDF. They are also used to initialize functions and variables used by page-level and field-level scripts. You can also associate scripts with actions such as a page opening or closing. See the ASN JavaScript training module on "Location Matters" for details.

- Dynamically modifying a document's appearance and function

  One major advantage of electronic form documents is the ability to dynamically alter their appearance in response to a user's data entry. For example, you could alter allowed per diem amounts on an expense report if an employee's business trip is chargeable to a client versus an internal project. Examples of such changes include:
  - Modifying field properties, such as hidden, read-only, required, and don't print
  - Populating list boxes and combo boxes with different choices
  - Altering actions associated with fields and buttons, such as adding new JavaScripts
  - Dynamically creating fields
  - Generating annotations

  See the ASN JavaScript training module on "Creating Fields with JavaScript," for additional information.

- Processing multiple PDF files using batch sequences

  Using a batch sequence, you can execute JavaScript on a set of PDF files. You can use batch sequences to do a variety of different tasks such as extract comments, identify spelling errors, automatically print PDF files, and so on. You can specify the set of files on which to operate either when you define the batch sequence or just prior to running the sequence. See the ASN JavaScript training module on "Batch Processing with Sequences," for details.

- Dynamically creating a new page based on an Acrobat page template

  Based on user input or action, you can use JavaScript to create or spawn a new page based on a template. A template is a blueprint for a page that can be added dynamically to an existing PDF document. See the ASN JavaScript training module on "Templates and JavaScript," for details.

- Overlaying templates on top of a page to change its appearance

  If a form has many fields and buttons that need to be displayed or hidden, you can better control these page elements by placing them into a separate template. A template is a blueprint for variable information, such as text, fields, buttons, and so on, that can be dynamically overlaid on a PDF document. To change a PDF document's appearance, use JavaScript to overlay the page elements contained in the template on top of the specified page. See the ASN JavaScript training module on "Templates and JavaScript," for details.

- Interfacing to a database

  Acrobat JavaScript provides objects to specifically interact with a database via ADBC, Acrobat's specific implementation of ODBC. Using JavaScript, you can read values from a database, update or insert new data, and delete information. See the ASN JavaScript training module on "Integrating with a Database," for details.

- Setting the comment repository preference

  If you plan on using Adobe Acrobat's collaborative review capabilities, you need to specify the type of comment repository to use, as well as its location. These are referred to as *comment repository preferences*. You can use JavaScript calls to set the comment repository for a particular document or for all documents you review in Acrobat. See the

ASN JavaScript training module on "Setting up a Comment Repository," for more information.

Acrobat JavaScript Object Overview

Acrobat JavaScript defines several objects that allow your code to interact with the Acrobat application, a PDF document, or fields and buttons on a PDF document. This section introduces you to some of the more commonly used objects.

The App object

The `App` object is a static object that represents the Acrobat application itself. It defines a number of Acrobat specific functions plus a variety of utility routines and convenience functions. By interacting with the `App` object, you can get to all of the currently open PDF documents and customize Acrobat by adding menus and menu items. You can also query `App` to determine which type of Adobe product (for example, Reader, Approval, or full Acrobat) and which version the end user is using.

The Doc Object

Since the primary focus of an Acrobat application is the PDF document, you can use the Doc Object to manipulate an actual PDF document. 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, move around within the document, and access other objects within a document. Many of the objects represent items found within a PDF document, such as bookmarks, fields, templates, annotations, and sounds.

NOTE: Those of you who are familiar with the HTML JavaScript object hierarchy will remember that it is a containment hierarchy, not an inheritance hierarchy. Acrobat JavaScript is no different. No object inherits properties or methods of an object higher up the chain. Likewise, there is no automatic message passing from object to object in any direction.

FIGURE 1.1 Doc Object Containment Hierarchy

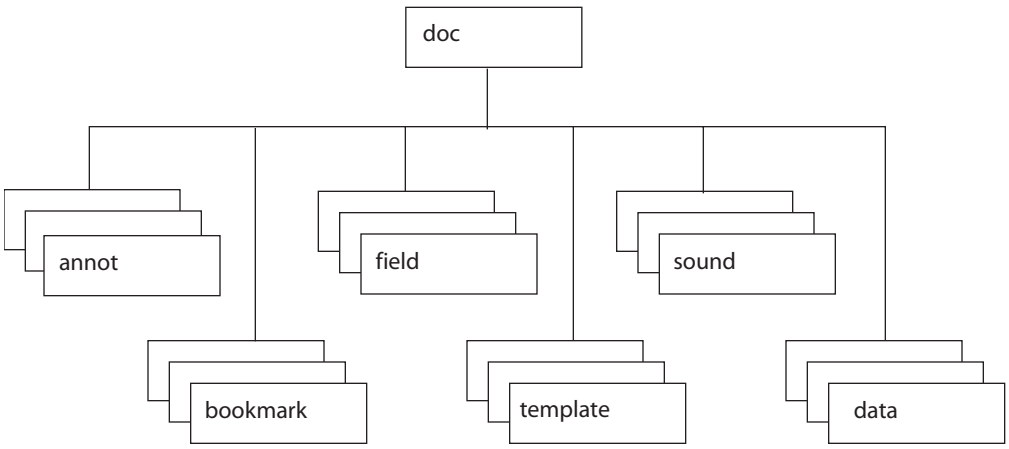


Figure 1.1 represents the containment hierarchy of objects related to the Doc Object.

Accessing the Doc Object from JavaScript can be done in a variety of ways. The most common method is using the **`this`** object, which usually points to the Doc Object of the current underlying document.

Other Common Acrobat JavaScript Objects

Several objects exist independently of the PDF document. In addition to **`App`**, some objects, such as **`Console`**, **`Global`** and **`Util`**, provide assistance to the programmer.

Console

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

Dbg

You can use the **`Dbg`** object, available only in Acrobat Pro, to optionally control the JavaScript debugger from a command line console. The **`Dbg`** object methods produce the same basic functionality as the buttons in the JavaScript debugger dialog toolbar. In addition, you can use the **`Dbg`** object to set, delete, and inspect breakpoints.

Global

Use the **`Global`** object to store data that is persistent, or permanent, across invocations of Acrobat. **`Global`** is also 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 process as a property of **`Global`**. If information about the documents needs to be stored in a **`Report`** object, that object is assigned is assigned to a property of **`Global`** so it is accessible. For more on using the **`Global`** object,

see the ASN JavaScript training module on "Location Matters" and the ASN JavaScript training module on "Batch Processing with Sequences."

Util

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

Database Objects

You can interface to a database by using the `ADBC`, `Connection`, and `Statement` objects. These Acrobat-specific JavaScript objects constitute Acrobat Database Connectivity (ADBC) and utilize ODBC calls to establish a connection to a database and access its data. As a developer, you can use SQL statements to insert, update, retrieve, and delete data. You simply pass the SQL statement to the `Statement` object's `execute()` method. These objects are discussed in detail in the ASN JavaScript training module on "Integrating with a Database."

JavaScript Language Caveats

One issue that comes up often with beginning JavaScript programmers deserves to be mentioned explicitly. It comes from the combination of two features of the JavaScript language. First, JavaScript is a case-sensitive language, which means that "`fillColor`" and "`fillcolor`" are two separate properties. Second, JavaScript creates local properties of objects you are working with "on demand", simply by your assigning a value to the new property name. The combination of these features can create programming bugs that are difficult to discover. If you assign a value to a property, but do not have the case correct in the property name, you will simply create a new property without affecting the value you thought you were changing.

2 Acrobat JavaScript Editor and Debugger Console

Introduction to the JavaScript Editor and Debugger Console

Acrobat provides a development environment in which you can implement and test Acrobat JavaScript functionality. On both Windows and Macintosh systems, you have a choice of using Acrobat's built-in editor or a third-party editor to develop your code. To introduce you to a simple method of evaluating short scripts, this chapter details how you can use Acrobat's JavaScript debugger console with the built-in JavaScript editor to evaluate scripts.

Chapter 3, "Acrobat JavaScript Debugger," will introduce the full feature set of Acrobat's debugger, which enables you to perform more sophisticated debugging tasks such as setting breakpoints, inspecting variables, and stepping through code—to name a few.

Chapter Goals

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

- Specify the type of editor you use to write your code.
- Identify the extra capabilities that Acrobat supports on some external editors.
- Invoke the JavaScript console and use it to interactively execute code and display print statements.

Contents

JavaScript Console

The Acrobat JavaScript console provides an interface for testing and debugging your JavaScript code. It is editable and interactive. The console is also handy for experimenting with object properties and methods before you use them in your code. You can use the console as an editor and interactively execute lines or blocks of code within the console.

You can use the `Console` object to manipulate the JavaScript console from within your code. `Console` is a static object. Consequently, you do not need to instantiate it to use it. Use `Console`'s `println()` method in your scripts to print debugging information. Other methods are available to show, hide, and clear the console.

Opening the JavaScript Console

To open the Acrobat JavaScript console from within the Acrobat application:

1. Open the debugger window using one of these methods:
   - Select the **Advanced > JavaScript > Debugger** menu command (Windows and Macintosh) *or*
   - Typing Ctrl-j (Windows) *or*
   - Typing Control-j (Macintosh)
2. Select **Console** or **Script and Console** from the debugger **View** window.

To open the console programmatically, use `console.show()`.

Executing JavaScript

One way of testing your JavaScript code as you develop it is to evaluate it in the JavaScript console. There are two basic ways of evaluating JavaScript code in the console:

- To evaluate a line of code, position the cursor on the line you want evaluated and press the Enter key on the numeric keypad (or type Ctrl+Return on the regular keyboard) to obtain the results.
- To evaluate a block of code, type in the code, highlight it, and press Enter on the numeric keypad (or type Ctrl+Return on the regular keyboard).

Any immediate results of evaluating the code are printed to the console. Be aware that when evaluating a block of code, Acrobat only automatically prints to the console the result from the last JavaScript expression in the block.

Formatting Code

You can format code in the JavaScript console using the Tab key.

- To insert fours spaces at the insertion point, press the Tab key.

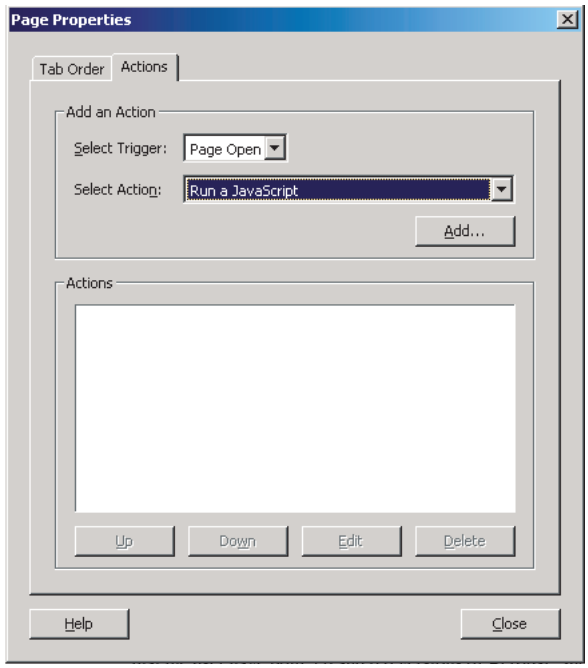
- To move the cursor four spaces to the left, press Shift+Tab. (This moves the cursor to the beginning of the line if it is fewer than four spaces from the left.)
- To move a whole line or block of code to the right four spaces, highlight the code, or a portion of a line, then press Tab.
- To move a whole line or block of code to the left four spaces (or as far left as possible when that is less than four spaces), highlight the code, or a portion of a line, then press Shift+Tab.

Using a JavaScript Editor

To define or edit JavaScript associated with a specific event for an object, you typically use a context-specific JavaScript editor, rather than the JavaScript console. The general procedure for accessing the editor is as follows:

1. Select the object (such as a field, link, bookmark, or page) with which you want to associate an action defined in JavaScript code.
2. Open the properties dialog box for that object. An example is shown in Figure 2.1.

FIGURE 2.1 Page Properties

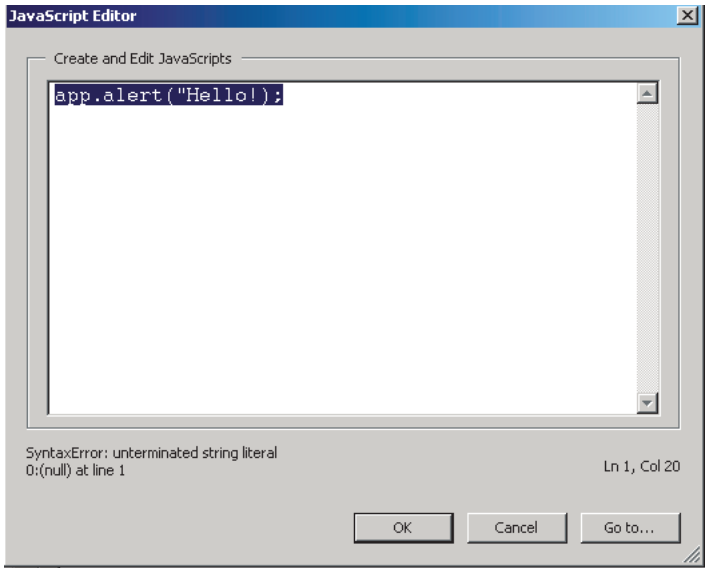


3. Select a particular trigger (for example, **Mouse Up** or **Page Open**) that will execute the JavaScript action.
4. Select **Run a JavaScript** from the **Select Action** drop-down list.

5. Click **Add** to open the JavaScript editor.
6. In the editor window, write the JavaScript script that you want to run when the user opens the page.
7. When you are done, click **Close** to close the editor.

   If there are errors in your code, the JavaScript editor will highlight the code line in question and display an error message, as shown in Figure 2.2.

FIGURE 2.2 Error detected by the JavaScript editor



In Figure 2.2, the quotation mark to the right of the string is missing.

