Datalogics

Sample Program Guide

Adobe PDF Java Toolkit 2.1

Document Updated 6/30/2014

For additional information, contact:

Datalogics, Incorporated
101 North Wacker Drive, Suite 1800
Chicago, IL 60606-7301
Phone: (312) 853-8200
Fax: (312) 853-8282
www.datalogics.com

Table of Contents

The PDF Java Toolkit Sample Program Interfaces

Adobe PDF Java Toolkit (PDFJT) is the core library used by Adobe LiveCycle, a product that allows users to build applications to automate business workflows. Popular with service organizations and government agencies, LiveCycle is an electronic form and document platform that is used to capture, process, and store information. For example, LiveCycle can be used to build tools to automate manual processes like setting up customer accounts, enrolling clients in benefit programs, and managing correspondence.

PDFJT is a Java Application Programming Interface (API) that provides much of the Adobe LiveCycle services for working with PDF files, forms, and the Adobe Reader. Developers can use PDFJT to build their own applications to offer a wide variety of services, such as working with electronic forms and managing digital signatures on PDF documents. To aid your development efforts, we provide a set of sample Java programs that show how to complete standard tasks using PDF Java Toolkit. You can use these sample programs to guide you in your own designs or copy the code from a program directly into one of your own applications.

The Sample Programs

When you install the PDF Java Toolkit files it will create the following directories for you:

```
/docs
/libs
/samples/input
/samples/src
```

PDFJT provides a set of sample Java program files, stored in the samples/src directory. The source code is provided as a single executable file, pdfjt.jar, stored in the libs directory. The dependencies needed by the sample programs are provided in a single executable as well, pdfjt-support.jar, also found in the libs directory.

Each of the sample programs completes a simple programming task. You can use these programs as models for tasks you would like to complete in your own applications. You can also copy and paste code from these samples into your own programs.

You can open and review the sample Java program files in the viewing tool of your choice, such as Eclipse or Microsoft Visual Studio. You can also run the programs from these tools, or run the executable Java program file for any sample program from a command line. Many of these sample programs create output files (commonly PDF or graphics files) and store them in the output directory under Samples. The software also installs with a set of input files, mostly PDF files.

Note that if you run a sample program a second or third time, it will overwrite any output files that were created and stored earlier. However, if you run a sample program, generate a PDF output file, and then open that PDF file and try to run that sample program again, you will see an error message. The program will not be able to overwrite an existing output file if that file is currently open in Adobe Reader or Adobe Acrobat.

We provide brief descriptions of each sample program in this document, including:

1. A discussion of the purpose of each program and how it works
2. A list of the most relevant Import Statements (Uses) for each sample program, so you know what part of the API Documentation to refer to
3. A list of the other sample programs in PDFJT that are most closely related to the sample program and that are most likely to be used with this program

We assume a basic level of technical understanding of the PDF file format, though we seek to review the features of each of these sample programs carefully.

Providing Reader Extensions for PDF Documents

The Reader Enablement feature in Adobe Acrobat allows a user to set up a PDF document so that another user can open it in Adobe Reader and have rights, in Reader, to edit that PDF document. This process applies the Reader Extension to the PDF document. With a PDF document enabled for use in Adobe Reader, a user can add comments to a PDF document, sign the document, and save the document with form fields completed.

To enable a PDF file, you must have a PFX file, which you can receive from Datalogics. The PFX, or Personal Information Exchange Format file, is an encrypted Windows file that stores a certificate used to authenticate a person or device, and that requires a password to open. You can use a PFX to transfer a certificate from one computer to another. PFX is based on the Public Key Cryptography Standards, or PKCS #12, which defines an archival file format that can be used to store a variety of kinds of cryptographic objects. For example, a PFX file could hold all of the elements of a chain of trust, or a set of individual certificates.

If you have access to a PFX file, you can use it to enable a set of PDF documents if you also use a simple product provided by Datalogics called RELite. RELite is an executable Java jar file that you can run at a command prompt with Java version 1.6 or later.

RELite is a fast and easy way to provide Reader Extensions to your PDF documents. For more information about RELite, and to download a copy of the RELite zip file, contact your Datalogics support representative.

Running RE Lite

To run RELite, at a command prompt, type:

```
java -jar relite.jar
```

The program will prompt you to enter the path name of the PDF file:

```
Enter path(s) to any input files to be reader enabled :
```

Enter the name of a single PDF document, if you have it stored in the same directory as the RELite.jar program file:

```
Sample.pdf
```

Or enter a path name with the PDF document, if you stored it elsewhere:

```
C:/PDFProject/input/sample.pdf
```

If you stored the PDF documents in a subdirectory under the directory where you stored RELite.jar, you don’t need to type in the full path name.

For example, if you stored the RELite.jar program file under C:/PDFProject and a set of PDF documents to enable under C:/PDFProject/Input, you could enter the path name for your PDF document like this:

```
input/sample.pdf
```

You can also enter the names of more than one input PDF file, and the program will process all of the files, one by one. Include the path name and file name for each file, each separated by a space, like this:

```
input/sample.pdf input/sample2.pdf input/sample3.pdf
```

RELite will ask you where you want to put the new output PDF file it will create.

```
Enter output file path(s) (press enter to use RELites default output
name):
```

You can enter your own path and file name, or simply press ENTER. By default, RELite will store your new PDF output file in a directory called "output" under the directory where you stored the RELite program, and add "-enabled" to the file name.

