Batch Sequences

Adobe® Acrobat® SDK

February 2011

Version 10.0

Adobe® Acrobat® X SDK Batch Sequences for Microsoft® Windows® and Mac OS®

Edition 3.0, February2011

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Contents

List of Examples

Preface

This document shows how to use JavaScript™ to do batch processing in Adobe® Acrobat® 5.0 and Acrobat Professional 6.0 or later, and offers sample solutions to some of the common tasks users encounter.

What's in this guide?

This guide discusses the fundamentals of writing a batch sequence that uses the Execute JavaScript batch command. The event object, the use of global variables, debugging and testing tips are discussed. There is also a chapter of batch sequence examples.

Who should read this guide?

This guide would be of interest to anyone who needs to automate repetitive tasks. It requires a knowledge of the Acrobat extensions to JavaScript, as documented in the *JavaScript for Acrobat API Reference*.

Related documentation

This document refers to the following sources for additional information about JavaScript and related technologies. The Acrobat documentation is available through the Acrobat Family Developer Center, http://www.adobe.com/go/acrobat_developer.

For information about	See
Known issues and implementation details.	*Readme*
Answers to frequently asked questions about the Acrobat SDK.	*Developer FAQ*
New features in this Acrobat SDK release.	*What's New*
A general overview of the Acrobat SDK.	*Overview*
Using JavaScript to develop and enhance standard workflows in Acrobat and Adobe Reader®.	*Developing Acrobat Applications Using JavaScript*
Detailed descriptions of JavaScript APIs for developing and enhancing workflows in Acrobat and Adobe Reader.	*JavaScript for Acrobat API Reference*

Using Batch Sequences

A batch sequence performs repetitive operations on a collection of *selected files*. Batch processing sequences, like all JavaScript code, are executed as a result of a particular event. During the event, the script is repeatedly executed on each of the selected files.

This chapter addresses some of the principles of writing a batch sequence that uses the Execute JavaScript batch command. With these principles in mind, and a knowledge of the JavaScript API for Acrobat, you will be able to write your own batch sequences.

Creating and running a batch sequence

A batch sequence is created through the UI of Acrobat Professional. In version 8, the batch sequence feature is accessed by selecting Advanced > Document Processing > Batch Sequences. The following batch sequence files are shipped with Acrobat Professional:

```
Embed Page Thumbnails.sequ
Open All.sequ
Print All.sequ
Save All as RTF.sequ
Fast Web View.sequ
Print 1st Page of All.sequ
Remove File Attachments.sequ
Set Security to No Changes.sequ
```

The batch sequence is saved to a text file with an extension of `sequ`. Acrobat Help covers how to use the batch processing feature, and, in particular, how to use these sequences.

You are more interested in writing your own, so let's walk through the creation of a simple, yet useful, batch sequence.

Task: For each of the selected files, set the `disclosed` property to `true`. (The `disclosed` property of the Doc object is very important to certain document processing operations. When `disclosed` is set to `true`, the `app.activeDocs` and `app.openDoc` methods return Doc objects rather than `null`. See the JavaScript for Acrobat API Reference for documentation on these two methods.)

The procedure below steps you through the creation of a batch sequence. The sequence inserts the script `this.disclosed = true` in each of the selected documents as document JavaScript.

➤ **To create a batch sequence that sets the disclosed property to true:**

1. Select **Advanced** > **Document Processing** > **Batch Sequences**.
2. Click **New Sequence**, and type `Make disclosed` in the **Name Sequence** dialog box.
3. In the **Edit Batch Sequence** dialog box, click the **Select Command** button to get to the **Edit Sequence** dialog box.

4. In the **Edit Sequence** dialog box, double-click **Execute JavaScript** to move it to the right side.
5. Click the **Edit** button to open the JavaScript editor.
6. Type the following code:

```
/* Make disclosed */
this.addScript("This Doc is Disclosed", "this.disclosed = true;");
```

7. Close all the dialog boxes.

The Edit Sequence dialog box now lists `Make disclosed`.

To run this new script you must first select the collection of files that the batch sequence will process. The general procedure below assumes the Edit Sequence dialog box is not open. Follow the procedure for the new `Make disclosed` batch sequence.

➤ **To select files:**

1. Choose **Advanced** > **Document Processing** > **Batch Processing**.
2. In the **Batch Sequences** dialog box, highlight one of the listed sequences, and click **Edit Sequence**.
3. In the **Edit Batch Sequence** dialog box, select one of the following items from the **Run Commands On** list:
   - **Selected Files**
   - **Selected Folders**
   - **Ask When Sequence is Run**
   - **Files Open in Acrobat**
4. Close all the dialog boxes.

Finally, run your new batch sequence.

➤ **To run a batch sequence:**

1. Choose **Advanced** > **Document Processing** > **Batch Processing**.
2. In the **Batch Sequences** dialog box, highlight one of the listed sequences, and click **Edit Sequence**.
3. In the **Edit Batch Sequence** dialog box, verify that the files selected are the ones you want to process. Click **OK**.
4. In the **Batch Sequences** dialog box, click **Run Sequence**. In the **Run Sequence Confirmation** dialog box, click **OK**.
5. After processing is over, close the **Batch Sequences** dialog box.

When creating a batch sequence to perform a task, you can use one or more of the batch commands listed in the Edit Sequence dialog box in combination with the Execute JavaScript batch command; for example, the sample sequence described in Sign all documents uses Execute JavaScript to sign the document, then uses the Security batch command to set the security of the document.

Batch processing objects

When you run a batch sequence that contains an Execute JavaScript batch command, a *batch event* occurs and an `event` object is created. This object contains the following properties:

`target`	During a batch sequence, the `target` property of the `event` object points to the Doc object of the file currently being processed. For example, `event.target.path` is the device-independent path of the file currently being processed. **Note:** The `this` object also points to the Doc object of the file currently being processed in the batch. See the following code lines in the section Using the this object.
`rc`	The `rc` property contains a return code for each application of the sequence. Setting `rc` to `false` aborts the batch process, so that no further files are processed (if any remain in the selected files collection).