Key Notes

You can open the JavaScript editor to associate a JavaScript action with an object at various locations (or "levels") in a PDF document. For example, you can associate a script at the location of a particular field in a form or at the document level, in which case the script is available from all other scriptable locations within the document. Access to the editor depends on where you want the script to take effect.

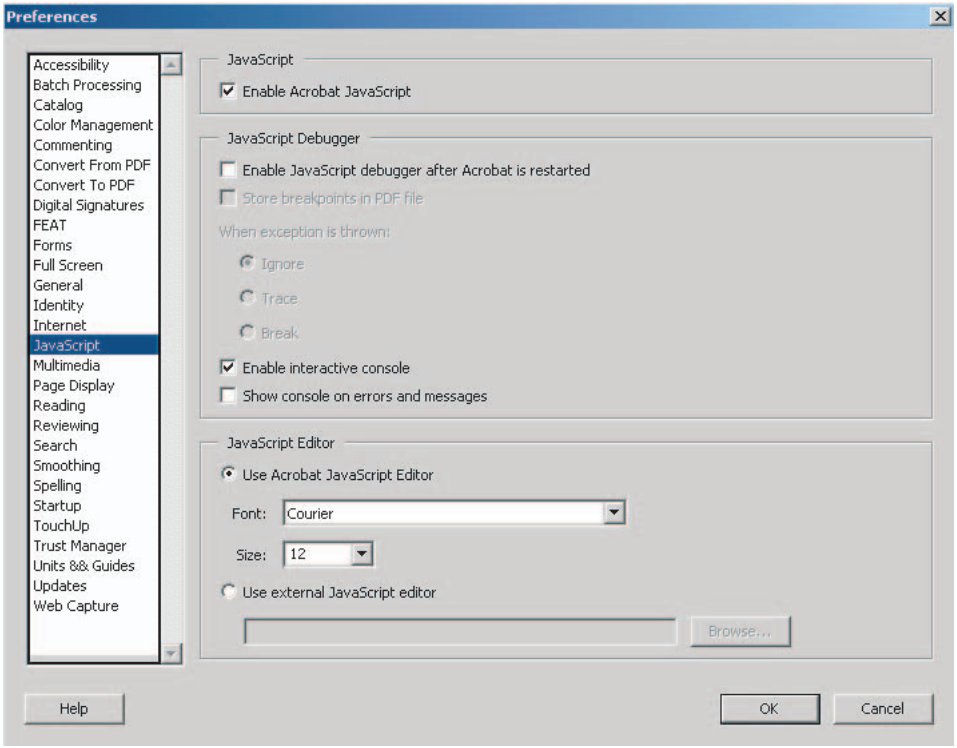
For details on accessing and using the JavaScript editor at different locations within a PDF document and at the application level, see the ASN JavaScript training module on "Location Matters."

Specifying the Default JavaScript Editor

You can choose to either use the built-in JavaScript editor that comes with Acrobat, or an external JavaScript editor of your choice. To set the default JavaScript editor that you want to use:

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 shown in Figure 2.3.
3. In the **JavaScript Editor** section, select the editor you want to use.

FIGURE 2.3 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. If you wish to use and external editor,type or paste the path to the editor in the text box, or click **Browse...** to navigate to your prefered JavaScript editor application.

NOTE: When you specify some external editors by browsing, Acrobat adds command options to the editor command. For details, see "Additional Editor Capabilities" on page 24.

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

Using the Built-in Acrobat JavaScript Editor

You can use the built-in Acrobat JavaScript editor to evaluate JavaScript code, just as you use the Acrobat JavaScript console. Simply select a line or block of code to evaluate, and press the Enter key on the numeric keypad or Ctrl+Return on the regular keyboard.

The editor behaves the same way as the debugger console except that the results of evaluating code in the console are output to the console window, rather than to the editor. This is so that code that you enter in the editor does not become mixed up with printed evaluation results.

You format code in the built-in JavaScript editor in the same way that you do in the console window. For details, see "Formatting Code" on page 20.

Using an External Editor

When you specify an external editor program as the default for editing JavaScripts in Acrobat, that program is used any time a JavaScript has to be edited from inside Acrobat. Acrobat generates a temporary file and opens it in the external editor program. When you are editing a file in an external editor, you must save the file for Acrobat to see your changes. In addition, Acrobat is inaccessible to you while you are editing the file in the external editor program. You cannot evaluate Acrobat JavaScript code from inside an external editor. Close the external JavaScript editor to regain access to Acrobat.

Additional Editor Capabilities

Acrobat is able to support additional editor capabilities of Windows-based editors if those capabilities can be specified on the command line. Two parameters are available to support command line syntax: the *file name* (`%f`) and the *target line number* (`%n`). Parameters for Macintosh-based editors are not supported.

One important editor capability is the launching of a new instance of the editor each time you start a new editing session. By default, many editors load new files into the same editor instance if one is already running. In such a case, opening a new session closes the existing one. Since Acrobat does not have access to your changes in the file until you have saved it in the temporary locaton, the result is that unsaved changes in your session are lost. To prevent this from occurring, you must either remember to close the editor application before starting a new editing session or you must instruct the editor to always launch a new editor instance.

If you can set an internal preference in your editor to always launch a new instance of the editor, do so. If you editor requires a command line parameter to instruct it to invoke a new editor instance, you may add that parameter to the editor command line specified in the

Edit > Preferences > JavaScript dialog. This ensures that your editing work is not discarded.

If your editor accepts a starting line number on the command line, Acrobat also supports the option of starting the editor on a line that contains a syntax error by making the line number available as a parameter (`%n`) to insert in the command line.

Acrobat has pre-defined, 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 pre-defined command line template, that template will be sustituted in the command line field. You may edit the command line options at this point if you wish. If your editor does not have a pre-defined template, you may need to specify the appropriate command-line parameters in the command-line field.

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.

If your editor is not one that Acrobat currently supports, go to the Web site for your editor and look up the editor documentation, or use the documentation provided with your editor. To determine your editor's capability, you need answers to the following questions:

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

  Switches vary depending on the editor and include `/NI` and `+new` followed by the file name, which is represented as `%f` and which should be enclosed in double quotes to ensure that spaces in file and folder names are properly handled.

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

  Some example line number command switches include `-#`, `-L`, `+`, and `-l` each followed by the line number, which is represented as `%n`.

  Because the value of `%n` may be null if there is no line number to jump to, Acrobat supports having that portion of the command be optional, only inserted if there is a line number to jump to. Any command switches or parameters enclosed in square brackets (`[...]`) will only be inserted if the line number parameter is to be used. Thus, unless your editor supports a null line number in a useful way, you should enclose the line number switch and the `%n` parameter in square brackets.

If you need to insert `%`, `[`, or `]` literally in your command line, you can use `%%`, `%[`, or `%]` respectively.

For instance, Acrobat recognizes the Visual SlickEdit editor as `vs.exe` and will substitute the command line:

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

This is what you will see in the **Edit > Preferences > JavaScript** dialog after you click **Open** in the **Browse** dialog. The program must be installed and must be found using the **Browse** dialog for Acrobat to properly recognize it and substitute the template.

When Acrobat needs to open the 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 be something like:

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

TABLE 2.1 Editors supported as External JavaScript editor with command line templates.

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 Supports Opening at Syntax Error Locations

On a few editors such as TextPad, Acrobat does not support opening the editor on a line number. To determine if this capability is supported,

1. Open a syntax error-free 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.

   Does a dialog display prompting you to fix the syntax error?

   If so, does it correctly specify the line containing the syntax error?

Saving and Closing a File with a Syntax Error

If you save and close a file you have been editing and it contains a syntax error (on line 123, for example), Acrobat displays a dialog box with the following message prompting whether you want to fix the error:

> 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 will expand the path to the editor to include the line number in the specified syntax. The editor will open and the cursor will be placed on line 123.

Exercise: Working with the JavaScript Console

To do this exercise, you must have Acrobat 6 Pro installed on your machine.

In this exercise you will verify that JavaScript is enabled for Acrobat and then 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 console in the debugger.
- Evaluate code in the console window.

Enabling JavaScript

To create and use JavaScript actions in Acrobat, you must have JavaScript enabled. To verify that JavaScript is enabled for your installation of Acrobat:

1. Launch the Acrobat application.
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. Keep the **Preferences** dialog box open to enable the JavaScript debugger and the debugger console.

Enabling the Interactive JavaScript Console

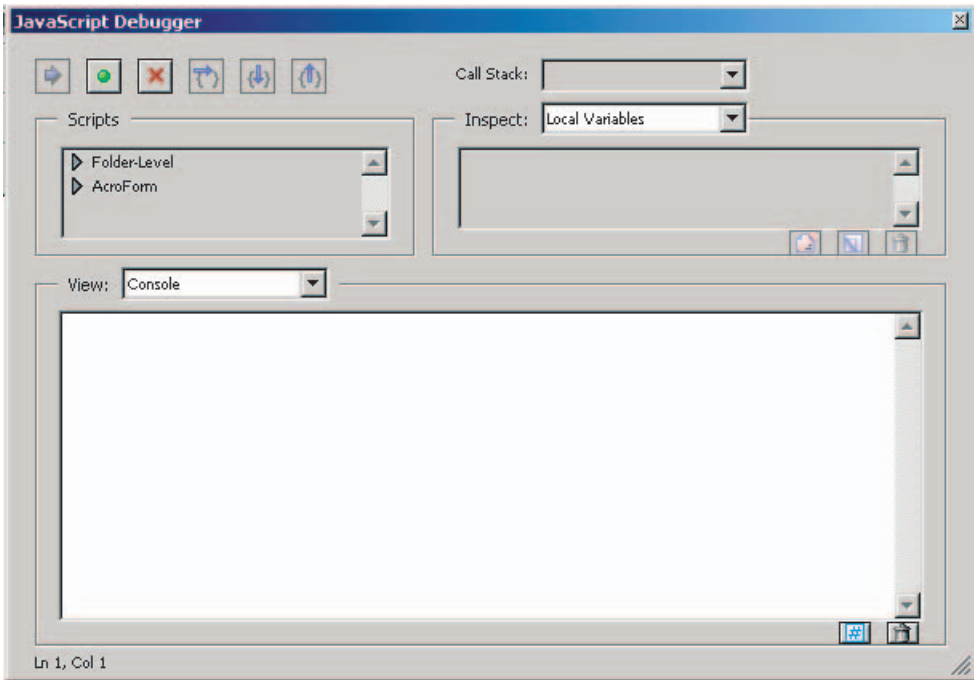
The console window is part of the JavaScript debugger. To enable the console, you must enable the debugger.

1. In the **Preferences** dialog, select **Enable JavaScript Debugger after Acrobat is restarted** from the JavaScript Debugger options.
2. Select **Enable interactive console**.
   This option enables you to evaluate code that you write in the console window.
3. Select **Show console on errors and messages**.
   In case you make mistakes, the console will display information to help you.
4. Click **OK** to close the Preferences dialog.
5. Close and restart Acrobat.

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 2.4.

FIGURE 2.4 Console window in debugger

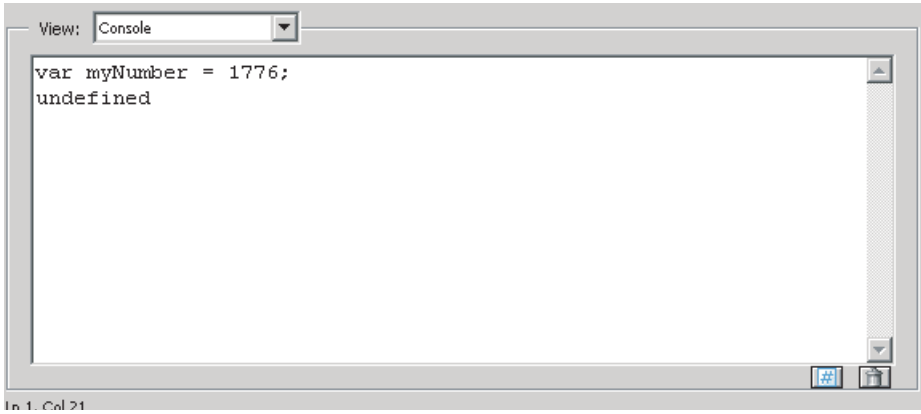


3. Click the Clear button (trash can icon) just below the console window to delete any contents that appear in the window.

4. Type the following code into the console:
```
var myNumber = 1776;
```

5. With the mouse cursor positioned somewhere in this line of code, press Enter on the numeric keypad or Ctrl+Return on the regular keyboard to evaluate the code. You should see the results shown in Figure 2.5.

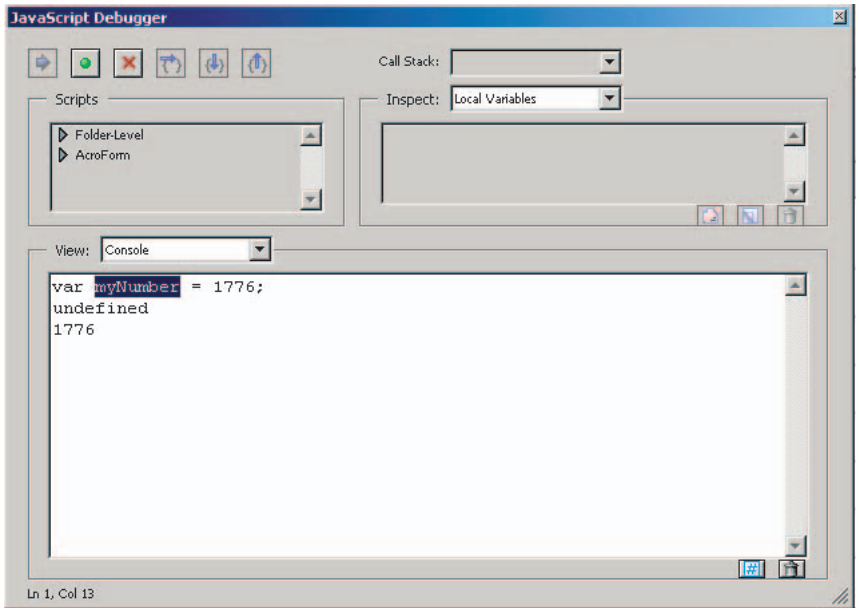
FIGURE 2.5 Evaluating the variable declaration



The console window prints out **undefined**, the return value of the variable declaration. Note that the result of an expression is not the same as the value of a variable set in the expression. In this case, the return value **undefined** does not mean that the value of **myNumber** is undefined, just that the expression as a whole returns **undefined**.