RELite will ask you to enter the path to your PFX file.

```
Enter path to pfx file:
```

Note that we provide a sample PFX file with the RELite download package, eval.pfx. But you would probably want to enter the path and file name of your own PFX file, or move that PFX file to the same directory where you have your RELite jar file.

RELite will ask for the password for your PFX. The password for the eval.pfx file provided with the RELite software is stored in the Readme.txt file that we provide.

Finally, RELite will ask you to define the rights to assign to the PDF file to be enabled.

```
Enter rights to enabled with this PDF ('All' for full reader
enablement):
```

Assigning Permissions to a PDF Document

RE Lite provides a set of 12 permissions to assign to a PDF document when it is opened in Adobe Reader.

You can simply type "All" and press Enter to assign every right to the PDF document. Or you can assign a limited set of permissions to the PDF document individually.

For example, if you assign the permission to *annotmodify* to a PDF document, that means that a person could open that PDF document in Adobe Reader and then create or change annotations in that document. That is, the user could add comments to the PDF or change comments added by another user who worked with the PDF document in Adobe Acrobat. If you assign *formadddelete* to a PDF document, a user can add form fields to that PDF document using Reader.

To assign a specific set of permissions to a PDF document, type a string of command names, each separated by a blank space, like this:

```
formfillinandsave formimportexport formadddelete spawntemplate
barcodeplaintext
```

Make sure that you enter each permission command name in all lower case letters. We provide a list of the available permissions below.

formfillinandsave	Fill in form fields on a PDF document and save the PDF document on a local machine, with the form values preserved.
formimportexport	Import and export form data from a PDF document to and from FDF, XFDF, XML and XDP files.
formadddelete	Add, change or delete fields and field properties on the PDF form.
submitstandalone	Submit data from a PDF form to a server when the PDF document is not open in a browser session, by email or offline.
spawntemplate	Create pages from template pages within the same PDF form.
signing	Digitally sign and save PDF documents, and clear digital signatures.
annotmodify	Create and change annotations in a PDF document, such as comments.
annotimportexport	Save annotations, such as comments, from a PDF document to a separate data file, and import comments from a separate file into a PDF document.
barcodeplaintext	When the user prints a PDF form document, the content entered in the form fields is captured in a 2D bar code that is then printed with the document. This permission prints a document with the form data added to the barcode in an unencrypted form that does not require licensed server software to decode.
annotonline	Upload and download annotations such as comments to and from an online document review and comment server. Note that this command is no longer used for Reader 7 or later.
formonline	Connect to web services or databases that are defined within a PDF form.
efmodify	Update embedded file objects associated with a PDF document.

When the process is complete, the program will confirm that it completed successfully:

```
Reader enabled PDF file successfully generated at sample-enabled.pdf
```

Note that you can also format an RE Lite command to run without prompts:

```
java -jar relite.jar -in input/doc.pdf input/doc2.pdf input/doc3.pdf -
out output/doc1-enabled.pdf output/doc2-enabled.pdf output/doc3-
enabled.pdf -pfx eval.pfx -pfxpass ka7GfzI9Oq -rights formimportexport
formadddelete spawntemplate
```

You will need to provide these tags:

`-in`	Input file path and name
`-out`	Output file path and name
`-pfx`	PFX file name
`-pfxpass`	Password for PFX file
`-rights`	List of permissions

With this format you could add a command to run RE Lite to a batch file, and run it automatically after business hours.

Annotations

Both Adobe Reader and Adobe Acrobat allow users to add text to a PDF file. Within the free Adobe Reader you can add notes and highlight text. With Adobe Acrobat you can make a variety of changes, including the following:

- add notes and highlights
- add, cross out, or delete text
- attach a file
- add a callout box to add text
- add images or stamps

When you save the PDF file these changes, called annotations, are added to the file.

QueryAnnotations.java

This sample program opens and scans a PDF form document to identify the annotations included in that document. The program lists at the command prompt or in an Eclipse console the type of each annotation or field. For hyperlinks, the program lists the web address if available. For a widget, the program provides the OnCursorEnter action, such as moving to a different page in the PDF document. In this sample, the program opens a PDF form file with nine text fields and a button. The response that describes one of the text fields looks like this:

```
PDF Field type: com.adobe.pdfjt.pdf.interactive.forms.PDFFieldText
```

Relevant APIs:

- com.adobe.pdfjt.pdf.interactive.forms
- com.adobe.pdfjt.pdf.interactive.annotation

See also these other samples:

- core.forms.QueryForms
- core.attachments.FileAttachmentAnnotations

RichMediaAnnotDemo.java

This sample program shows how to how to create a Rich Media annotation from a SWF source file. The SWF file format refers to Small Web Format. This file extension (.swf) is used for Adobe Flash files.