Aborting a script

Setting `event.rc = false` aborts the batch process, but does not abort the script itself. The remaining code in the program flow is executed. One strategy to avoid the continuation of the flow is to throw an exception, and set `event.rc = false` with the catch:

```
try {
   ... JS script for batch processing .....
   if ( somethingGoesWrong ) throw "Aborting Process"
   ....JS script for batch processing ....
   } catch (e)
   {
   ... clean up code lines ....
   event.rc = false;
   }
```

Using the this object

The this object points to the Doc object, for example, the following script gathers all annotations in the document currently being processed, sorts them by author, and returns an array of annotation objects for further manipulation:

```
var annots = this.getAnnots({nSortBy: ANSB_Author});
```

Alternatively, you can also use `event.target`:

```
var annots = event.target.getAnnots({nSortBy: ANSB_Author});
```

Global variables

Variables that must hold their values across document processing must be declared as global.

```
global.report = new Report()
global.counter = 0;
```

This declaration causes the `global.report` object to be available for each application of the script to a selected file. For example, the script that processes a file might contain the following code:

```
global.counter++
global.report.writeText("Report on File \#"+global.counter);
```

At the end of the batch job, any global variables can be removed. For example:

```
delete global.counter;
```

Beginning and ending a batch job

More complicated batch jobs that involve cross-document reporting may need some start-up, or *Begin Job* code, to initialize global variables before processing begins. After the files are processed, you can use *End Job* code to clean up or write a report.

The batch feature of Acrobat Professional does not have a built-in capability for *Begin Job* and *End Job* code. The approach the sample sequences takes for detecting the begin and end of job is to use two global variables, `global.counter` and `global.FileCnt`.

- `global.counter`: This variable detects the beginning of the job, counts the files as they are processed and determines when the end of the job occurs. When the batch is run, the variable is initialized to zero and incremented with each iteration of the script, or is initialized to the number of files to be processed and decremented with each iteration. The *End Job* code runs when the variable reaches the number of files being processed, or 0.
- `global.FileCnt`: Most of the work is done by `global.counter`; however, you do need to know how many files are to be processed. This is the role the `global.FileCnt` plays. The value of `global.FileCnt` can be manually set (by executing `global.FileCnt = 5` in the console, for example) or programmatically by another batch sequence. The batch sequence Count PDF files sets the value of `global.FileCnt` to the number of files selected.

Example: *Specify the number of files to process*

The following code is a general outline of how you can insert *Begin Job* and *End Job* code: it uses `global.counter` to detect the beginning of the job and uses another global variable `global.FileCnt`, which is set earlier by another batch sequence that counts the number of files to be processed, see Count PDF files to detect the end of the job.

```
// Begin job
if ( typeof global.counter == "undefined" ) {
     console.println("Begin Job Code");
     global.counter = 0;
     // insert beginJob code here
     ................
}
// Main code to process each of the selected files
try {
   global.counter++
```

```
   console.println("Processing File #" + global.counter);
   //  insert batch code here.
   ..............
} catch(e) {
   console.println("Batch aborted on run #" + global.counter);
   delete global.counter;    // Try again, and avoid End Job code
   event.rc = false;         // Abort batch
}
// End job
if ( global.counter == global.FileCnt ) {
   console.println("End Job Code");
   // Insert endJob code here
   ................
   // Remove any global variables used in case user wants to run
   // another batch sequence using the same variables
   delete global.counter;
}
```

Debugging and testing tips

The batch feature is very useful for performing a number of tasks on large scale. However, it poses its own dangers. Any automated utility that saves files to a hard drive can potentially create havoc on that drive.

Here are some debugging and testing tips:

- When testing the script, use test documents rather than real documents.
- Initially, work with a small number of test documents.
- Use test directories containing files that can be safely overwritten.
- Make sure your paths exist. If an `app.openDoc` is executed with a path that does not exist, an exception is thrown.
- Use `try/catch/throw` to exit gracefully from a batch if something goes wrong (such as bad paths), and to better control the flow of the code. (See the code snippet in Aborting a script.)
- Use `console.println()` to write information to the console for feedback as to the value of different variables as the batch runs.
- When potentially executing an infinite loop, during the testing phase limit the loop to a finite number of executions until you are sure the code is correct. (See the batch sequence described in Populate a form from a database and save it.)
- If you use the *Begin Job/End Job* technique illustrated above, make sure the global counter variable is undefined before the job runs.

2 Batch Sequence Examples

This chapter discusses some common tasks that can be done with batch processing, and offers sample solutions. Some of these solutions are fairly general, while others are specific to the system on which they were developed—that is, they give specific paths, icons, and so on. You should study them carefully, and modify them to suit your needs.

Topics	Description
"Count PDF files" on page 12	Counts the number of PDF files to process, and sets the value of `global.FileCnt`.
"Working with bookmarks" on page 12	Sequences for dealing with bookmarks: gathering, listing, counting, copying and inserting.
"Working with icons" on page 19	Sequences for dealing with icon files: importing and creating navigation buttons.
"Extract pages to a folder" on page 22	For each file from the selected files, extracts each of its pages and saves the extracted pages to a particular folder.
"Populate a form from a database and save it" on page 22	Populates a form with values from a database, and saves the form to a folder.
"Working with security" on page 24	Security related tasks: removing security settings, getting signature information and signing.
"Inserting comments and form fields" on page 26	Tasks that insert an annotation or form field into a specific location on a page.
"Spell check a document" on page 28	Spell checks a document and gives suggested alternative spellings.
"Summarize comments from selected files" on page 28	Writes a report of all comments in the selected files.
"Write comments to a tab-delimited file" on page 31	Writes all comments in the selected files to a tab delimited file.

Installing the sample batch sequences

All batch sequence files discussed in this chapter are contained in `batchseq.zip`. This file is available for download from the Acrobat Developer Center. Though fully tested, these batch sequences are offered with no guarantees.

To use the sample batch sequences, open `batchseq.zip` and extract all files with an extension of `.sequ` into the user `Sequences` folder. To find the location of this folder, execute the following JavaScript method from the console

```
app.getPath("user","sequences");
```

Note: For information on using the JavaScript console in Acrobat, see Developing Acrobat Applications Using JavaScript or the JavaScript for Acrobat API Reference.

➤ **To verify that the batch sequences are installed:**

1. Acrobat reads the `Sequences` folder only on start-up; if you have not done so already, restart Acrobat.
2. Choose **Advanced** > **Document Processing** > **Batch Processing**. In the Batch Sequences dialog box, you will see such sequences as `Change Security to None`, `Count Bookmarks`, and `Count PDF files`.

If the batch sequences are not listed, check the name and location of the folder that contains the sequences.