6. To evaluate just the **myNumber** variable (part of a whole JavaScript statement) in the console window, highlight the variable name and press Enter or Ctrl+Return. You should see the results shown in Figure 2.6.

FIGURE 2.6 Evaluating myNumber

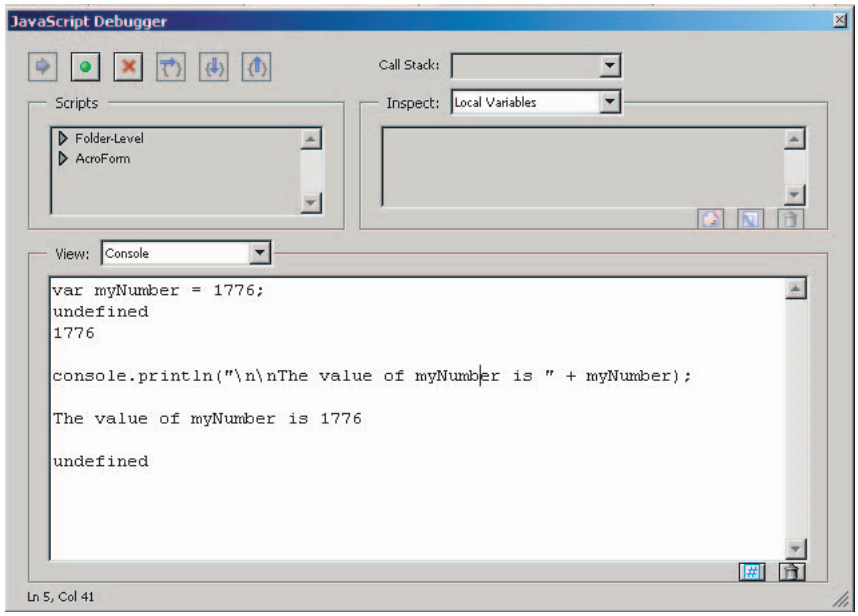


7. Now evaluate the following line of code by typing it in below the existing console contents and pressing Enter with the cursor positioned on the code to be evaluated:

```
console.println("\n\nThe value of myNumber is " +
                myNumber);
```

Notice the use of the C-style formatting character **\n** to force a new line. The results in the console window should appear as shown in Figure 2.7.

FIGURE 2.7 Evaluating a code line



The **`console.println()`** method can be very useful to include in your JavaScript code to print debugging information and other messages to the console. It prints the text that you specify and then returns **`undefined`**.

8. Click **Close** to close the debugger dialog.

3 Acrobat JavaScript Debugger

Introduction to the Acrobat JavaScript Debugger

The Acrobat JavaScript Debugger allows debugging of JavaScript code in Acrobat. It is a fully capable JavaScript debugger which lets you set breakpoints and inspect variable values while stepping through code. The debugger is not available in the Acrobat Reader.

Debugging is controlled with a single dialog that you can open from the Acrobat menu item **Advanced > JavaScript > Debugger**. If the debugger dialog is not open and you happen to run a script that throws an exception or for which a breakpoint has been set, the debugger dialog displays so that you can proceed with debugging without having to manually open the debugger.

NOTE: Debugging JavaScript stored in HTML pages viewed by Web browsers such as NetScape or Internet Explorer, or any other kind of scripting languages, is not possible using this debugger.

Chapter Goals

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

- Identify and understand how to use the controls provided by the debugger.
- Start the debugger at the beginning of a script or from anywhere within a script.
- Interactively execute code and display output to the console window.
- Set customized watches and breakpoints.
- Inspect the details of variables.
- Complete a debugging session and start a new one while in the debugger dialog.

Contents

Topics and Exercises
Inspect Details Window
Starting the Debugger
Exercise: Calculator
Known Issues

Enabling the Acrobat JavaScript Debugger

Before you can use the debugger, both JavaScript and the debugger must be enabled. Use the **Edit > Preferences** dialog to control the behavior of the Acrobat JavaScript development environment. Enabling JavaScript and the JavaScript editor are described in Chapter 2. 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.

Other options for the debugger are located in the **JavaScript Debugger** section of the **Preferences** dialog, as shown in Figure 3.1.

FIGURE 3.1 Specifying JavaScript Development Tool Preferences

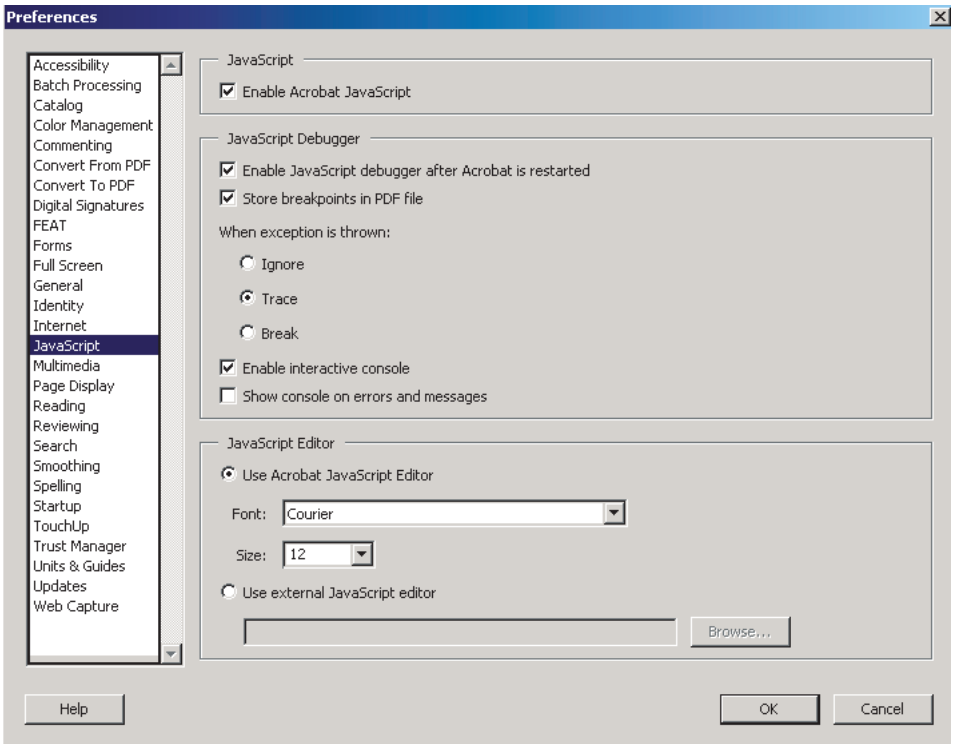


Figure 3.1 shows typical JavaScript debugger option settings. Each of the options is described in Table 3.1.

TABLE 3.1 JavaScript debugger options

Option	Meaning
Enable Javascript debugger after Acrobat is restarted	To enable the debugger you must check this option. The debugger features such as setting breakpoints become available for you to use the next time you launch Acrobat.
Store breakpoints in PDF file	This option enables you to store breakpoints you have set so they will be available the next time you start Acrobat or open the file. To remove the breakpoints when the file is debugged and ready for delivery: • 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 what you want to happen 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 and function where the exception occurred.
Enable interactive console	This option allows you to edit in the console window. If this option is not checked and you click the console window as if to edit text there, the following message displays: The interactive console is not enabled. Would you like to enable it now? Clicking **Yes** enables 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 and displays the appropriate message. If the debugger is not enabled when you check this option, the debugger dialog still opens when an error occurs and displays the error message to the console window. If, however, you click the console window, the following message displays: The interactive console is not enabled. Would you like to enable it now? Clicking **Yes** enables the interactive console from within the debugger.

Debugger Dialog Window

You can open the debugger dialog window without having a script to debug by using 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" on page 40 outlines the types of scripts that can be debugged and where they are located. The section "Starting the Debugger" on page 46 details the ways of automatically starting the debugger on a script to initiate an interactive debugging session.

Main Groups of Controls

The debugger dialog window, shown in Figure 3.2, consists of three main groups of controls. The toolbar on the top left contains six button controls for stepping through code in different ways and for quitting the debugger.

Immediately below the toolbar, a **Scripts** window displays the names of scripts that you can debug. Scripts are organized in a tree hierarchy, such as the the one illustrated in Figure 3.2. Please note the distinction between the **Scripts** hierarchy, in this window, and the **Script** window below, which shows a single script.

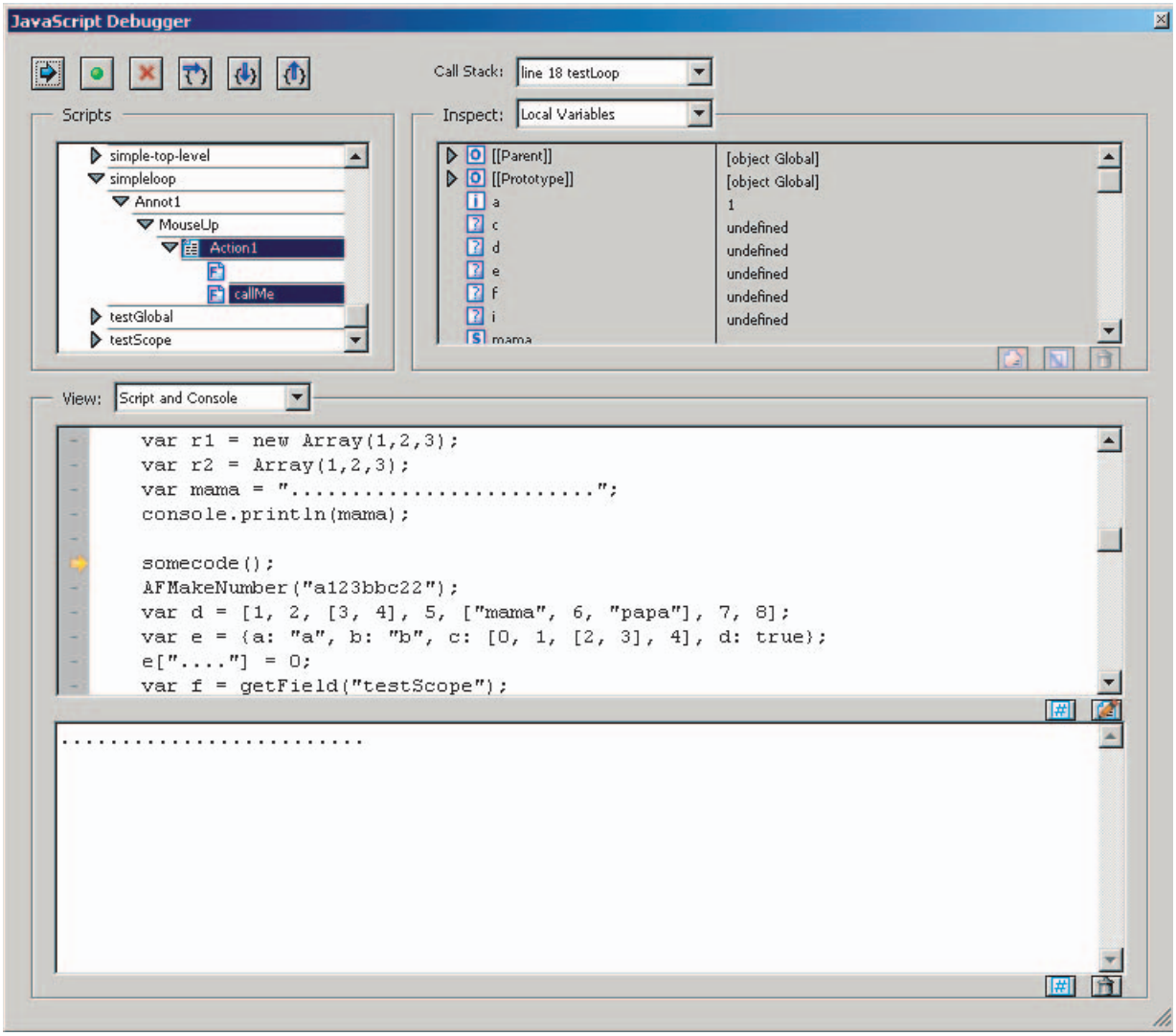
The **Call Stack** and **Inspect** drop-down lists are located at the top right of the debugger dialog. Selecting entries in these lists enables you to view nested functions and to inspect the details of variables, watches, and breakpoints in the **Inspect** details window.

Debugger View Windows

Below the main group of controls, the debugger has a view window. From the **Views** drop-down list, you can choose:

- **Script**, to view a single JavaScript script that you have selected from the Scripts hierarchy window, above.
- **Console**, to 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. This is described in detail in Chapter 2, "Acrobat JavaScript Editor and Debugger Console".
- **Script and Console**, to view the console window and script window at the same time. The script window displays above the console window, as shown in Figure 3.2.

FIGURE 3.2 Debugger dialog



Debugger Buttons

The Acrobat JavaScript Debugger allows you to easily control what portions of the script you want to see executed. You can start the debugger using any of the ways described in "Starting the Debugger" on page 46.

Figure 3.3 shows the debugger buttons on the toolbar. Figure 3.2 summarizes the functionality of each of these buttons. Details describing each button follow the summary table. An overview of their use together is in the section on "Stepping Through Your Code" on page 47.

FIGURE 3.3 Debugger buttons

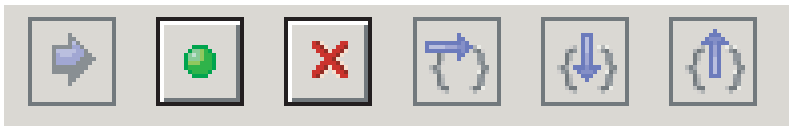


TABLE 3.2 Debugger buttons summary

Button	Description
Resume Execution	Runs a script stopped in the debugger.
Interrupt	Halts execution.
Quit	Terminates script execution and closes the debugger dialog.
Step Over	Single steps through instructions but does not enter function calls encountered.
Step Into	Single steps through instructions and enters each function call that is encountered.
Step Out	Executes the code of a function call, and stops on the instruction immediately following the call to the function 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 execute
- The next breakpoint
- The next error encountered
- The end of the script

You can also halt execution by clicking either the **Interrupt** or **Quit** buttons.