Relevant APIs:

- com.adobe.pdfjt.pdf.interactive.annotation.PDFAnnotationRichMedia
- com.adobe.pdfjt.pdf.multimedia

See also these other samples:

- core.annotations.RichMediaAnnotExtract

RichMediaAnnotExtract.java

This sample program shows how to how to extract the embedded file from a Rich Media annotation contained within a source PDF document.

Relevant APIs:

- com.adobe.pdfjt.pdf.interactive.annotation.PDFAnnotationIterator
- com.adobe.pdfjt.pdf.interactive.annotation.PDFAnnotationRichMedia

See also these other samples:

- core.annotations.QueryAnnotations
- core.annotations.RichMediaAnnotDemo

Attachments

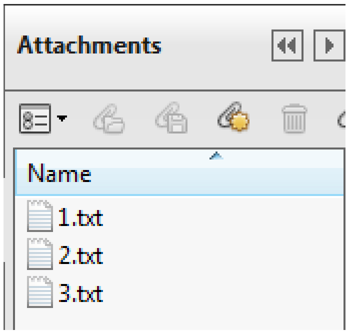
EmptyPDFWithFileAttachments.java

Like an email message, a PDF file can hold attached files. You can add any kind of file, such as an MS Word document, a JPG photograph image, a spreadsheet, or an MP3 audio file, to a PDF file and then save that attachment as part of the PDF. And you can add as many attached files to a PDF document as you like.

This sample program opens a blank PDF file and saves it as an output document with three file attachments added, all text (TXT) files.

After you run the program and open the output file in Adobe Acrobat, you can display the list of attached files and open them from within the PDF file.

Click the attachments icon 📎:



Attachments

Name

1.txt

2.txt

3.txt

Relevant APIs:

- com.adobe.pdfjt.pdf.document.PDFEmbeddedFile
- com.adobe.pdfjt.pdf.document.PDFNamedEmbeddedFiles

See also these other samples:

- core.collection.CreatePDFPackage

FileAttachmentAnnotations.java

Like an email message, a PDF file can hold attached files. You can add any kind of file, such as an MS Word document, a JPG photograph image, a spreadsheet, or an MP3 audio file, to a PDF file and then save that attachment as part of the PDF. Also, a PDF file can have an attached file represented as an annotation on a page. For example, an icon such as a thumb tack or paper click could appear on a PDF page, and when a user clicked on that icon, open an attached JPG or text file, the attached file related to the icon can be opened or deleted from the PDF document.

This sample program searches a PDF document for any attachments embedded in the file as an annotation, and exports it to a separate file. In this example, the program finds an embedded bitmap graphic (a Datalogics logo) and exports it to a separate BMP file.

The program also prints the result of the search at the command prompt or in the Eclipse console, describing the type of annotation and providing the name of the export file.

Relevant APIs:

- com.adobe.pdfjt.pdf.interactive.annotation.PDFAnnotationFileAttachment
- com.adobe.pdfjt.pdf.interactive.annotation.PDFAnnotationIterator

See also these other samples:

- core.annotations.QueryAnnotations

The PDF Collection Feature

PDF files can hold embedded or attached files, rather like email messages. You can also use a PDF collection or PDF package as a means to store a group of attached files. A PDF Collection is a set of files embedded in a PDF document, where this PDF document serves only as a container, or a cover sheet. Within the collection, each of the embedded PDF documents works as its own PDF file, with its own security settings, digital signatures, and so on. You can open a PDF collection just as you would a regular PDF document, but the collection does not have its own pages, and you will see a different layout.

In the years since Adobe Systems first introduced the PDF Collection feature, new upgrades to the Acrobat system have added a new user interface and other features to PDF Collection. With these later versions of the product Adobe Systems has referred to a PDF document with a collection dictionary as a PDF Package or a PDF Portfolio. With Acrobat 9.0 the term PDF Portfolio is used to describe any document that contains a collection dictionary.

For more information, see the Knowledge Base article on this topic.

CopyPDFCollection.java

This sample program identifies the attachment files included in a PDF Collection and copies each one to a separate PDF document. The original PDF Collection used as an input document includes four different embedded PDF files:

- Ducky.pdf
- Images.pdf
- Layers.pdf
- Test-collection.pdf

As a result the program will generate four different PDF output documents.

Relevant APIs:

- com.adobe.pdfjt.pdf.interactive.navigation.collection
- com.adobe.pdfjt.pdf.document.PDFNamedEmbeddedFiles

See also these other samples:

- core.collection.CreatePDFPackage

CreatePDFPackage.java

This sample creates a PDF Package that holds three files. Each of the attached files in the PDF package has two attributes, found in Properties: Author and Subject. The files are sorted by Subject and then by Author.

Relevant APIs:

- com.adobe.pdfjt.pdf.document.PDFNameDictionary
- com.adobe.pdfjt.pdf.interactive.navigation.collection

See also these other samples:

- core.attachments.EmptyPDFWithFileAttachments

SortPDFCollection.java

This sample lists the names of the files held within a PDF Collection, showing the files and path names in order, both sorted and unsorted. The file names appear at a command prompt or within the Eclipse console.