Note: Read this document before attempting to use these sequences. When you want to run one of these sample batch sequences, and you've read the documentation for it, review "Creating and running a batch sequence" on page 6. The information there includes how to select the files on which the batch sequence is to operate, and how to run the batch sequence.

The rest of this chapter consists of a discussion of common tasks that have a solution using the Execute JavaScript batch command. At the beginning of a section that describes the task, the name is given of the corresponding sample batch sequence that performs the task; generally, a verbose listing of the script is also presented in the form of an example.

Count PDF files

Batch sequence name: `Count PDF files.sequ`.

The batch sequence `Count PDF files.sequ` counts the number of PDF files selected for processing, and assigns that value to the global variable `global.FileCnt`

The following routine is used by some of the sample sequences.

Example: *Count PDF files*

```
/* Count PDF files */
if ( typeof global.FileCnt == "undefined" ) global.FileCnt = 0;
global.FileCnt++
```

This sequence sets the global variable, `global.FileCnt`, equal to the number of selected files. If used prior to running another batch sequence that depends on this value, be sure the selected files are the same.

Working with bookmarks

There are many tasks associated with bookmarks that can be easily performed using the batch sequencing capability of Acrobat. These samples gather, list, and count bookmarks.

Gather bookmarks to an array

Batch sequence name: `Gather Bookmarks to Array.sequ`.

This sequence gathers up bookmark information from the selected files and saves it in a global array. The information in this array is used by the sequence `Copy Bookmarks from Array.sequ` described in "Copy bookmarks from an array" on page 16.

Example: *Gather bookmarks to an array*

```
/* Gather Bookmarks into an Array */
// Begin job setup
if ( typeof global.bmArray == "undefined" ) {
   global.indexCnt = 0;
   global.bmFileCnt = 0;
   global.bmArray = new Array();
}
global.bmFileCnt++; // Count files

// Get the path of the file Acrobat has taken from the selected folder
var path2file = this.path;

// Regular expression that acquires the base name of file
var re = /.*\/|\.pdf$/ig;

// filename is the base name of the file Acrobat is working on
var filename = path2file.replace(re,"");

// If the title is available, use it for a bookmark, else use filename
if ((this.info.Title==null) || (this.info.Title.replace(/\s/g,"")==""))
   var bmToFile = filename;
else
   var bmToFile = this.info.Title;

global.bmArray[global.indexCnt] = [bmToFile,filename];
global.indexCnt++;
```

Once the array `bmArray` is in memory, you can sort the entries with respect to some criteria. The following code sorts the bookmark names alphabetically. The script could be run from the console, or you could insert it into the Begin Job section of "Copy bookmarks from an array" on page 16.

```
function compare (a,b) {     // Define a compare function
   if (a[0] < b[0] ) return -1;
   if (a[0] > b[0] ) return 1;
   return 0;
}
global.bmArray.sort(compare);
```

List all bookmarks

Batch sequence name: `List all Bookmarks.sequ`.

Task: Create a PDF document using the Report object that lists the bookmarks in another PDF document. The new document can be saved and printed.

Example: *List all bookmarks*

```
/* List all Bookmarks */
/* Recursively work through bookmark tree  */
function PrintBookmarks(bm, nLevel)
```

```
{
   if (nLevel != 0) { // don't print the root
      bmReport.absIndent=bmTab*(nLevel-1);
      bmReport.writeText(util.printf("%s",bm.name));
   }
   if (bm.children != null)
      for (var i = 0; i < bm.children.length; i++)
      PrintBookmarks(bm.children[i], nLevel + 1);
}
// Set up the parameters to write a report
bmTab = 20;
bmReport = new Report();
bmReport.size = 2 // Large font size for title
bmReport.writeText(this.title);
bmReport.writeText(" "); // Skip a line
bmReport.size = 1.5; // Slightly smaller for heading
bmReport.writeText("Listing of Bookmarks");
bmReport.writeText(" ");
bmReport.size = 1; // Default size for everything else
PrintBookmarks(this.bookmarkRoot, 0); // Start moving through bookmarks

// Make it global so the object will be "remembered" after batch is done.
// global.bmRep = bmReport;
// Make global for next step

/*
   We now want to open our report (Report.open), but in Acrobat 6 and earlier
   we cannot open or save (Report.save) a report while a modal dialog box
   is open, as there is while the batch is running. So, if the version is 6 or
   earlier, wait until the batch is done, then open the report.
*/
if ( app.viewerVersion < 7 ) {
   global.wrtDoc = app.setInterval(
      'try {'
      +'      reportDoc = global.bmRep.open("Listing of Bookmarks");'
      +'      console.println("Executed Report.open");'
      +'      app.clearInterval(global.wrtDoc);'
      +'      delete global.wrtDoc;'
      +'      console.println("Executed App.clearInterval");'
      +'      reportDoc.info.title = "Bookmark Listings";'
      +'      reportDoc.info.Author = "A. C. Robat";'
      +'} catch (e) {console.println("Waiting...: " + e);}'
      , 100); // Check every 1/10th of a second. You can adjust this
} else {
   reportDoc = global.bmRep.open("Listing of Bookmarks");
   console.println("Executed Report.open");
   reportDoc.info.title = "Bookmark Listings";
   reportDoc.info.Author = "A. C. Robat";
}
```

You can edit the script and customize the look for the report document; for instance, you can add bullets at different levels, or only report on the top level bookmarks. You can use the TouchUp Object Tool to copy and paste the column of bookmarks to make a two or three column format.

Count bookmarks

Batch sequence name: `Count Bookmarks.sequ`.

Task: Count the number of bookmarks in a given PDF document. This sequence is general enough that it can be run on any of the file selection options.

Example: *Count bookmarks*

```
/* Count Bookmarks */
/* Recursively work through bookmark tree  */
function CountBookmarks(bm, nLevel)
{
   if (nLevel != 0) counter++       // Don't count the root
   if (bm.children != null)
   for (var i = 0; i < bm.children.length; i++)
      CountBookmarks(bm.children[i], nLevel + 1);
}
var counter = 0;
CountBookmarks(this.bookmarkRoot, 0);
console.show();
console.println("\nFile: " + this.path);
console.println("The number of bookmarks: " + counter);
```

Gather bookmarks to a document

Batch sequence name: `Gather Bookmarks to Doc.sequ`.