Interrupt

The **Interrupt** button halts execution of the current script. Click this button to activate it. When active, it displays in red. Performing any action that results in running a JavaScript script in Acrobat will cause execution to stop at the beginning of the script. The **Interrupt** button deactivates when it is used and must be clicked again to interrupt another script.

Quit

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

Step Over

The **Step Over** button single-steps through instructions. Clicking **Step Over** at a function call executes the entire function in a single step rather than executing the function's instructions one at a time with each click of the button. Say, for example, the position indicator (yellow arrow) in the debugger is to the left of a function call, as shown in Figure 3.4.

FIGURE 3.4 Position indicator at a function call

```
callMe();
```

This location is immediately 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 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 individual statements within a function. Say, for example, the debugger position indicator is as shown in Figure 3.4. Clicking **Step Into** at this point takes you to the first statement within the `callMe()` function.

NOTE: Be aware that native functions, for which there is no JavaScript implementation, will not allow you to step into them. This applies to Acrobat native functions as well as core JavaScript functions.

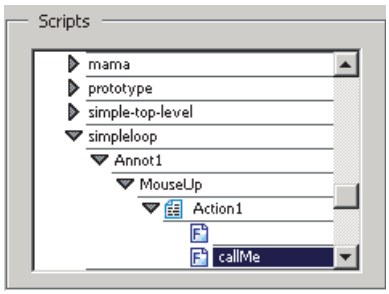
Step Out

The **Step Out** button executes the code out of a function call and stops on the instruction immediately following the call to the function. Using this button, you can quickly finish executing the current function after determining that a bug is not present. If you are not inside of a function call and assuming there are no errors, clicking the **Step Out** button continues executing code to the end of the script or until a breakpoint is reached.

Debugger Scripts Window

Scripts are automatically collected from a PDF file open in Acrobat when you open the debugger dialog. The debugger displays these in the **Scripts** window. Figure 3.5 shows an example.

FIGURE 3.5 Scripts window



Accessing Scripts in the Scripts Window

To access scripts you want to display, you click the triangle next to an entry in the **Scripts** window to open the next level down. Continue to click triangles to open the next lower level. Use the scroll bar, if necessary, to view the contents in the window as you open lower levels. A script icon indicates that a script is defined for an action or function. In Figure 3.5, a function is defined for a mouse-up action on a button-type form named `Button1`. Clicking on the script icon opens the debugger at the location of the first script statement.

Acrobat JavaScripts can be stored in several places, either inside or outside PDF files. Scripts that live inside a PDF file are associated with an action of type JavaScript. The following sections contain lists of possible locations for scripts, inside or outside a PDF file. These scripts are immediately available for viewing and debugging in with the Acrobat JavaScript Debugger

Scripts Inside PDF Files

Table 3.3 lists the PDF-file script types that can be displayed in the Scripts window. You can view and debug these scripts anytime that you open the debugger dialog. You can edit these scripts from inside the debugger and you can set breakpoints as described in Breakpoints on page 44. Changes to scripts do not take effect until the scripts are re-run; you cannot make changes to a running script.

TABLE 3.3 Scripts inside PDF files

Location	Access
Document level	**Advanced > JavaScript > Document JavaScripts**

TABLE 3.3 Scripts inside PDF files

Location	Access
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, you need to access Acrobat's Advanced Editing toolbar. If the Advanced Editing toolbar is not visible, choose **Tools > Advanced Editing > Show Advanced Editing Toolbar**. You may want to dock this toolbar, as you will be using it frequently when you write scripts.

Scripts Outside PDF Files

Scripts outside of Acrobat are also listed in the Scripts window and are available for debugging in Acrobat. Table 3.4 lists these script types and how to access them.

TABLE 3.4 Scripts outside PDF files

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

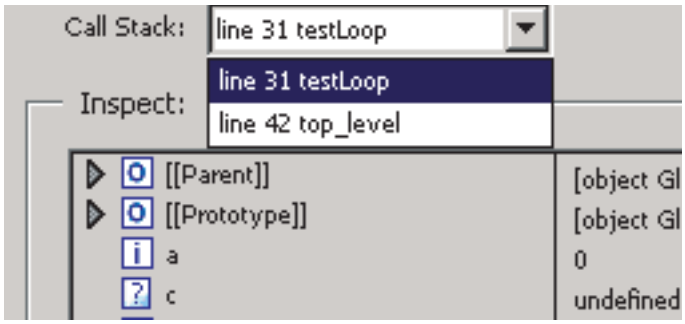
Folder-level scripts normally can be viewed and debugged but not edited in Acrobat. Console and batch processing scripts are not visible to the debugger until they are executed. For this reason, you cannot set breakpoints prior to executing these scripts. You can access the scripts either using the **Debug From Start** option or by using the `debugger` keyword. See Starting the Debugger on page 46 for details.

Call Stack List

To the right of the debugger control buttons is the is the **Call Stack** List. The **Call Stack** drop-down list displays the current scope of execution in a script. See Figure 3.6. When text entries are displayed in the call stack, you are currently stopped in the debugger. Each text

entry represents a function call, or frame, in the stack. The text of an entry displays the line number of the function in the script and the function's name. The most recent stack frame is the top of the stack and 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. The variables appear in the **Inspect** details window immediately below the call stack list.

FIGURE 3.6 Call stack



Selecting a different entry in the call stack activates that stack frame and displays that location in the script in the **View** window. A green triangle to the left of the script in the **View** window points to the exact frame location. 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.

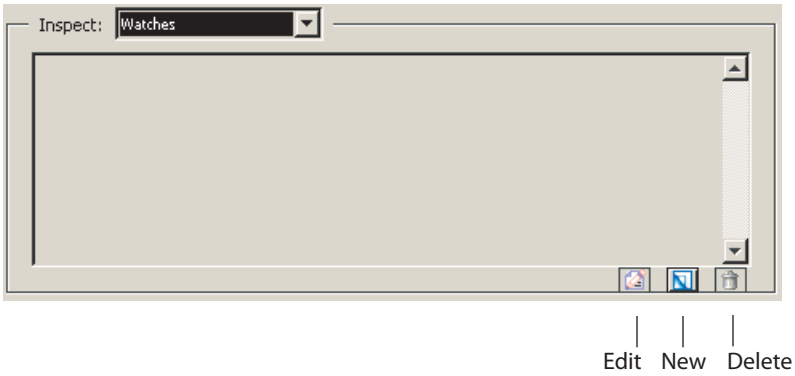
Inspect Details Window

To the right of the **Scripts** window and below the **Call Stack** list is the **Inspect** details window. You can inspect the values of variables, customize your inspection of selected variables (also referred to as setting watches), and obtain detailed information about breakpoints.

Details Window Controls

Three buttons just below the **Inspect** details window can be used to edit, create, or delete items. These buttons are dimmed until you select items in the **Inspect** drop-down list that can be created, edited, or deleted. Figure 3.7 identifies the **Edit**, **New**, and **Delete** buttons in the **Details** window.

FIGURE 3.7 Inspect details window button controls



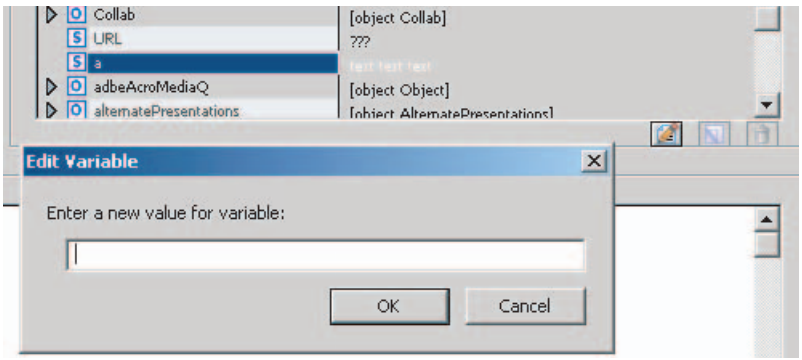
Inspecting Variables

The capability of inspecting variables is a powerful tool that you can use to examine the current state of JavaScript objects and variables. It enables you to see the values for variables in the current stack frame, or local scope, being debugged. If a property is an object, you can recursively inspect the tree of objects and properties.

To start inspecting variables, select **Local Variables** from the **Inspect** drop-down list. If debugging currently is stopped in a script, a paired list of variables and values displays in the **Inspect** details window just beneath the list. To edit a variable, highlight in the variable in the details window. This activates the Edit button just below the window. Clicking **Edit** displays a popup window in which you can enter a new value for the variable, as shown in Figure 3.8.

A triangle next to the property name means that property is an object. To display that object's properties, click the triangle to expand the object's properties. To contract the list again, click again on the triangle.

FIGURE 3.8 Local variable details



Watches

The **Watches** list enables you to customize your inspection of variables. Watches are JavaScript expressions evaluated every time the debugger stops at a breakpoint or a step in execution. You can add as many watches as you need in the **Watches** list. You can edit, add, or delete watches using the three buttons just below the **Inspect** details window. Results are evaluated and displayed in the **Inspect** details window in the order they are entered in the Watches list that you create.

To start working with watches, select **Watches** from the **Inspect** drop-down list. Clicking **New** below the details window displays a popup window in which you can enter the JavaScript variable or expression that you want to be evaluated.

To change the value of a watch, select the watch from the list. Clicking **Edit** displays a popup, allowing you to specify a new expression for evaluation. To delete a watch, select it from the **Inspect** drop-down list and click **Delete**. This removes the watch from the details window.

Breakpoints

The Breakpoints option in the **Inspect** drop-down list allows you to manipulate program breakpoints. A breakpoint is the exact location where you want your script execution to halt so you can check the values of local variables at that point. Breakpoints are defined for a particular line of code, and they can be either unconditional or conditional. (See "Using Conditional Breakpoints" on page 45.)

When a breakpoint is reached, JavaScript execution stops and the debugger shows that line of code.

To add a breakpoint, click on the gray strip to the left of the script in the script view. The lines on which breakpoints are allowed are indicated by a small horizontal line. When a breakpoint has been set, a red dot indicates the position. Clicking the red dot removes the breakpoint.

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

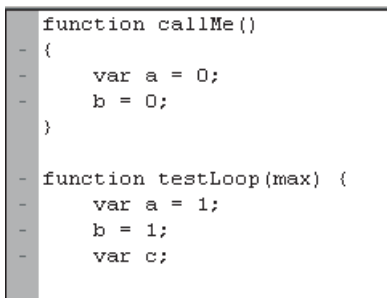
```
function callMe()
{ // curly brace is on a separate line
     var a = 0;
}
```

The Acrobat JavaScript debugger will not let you set a breakpoint next to the function name for Style 2. Attempting to set a breakpoint in this location causes the debugger to

display an alert telling you that you cannot set a breakpoint on that location. You can only set a breakpoint next to the codeline containing the left curly brace.

In Figure 3.9, you can set the breakpoint on the line with the left curly brace following the function name `callMe()`. You can set a breakpoint on the line containing the curly brace and the `testLoop()` function name. In either case, the effect is the same. Executing from the breakpoint evaluates that function.

FIGURE 3.9 Setting a Breakpoint Before a Function Definition



Setting a breakpoint just before the call to a function and then running the script containing it causes execution to stop at the first statement within the function.

Listing Breakpoints

Selecting the Breakpoints option from the **Inspect** drop-down list allows you to see a list of all the breakpoints that you set during a debugging session. You can edit and delete breakpoints using the button controls just beneath the **Inspect** details window, as shown in Figure 3.7. Select the breakpoint you want to edit or delete in the **Inspect** details window to activate the button controls for breakpoints.

Using Conditional Breakpoints

Conditional breakpoints are sometimes necessary when debugging. If you are debugging an iteration loop, for example, and your problem only occurs after the loop control variable reaches a certain value, it may take too long for you to step through every iteration. A conditional breakpoint can help you determine when to stop.

A conditional breakpoint causes the interpreter to stop the program and activate the debugger if a certain condition is met. You specify the condition as a JavaScript expression. If the result of the expression evaluates to true, the interpreter stops the program at the breakpoint. Otherwise, the interpreter does not stop the program. An unconditional breakpoint always causes the interpreter to stop the program and to activate the debugger when it reaches the line of code where the breakpoint is set.

By default, all breakpoints created in Acrobat have their conditions set to true. To change a breakpoint condition, select Breakpoint from the **Inspect** drop-down list and click Edit. This displays a popup window in which you can change the breakpoint condition.

Starting the Debugger

You can start the debugger in four ways. Two of these ways allow you to debug from the start of execution. The other two allow you to start debugging from an arbitrary point in the script.

Debugging From the Start of Execution

Both of these methods start the debugger at the first line of the script. Use the Step Into button to go into the script from this point.

Debug From Start

Choose **Advanced > JavaScript** and, if the option is not already checked, click on **Debug From Start** to select this option. Clicking the checked option will turn it off again.

When this option is checked on, execution stops at the first line when you run any script in Acrobat. The debugger opens, allowing you to start debugging from the first statement in the script.

NOTE: **Debug From Start** does not turn off automatically. Be sure to turn off this option when you are done using it. Otherwise it will continue to stop on every new script you execute in Acrobat.

Click Interrupt

Open the debugger window and click the **Interrupt** button. (It will display in red.) With Interrupt activated, performing any action that results in running a JavaScript script in Acrobat causes execution to stop at the beginning of the script.

Unlike **Debug From Start**, the **Interrupt** button deactivates after you use it once to stop at a script. 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 on page 44 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.

Stepping Through Your Code

The Acrobat JavaScript Debugger allows you to easily control what portions of the script you want to see executed. Four of the toolbar buttons allow you to step through your code when a script is stopped in the debugger. These are the **Resume Execution**, **Step Over**, **Step Into**, and **Step Out** buttons, described individually in the section on "Debugger Buttons" on page 37. This section describes how the buttons are used together to control the debugger.