Relevant APIs:

- com.adobe.pdfjt.pdf.interactive.navigation.collection
- com.adobe.pdfjt.pdf.document.PDFNamedEmbeddedFiles

See also these other samples:

- core.collection.CreatePDFPackage

Working with Digital Signatures

PDFJT provides an array of sample programs that demonstrate processes related to creating and managing digital signatures in PDF documents.

Note that Adobe Acrobat and Adobe Reader may not validate signatures in the PDF output files created by the sample programs described below. This is because the evaluation certificates in these output PDF files are self-signed by Datalogics.

Many of the sample programs apply a certificate and key to the document provided by Datalogics, drawn from the key and certificate files pdfjt-eval-key.der and pdfjt-eval-cert.der, both stored in the input/sigsig directory.

These certificates rely on the Distinguished Encoding Rules (DER). DER is part of a set of rules called Abstract Syntax Notation (ASN.1) that define how data structures are encoded and decoded when they are transferred from one computer to another. The original rule set was called Basic Encoding Rules (BER), and DER was developed later as a subset of BER to satisfy the requirements for X.509 specification for secure data transfer using certificates and public keys.

AddLongTermValidationInfo.java

This program demonstrates how to find and evaluate the digital signatures in Acrobat Form (AcroForm) fields in a PDF file, and then apply Long Term Validation (LTV) criteria to each signature.

Signed documents are frequently kept in storage indefinitely, and retain their legal authority years or even decades after they were first executed. The Long Term Validation Profile provides the same service for electronically signed PDF files. With LTV, a user will be able to confirm a signature was valid at the time the electronic signature was applied to the PDF many years after the PDF file is created.

To have Long Term Validation for a signed PDF document, all of the required elements for the validated signature must be embedded in the signed PDF. The LTV criteria for a signature can include, for example, the certificate, the certificate's revocation status, and a time and date stamp.

Relevant APIs:

- com.adobe.pdfjt.pdf.digsig.PDFValidationRelatedInfo
- com.adobe.pdfjt.services.digsig

See also these other samples:

- core.digitalSignatures.pades.AddPAdESLTV

ApplyUsageRights.java

This sample program demonstrates how to add usage rights, also known as Reader Enablement or Reader Extensions, to a PDF document. The Reader Enabling feature in Adobe Acrobat allows a user to set up a PDF document so that another user can open it in Adobe Reader and still have rights to add comments, sign the document, and save the PDF document with form fields completed.

In order to use this sample a special set of credentials is required. You can run through the steps provided in the comments at the beginning of this program in order to download the PFX file, generate the .der files you will need, and update the sample Java code.

We recommend that you contact your support representative at Datalogics. Your representative can send you a zip file as an email attachment that includes the .der files you will need. The representative can also tell you where to store the files and how to work with the ApplyUsageRights.java program to make sure that the program can access and use them to run successfully.

To learn more see the section "Providing Reader Extensions for PDF Documents" on page 2.

Relevant APIs:

- com.adobe.pdfjt.services.digsig.SignatureOptionsUR
- com.adobe.pdfjt.pdf.digsig.PDFURDocument

See also these other samples:

- core.digitalSignatures.RemoveUsageRights

CalculateLegalAttestation.java

This sample shows how to create a list of the legal attestation dictionary for a PDF document.

A legal attestation is a declaration by a witness that a document has been executed formally in his or her presence. An example of an attestation would be a notarized signature on a contract; the attestation demonstrates that the document is legally valid. Within a PDF document, a legal attestation dictionary lists any elements within the PDF document that may result in an unexpected rendering of the document contents, and that might in turn affect the legal integrity of the document after it has been signed. For example, the legal attestation dictionary might list the number of annotations added to the document, the number of layers, the number of True Type fonts, and a text string written by the author of the document explaining any of the entries that appear in the dictionary itself.

When you run the program the items in the legal attestation dictionary and the number of each one appear at the command prompt or in the Eclipse console, like this:

```
/CatalogHasOpenAction 1
/DevDepGS_TR 5
/JavaScriptActions 1
/NonEmbeddedFonts 4
/OptionalContent 1
/TrueTypeFonts 1
```

Relevant APIs:

- com.adobe.pdfjt.pdf.digsig.PDFLegalAttestation
- com.adobe.pdfjt.services.digsig.LegalScrubber

See also these other samples:

- core.document.RedactionSample
- core.document.SanitizationSample

CertifyAndSignExistingSignatureFields.java

This program shows how to sign signature fields in a PDF document, and how to group digital signature fields into locked signatures.

This program is very similar to the sample program SignExistingSignatureFields. The SignExisting program builds on the CertifyandSignExisting program, in that it adds more information to the PDF document.

Relevant APIs:

- com.adobe.pdfjt.pdf.digsig.PDFFieldAction
- com.adobe.pdfjt.services.digsig

See also these other samples:

- core.digitalSignatures.SignExistingSignatureFields

CertifyDocument.java

This sample illustrates how to apply a simple certifying (or author) signature to a PDF document. An "author" signature refers to a digital signature applied to a PDF document by a person who wrote or created the PDF document.