Task: Given a collection of selected files *not open in Acrobat*, and a document that is open in Acrobat, create a series of bookmarks in the open document which link to the selected files.

- This sequence can be used by itself to insert a series of navigation bookmarks, or in conjunction with `Insert Bookmarked Pages.sequ`, as discussed in "Insert bookmarked pages" on page 18.
- This script strips away the path to the file, so the links will work if the PDF file containing bookmarks are in the same directory as the targeted files. This is suitable for use with `Insert Bookmarked Pages.sequ` (see "Insert bookmarked pages" on page 18).

Example: *Gather bookmarks to a document*

Note: To see the bookmarks, you may have to do a "Save As" after running this sequence.

```
/* Gather Bookmarks to a Doc */
/*
** Assumes that one document is open in the viewer, and inserts
** bookmarks to the selected files into this document.
*/
try
{
   // Get active docs in viewer.
   var d = app.activeDocs;
   if (d.length != 1) // If not exactly one doc, exit batch
      throw "One and only one file must be open in the Viewer";
   // If there is a doc open, assume it is the one the user wants
   else {
      var tDoc = d[0];
```

```
      var bmRoot = tDoc.bookmarkRoot;
   }

   // Get path of the file Acrobat has taken from the selected folder
   var path2file = this.path;

   // A regular expression to acquire the base name of the file
   // (modify this to reference files in different directories)
   var re = /.*\/|\.pdf$/ig;

   // filename is the base name of the file Acrobat is working on
   // (modify this to reference files in different directories)
   var filename = path2file.replace(re,"");

   // If the title available, use for a bookmark name, else use filename
   if ((this.info.Title == null) ||
        (this.info.Title.replace(/\s/g,"")== ""))
      var bmToFile = filename;
   else
      var bmToFile = this.info.Title;

var nIndex = (bmRoot.children == null) ? 0 : bmRoot.children.length;

   // Make a bookmark with the action
   bmRoot.createChild(bmToFile,
      'if (this.external)\r\t'
      + 'this.getURL("' + filename + '.pdf");\r'
      + 'else {\r\t'
      + 'var that = app.openDoc("' + filename + '.pdf", this);\r\t'
      + 'if ( this != that ) this.closeDoc();\r'
      + '}', nIndex);
}
catch (e)
{
   console.println("Aborting Batch: " + e)
   event.rc = false;
}
```

Copy bookmarks from an array

Batch sequence name: `Copy Bookmarks from Array.sequ`.

Task: Take the collection of bookmarks gathered by the sequence `Gather Bookmarks to Array.sequ` (see "Gather bookmarks to an array" on page 12) and insert them into each of the selected files.

Example: *Copy bookmarks from array*

```
/* Copy Bookmarks from Array  */
/*
** Run Gather Bookmarks to Array first.
** Uses: global.bmArray, global.bmFileCnt from the sequence
*/
// Begin job. Executed when first file is processed.
if ( typeof global.counter == "undefined")  {
   console.println("Begin Job...");
```

```
   global.counter = 0;
   // If Count PDF Files has not been run, then...
   if ( typeof global.FileCnt == "undefined" )
   // Then use the file count from Gather Bookmarks From Array
     global.FileCnt = global.bmFileCnt;
/* Note: A routine to sort the global.bmArray can be inserted here */
}

// This script is executed for each of the selected files
try
{
   global.counter++
   console.println("Processing file \#"+global.counter);
    // Now, insert a bookmark at the end of the bookmark tree
   for (var i = 0; i < global.bmArray.length; i++) {
     var nIndex = (this.bookmarkRoot.children == null ) ?
        0 : this.bookmarkRoot.children.length;
     this.bookmarkRoot.createChild(global.bmArray[i][0],
        'if (this.external) \r\t'
        +'this.getURL("' + global.bmArray[i][1] + '.pdf");\r'
        +'else {\r\t'
        +'var that =
          app.openDoc("'+ global.bmArray[i][1]+'.pdf",
               this);\r\t'
        +'if ( this != that ) this.closeDoc();\r'
        +'}',
     nIndex);
   }
}
catch (e)
{
   console.println("Aborting Batch: " + e);
   console.println("Current File: " + this.path);
   event.rc = false;
   delete global.counter;
}
// End of Job. Executed when last file is processed.
if (global.counter == global.FileCnt) {
   console.println("End of Job...");
   delete global.indexCnt;
   delete global.bmFileCnt;
   delete global.bmArray;
   delete global.counter;
   delete global.FileCnt;
}
```

Insert bookmarked pages

Batch sequence name: `Insert Bookmarked Pages.sequ`.

Task: Merge a number of one-page PDF files (perhaps image files) into one file with bookmarks to each page. This example places a cover sheet and a PDF file with one or more pages in the viewer. It takes a selection of files, assumed to be one page. (They can be multiple pages, but only the first page of each file is inserted.)

Example: *Insert pages with bookmarks*

```
/* Insert Pages with Bookmarks */
/*
** Insert PDF page one from the selected files into the current
** document open in Acrobat. This open document is assumed to be the
** only document in the viewer.
*/
try
{
   // Get active docs in viewer.
   var d = app.activeDocs;
   // The document into which we are going to insert pages is
   // assumed to be d[0], since only one file is open in the viewer.
   if (d.length != 1) throw "Need exacly one file open in Viewer";
   var tDoc = d[0];

   // Get the number of pages in the document
   var lastpage = tDoc.numPages;
   // Get path of the file Acrobat has taken from the selected files
   var path2file = this.path;

   // A regular expression to remove the path to the folder of this file
   var re = /.*\/|\.pdf$/ig;

   // filename is the name of the file Acrobat is working on
   var filename = path2file.replace(re,"");

   console.println("Processing: " + filename);

   // If the title is available, use it for a bookmark, else use filename
   if ((this.info.Title == null)||
        (this.info.Title.replace(/\s/g,"") == ""))
     var bmToFile = filename;
   else
     var bmToFile = this.info.Title;

   // Now, insert pages into the document open in the viewer.
   // If you want the whole document inserted, then you would insert
   // nEnd: this.numPages-1 in argument list of insertPages method
   tDoc.insertPages
   ({
     nPage: lastpage - 1,
     cPath: path2file
   });
```

```
   // Insert a bookmark
   var nIndex = (tDoc.bookmarkRoot.children == null) ?
          0 : tDoc.bookmarkRoot.children.length;
   tDoc.bookmarkRoot.createChild(bmToFile,"this.pageNum=" +
               lastpage,nIndex);
}
catch (e)
{
   console.println("Aborting Batch: " + e );
   console.println("Current File: " + this.path );
   event.rc = false;
}
```

Working with icons

This section presents several batch sequences that deal with icons. The first one, Import Named Icons, imports named icons into a document, the second one, Insert Navigational Icons is a sequence for using button fields and icon appearances to create navigational icons for a document.