When the line you have stopped contains a call to a JavaScript function, you may wish to execute the individual statements inside that function by clicking on the **Step Into** button on the toolbar. This is called stepping into a function and allows you to investigate exactly what is going on with your script when a function is called. Be aware that native functions, for which there is no JavaScript implementation, will not allow you to step into them.

When the line you have stopped contains a function call, you may avoid executing the individual statements in that function by clicking on the **Step Over** button on the toolbar This is called stepping over a function. If the statement you have stopped does not contain a function call, stepping over will simply execute that statement.

If the line you are stopped at is inside a function call, you may choose not to step through the rest of the statements of that function by clicking the **Step Out** button on the toolbar. This is called stepping out of a function and allows you to return to the statement which called that function. If you are not inside a function call, clicking on the Step Out button will continue execution to the end of the script or until a breakpoint is reached.

When stopped in the debugger, you may continue execution from the current statement by clicking on the **Resume Execution** button on the toolbar. Your program will either run to completion or stop again if it reaches a new breakpoint.

Exercise: Calculator

To do the following exercises, you must have unzipped the file **`TestDebugger.zip`**. to get the **`Calc.pdf`** file.

In this exercise, you will set breakpoints in a script and create watches to view how a variable changes as you execute the script in the debugger. 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 these debugger buttons to control how the interpreter executes the code in the script.

- **Step Into**
- **Step Out**
- **Quit**

Calculator

Calc.pdf uses document-level and form field-level scripts to create the behavior of a simple calculator. The key pad consists of numeric and function keys, the period (**.**), the equals sign (**=**), as well as Cancel and Cancel Entry keys.

How the Calculator Displays Output

Calc.pdf uses the Acrobat Doc Object method **getField()** to display the values of keys that the user enters and the results of evaluating expressions to the calculator display window. The calculator display window is the text field **Display**. In this statement, **getField()** binds the variable **display** to this field:

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

Each numeric key is a button field with this Mouse Up action whose return value is assigned to the calculator display window:

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

For example, when the user presses the **"Seven"** button, this assigns the value returned by **digit_button(7)** to the calculator window.

Calc.pdf uses a similar mechanism to display the strings representing arithmetic operations to the **Func** text field. In this statement, **getField()** binds the variable **display** to this field:

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

Here is the Mouse Up action for the division (**/**) key:

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

The action passes the string **"DIV"** to the **func_button()** function. The parameter is assigned the name **req_func**, so when **func_button()** is executed,

```
req_func = "DIV"
```

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.

Adjusting For 2-Digit Values

To adjust for values entered that are greater than or equal to 10, the calculator multiplies the current value by 10. For values less then 1, it multiplies the divisor 10. Say, for example, the user enters the values 2 and 4 consecutively. The code multiplies the 2 by 10 and adds 4 to the result. The point button Mouse Up action updates the divisor to 10 if it is 1. This

causes a subsequent value entered (to the right of the point) to be divided by 10 to represent the tenths position.

Testing the Calculator

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

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

Getting Started

This exercise introduces runtime errors into the calculator example. It takes you through steps in the Debugger to correct the errors so that this version of the calculator works like the one you just experimented with.

NOTE: Before you start this exercise, be sure that the Acrobat JavaScript debugger is enabled. See Enabling the Acrobat JavaScript Debugger on page 34 for details.

1. Open the file **`CalcWithRTErrors.pdf`** in Acrobat.
2. Just as you did with **`Calc.pdf`**, experiment with entering some expressions. Observe the results of:

   Button values that you enter and the resulting value displayed.

   Function keys that you enter and the resulting string values displayed.
3. Click Cancel. This clears **`Display`** to 0s.
4. Test values, and compare your results to **`Results:`**.

   Test:

   Enter this expression. Evaluate:
   ```
   9 - 6 = ???
   ```
   Click Cancel. Evaluate:
   ```
   9 * 3 = ???
   ```
   Add 3 to the result. Evaluate:
   ```
   ((9 * 3) +3 )=
   ```
   Add 3 again. Evaluate:
   ```
   (((9 * 3) +3) + 3 =
   ```
   Click Cancel. Add 3 to 9. Evaluate:
   ```
   9 + 3 = ???
   ```

   Results:
   ```
   9 - 6 = -6
   9 * 3 = 0
   ((9 * 3) +3 )= 0
   (((9 * 3) +3) = 0) + 3 = 3
   9 + 3 = 3
   ```

5. Summarize your findings.

   From the above, note that:

   – Button values display correctly. Clicking 9 displays 9, clicking 3 displays 3, and so forth in the `Display` field.

   – Function strings display correctly in the `Func` field. Clicking `*`, `-`, `+`, and so forth displays the correct function string.

   – Click Equals (`=`). In most cases, results are incorrect.

   – Assumption: There is at least one runtime error in the code. Something isn't working correctly between the value of the `entry` and `accum` variables.

Debugging a runtime error

1. Click **C** to Cancel and clear the calculator display to zeros, if necessary.
2. Select **Advanced** > **JavaScript** > **Debugger** to display the debugger dialog.
3. In the Scripts window, navigate to the Document script Mult, and set a breakpoint at the call to `func_button(MULT);`
4. In the **Inspect** details window, create a watch for entry and accum.

   At this point both variables have the value 0.???
5. Click **Quit** to close the debugger.
6. Enter the expression 9 * 3 on the calculator pad.

   The debugger executes up to the breakpoint.
7. Click **Step Into** (single step) to execute the first statement in `func_button()`.
8. Single Step two more times.

   This statement contains an error.

   ```
   accum == entry; \\ This is incorrect.
   ```

   The statement should assign the value of `entry` to `accum`. Executing this statement does not update `accum`. See the watches in Figure 3.10.
9. You can either click **Edit** to correct the error or you can click **Step Out** to end the session.

   Clicking **Edit** to correct the error causes the debugger to display this message:

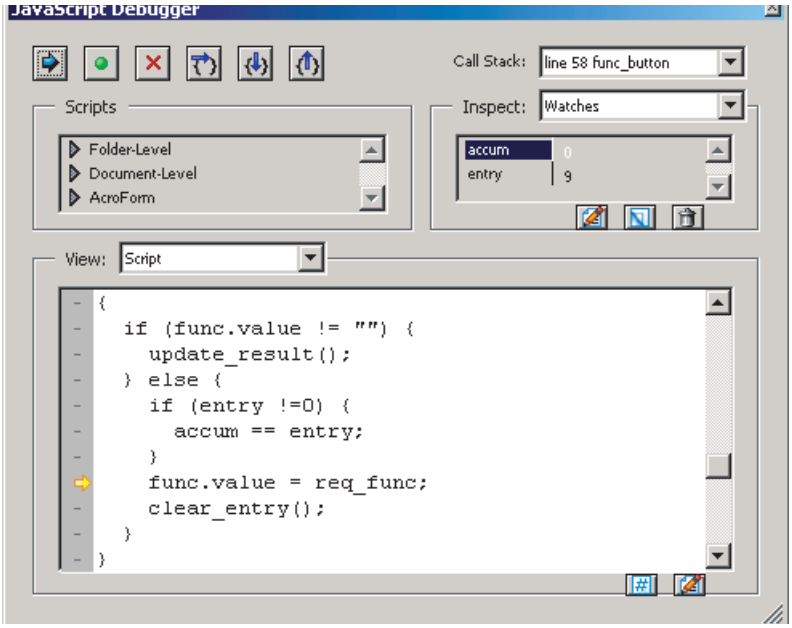
   > If you edit the file, the debugger session will stop. Would you like to edit the file anyway?

   Clicking **Yes** enters the editor where you can correct the error. Change `==` to `=`.

   NOTE: You will still need to clear the breakpoint. You can quit the debugger, execute the calculator again to the breakpoint and clear it.

   Alternately, you can click **Step Out** to end the debugger session. Then correct the error in the editor and click **Quit**.
10. Clear the calculator display.

FIGURE 3.10 The value of accum is not updated



Another runtime error

1. Enter 9 + 3 =

   The result of evaluation is 3. This is indicates another runtime error.

2. If you didn't delete your watches from the previous session, you can reuse them. For each watch, select the watch in the **Inspect** details window. Click **Edit** and **OK**.

   The variable becomes undefined.

3. In the **Scripts** window, navigate to the Equals script.

4. Set a breakpoint at the function call `update_result();`

5. Quit the debugger.

6. Clear the display if necessary.

7. Enter 9 + 3 =

   This enters the debugger and stops execution at the breakpoint.

8. Single step through the function `update_result()`.

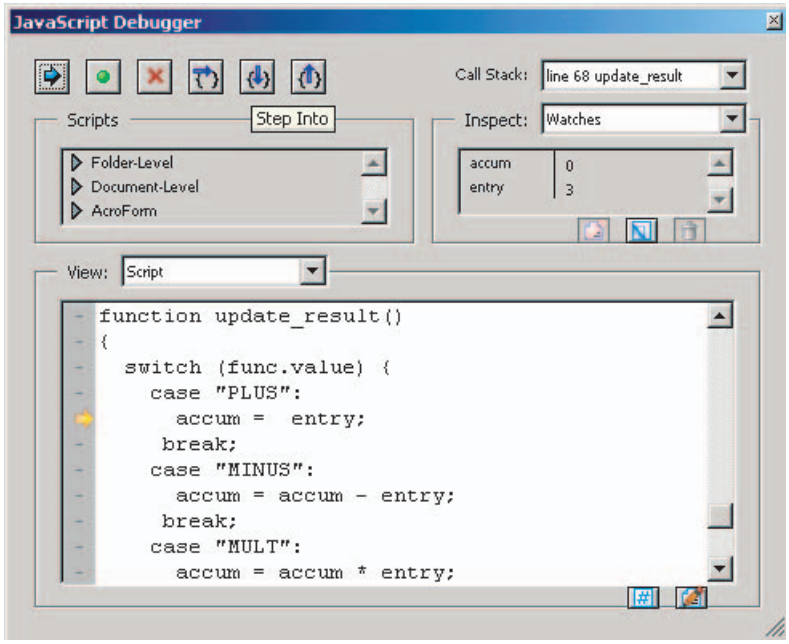
   Note the statements executing and the values of the watches. Execute the first case statement:

   ```
   accum = entry; // This is incorrect.
   ```

   This is incorrect. The value of `entry` should be added to `accum`. Correct the error:

   ```
   accum = accum + entry;
   ```

FIGURE 3.11 Second Runtime Error



Known Issues

Due to some limitations in the JavaScript Debugger, debugging of certain classes of scripts in Acrobat is disabled or unavailable. Fortunately, there are plenty of other places and situations where scripts can be debugged, so that your script can be tested and then transported to its definitive place in case debugging is unavailable. Below is an incomplete list of issues we have found during testing and development of the JavaScript Debugger in Acrobat 6.0.

Menu-item scripts cannot be debugged or reached via the "Debug From Start" menu option. You'll notice that if the "Debug From Start" menu item option is checked on, menu items created via the JavaScript method app.addMenuItem() will be disabled. The correct way to debug these scripts is to put a breakpoint on the function called from the "cExec" parameter defined in the respective addMenuItem() call. This function may exist either in a Config.js file, or in a doc-level script, both available for perusal and setting breakpoints via the debugger dialog.

When trying to debug scripts at PDF file open time (e.g.: Doc-Level scripts) by turning on the "Debug From Start" menu item option, other scripts may be triggered while we're stopped in the debugger (e.g.: Page 1 Open scripts). The posterior execution of these scripts may be incorrect, especially if they depend on definitions contained in the halted doc-level script being debugged.

The events DocWillPrint and DocDidPrint cannot be debugged because the printing progress bar conflicts with the debugger. Press the Escape key to clear this situation. Debug scripts using different event triggers and then change the trigger to the required event.

Debugging scripts in Acrobat from inside a browser (IE, NS) has some limitations. They're mostly due to the fact that not all scripts contained in a PDF file may be available if the PDF file has not been completely downloaded.

Debugging is not possible with a model dialog open. This occurs, for example, when debugging a batch sequence. A script may create and define a dialog and become not debuggable due to the modal characteristic of these dialogs. Debugging these scripts may hang the Acrobat application, in which case the correct exit is to hit the Esc key.

Debugging script with a running event initiated by either `app.setInterval` or `app.setTimeOut` methods may cause a recurring alert boxes to appear. Use the Esc key after the model dialog is dismissed to resolve the problem.

Hitting the Esc key is the last resort in case debugging of scripts causes the Acrobat application to hang.

Summary

The Acrobat JavaScript Debugger is a is a fully capable tool for troubleshooting problems in Acrobat JavaScript scripts. In combination with the **Edit > Preferences** dialog, the debugger enables you to make very specific choices about how to control the behavior of your Acrobat JavaScript development environment. It consists of three main sets of controls for selecting and executing scripts. Six button controls allow starting and quitting the debugger, as well as single stepping through code, skipping over functions, and exiting functions at any point. A **Scripts** window enables you select the script you want to debug. **Call Stack** and **Inspect** lists, in combination with an **Inspect** details window, enable you to view nested functions and to inspect the details of variables, watches, and breakpoints. You can use the details window controls to set and clear breakpoints, inspect variable values while stepping through JavaScript code, and to create and delete custom watches to filter what you want to examine during script execution. You can also trace the stack and view the exact locations of stack frames in a script.

4 Using Acrobat JavaScript in Forms

Creating simple JavaScripts

There are a number of simple JavaScripts you can integrate into your forms to enhance their interactive capabilities. The scripts described here are commonly used with Acrobat forms. Trying out these scripts in the forms you create will give you a glimpse of what JavaScript offers.

Careful selection of field names when creating forms is an important factor in data collection. If two fields share the same name, they also share the same value. You can use this capability to create fields that have different appearances (that is, appear on different pages and have different background colors) but have the same value. This means you can modify one field and the other field is updated automatically.

NOTE: In order to create and use JavaScript in Acrobat, you must have JavaScript enabled. Make sure that JavaScript is enabled by choosing **Edit > Preferences > General** and selecting **JavaScript** from the list on the left. Select **Enable Acrobat JavaScript** if it is not already selected, and click **OK**.