Relevant APIs:

- com.adobe.pdfjt.pdf.interactive.forms.PDFInteractiveForm
- com.adobe.pdfjt.services.digsig

See also these other samples:

- core.digitalSignatures.SignDocument
- core.digitalSignatures.CertifyDocumentAnndLockSigField

CertifyDocumentAndLockSigField.java

This sample program builds on the related sample program, CertifyDocument.java. Like CertifyDocument, this program applies a signature to a PDF document but it then locks the signature field against any further changes.

Relevant APIs:

- com.adobe.pdfjt.services.digsig.SigFieldLock

See also these other samples:

- core.digitalSignatures.CertifyDocument

CertifyDocumentWithObjectDigestControl.java

This sample program builds on the related sample program, CertifyDocument.java. Like CertifyDocument, this program opens a PDF document and applies a signature to that document, but it then adds a call that adds an Object Digest Control. A Digest Method is an array of names that indicate the acceptable digest algorithms that can be applied when signing a PDF document.

The value will be one of the following:

- SHA1
- SHA256
- SHA384
- SHA512
- RIPDMD160

These algorithms create encrypted hash values that can be used to verify the integrity of a dataset without providing any means to access and read the values in that dataset in their original form.

Relevant APIs:

- com.adobe.pdfjt.services.digsig.ObjectDigestMode
- com.adobe.pdfjt.services.digsig

See also these other samples:

- core.digitalSignatures.CertifyDocument
- core.digitalSignatures.SignDocumentWithDigestOptions

CreateUnsignedSignatureField.java

This sample illustrates how to add two unsigned signature fields to a document. It also shows how to apply seed values to a signature field. A set of seed values can be used to present a limited set of rationales to choose from when signing a PDF document.

For example, if a PDF form had a series of signature fields, as part of a workflow for an approval process, you could use seed values to require that the signatures be completed in a specific order, and that each signature field must be signed by a specific individual or one person from an approved group of individuals.

Or you could create a drop down menu item connected to a signature field showing a list of reasons for signing, such as:

- Legal Evaluation Complete
- Testing Complete
- Project Complete
- Final Approval
- Payment Approved for Release

Relevant APIs:

- com.adobe.pdfjt.pdf.interactive.forms.PDFInteractiveForm
- com.adobe.pdfjt.services.digsig.SignatureFieldFactory

See also these other samples:

- core.digitalSignatures.SignExistingSignatureFields

CredentialCreation.java

This sample shows how to import keys and certificates for use with the PDF Java Toolkit security and digital signature capabilities.

The program replaces the certificate key applied to a PDF document with a certificate and key provided by Datalogics, drawn from the key and certificate files pdfjt-eval-key.der and pdfjt-eval-cert.der, both stored in the input/digsig directory.

DER refers to Distinguished Encoding Rules, a standard for encoding and decoding data structures.

Relevant APIs:

- com.adobe.pdfjt.core.credentials.Credentials
- com.adobe.pdfjt.core.credentials.PrivateKeyHolder

See also these other samples:

- core.digitalSignatures.CertifyDocument
- core.digitalSignatures.ApplyUsageRights

EmbedCRLInSignature.java

This sample demonstrates how to add a Certificate Revocation List (CRL) to a signature field. A CRL is a list of serial numbers for certificates that have been revoked by certifying authorities. These revoked certificates should no longer be trusted.

To add the certificate revocation data to a signature, the software provides information about the revocation information provider and registers that provider with SignatureOptions.

SignatureOptions is a Java data structure that holds parameters related the signatures that appear on a PDF document. It can provide overrides for various parts of the process followed to create a signature.

Relevant APIs:

- com.adobe.pdfjt.services.digsig.spi.RevocationInfoProvider
- com.adobe.pdfjt.services.digsig.SignatureOptions

See also these other samples:

- core.digitalSignatures.EmbedOCSPInSignature

EmbedOCSPInSignature.java

This sample demonstrates how to add revocation information checks to a signature field via Online Certificate Status Protocol (OCSP). OCSP is an Internet protocol used to request the revocation status of a digital certificate. OCSP is similar to CRL; messages communicated to vendors or providers via OCSP are usually communicated over Hypertext Transfer Protocol (HTTP).

To add the certificate revocation data to a signature, the software provides information about the revocation information provider and registers that provider with SignatureOptions.

SignatureOptions is a Java data structure that holds parameters related the signatures that appear on a PDF document. It can provide overrides for various parts of the process followed to create a signature.

Relevant APIs:

- com.adobe.pdfjt.services.digsig.spi.RevocationInfoProvider

See also these other samples:

- core.digitalSignatures.EmbedCRLInSignature

FlattenSignatures.java

This sample illustrates how to flatten all of the digital signatures and/or signature fields in a signed PDF document. The program has code to flatten all of the signatures in a document and to flatten just one signature, and it produces two PDF output documents. Each has three signatures. For one, all of the signature records are flattened; for the other, only one is flattened. In that case the other signature field in the PDF document remains intact, and you can click on the digital signature and display properties.

The flatten process, in working with PDF documents, combines layers of content, or a stack of transparent images or colors on a PDF page and renders the result as a single image, color, or set of text. When a signature is flattened, the digital certificate key and related properties are removed from the signature field. The name of the person who signed the document and related information, such as the date and time stamp and the signer's email address, appear on the page as text. The signature field is no longer presented as an annotation.