Import named icons

Batch sequence name: `Import Named Icons.sequ`.

Task: Import a series of icons into the document currently open in Acrobat.

This solution assumes that there is only one document open in Acrobat; this is the document into which you want to import some named icons. The icons are the selected files. Each of the selected PDF files contains only one icon on the first page (page 0). It is also assumed that each icon file has a document title; this title is used to assign the icon a name.

Example: *Import named icons*

```
/*Import Named Icons*/
/*
** Requires a file open in Acrobat into which the icons are inserted.
*/

try
{
   var d = app.activeDocs;
   if ( d.length != 1)
     throw "Batch requires a file to be open in Acrobat to insert Icons."
   var myDoc = d[0]; // target doc is only one open in Viewer

   // Get path of the file Acrobat has taken from the selected folder
   var path2file = this.path;
   console.println("\nPath: " + path2file);

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

   // filename is the base name of the file Acrobat is working on
   var filename = path2file.replace(re,"");
```

```
    // If the title is available, use for the icon name, else use filename
    if ((this.info.Title==null) ||
       (this.info.Title.replace(/\s/g,"")== ""))
       var nameIcon = filename;
     else
       var nameIcon = this.info.Title;

    console.println("Name used: " + nameIcon );

    // Insert icons!
    myDoc.importIcon(nameIcon, this.path);
  } catch (e) {
    console.println("Aborted Batch: " + e);
    event.rc = false;  // abort batch
  }
```

Once the icons have been introduced into the document by name, it is easy to associate any of these icons with any particular button. For example,

```
var f, i;
f = this.getField("iconButton1");
i = this.getIcon("anIconName1");
f.buttonSetIcon(i);

f = this.getField("iconButton2");
 = this.getIcon("anIconName2");
f.buttonSetIcon(i);
... ...
... ...
```

These lines can be a part of another batch sequence which runs after the icons have been imported. Repetitive code, such as the one shown, can then associate each button requiring one or more icons with one or more named icons.

You can remove the icons with the Doc object `removeIcon` method, then import new icons into the same document; this makes it fairly easy to maintain updates.

Insert navigation icons

Batch sequence name: `Insert Navigation Icons.sequ`.

Task: Insert a set of navigational icons on each page of each file in the selected files. This script is a representative example of the type of code you need to write. You will have to modify it as appropriate.

Example: *Insert navigation icons*

```
/* Insert Navigation Icons  */
// getPageBox, the default is "Crop"
var aPage = this.getPageBox();
var w = 36; // width of each icon
var nNavi = 4; // number of navigation icons to be placed
var g = 6; // gap between icons
var totalWidth = nNavi * w + (nNavi - 1) * g; // total width of navi bar
```

```
var widthPage = aPage[2] - aPage[0];
// Horizontal offset to left-most edge of navi bar needed to center it
var hoffset = (widthPage - totalWidth) / 2;
var voffset = 12;  // vertical offset from bottom

/*
   Import icons into the document. This uses the file named Arrows.pdf, but
   any set of icons will do.
*/
this.importIcon("PrevPage", "/F/Adobe/Batch/Arrows.pdf",48);
this.importIcon("PrevPage_p", "/F/Adobe/Batch/Arrows.pdf",52);
this.importIcon("NextPage", "/F/Adobe/Batch/Arrows.pdf",49);
this.importIcon("NextPage_p", "/F/Adobe/Batch/Arrows.pdf",53);
this.importIcon("PrevView", "/F/Adobe/Batch/Arrows.pdf",50);
this.importIcon("PrevView_p", "/F/Adobe/Batch/Arrows.pdf",54);
this.importIcon("NextView", "/F/Adobe/Batch/Arrows.pdf",51);
this.importIcon("NextView_p", "/F/Adobe/Batch/Arrows.pdf",55);

for ( var nPage = 0; nPage < this.numPages; nPage++) {
     // Create the fields
  var pp = this.addField("PrevPage", "button", nPage,
     [ hoffset, voffset, hoffset + w, voffset + w ] );
  var np = this.addField("NextPage", "button", nPage,
     [ hoffset + w + g, voffset, hoffset + 2*w + g, voffset + w ] );
  var pv = this.addField("PrevView", "button", nPage,
     [ hoffset + 2*w + 2*g, voffset, hoffset + 3*w + 2*g, voffset + w]);
  var nv = this.addField("NextView", "button", nPage,
     [ hoffset + 3*w + 3*g, voffset, hoffset + 4*w + 3*g, voffset + w]);

     // Place the fields objects in an array for convenience
     var NaviFields = new Array( pp, np, pv, nv );

     for ( var i = 0; i < NaviFields.length; i++ ) {
        var o = NaviFields[i];
     // Set the properties
        o.buttonPosition = position.iconOnly;
        o.highlight = highlight.p;
     // Set appearance faces
        o.buttonSetIcon(this.getIcon(o.name),0);
     o.buttonSetIcon(this.getIcon(o.name+"_p"),1);
     }

     // Set the actions
     pp.setAction("MouseUp", "this.pageNum--");
     np.setAction("MouseUp", "this.pageNum++");
     pv.setAction("MouseUp", "app.goBack()");
     nv.setAction("MouseUp", "app.goForward()");
}
```

Extract pages to a folder

Batch sequence name: `Extract Pages to Folder.sequ`.

Task: For each file from the selected files, extract each of its pages and save the extracted pages to a particular folder.