Creating an automatic date field

Many forms require a date for tracking purposes. The following procedure shows you how to create a text field that automatically displays the current date when the document is opened.

The script you create to display the current date when the document is opened is a document level script.

To create an automatic date field:

1. Select the form tool , and create a text field. (For more information, see the section on Creating Form Fields in the Acrobat online help.) Select the **General** tab and name the field *Today*.
2. Select the **Format** tab, choose **Date** for the format category, and choose a month, day, and year format option (for example, "**mmm d, yyyy**"). On the **General** tab, make sure the field is read-only because it will be a calculated field, and click **Close**.
3. To create a document level script that is executed each time the document is opened, choose **Advanced > JavaScript > Document JavaScripts**. Name the script `Today`, and click **Add**.

4. Delete the automatically generated text, defining a **Today** function, that is displayed in the script window. Type in the following text in the exact format (line wraps are OK) and click **OK**.

```
var f = this.getField("Today");
f.value = util.printd("mm/dd/yyyy", new Date());
```

This script binds the **Today** field to the variable **f**, and then calculates the value. The **new Date()** expression creates a new date object initialized to the current date and time. The **util** object is used to format the date into the month/day/year format.

5. Click OK in the JavaScript Edit dialog box, and then click Close in the JavaScript Functions dialog box.

Notice that the date is displayed in the format you selected for the text field, not in the format produced by **util.printd**.

Performing Arithmetic Calculations

You can use JavaScript to automatically perform arithmetic calculations using the values in two or more fields and display the results in another field by using Simplified Field Notation. In the following example, you create three form fields in which the value in the second field is subtracted from the value in the first. The results are calculated, and the value is automatically displayed in the third field.

To create an arithmetic calculation of fields:

1. Select the form tool , and create a text field. (For more information, see the section on Creating Form Fields in the Acrobat online help.) Name this field **ValueA** with no spaces.
2. Click the **Format** tab, choose **Number**, and choose the number of decimal places, a currency symbol if needed, and a separator style.
3. Click the **Options** tab, specify a default value (for example, 1), and click **OK**.
4. Create a second text field, and name this field **ValueB** with no spaces.
5. Click the **Format** tab, and make the format of this field the same as for the previous field.
6. Click the **Options** tab, specify a default value (for example, 1), and click **OK**.
7. Create a third text field, and name this field **ResultsC** with no spaces.
8. Click the Format tab, and make the format of this field the same as for the previous field.
9. Click the **Calculate** tab, click the **Simplified field notation** button, and then click **Edit**.
10. In the JavaScript Editor window, type in the following text exactly as shown:

```
ValueA - ValueB
```

11. Click **OK**.

12.•Click **Close** in the Text Field Properties dialog.

In Acrobat 5 you would have had to explicitly get each field from the **this** object and use the value properties of each field object, as in the following example (equivalent to the symplified syntax in step 10, above):

```
var f = this.getField("ValueA");
var g = this.getField("ValueB");
event.value = f.value - g.value;
```

The simplifed field notation makes it very easy to define relatively complex calculations. Instead of typing:

```
event.value = ( getField("income.interest").value
     + getField("income.rental").value )*0.45
     - getField("deductible").value;
```

the new syntax would require only:

```
(income\.interest + income\.rental)*0.45 - deductible
```

Note the backslash (\) before the period or dot (**.**) in the above example. To avoid ambiguity, all operators (including the dot in the example), numbers, and whitespace characters are assumed to separate field names. If those characters are used in a field name, they may be specified in the Simplified Field Notation by escaping them with a backslash character, as in the example. Quote characters used in a SimplifiedField Notation script will be treated as part of a field name.

For even more complex calculations and logic, you can still use the **Custom calculation script** choice. Scripts entered in this window require the full object and property syntax.

Assigning a 'go to page' action

If you create a multiple page form, it is useful to add a button that automatically takes you to the next page. This type of action is most commonly associated with the Mouse Up action.

The JavaScript you use to take you to the next page at the click of a button can be easily modified to automatically take you to the previous page of the form, the first page of the form, or the last page of the form. All these variations are presented in the following procedure.

To specify a "go to page" action for a button:

1. Select the form tool , and create a form field. (For information, see the section on Creating Form Fields in the Acrobat online help.) Name this field **GoNext**.

If you want to create several 'go to page' buttons on the same form, name each field accordingly: **GoNext**, **GoPrev**, **GoFirst**, and **GoLast**.

2. Choose Button from the Type menu, and specify the border, background, text, and field appearances. Click the Options tab, and specify selections as needed. (For more information, see "Creating interactive buttons" in the Acrobat online help.)

3. Click the Actions tab, choose Mouse Up, and then click Add.
4. Choose JavaScript from the Type menu, and then click Edit.
5. To specify *go to the next page* when the button is selected, in the script window, type in the following text in the exact format, and click OK:

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

For other go to page buttons, use the following scripts with the appropriate button fields:

Go to the previous page:

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

Go to the first page:

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

Go to the last page:

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

6. Click Set Action, and then click OK in the Field Properties dialog box.

Sending a document or form via e-mail

You can create a button on your form that automatically mails the PDF document to a specified e-mail address when selected. You can also specify that only the form data is mailed as an FDF file.

In the following example, the ***name@address.com*** variable represents the e-mail address to which the form is to be sent. Most e-mails have a message subject that gives a brief description of the content of the message. The *Message Subject Description* variable in the example represents the description that would accompany an e-mail message. The double sets of quotes are where you can enter a cc: e-mail address and (blind) bcc: e-mail address, if desired.

To assign an action that e-mails a document or form:

1. Select the form tool , and create a form field. (For more information, see the section on Creating Form Fields in the Acrobat online help.) Name this field *MailPDF*.

If you want to create a second button that mails only the forms data (an FDF file), do so, and name the field *MailFDF*. An FDF file is smaller in size because it contains only the data entered into the form, and not the form itself.

2. Choose Button from the Type menu, and specify the border, background, text, and field appearances. Click the Options tab, and specify selections as needed. (For more information, see "Creating interactive buttons" in the Acrobat online help.)
3. Click the Actions tab, choose Mouse Up, and then click Add.
4. Choose JavaScript from the Type menu, and then click Edit.

5. To mail the PDF document to the specified e-mail address when the button is selected, in the script window, enter the following text in the exact format, and click OK:

```
this.mailDoc(true, "name@address.com", "", "", "Message Subject
Description");
```

To mail the forms data (only) as an FDF file, use the following script instead:

```
this.mailForm(true, "name@address.com", "", "", "Message Subject
Description");
```

6. Click Set Action, and then click OK in the Field Properties dialog box.

Hiding a field until a condition is met

In more complex forms, you might want to have one field that is hidden, or inactive, until a specific condition is met. For example, a field could be hidden, grayed out, or read only until a dollar amount greater than a specified number is entered into another field.

In our example, a dollar amount greater than 100 must be entered in the ActiveValue field to activate the GreaterThan field. The active field is called *ActiveValue*, and the inactive field is called *GreaterThan*.

To activate a field when a condition is met in another field:

1. Select the form tool , and create a text field. (For more information, see the section on Creating Form Fields in the Acrobat online help.) Name the field *ActiveValue*.
2. Click the Format tab, and choose Number from the Category list. Choose two decimal places, Dollar as the Currency Symbol, and the common Separator Style (the default). Click OK.
3. Create a second text field, and name it *GreaterThan*.
4. Click the Format tab, and choose Number from the Category list. Choose two decimal places, Dollar as the Currency Symbol, and the common Separator Style (the default). Click OK.
5. Double-click the ActiveValue field. Click the Validate tab, select Custom Validation Script, and click Edit.
6. Here are three variations in the behavior of the GreaterThan field, with the corresponding code:
   - To keep the GreaterThan field hidden until an amount greater than 100 is entered in the ActiveValue field, in the script window, type in the following in the exact format, and click **OK**:

```
var f = this.getField("GreaterThan");
f.hidden = (event.value < 100);
```

- To keep the GreaterThan field read only until an amount greater than 100 is entered in the ActiveValue field, in the script window, type in the following in the exact format, and click **OK**:

```
var f = this.getField("GreaterThan");
f.readonly = (event.value < 100);
```

- To keep the GreaterThan field grayed out and read only until an amount greater than 100 is entered in the ActiveValue field, in the script window, type in the following in the exact format, and click **OK**:

```
var f = this.getField("GreaterThan");
f.readonly = (event.value < 100) ;
f.textColor = (event.value < 100) ? color.gray : color.black;
```

7. Click **OK** in the JavaScript dialog box, and then click **OK** in the Field Properties dialog box.

Working with JavaScript actions

A JavaScript action allows you to invoke a JavaScript from a form field, a link, a bookmark, a document, or a page action. Familiarity with JavaScript is required. Storing a JavaScript for a commonly used function as a field level script allows you to invoke the function from other JavaScripts. Storing a function as a document level JavaScript makes the function available to all JavaScripts in the current document. Storing a function as a plug-in level script makes the function available to all JavaScripts in the application. Plug-in level scripts are contained in files with a .js extension. These scripts should be located within the Plug-ins folder in the JavaScripts subfolder.

To choose the JavaScript action:

1. Create or select a form field, link, bookmark, or page action.
2. Press the right mouse button (Windows) or Control-click (Mac OS), and choose Properties.
3. Select JavaScript as the action. For information about selecting an action for a form field, link, bookmark, or page action, see the section in the Acrobat online help, "Adding Navigation to Adobe PDF Documents" > "Using Actions for special effects" > "About action types".
4. Click Edit.
5. Copy and paste a predefined custom script, or type the script in the text box provided, and then click OK.
6. Click Set Action or Set Link, as appropriate.

To create a document level JavaScript:

1. Choose **Advanced > JavaScript > Document JavaScripts**.

2. Type the name of the script in the text box.
3. Click Add.
4. Copy and paste a predefined custom script, or type the script in the text box provided, and then click OK. The name of the script appears in the lower text box.
5. Click Close.
6. Choose **Advanced > JavaScript > Debugger** to open the console window. When a JavaScript is executed, you are alerted to any script errors by a message in the console window. Click Clear to clear the results, or Close to close the window.

To edit or delete an existing document level JavaScript:

1. Choose **Advanced > JavaScript > Document JavaScripts**.
2. To edit a document level JavaScript, select the JavaScript from the list, and click Edit. Change the existing text, or paste a predefined custom script into the text box provided. Click OK to accept and conclude the editing.
3. To delete a document level JavaScript, select the JavaScript from the lower text box, and click Delete.
4. Click Close.

To create a plug-in level JavaScript:

1. Create a text file containing the JavaScript function. Name and save the file with a .js extension.
2. Copy the text file into the JavaScripts directory inside the Acrobat folder, or into the JavaScript folder in your system directory.

Working with document level JavaScript actions

In addition to JavaScript functions that can be accessed by any JavaScript in the current document, you can create document level JavaScript actions that apply to the entire document. When you do something that affects the entire document, like printing it, the document level action applied to that occurrence will run. You create and edit the JavaScript code in the editor of your choice. You can later edit all your document level JavaScript actions at once.

To set a document level JavaScript action:

1. Choose **Advanced > JavaScript > Set Document Actions**.
2. In the Document Actions dialog box, choose the appropriate document occurrence.

- Document Will Close runs the JavaScript while the document closes.
- Document Will Save runs the JavaScript while the document is saved.

- Document Did Save runs the JavaScript after the document is saved.
- Document Will Print runs the JavaScript while the document is printed.
- Document Did Print runs the JavaScript after the document is printed.

3. Click Edit. A JavaScript editor appears (you can set a default JavaScript editor in the Preferences dialog box).
4. Enter your code into the editor and click OK. If you are using an external editor, follow the editor's instructions. You will have to save your code and close the editor window before returning to Acrobat. The code is entered into the Execute this JavaScript section of the dialog box, and a green circle appears next to the occurrence to show that a JavaScript action is set for it.
5. Click OK.

To delete a document level JavaScript action:

1. Choose **Advanced > JavaScript > Set Document Actions**.
2. In the Document Actions dialog box, choose the document occurrence whose JavaScript you want to remove.
3. Click Edit.
4. In the JavaScript editor, delete the code and click OK.
5. Click OK to close the Document Actions dialog box.

To edit all document level JavaScript actions:

1. Choose **Advanced > JavaScript > Set Document Actions**.
2. Click Edit All. All scripts for the file will appear. Select the document level scripts.
3. Edit the code in the editor, then click OK.
4. In the dialog box, click OK.

Creating form fields programmatically

Acrobat provides a large number of JavaScript properties and methods for creating form fields and setting their appearance and associated actions, if any. This section presents these properties and methods, organized following the Acrobat user interface (UI). If you create a prototype of your form fields using the Forms toolbar and the Properties dialogs in the Acrobat UI, you can then use this section to find the corresponding JavaScript properties and/or methods.

There are seven types of form fields, described in detail in the following sections:

- Button
- Check Box
- Combo Box
- List Box
- Radio Button
- Signature
- Text

A form field may be created either through the Form toolbar in the Acrobat UI, or by the **addField** method of the **Doc Object**. Programmatically, a form field is created as follows:

```
var f = this.addField("myField", "fieldtype", 0, [100, 472, 172, 400]);
```

The *fieldtype* can be one of **button**, **combobox**, **listbox**, **checkbox**, **radiobutton**, **signature**, or **text**. All of the fields except Radio Button can be created this way. Creating a Radio Button requires more, to connect the multiple buttons. For details of the special requirements, see the Radio Button section on page 71.

The page numbering starts with zero and the coordinates are upper-left *x*, upper-left *y*, lower-right *x* and lower-right *y*. Thus the above line would create a field on the first page of the document located at `[100, 472, 172, 400]`; i.e., the field would be one inch wide (72 points) and 1 inch high, 100 points in from the left and 400 points up from the bottom of the page. Depending on your design requirements, check boxes are typically smaller than this example, while text boxes can often be larger.

A default appearance is given to the field. The return value of this method is a *Field object*, **f**, that will be used throughout the rest of this section.