Relevant APIs:

- com.adobe.pdfjt.services.digsig.SignatureManager

See also these other samples:

- core.forms.FlattenForm

GetLongTermValidationInfo.java

This sample demonstrates how to find information related to Long Term Validation information in a signed PDF document and display relevant details.

Signed documents are frequently kept in storage indefinitely, and retain their legal authority years or even decades after they were first executed. The Long Term Validation Profile provides the same service for electronically signed PDF files. With LTV, a user will be able to confirm a signature was valid at the time the electronic signature was applied to the PDF many years after the PDF file is created.

This program opens a pair of PDF input documents with digital signatures, looks for a Document Security Store (DSS) dictionary, and lists all of the certificates found there. The program also finds all Certificate Revocation Lists (CRL) and Online Certificate Status Protocol (OCSP) responses referenced in the Validation Related Information (VRI).

A CRL is a list of serial numbers for certificates that have been revoked by certifying authorities. These revoked certificates should no longer be trusted. OCSP is an Internet protocol used to request the revocation status of a digital certificate. OCSP is similar to CRL; messages communicated to vendors or providers via OCSP are usually communicated over Hypertext Transfer Protocol (HTTP). VRI includes indirect references to validation data.

The program lists the information it finds about the certificate records and the digital signatures in a command prompt or in the Eclipse console.

Relevant APIs:

- com.adobe.pdfjt.pdf.digsig.PDFValidationRelatedInfo
- com.adobe.pdfjt.pdf.digsig.PDFDocumentSecurityStore

See also these other samples:

- core.digitalSignatures.AddLongTermValidationInfo
- core.digitalSignatures.pades.AddPAdESLTV

JSSigInfoExtensionDemo.java

This program demonstrates how PDF Java Toolkit can run JavaScript in PDF documents. A JavaScript embedded in a PDF document would normally run with the PDF document displayed in Adobe Acrobat. PDF Java Toolkit offers a utility called the JavaScript Handler that allows a JavaScript to *imitate* Adobe Acrobat. Thus the script can work with a PDF document without Acrobat being installed on the machine being used.

For this program the JavaScript Handler uses a callback interface JSSigValidateProvider to provide support for setting a number of parameters in SignatureInfo.

SignatureInfo is a generic JavaScript Application Programming Interface (API) that holds the properties of a digital signature. For example, SignatureInfo holds the Date object, that provides the date and time when the signature was created, the Digest Method string, such as SHA1 or SHA256, the name of the user who signed the document, and the location (often the city) where the document was signed.

An Extension allows a system to extend the virtual functions available when the JavaScript Handler imitates utilities provided by Adobe Acrobat. For example, when verifying a signature in a PDF document, it will be necessary to imitate the SignatureInfo object, and run a callback function that will create the same function as its equivalent process in Adobe Acrobat. The extension provides JavaScript that will replace an equivalent feature within Acrobat.

A callback is a method executed when an event is completed, or when called by a timer.

Relevant APIs:

- com.adobe.pdfjt.services.javascript.JavaScriptHandler;
- com.adobe.pdfjt.services.javascript.extension.JSSigInfo

See also these other samples:

- core.forms.JSExtension
- core.forms.ExecuteJavaScriptInFieldCalculation

PKCS7ContentParser.java

This sample program demonstrates how to draw the Public Key Cryptography Standard (PKCS) #7 certificate from a digital signature in a PDF document, and then decode the certificate to collect information about the signature. The PKCS7 standard is used to sign and encrypt messages. The information includes the time and date stamp and revocation information. The sample program lists the full values of the certificate key, the time stamp token, and any CRL and OCSP responses embedded in the digital signature at the command prompt or on the Eclipse console. As a result the response returned runs for five or six pages for the default input PDF document used.

A Certificate Revocation List (CRL) is a list of serial numbers for certificates that have been revoked by certifying authorities. These revoked certificates should no longer be trusted. The Online Certificate Status Protocol (OCSP) is an Internet protocol used to request the revocation status of a digital certificate. OCSP is similar to CRL; messages communicated to vendors or providers via OCSP usually come over Hypertext Transfer Protocol (HTTP).

Relevant APIs:

- com.adobe.pdfjt.services.digsig.PKCS7SignatureParser
- com.adobe.pdfjt.services.digsig.PKCS7SignerInfo

See also these other samples:

- core.digitalSignatures.QuerySignatureFields

QueryPermissionsSignatures.java

This sample collects and then prints out metadata related to permissions signatures for a PDF input document. The information appears at a command prompt or in an Eclipse console, and includes information about Reader Enabling and DocMDP restrictions.

```
Document: AuthorSignature.pdf
      Signature Type: author (certifying)
      FieldMDP Restrictions: Yes
```

The Reader Enabling feature in Adobe Acrobat allows a user to set up a PDF document so that another user can open it in Adobe Reader and still have rights to add comments, sign the document, and save the PDF document with form fields completed. This sample program will determine if a signature in a PDF document is enabled for Reader.