Example: *Insert navigation icons*

```
/* Extract Pages to Folder */
// A regular expression to acquire the base name of the file.
var re = /.*\/|\.pdf$/ig;
var filename = this.path.replace(re,"");

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

Before running this script, make the appropriate changes to `cPath` (the path, the file name, and the method of referencing the nth page of the target document).

Populate a form from a database and save it

Batch sequence name: `Populate and Save.sequ`.

The sequence uses an "intelligent" PDF form, and queries a database based on information given in certain fields in the form. For each line of data that satisfies the query, the batch sequence populates the form with that data and saves the form to a pre-selected folder. This batch sequence creates a populated PDF form for each line of data that is returned by the query.

Note: This batch sequence uses ADBC, a Windows-only feature of Acrobat.

The file `batchseq.zip` comes contains another zip file named `formsbatch.zip`. The file `formsbatch.zip` contains three files: `FormsBatch.pdf`, `ADBCdemo.mdb` and `ADBCdemo.pdf`. The first two of these are used in the demonstration of the `Populate and Save.sequ` batch sequence, while the third one is a demonstration of ADBC using the same database.

Preparing to use FormsBatch.pdf

The file `FormsBatch.pdf` is used to demonstrate the `Populate and Save.sequ` sequence. Begin by unzipping `formsbatch.zip` to a folder on your computer.

Register the Access file, `ADBCdemo.mdb`, with the ODBC Data Source Administrator, as per the instructions contained on the first page of the demo file `ADBCdemo.pdf`. (Once the database has been registered, you can experiment with ADBC using `ADBCdemo.pdf`.)

Filling in FormsBatch.pdf

Our approach is to have a fairly general batch sequence and an "intelligent" form. In addition to the fields pertinent to the use of the form, the form contains four additional fields used by the batch routine:

`DB` — A text field that contains the name of the database to be used for retrieving data; for example: `ADBCdemo`.

`SQL` — A text field that contains a SQL select command, a query, for choosing data from a database, as defined in the `DB` field; for example: `Select * from ClientData Where Income > 100000`.

`DIPath` — A text field that contains a device independent path to the folder in which the saved PDF files are to be deposited; for example: `/F/database/`. The path can contain a suffix; for example, `/F/database/ID`. The `DIPath` is concatenated to the `NameRule` to obtain the complete path name of the file to be saved.

`NameRule` — A unique file name to give the form about to be saved. Each row of data retrieved from the database should contain a field that uniquely characterizes the record, such as a client ID. The ID can be used to build the name of the output file; for example, the `ADBCdemo` database has an ID field for each data row, so we can put the value of the text field "NameRule" to `row.ID.value`. When the batch sequence is run, each file will have a path of `/F/database/ID1.pdf`, `/F/database/ID2.pdf`, and so on.

The form also contains four document JavaScript functions called by the batch sequence listed below. This document JavaScript can be copied into your own form, and modified as needed.

`getConnected()` — Reads the values of `DB` and `SQL`, connects to the database and executes the requested SQL.

`getNextRow()` — Gets the next row of data; it returns either a row object, or `null`.

`populateForm(row)` — Populates the form with the row of data passed to it.

`saveToFolder()` — Saves the populated form to the path obtained by concatenating the value of `DIPath` and the evaluated value of `NameRule`.

Running Populate and Save

- If you haven't done so already, move the batch sequence file `Populate and Save.sequ` to the user's Sequences folder. To find the location of this folder, execute the following JavaScript method from the console:

  ```
  app.getPath("user","sequences");
  ```

  Remember to restart Acrobat to read this new batch sequence.
- After reading about "Filling in FormsBatch.pdf" on page 23, open the file `FormsBatch.pdf` in the Acrobat.
- Modify the text field named `DIPath` to refer to a folder on your own hard drive.
- Save `FormsBatch.pdf`, then run the sequence Populate and Save. If there are no errors, for each row of data from `ADBCdemo.mdb` that satisfies the query criterion, a copy of the first page of `FormsBatch.pdf` with the form fields populated by that row of data is saved to the folder you specified. This is like creating a personalized invoice for each customer.

Example: *Populate and save a form*

```
/* Populate and Save */
try
{
   // This counter is not really needed, but can be used to count the
   // number of files processed, or use it while in a testing stage.
   var counter=0;
   if ( !getConnected() ) throw "Could not connect!";
   do
   {
//   if (counter > 2) throw "Bailing out!" // Debug line
     var row = getNextRow();
     if ( row == null )
        throw "No more rows. Total processed: " + counter;
     counter++
     populateForm(row);
     saveToFolder();
   } while (true)
}
catch(e)
{
     console.println("Batch aborted: " + e);
     event.rc = false;   // Abort batch
}

console.println("End of Job Reached!");
this.closeDoc(false);   // Close the last form that was populated.
```

Working with security

Security related tasks are especially important in an enterprise settings, this section covers sequences for removing security, reporting on the signatures, and signature signing.

Change security to no security

Batch sequence name: `Change Security to None.sequ`.

Task: Given the selected password-protected files *all with the same password*, remove the password protection.

This task can be performed without the aid of JavaScript by using the **Document** > **Security** command sequence found in the Edit Sequence dialog box. If you edit this sequence, you see that it is set to "None" ("No Security").

To make `Change Security to None.sequ` run correctly, you must first choose **Edit** > **Preferences** > **Batch Processing** and change the Security Method to Password Security. When you run the sequence, Acrobat asks for a password. When that is entered, the batch continues.

After you are finished with this batch sequence, be sure to change Security Method of the Batch Sequences user preferences to "Do not ask for password".

Gather digital signature information

Batch sequence name: `Gather DigSig Info.sequ`.

Task: Write a report of the state of all digital signatures in the selected files.

This sequence is too long to be listed in this document. This is an example of cross-document processing and requires that the `global.FileCnt` be set by running "Count PDF files" on page 12. Before running this sequence, execute `delete global.counter` to be sure that `global.counter` is undefined.

The output is a report in which all signatures are grouped by their status: blank, invalid, unknown status and valid. A complete listing of digital signature information, using Field object `signatureInfo` method, is given. If a signature is blank, invalid or has an unknown status, it is highlighted in red.

Sign all documents

Batch sequence name: `Signature Sign All.sequ`.

Task: Create an invisible signature on each of the selected files.

Example: *Sign all documents*

```
/* Signature Sign All  */

// Choose handler
var ppklite = security.getHandler("Adobe.PPKLite");

// Login -- change as appropriate
ppklite.login("dps017", "/C/Profiles/DPSmith.pfx");

// Add a signature field with zero dimensions (invisible)
var f = this.addField("Signature", "signature", 0, [0,0,0,0]);