In all of the Acrobat UI Field Properties dialogs, there is a checkbox labeled **Locked** at the lower left corner. Checking this locks the field against being selected with the form editing tools with a single click and thus possibly accidentally changed through the UI. When this is checked it is still possible to edit the field properties by double clicking. Like the **Close** button at the lower right of each dialog, this checkbox is only accessible through the UI and is not available programmatically.

Note that in Acrobat 5, the form tool UI had a lot in common with the above methodology. You created a form field and then set its type to **button**. In Acrobat 6, instead of a single form field tool, there are separate tools for each type of form field. The programmatic methodology is still the same.

The field object for a field that already exists can be obtained by the **getField** method of the Doc Object:

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

For any field object, you can determine its type using the **type** property. For example, to display the type of the field object **f**, use the command:

```
console.println(f.type);
```

Note that **type** is a read-only property. Contrary to what might be implied by the older Acrobat 5 UI, you cannot change the appearance of a field by changing its type. Each field type is a different object type.

All field types have the same properties availabe in the **General** tab of the **Properties** dialog. It is possible through the UI in Acrobat to specify the *Appearance* of the field, its *Options* and its *Actions*, all of which appear as tabs in the UI for a button field. All these properties can be accessed through field-level JavaScript properties and methods.

Button

The **Button Properties** dialog has four tabs: **General**, **Appearance**, **Options**, and **Actions**.

General Tab

The General tab allows you to set the the following properties.

Name	See Reference	Example
Name	`name`	`console.println(f.name);`
Tooltip	`userName`	`f.userName = "Submit Button"`
Common Properties		
Read Only	`readonly`	`f.readonly = true;`
Form Field	`display`	`f.display = display.visible`
Orientation	`rotation`	`f.rotation = 90;`

Table Notes.

- `name` is a read-only property. It can be set at creation time, either through the UI or programmatically, with `addField`. See the example below.
- Beginning with Acrobat 6.0, the `Field.rotation` property can rotate a button or other field in multiples of 90 degrees. For example, the code below creates a vertically oriented button.

```
var f = this.addField("actionField", "button", 0,
        [200, 250, 250, 400]);
f.strokeColor = color.black;
f.fillColor = color.ltGray;
f.borderStyle = border.b;
f.buttonSetCaption("Push Me");
f.rotation = 90;
```

In Acrobat 5, a rotated button can be created by rotating the page, creating the button, then rotating back again, as in this example:

```
this.setPageRotations(this.pageNum, this.pageNum, 90);
var f = this.addField("actionField", "button", 0, [200, 250, 300, 200]);
f.delay = true;
f.strokeColor = color.black;
f.fillColor = color.ltGray;
f.borderStyle = border.b;
f.delay=false;
this.setPageRotations(this.pageNum, this.pageNum);
```

Appearance Tab

The **Appearance** tab allows you to set the basic appearance of the field. The table below summarizes how the appearance can be set programmatically using JavaScript.

Region/Name	See Reference	Example
Borders and Colors		
Border Color	`strokeColor`	`f.strokeColor = color.black;`
Fill Color	`fillColor`	`f.fillColor = color.ltGray;`
Line Thickness	`lineWidth`	`f.lineWidth = 1;`
Line Style	`borderStyle`	`f.borderStyle = style.b`
Text		
Font Size	`textSize`	`f.textSize = 16;`
Text Color	`textColor`	`f.textColor = color.blue;`
Font	`textFont`	`f.textFont = font.Times;`

Options Tab

The **Options** tab allows you to set the highlighting, the layout and the button-face attributes.

Region/Name	See Reference	Example
Layout	`buttonPosition`	`f.buttonPosition = position.iconOnly;`
Behavior	`highlight`	`f.hightlight = hightlight.p`
Advanced ...		
When to Scale	`buttonScaleWhen`	`f.buttonScaleWhen = scaleHow.always`
Scale	`buttonScaleHow`	`f.buttonScaleHow = scaleHow.proportional`
Fit to bounds	`buttonFitBounds`	`buttonFitBounds = true;`
Button/Icon position scale	`buttonAlignX` and `buttonAlignY`	`f.buttonAlignX = 50;` `f.buttonAlignY = 50;`
Icon and Label		
State	`buttonSetIcon`	`f.buttonSetIcon(i);`
Label	`buttonSetCaption` and `buttonGetCaption`	`f.buttonSetCaption("Push Me");`

Region/Name	See Reference	Example
Icon	`buttonSetIcon`	See table notes

Table Notes.

- For "Select Icon" under "Button Face Attributes", the basic tool for associating a icon with a button face is `buttonSetIcon`; however, this assumes there is a named icon already in the PDF file. See the Doc Object `addIcon` for a complete example of inserting an icon into a button face.
- `buttonImportIcon` can be used to introduce named icons into the document.

Actions Tab

The action of the button can be set with the field level `setAction` method with trigger names "MouseUp", "MouseDown", "MouseEnter", "MouseExit", "OnFocus", "OnBlur". For example,

```
f.setAction("MouseUp", "app.beep(0);")
```

Check Box

The **Check Box Properties** dialog has four tabs: **General**, **Appearance**, **Options**, and **Actions**.

General Tab

The **General** tab shows the same set of properties as for buttons. For details, see the General Tab for buttons.

Appearance Tab

The **Appearance** tab shows the same set of properties as for buttons. For details, see the Appearance Tab for buttons. There is one difference, however — there is no choice for `textFont`. In the case of a check box, the font is always *Adobe Pi*.

Options Tab

The Options tab allows you to set the style of check mark used in the field and the export value.

Region/Name	See Reference	Example
Check Style	`style`	`f.style = style.ci;`
Export Value	`setFocus`	`f.setExportValues(["buy"]);`
Check box is checked by default	`defaultIsChecked`	`f.defaultIsChecked(0);` See table notes

Table Notes.

- Check box is checked by default: After using `defaultIsChecked`, the field is not necessarily checked. To check the field, either reset the field, `this.resetForm([f.name])`, or apply the `checkThisBox`: `f.checkThisBox(0);`
- Check box is checked by default: To determine if the "Check box is checked by default" check box on the Options tab is "checked", use `isDefaultChecked`.
- To determine if a check box is "checked", use `isBoxChecked`.

Actions Tab

The **Actions** tab shows the same set of properties as for buttons. For details, see the Actions Tab for buttons.

Combo Box

The **Combo Box Properties** dialog has seven tabs: **General**, **Appearance**, **Options**, **Actions**, **Format**, **Validate**, and **Calculate**.

General Tab

The **General** tab shows the same set of properties as for buttons. For details, see the General Tab for buttons.

Appearance Tab

The **Appearance** tab shows the same set of properties as for buttons. For details, see the Appearance Tab for buttons.

Options Tab

The Options tab allows the UI user to manipulate the item list of Combo Box. Items can be added and their names and their corresponding export value can be set. The order of items in the list can be manipulated and sorted. Items can be deleted. Direct programatic equivants to these individual operations are limited, but the same operations may be performed using the methods and properties: `setItems`, `numItems`, `getItemAt`, `insertItemAt`, `deleteItemAt`, `clearItems`, and `currentValueIndices`.

In addition, the following settings in the UI do have direct programmatic equivalents:

Region/Name	See Reference	Example
Allow user to enter custom text	`editable`	`f.editable = true;`
Check spelling	`doNotSpellCheck`	`f.doNotSpellCheck = true;`
Commit selected value immediately	`commitOnSelChange`	`f.commitOnSelChange = true;`

Table Notes.

- Item and Export Value: After a combo box is created with **addField**. the item names their export values can easily be introduced using **setItems**; for example

```
            f.setItems([ ["California", "CA"], ["Ohio", "OH"],
["Arizona", "AZ"] ]);
```

- Sort: There is no direct hook to this check box on the Options tab. This check box tells Acrobat to sort the list as it is entered in the UI. In the above example of **setItems**, the items are not entered in alphabetical order. Programmatically, the list can be sorted using the sort method of the array object. For example:

```
function compare (a,b) {         // define a compare function
      if (a[0] < b[0] ) return -1;
      if (a[0] > b[0] ) return 1;
      return 0;
}
var tmp = new Array();
var f = this.getField("myCombo");
                            // load [item, exportvalue]
for (var i = 0; i < f.numItems; i++)
      tmp[i] = [f.getItemAt(i,false), f.getItemAt(i)];
tmp.sort(compare);                  // sort of first component
f.clearItems();                     // out with the old
f.setItems(tmp);                    // in with the new
```

Actions Tab

The **Actions** tab shows the same set of properties as for buttons. For details, see the Actions Tab for buttons.

Format Tab

The action of the combo box can be set with the field level **setAction** and a trigger name of "Format". The UI has several categories of formatting, the JavaScript counterparts are listed in the table below. Except for custom formatting, all formats can be realized by using the formatting functions contained in **Javascripts\aform.js**.

Region/Name	See Reference	Example
Number	AFNumber_Format() in Javascripts\aform.js	f.setAction("Format", 'AFNumber_Format(2, 0, 0, 0, "\u20ac", true)'); f.setAction("Keystroke", 'AFNumber_Keystroke(2, 0, 0, 0, "\u20ac", true)');
Percentage	AFPercent_Format()	
Date	AFDate_FormatEx()	

Region/Name	See Reference	Example
Time	`AFTime_Format()`	
Special	`AFSpecial_Format()`	
Custom		see table notes

Table Notes.

- Number: The example in the table corresponds to a comma delimited euro currency with two decimal points in the UI.
- Custom: Any format script that does not use the above mentioned format functions is classified as custom formatting script. Custom keyboard script is set using the `setAction` with a trigger name of "Keystroke".

Validate Tab

The action of the combo box can be set with the field level `setAction` and a trigger name of "Validate". Validation can consist of either verifying that an input value is between given limits, or validation can be performed by a custom script. See Field/Validate for more information.

Region/Name	See Reference	Example
Field value is in range	`AFRange_Validate()` in `Javascripts\aform.js`	`f.setAction("Validate", 'AFRange_Validate(true, 0, true, 100)'); /* value between 0 and 100, inclusive */`
Run custom validation script		see table notes

Table Notes.

- Custom: Any validate script that does not use the `AFRange_Validate()` function is classified as custom.

Calculate Tab

The action of the combo box can be set with the field level `setAction` and a trigger name of "Calculate". The UI has three categories of Calculate, the JavaScript counterpart is the `AFSimple_Calculate()` method, as listed in the table below.

Region/Name	See Reference	Example
Value is the sum (product, average, minimum, maximum) of the following fields:	`AFSimple_Calculate()` in `Javascripts\aform.js`	`f.setAction("Calculate", 'AFSimple_Calculate("SUM", new Array ("line.1", "line.3"))' );`
Custom		see table notes

Table Notes.

- Custom: Any calculate script that does not use the `AFSimple_Calculate()` function is classified as custom.

Miscellaneous Programming Notes

- The number of items in a combo (or list) box can be queried using the property `numItems`.
- `getItemAt` can be used to get the face name (the item name) and/or the export value of that item.
- `insertItemAt` can be used to insert a new item into a combo (or list) box.
- `deleteItemAt` can be used to delete an item from a combo (or list) box.
- `clearItems` can be used to delete the whole list from the combo (or list) box.
- `currentValueIndices` can be used to change the current value of the combo (or list) box. For example, putting f.currentValueIndices = 2 will make the third item (0 based) the current value of combo box. (Its export value will be exported, if the form is submitted.)

List Box

The **List Box Properties** dialog has five tabs: **General**, **Appearance**, **Options**, **Actions**, and **Selection Change**.

General Tab

The **General** tab shows the same set of properties as for buttons. For details, see the General Tab for buttons.

Appearance Tab

The **Appearance** tab shows the same set of properties as for buttons. For details, see the Appearance Tab for buttons.

Options Tab

The Options Tab show the same set of properties as the comobox, with the exception that **Allow user to enter custom text** and **Check spelling** are not available and Multiple selection is available.

Region/Name	See Reference	Example
Multiple Selection	`multipleSelection`	`f.multipleSelection = true;`

Table Notes. See Table Notes of the combo box.

Actions Tab

The **Actions** tab shows the same set of properties as for buttons. For details, see the Actions Tab for buttons.

Selection Change Tab

The action of the list box can be set with the field level `setAction` and a trigger name of "Keystroke".

Example:

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

Miscellaneous Programming Notes

See the Miscellaneous Programming Notes of the combo box.

Radio Button

A radio button field may be created either by the UI for Acrobat, or by the `addField` of the `Doc Object`. Unlike other form fields, a radio button requires multiple fields to be created with the same name. Programmatically, a radio button field is created as follows:

```
this.addField("myRadio", "radiobutton", 0, [400, 442, 412, 430]);
this.addField("myRadio", "radiobutton", 0, [400, 427, 412, 415]);
var f = this.addField("myRadio", "radiobutton", 0,
            [400, 412, 412, 400]);
f.setExportValues(["Yes", "No", "Sometimes"]);
```

This would create a series of three radio buttons on page 0; each radio button would be 12 points wide and 12 points high. A default appearance is given to the field. The export values of the different buttons are defined by using `setFocus`.

The UI for a radio button is exactly the same as that of a check box. See the section on Check Box to see how to change the appearance, set the options and actions of a radio button field.

General Tab

The **General** tab shows the same set of properties as for buttons. For details, see the General Tab for buttons.

Appearance Tab

The **Appearance** tab shows the same set of properties as for buttons. For details, see the Appearance Tab for buttons.

Options Tab

The Options Tab of a radio button field is the same as that of a check box. The Table Notes are applicable as well.

Actions Tab

The **Actions** tab shows the same set of properties as for buttons. For details, see the Actions Tab for buttons.

Signature

The **Signature Properties** dialog has four tabs: **General**, **Appearance**, **Actions**, and **Signed**.

General Tab

The **General** tab shows the same set of properties as for buttons. For details, see the General Tab for buttons.

Appearance Tab

The **Appearance** tab shows the same set of properties as for buttons. For details, see the Appearance Tab for buttons.

Actions Tab

The **Actions** tab shows the same set of properties as for buttons. For details, see the Actions Tab for buttons.