MDP refers to Modification Detection and Protection. The DocMDP transform method is used to detect changes to a signature field that has been signed by the author of a document. A PDF document can hold only one signature with a DocMDP transformation method, and it makes it possible for the author of the document to specify what changes are permitted to be made to the document and what changes invalidate the author's signature.

Relevant APIs:

- com.adobe.pdfjt.pdf.digsig.PDFSignature
- com.adobe.pdfjt.services.digsig.SignatureUtils

See also these other samples:

- core.document.PermissionsQuerySample
- core.security.ReEncryptDocument

QuerySignatureFields.java

This sample program shows how to search for digital signatures in a PDF document. The program displays the results at the command prompt or in the Eclipse Console, including the name of each signature field, whether the field is signed, the signature type, and Field MDP Restrictions.

Relevant APIs:

- com.adobe.pdfjt.pdf.digsig.PDFSignature
- com.adobe.pdfjt.services.digsig

See also these other samples:

- core.annotations.QueryAnnotations
- core.forms.QueryForms

RemoveSignatures.java

This sample illustrates how to remove all of the signatures and signature fields, with associated content, from a PDF document. The program takes a PDF input document with a blank signature field and two completed signature fields, deletes them, and saves the result as a blank PDF.

Relevant APIs:

- com.adobe.pdfjt.services.digsig.SignatureManager

See also these other samples:

- core.digitalSignatures.UnSignSignatures
- core.digitalSignatures.RemoveSignaturesUsingIterator

RemoveSignaturesUsingIterator.java

This sample program shows how to look through the digital signatures in a PDF document and remove each one individually. When you run the program it will delete the first of three signatures found in the source PDF document and save the result to a separate PDF.

You could use this code as a the basis for a system that might, for example, find all of the signatures in a PDF document, and then prompt a user to confirm whether each of the signatures in that document should be deleted, one at a time. Or your program could be set up to delete the first two signatures in a PDF document and leave the last one in place.

Relevant APIs:

- com.adobe.pdfjt.services.digsig.SignatureManager
- com.adobe.pdfjt.services.digsig.SignatureFieldInterface

See also these other samples:

- core.digitalSignatures.RemoveSignatures

RemoveUsageRights.java

This sample illustrates how to remove the usage rights (Reader Enablement) from a Reader-Enabled PDF Document. The program will open a PDF document, remove the rights, and save the result to a separate PDF document.

If you open the output document and click the Signatures icon ✍ you will see that the “Validate All” option to validate the signatures in the document has been disabled.

The Reader Enabling feature in Adobe Acrobat allows a user to set up a PDF document so that another user can open it in Adobe Reader and still have rights to add comments, sign the document, and save the PDF document with form fields completed. This sample program will remove the ability of a user to edit the PDF document in Adobe Reader.

Relevant APIs:

- com.adobe.pdfjt.services.digsig.SignatureManager

See also these other samples:

- core.digitalSignatures.ApplyUsageRights

Rollback.java

This sample program shows how to return to a previous version or revision of a PDF document that has been updated using incremental saves. The program starts with an input PDF document and generates two PDF output documents, with previous signature changes reversed, so that the signatures are returned to their original state before the document was updated and saved.

Relevant APIs:

- com.adobe.pdfjt.pdf.document.PDFOpenOptions
- com.adobe.pdfjt.pdf.document.PDFSaveIncrementalOptions

SignDocument.java

This program shows how to open a blank PDF file and add a simple digital signature to that file. SignDocument opens "Simple.PDF" from the input directory of PDFJT, attaches a signature, and saves the result to an output file called "Simple_signed.PDF." This result file is locked against any further changes, though other readers can open the file and attach their own signatures or add comments. If the PDF file used is a form, a user could fill in form fields and save the file with those changes as well.

Relevant APIs:

- com.adobe.pdfjt.core.credentials.Credentials;
- com.adobe.pdfjt.services.digsig.SignatureManager

See also these other samples:

- core.digitalSignatures.CerifyDocument
- core.digitalSignatures.SignDocumentWithAppBuildOptions

SignDocumentwithAppBuildOptions.java

Like SignDocument, this program shows how to open a blank PDF file, add a simple digital signature to that file, and then save the result to an output PDF document. But it also shows how to select Build Properties Dictionary entries as options to be applied when a user adds a signature to the document.

The Build Properties Dictionary can be used to store information that describes the state of the computer used when signing the PDF document.

In this sample program the entries include:

- the page number where the signature is found
- the Digest method used (defaults to SHA256)
- the Operating System on the computer
- the application used to create the signature (defaults to My Signing Application 8.0)

Relevant APIs:

- com.adobe.pdfjt.pdf.digsig.PDFAppBuildData
- com.adobe.pdfjt.services.digsig.SignatureOptions
- com.adobe.pdfjt.pdf.digsig.PDFBuildProp

See also these other samples:

- core.digitalSignatures.SignDocument

SignDocumentWithCustomAppearance.java

This sample builds on the SignDocument sample program and adds a custom appearance to the signature field. Specifically, the program adds an avatar drawing to the signature field, and an image of a hand-written signature, both appearing on the final PDF file:

Relevant APIs:

- com.adobe.pdfjt.services.digsig.SignatureAppearanceOptions
- com.adobe.pdfjt.services.digsig.SignatureOptions

SignDocumentWithDigestOptions.java

This sample builds on the SignDocument sample program. This program shows how to open a blank PDF file and add a simple digital signature to that file. It also shows how to specify the digest option to be applied to the signature. The digest options are an array of names that indicate the acceptable algorithms that can be applied when signing a PDF document. The value will be one of the following:

- SHA1
- SHA256
- SHA384
- SHA512
- RIPDMD160
- MD5

These algorithms create encrypted hash values that can be used to verify the integrity of a dataset without providing any means to access and read the values in that dataset in their original form.