// Sign it and log out. Change as appropriate
f.signatureSign(ppklite,
   { password: "dps017",
     location: "San Jose, CA",
     reason: "I am approving this document",
     contactInfo: "dpsmith@mycompany.com",
     appearance: "DPSmith"});
ppklite.logout();
```

In addition to signing each document, you may want to set the security of each document. Security must be set before signing. Edit the Signature Sign All batch (or a renamed copy of it) and do the following procedure.

➤ **To set security:**

1. Choose **Advanced** > **Document Processing** > **Batch Processing** > **Edit Sequences...**.
2. From the list, choose **Signature Sign All** (or the sequence as you have renamed it).
3. Click **Edit Sequence...**, **Select Commands...**.
4. Select **Security** from the left panel, then click the **Add >>** button to move **Security** to the right panel.

5. Move **Security** above the **Execute JavaScript** statement corresponding to **Signature Sign All**.
6. Edit the Security batch sequence.
7. Select **Password Security** then click **Change Settings**.
8. Choose the 128-bit RC4 encryption level and select **Restrict Editing and Printing of the Document**.
9. Under Changes Allowed, select, for example, **Filling in Form Fields and Signing Existing Signature Fields**. You can optionally, add a password.
10. Close all the dialog boxes.

Inserting comments and form fields

You can use a batch method to insert a comment, such as a stamp, or a form field into each of the selected files at a pre-designated location.

Insert a stamp

Batch sequence name: `Insert Stamp.sequ`

Task: Insert a stamp on the first page of each of the selected files.

This script places the stamp in the geometric center of the page. The width of the stamp is set to 67% of the width of the cropped page; the height set at 25% of that.

An annotation, such as a stamp, calculates all the coordinates in default user space. The form plug-in calculates the coordinates in rotated user space. Some undocumented JavaScript methods are used to convert between these two spaces.

Example: *Insert Stamp*

```
/* Insert Stamp */

// Get the crop box of the first page
var aCrop = this.getPageBox()

var semiWidth  = aCrop[2]/2;
var semiHeight = aCrop[1]/2;

// Get the center of the crop box.
var x =  semiWidth;
var y =  semiHeight;

// Make the width of the stamp roughly .67% of the cropped page width, and
// make the height of the stamp 25% of the stamp's width
semiWidth = 0.67*semiWidth;    // Make width .67 of width of page
semiHeight = 0.25*semiWidth;   // Make height .25  of semiWidth

var Rect = new Array (x-semiWidth,y-semiHeight,x+semiWidth,y+semiHeight);

// The matrix m transforms from rotated to default user space
var m = (new Matrix2D).fromRotated(this,0)
```

```
// Transform Rect from rotated to default user space
Rect = m.transform(Rect)

// Now add our annotation using the Rect array for the rect property
var annot = this.addAnnot
({
   page: 0,
   type: "Stamp",
   name: "myStamp",
   rect: Rect,
   contents: "This document has been approved.",
   AP: "Approved"
});
```

Insert a barcode

Batch sequence name: `Insert Barcode.sequ`

Task: Insert a barcode on the first page of each of the selected files.

Because we operate in rotated user space, the problem of inserting a form at a specific location in a document is much easier than inserting an annotation (see the discussion in "Insert a stamp" on page 26).

The following example is incomplete. Details specific to your needs must be added.

Example: *Insert a barcode*

This script inserts a text field on the first page of each the selected documents. The text font is set to a barcode font. A barcode font is required.

```
/* Insert Barcode */

// getPageBox, the default is "Crop"
var aPage = this.getPageBox();

// Put a barcode in the upper right-hand corner of page 0
// The text field is 144 pts wide and 20 high
aPage[3] = aPage[1] - 20;
aPage[0] = aPage[2] - 144;

var f = this.addField("myText", "text", 0, aPage);
f.delay = true;
f.readonly = true;
f.alignment = "right";
f.textFont = "Code39Two";      // a barcode font is required here
f.fillColor = color.transparent;
// A fixed code is inserted. In a real application, this would be obtained,
// perhaps, from a database, perhaps through the ADBC plug-in.
f.value = "ID-123-4567";
f.delay = false;
```

Spell check a document

Batch sequence name: `Spell Check a Document.sequ`

Task: Go through a PDF file (or selected PDF files) and mark each misspelled, or questionable word with a wavy underline annotation. The contents of the annotation contain suggested alternative spellings.

Example: *Spell check a document*

```
/* Spell Check a Document */
var ckWord,numWords, i, j;
for (var i = 0; i < this.numPages; i++ )
{
   numWords = this.getPageNumWords(i);
   for ( j = 0; j < numWords; j++)
   {
      ckWord = spell.checkWord(this.getPageNthWord(i,j))
      if ( ckWord != null )
      {
         annot = this.addAnnot
         ({
          page: i,
          type: "Squiggly",
          quads:  this.getPageNthWordQuads(i,j),
          author: "A. C. Acrobat",
          contents: ckWord.toString()
         });
      }
   }
}
```

You can use `Doc.spellDictionaryOrder` and `spell.dictionaryOrder` to determine the dictionaries used, and the order in which they are checked.

Summarize comments from selected files

Batch sequence name: `Cross Doc Comment Summary.sequ`.

Task: Gather all comments from the selected files, sorted by author, and create a combined report.

This script has both Begin Job and End Job sections and illustrates how to write and manage a more complex batch sequence. It uses the global variable `global.FileCnt`. You can either run "Count PDF files" on page 12, which sets this value, or you can enter the value through the console before you run the sequence.

Type `delete global.counter` in the console before you start this sequence to be sure `global.counter` is undefined. The sequence executes `delete global.counter` if the batch aborts or finishes successfully.

Example: *Summarize comments from selected documents*

```
/* Cross Doc Comment Summary */
/*
   Requires the Count PDF Files batch to be run first to give
```

```
   global.FileCnt a value (or enter through console)
*/
// The begin job routine. Executed while first file is processed.
if ( typeof global.counter == "undefined" ) {
   console.println("Begin Job");
   console.clear();
   global.counter = 0;
   // insert beginJob code here
   global.rep = new Report();
   global.rep.size = 1.2;
   global.rep.style = "NoteTitle";
}