Signed Tab

The action of the signature field can be set with the field level `setAction` and a trigger name of "Format". The signing of the form by the user and the subsequent changes to the status of various fields is handled programmatically as reformatting. The UI has several categories of Signed, the JavaScript counterparts are listed in the table below. Except for custom formatting, all formats can be realized by using the formatting functions contained in `Aform.js`.

Region/Name	See Reference	Example
Nothing happens when signed		The default behavior, values null.
Mark as read-only	`setLock`	`oLock.action = "Include";`
Pick	`setLock`	`new Array ("mySignature")`

Region/Name	See Reference	Example
This script executes when field is signed	setAction	new Array ("mySignature")

An Acrobat 5 Example

Here is a complete example to create, sign, and lock a signature field using JavaScript using the **AFSignature_Format** method defined in **Aform.js** in Acrobat 5 and later:

```
// Create signature field dynamically
var f = this.addField("mySignature", "signature", 0,
            [200, 500, 500, 400]);
f.strokeColor = color.black;

// set it to lock when signed
f.setAction("Format",
     'AFSignature_Format("THESE", new Array ("mySignature"));' );

var ppklite = security.getHandler("Adobe.PPKLite");    // choose handler
ppklite.login("dps017", "/C/signatures/DPSmith.pfx");  // login
f.signatureSign(ppklite,                               // sign it
     { password: "dps017",
           location: "San Jose, CA",
           reason: "I am approving this document",
           contactInfo: "dpsmith@adobe.com",
           appearance: "Fancy"});
ppklite.logout();                                      // logout
```

Text

The **Text Field Properties** dialog has seven tabs: **General**, **Appearance**, **Options**, **Actions**, **Format**, **Validate**, and **Calculate**.

General Tab

The **General** tab shows the same set of properties as for buttons. For details, see the General Tab for buttons.

Appearance Tab

The **Appearance** tab shows the same set of properties as for buttons. For details, see the Appearance Tab for buttons.

Options Tab

In the Options tab, the default text can be entered as various text field attributes can be set.

Region/Name	See Reference	Example
Alignment	alignment	f.alignment = "center";

Region/Name	See Reference	Example
Default Value	`defaultValue`	`f.defaultValue = "Name: ";`
Multiline	`multiline`	`f.multiline = true;`
Scroll long text	`doNotScroll`	`f.doNotScroll = true;`
Allow Rich Text Formatting	`richText`	`f.richText = true;`
Limit of # characters	`charLimit`	`f.charLimit = 40;`
Password	`password`	`f.password = true;`
Field is used for file selection	`exportValues`	`f.fileSelect = false;`
Check Spelling	`doNotSpellCheck`	`f.doNotSpellCheck = true;`
Comb of # characters	`comb`	`f.comb = true;` Note: the number of characters allowed in the comb field is determined by `f.charLimit`.

Actions Tab

The **Actions** tab shows the same set of properties as for buttons. For details, see the Actions Tab for buttons.

Format Tab

The Format Tab tab is the same as that of the combo box.

Validate Tab

The Validate Tab tab is the same as that of the combo box.

Calculate Tab

The Calculate Tab tab is the same as that of the combo box.

A Short Acrobat JavaScript FAQ[1]

Where can JavaScripts be found and how are they used?

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

Folder Level JavaScripts

JavaScripts can be placed in individual files with the "`.js`" extension. For such files to be read by the viewer at startup they need to be placed in either the Acrobat Application `JavaScripts` folder or in the user's `JavaScripts` folder. See App/Init in the Event Object section of the *Acrobat JavaScript Scripting Reference* for a discussion of the order these files are loaded into the application on startup.

You should place variables and functions that might be generally useful to the application in these folders. Note that some JavaScripts methods can only be executed from within one of these JavaScript files at startup, for example, `addMenuItem` and `addSubMenu`.

There are some restrictions when writing JavaScript files, particularly with respect to the use of the default this Object.

The standard Acrobat JavaScript implementation comes with three JavaScript files: `Aform.js` which contains built-in pre-canned functions, `Annots.js` which is used by the Annotations plug-in and `ADBC.js` used by the ADBC plug-in. These are located in the application `JavaScripts` folder.

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

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

Document level

By using the Adobe Acrobat menu item **Advanced > JavaScript > Document JavaScripts...**, you can add, modify, or delete document level scripts. These scripts should be function definitions that are generally useful to the document but are not applicable outside the document. Document level scripts are executed after the document has opened and after the plug-in level scripts are loaded. Document level scripts are stored

1. Frequently Asked Questions

within the PDF document. Therefore, the forms programmer should make every effort to package scripts as effectively as possible.

FDF programmers should also be aware that FDF files can include document-level scripts and can also include scripts specified to be executed just before the FDF gets imported or executed right after the FDF gets imported.

Field level

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

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

How should I name my form fields?

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

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

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

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

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

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

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

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

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

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

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

How do I use date objects?

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

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

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

To convert a **Date** object into a string, the printd 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: 01/15/99
Format yy/mm/dd looks like: 99/1/15
Format mmmm dd, yyyy looks like: January 15, 1999
Format dd-mmm-yyyy looks like: 15-Jan-1999
```

NOTE: (TIP) Given the ever increasing length of the human lifespan and the lessons we've learned from Y2K coding issues, it is advised that you output dates with a four digit year to avoid ambiguity.

Converting from a string to a date

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

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

The output of this script would look like:

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

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

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

Date arithmetic

It is often useful to do arithmetic on dates to determine things like the time interval between two dates or what the date will be several days or weeks in the future. The JavaScript **Date** object provides several ways to do this. The simplest and possibly most easily understood method is by manipulating dates in terms of their numeric

representation. Internally, JavaScript dates are stored as the number of milliseconds (one thousand milliseconds is one whole second) since a fixed date and time. This number can be retrieved through the valueOf method of the **Date** object. The **Date** constructor allows the construction of a new date from this number.

```
/* Example of date arithmetic. */
/* Create a Date object with a definite date. */
var d1 = util.scand("mm/dd/yy", "4/11/76");
/* Create a date object containing the current date. */
var d2 = new Date();
/* Number of seconds difference. */
var diff = (d2.valueOf() - d1.valueOf()) / 1000;
/* Print some interesting stuff to the console. */
console.println("It has been "
     + diff + " seconds since 4/11/1976");
console.println("It has been "
     + diff / 60 + " minutes since 4/11/1976");
console.println("It has been "
     + (diff / 60) / 60 + " hours since 4/11/1976");
console.println("It has been "
     + ((diff / 60) / 60) / 24 + " days since 4/11/1976");
console.println("It has been "
     + (((diff / 60) / 60) / 24) / 365 + " years since 4/11/1976");
```

The output of this script would look something like:

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

The following example shows the addition of dates.

```
/* Example of date arithmetic. */
/* Create a date object containing the current date. */
var d1 = new Date();
/* num contains the numeric representation of the current date. */
var num = d1.valueOf();
/* Add thirteen days to today's date, 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 my document secure?

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

Restricting Access to the Document

If the author desires to restrict access to the form in its entirety then the standard security model in Acrobat can be selected and an open password defined that requires a user to type in a password before opening the form. Other security handlers exist and are provided by third party developers as plug-ins and may also be useful. E.g. using a public/private key infrastructure to lock a form for a particular set of people or allowing a form to expire after a certain time period.

The ability to set a user password is accessed by choosing **File > Document Properties...** from the **Acrobat** menu, then select **Security** from the left-hand pane. From the drop down menu, select **Password Security**. You can now set the password and permissions as desired.

Restricting Permissions

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

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

If the *No Changing the Document* restriction is selected, the user can fill-in form fields and add annotations but cannot author or modify form fields or change the background text using the TouchUp plug-in.

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

Digital Signatures

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

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

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

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

See the section on the Signature field on page 72 for a discussion on how to create and sign a digital signature field programmatically. Also see the section on "How can I lock a document after a signature field has been signed?" on page 82.

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 "Creating a certifying signature" in Adobe Acrobat 6.0 Help.
- The recipient should trust the signer for certified documents and JavaScript.

  For details, see "Managing digital ID certificates" in Adobe Acrobat 6.0 Help.

All trusted certificates can be accessed by selecting Certificates from **Advanced > Manage Digital IDs > Trusted Identities** in Acrobat's main menu. Click Edit open and view a certificate. To enable JavaScript execution, check the "Trusted for JavaScript" check box in the certificate.

How can I lock a document after a signature field has been signed?

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

A signature field's value is read-only. An unsigned signature has a null value. Once the field has been signed its value is non-null. When crafting a custom script for when the signature field is signed, remember to allow for resetting the form (i.e. the field's value is set to null).

The **setLock** method of the Field Object object controls which fields are to be locked when a signature is applied to the corresponding **signature** field.

In addition, there is a convenience routine available for your use called **AFSignatureLock()** in **AForm.js** (see the section "Where can JavaScripts be found and how are they used?") that is available in Acrobat 5 and performs the programmatic equivalent of the simple locking user interface in the signature properties dialog. This allows you to easily lock and unlock all fields, a particular list of fields, or all fields but those specified. The example is re-coded using this convenience routine below:

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

See the comments in **AForm.js** for more details.

How can I make my documents accessible?

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. Beginning with the 4.05 release of Acrobat, Adobe has worked to allow motion and vision impaired users to fill out Acrobat Forms. Version 6 of Acrobat is more fully speech-enabled than ever.

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 filename is used. Often, file names are abbreviated or changed and as such it is highly encouraged that the document author specify a title for the document. For example, if a document has a file name of "**IRS1040.pdf**" a good document title would be "Form 1040: U.S. Individual Income Tax Return for 1998".

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 name (tooltip) that is user friendly. 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 his 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 globals in JavaScript?

The Acrobat extensions to JavaScript define a **`Global`** object to which you can attach global variables as properties. To define a new global called '**`myVariable`**' and set it equal to the null string, you would type:

```
global.myVariable = "";
```

All of your scripts, no matter where they are in a document, will now be able to reference this variable.

Making Globals Persistent

Global data does not persist across user sessions unless you specifically make your globals persistent. The predefined **Global** object has a method designed to do this. To make a variable named '**myVariable**' persist across sessions, do this:

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

In future sessions, the variable will still exist (with its previous value intact).

How can I send form data to an e-mail address?

You can use the **submitForm** of the Field Object to accomplish this:

```
var url = "mailto:johndoe@doe.net";
this.submitForm(url,false);
```

In this instance, the form contents will be sent to the address given in the variable **url**. The second argument of **submitForm()** determines whether the form contents are sent as url-encoded data using the POST method, or sent as FDF (Forms Data Format). A value of "false" means the data will be sent in url-encoded fashion.

How can I hide a field based on the value of another?

Use the **display** of the Field Object.

```
var title = this.getField("title");
if (this.getField("showTitle").value == "Off")
   title.display = display.hidden;
else
   title.display = display.visible;
```

How can I query a field value in another open form from the form I'm working on?

One way would be to use the Global Object's **subscribe** method to make the field(s) of interest available to others at runtime. For example, a form could contain a document-level script (invoked when that document is first opened) that defines a global field value of interest:

```
function PublishValue( xyzForm_fieldValue_of_interest ) {
     global.xyz_value = xyzForm_fieldValue_of_interest;
}
```

Then, when your document (Document A) wants to access the value of interest from the other form (Document B), it can subscribe to the variable in question:

```
global.subscribe("xyz_value", ValueUpdate);
```

In this case, ValueUpdate refers to a user-defined function that gets called automatically whenever **xyz_value** changes. If you were using **xyz_value** in Document A as part of a field called **MyField**, you might define the callback function this way:

```
function ValueUpdate( newValue ) {
     this.getField("MyField").value = newValue;}
```

How can I intercept keystrokes one by one as they occur?

Create a Custom Keystroke Filter script (see the Format tab in the **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 build a nested popup menu?

Use the **app.popUpMenu()** method. Create an array of menu selections, then call **app.popUpMenu(arrayName)** from the mouse-down or mouse-up event of a given field to pop the menu. For Acrobat 6.0, you can also use the **app.popUpMenuEx()** method.

Example:

```
var cChoice = app.popUpMenu("one", "two", "-",
     [ "three", "three.one", "three.two" ] );
app.alert("You chose " + cChoice);
```

How can I construct my own colors?

Colors are **Array** objects in which the first item in the array is a string describing the color space ('G' for grayscale, 'RGB' for RGB, 'CMYK' for CMYK) and the following items are numeric values for the respective components of the color space. Hence:

```
color.blue = new Array("RGB", 0, 0, 1);
color.cyan = new Array("CMYK", 1, 0, 0, 0);
```

To make a custom color, just declare an array containing the color-space type and channel values you want to use.

How can I prompt the user for a response in a dialog?

Use the **response** defined in the App Object class. This method displays a dialog box containing a question and an entry field for the user to reply to the question. (Optionally, the dialog can have a title or a default value for the answer to the question.) The return value is a string containing the user's response. If the user presses the dialog's Cancel button, the response is the null object.

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

How can I fetch an URL from JavaScript?

Use the **getURL** of the Doc Object class. This method retrieves the specified URL over the internet using a GET. If the current document is being viewed inside the browser or Acrobat Web Capture is not available, it uses the Weblink plug-in to retrieve the requested URL.

How can I change the hot-help text for a field dynamically?

The **userName** of the Field Object returns/sets the "tooltip" of the field in question, as a string. This property is intended to be used as a short description or hot-help popup whenever the mouse cursor loiters over a field. It can be a great help to the user if you put useful suggestions (or descriptive language of some sort) in the **userName** .

How can I change the zoom factor programmatically?

Use the **zoom** of the Doc Object class. For example, the following code shows two ways to set the zoom factor of the current page:

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

// This sets the zoom to 73%:
this.zoom = 73;
```

Note that the value used for the zoom factor is in full percent, so 73% is specified as `73` and not as `0.73`.

How can I determine if the mouse has entered/left a certain area?

Create an invisible, read-only text field at the place where you want to detect mouse entry or exit. Then attach JavaScripts to the mouse-enter and/or mouse-exit actions of the field.

Index