// This script is executed for each of the selected files
try{
   //  insert regular batch code here.
   global.counter++
   console.println("global.counter =  " + global.counter)
   global.absindent = 0;
   this.syncAnnotScan();
   global.annots=this.getAnnots({nSortBy: ANSB_Author});
   global.rep.color = color.blue;
   if (global.counter != 1) global.rep.writeText(" ");
   global.rep.writeText("Annotation Summary: By Author");
   global.rep.writeText("Comments from file: "+this.path);
   global.rep.color = color.black;
   global.rep.writeText(" ");
   console.println("annots.length = "+global.annots.length);
   global.rep.writeText("Number of Annots: "+global.annots.length);
   global.rep.writeText(" ");
   var msg = "\u00B7 (%s) page %s: \"%s\""; // \u00B7 unicode bullet
   var theAuthor=global.annots[0].author
   global.rep.writeText(theAuthor);
   global.rep.indent(20);
   for (var i=0; i < global.annots.length; i++)
   {
   if (theAuthor != global.annots[i].author) {
      theAuthor = global.annots[i].author;
      global.rep.writeText(" ");
      global.rep.outdent(20);
      global.rep.writeText(theAuthor);
      global.rep.indent(20);
      }
      global.rep.writeText(util.printf(msg, global.annots[i].type,
1+global.annots[i].page, global.annots[i].contents));
   }
   global.rep.outdent(20);
} catch(e) {
      console.println("Aborted on run number " + global.counter);
      delete  global.counter;
      event.rc = false;   // abort batch
}
// The end job routine. Executed while last file is being processed.
if  (global.counter ==  global.FileCnt) {
   console.println("End Job");
```

```
    // insert endJob code here
    console.println("Before open report");
    /*
    ** We now want to open our report (Report.open).If the Acrobat
    ** version is earlier than 7, we cannot open or save (Report.save)
    ** a report while a modal dialog box is open, as it is while
    ** the batch is running.
    */
    if ( app.viewerVersion < 7 ) {
       global.wrtRep = app.setInterval(
          'try {'
       +'       docRep = global.rep.open("myreport.pdf");'
       +'       console.println("Executed Report.open");'
       +'       app.clearInterval(global.wrtRep);'
       +'       delete global.wrtRep;'
       +'       console.println("Executed App.clearInterval");'
       +'       docRep.info.Title = "Monthly Report: August 2004";'
       +'       docRep.info.Subject = "Summary of the August meeting";'
       +'       docRep.info.Author = "A. C. Robat";'
       +'} catch (e) {console.println("Waiting...: " + e);}'
       +'finally { delete global.counter;'
       +'       console.println("Executed delete global.counter"); }'
       , 100);
  } else {
       docRep = global.rep.open("myreport.pdf");
       console.println("Executed Report.open");
       docRep.info.Title = "Monthly Report: August 2004";
       docRep.info.Subject = "Summary of the August meeting";
       docRep.info.Author = "A. C. Robat";
       delete global.counter;
       console.println("Executed delete global.counter");
    }
  }
```

The information written to the report should be customized. Use this sequence as the starting point for designing your own report summary of comments.

In the Batch Sequence commands there is an item called Summarize Comments that summarizes comments of each of the selected files; however, the comments are not combined. The comments from the file `myDoc.pdf`, for example, are saved in a separate file named `myDoc_sum.pdf`. You can merge the summary documents, if you want.

Write comments to a tab-delimited file

Batch sequence name: `Comments to tab delimited file.sequ`.

Acrobat 7.0 introduced some powerful JavaScript methods for working with attachments. This batch sequence is an example of the types of problems you can attack with these file attachment techniques.

In this batch sequence, all comments from the selected files are placed in a tab-delimited file, rather than in a report, as does `Cross Doc Comment Summary.sequ`, discussed on "Summarize comments from selected documents" on page 28. Recall that this sequence needs to know when all files have been processed before it can write its report; `Comments to tab delimited file.sequ` does not have this limitation.

When the sequence is started, there should be no files open in Acrobat. The sequence creates a blank PDF document to hold the gathered data as a file attachment.

Once the sequence has finished, you can open the attachment to see a summary of all comments from the selected files. The file will open in the default spreadsheet application.

The same approach can be used to extract data that has been entered into form documents of the same type. You can create a batch similar to this one, to extract the form data to a tab-delimited file for later analysis.

Note: Be sure the global variable `global.startBatch` is undefined at the beginning of this sequence. If necessary, execute `delete global.startBatch` in the console.

Example: *Write comments to a tab delimited file*

```
/* Copy comments to a tab-delimited file */
/* Acrobat 7.0 or later required
** Prior to running this script for the first time, be sure the global
** variable global.startBatch is undefined. If necessary, execute
** delete global.startBatch from the console.
*/
var re = /\"/g; // find all double quotes ("), used with annots.contents
if ( typeof global.startBatch == "undefined" ) {
   global.startBatch = true;
   // When we begin, we create a blank doc in the viewer to hold the
   // attachment.
   global.myContainer = app.newDoc();
   // Create an attachment and some fields separated by tabs
   global.myContainer.createDataObject({
      cName: "mySummary.xls",
      cValue: "File Name\tName\tType\tPage"
         +"\tModification Date\tComment"
   });
   var dataLine = "";
}
try {
   // Wait until all comments have been scanned.
   this.syncAnnotScan();
   // Add the document name to the this line.
   dataLine += "\r\n"+this.documentFileName;
   // Sort through the comments, adding tab-delimited lines to data
   // string.
```

```
   var annots=this.getAnnots({nSortBy: ANSB_Author});
   if ( annots != null ) {
      for (var i=0; i < annots.length; i++)
      {
         var cContent = annots[i].contents.replace(re, "\"\"");
         dataLine += "\r\n\t"+ annots[i].author
            + "\t" + annots[i].type
            + "\t" + annots[i].page
            + "\t" + util.printd(2, annots[i].modDate)
            + "\t\""+ cContent+"\"";
      }
      // Get the data object contents as a file stream
      var oFile =
         global.myContainer.getDataObjectContents("mySummary.xls");
      // Convert the stream to a string
      cFile = util.stringFromStream(oFile, "utf-8");
      // Concatenate the new lines.
      cFile += dataLine;
      dataLine = "";
      // Convert back to a file stream
      oFile = util.streamFromString( cFile, "utf-8" );
      // and update the file attachment
      global.myContainer.setDataObjectContents({
         cName: "mySummary.xls", oStream: oFile
      });
   } else console.println("The document " + this.documentFileName + "
contains no annots.");
} catch(e) {
   console.println("Error on line " + e.lineNumber + ": " + e);
   delete typeof global.startBatch
   event.rc = false;   // abort batch
}
```