Relevant APIs:

- com.adobe.pdfjt.pdf.digsig.PDFSignature
- com.adobe.pdfjt.services.digsig.SignatureOptions

See also these other samples:

- core.digitalSignatures.SignDocument
- core.digitalSignatures.CertifyDocumentWithObjectDigestControl

SignExistingSignatureFields.java

This sample program signs multiple unsigned signature fields in a PDF document and sets the field lock actions. It shows you how to work with the field lock feature.

The program generates a set of sample PDF form documents, each one with two groups of check boxes, radio buttons, push buttons, and text fields. Each group of these form fields is controlled by a signature field. When the program assigns a signature to the signature field in one of these groups, all of the forms items in that group are locked. It is no longer possible to enter a value in a text field or change a radio button option.

Relevant APIs:

- com.adobe.pdfjt.services.digsig.SignatureManager
- com.adobe.pdfjt.services.digsig.SignatureFieldInterface

See also these other samples:

- core.digitalSignatures.CreateUnsignedSignatureField
- core.digitalSignatures.UnsignSignatures

TimeStampSignature.java

This sample demonstrates how to apply an ordinary (non-author) signature with a time stamp to a document. Applying timestamps to signatures is completed by first creating a timestamp provider implementation, and then registering the timestamp provider with SignatureOptions.

SignatureOptions is a Java data structure that holds parameters related the signatures that appear on a PDF document. It can provide overrides for various parts of the process followed to create a signature.

A “non-author” signature refers to a digital signature applied to a PDF document by a person who did not write or create that document.

The emphasis of this sample program is on applying a time stamp to a digital signature in a PDF document, and creating that time stamp in a standard way that is easy to extract from the file.

Relevant APIs:

- com.adobe.pdfjt.services.digsig.spi.TimeStampProvider
- com.adobe.pdfjt.services.digsig

See also these other samples:

- core.digitalSignatures.pades.AddPAdESLTV

UnsignSignatures.java

This sample illustrates how to remove signatures from all signed signature fields in a PDF document. It starts with an input PDF document that has three signature fields, two of them signed. In the output file, all three of the signature fields are blank.

Relevant APIs:

- com.adobe.pdfjt.services.digsig.SignatureManager

See also these other samples:

- `core.digitalSignatures.RemoveSignatures`
- `core.digitalSignatures.RemoveSignaturesUsingIterator`

ValidateSignature.java

ValidateSignature takes a PDF file with a signature attached and validates that signature. The program selects a signed PDF file called "Signed_Gibson_PKCS7_DETACHED_ Sha256.pdf" in the input/digsig directory and finds the signature.

The program evaluates the signature in the PDF file and responds:

```
Signature field "Signature1" is signed.
The signature is VALID.
The signature COVERS the entire document.
```

Note that the program performs only signature validation, not signer validation. That is, the sample program only validates the signature, not the party that provided the signature; the sample does not provide the code needed to contact an external Certifying Authority (a vendor) to verify that the signature shown on the PDF document is authentic. Rather it simply verifies that the signature can be read properly. Your organization will need to complete the Certificate Path Building and Certificate Path Validation processes separately.

To complete the Certificate Path Validation in Windows, open the Local Group Policy Editor. Then, select Computer Configuration, Windows Settings, Security Settings, Public Key Policies, and Certificate Path Validation Settings.

To validate the digital signature in a PDF file, a computer system must determine the validity of the certificate key associated with that signature. It completes this process by linking the digital signature to a certificate path and working through that path until it links the digital signature to a certificate key that the system has already validated and knows it can trust. This certificate is called the Trust Anchor.

The certificate path is an ordered series of certificates that starts with the Trust Anchor. The system validates the digital signature by working forward from the Trust Anchor through each certificate until it reaches the digital signature. Each certificate verifies the authenticity of the certificate before it in the sequence. The computer system may retrieve the intermediate certificates in the certificate path from a network or over the Internet using any means available to that application, including LDAP, HTTP, SQL, a local server cache, or other methods.

Relevant APIs:

- com.adobe.pdfjt.services.digsig.SignatureUtils
- com.adobe.pdfjt.services.digsig.cryptoprovider

Digital Signatures and PDF Advanced Electronic Signatures (PAdES)

PDF Advanced Electronic Signatures, or PAdES, is a European standard for digital signatures in PDF documents. Effectively, PAdES is a set of restrictions and extensions for PDF that ensures compliance with legal requirements for digitally-signed documents.

- **Basic**. The initial Electronic Signature standard.
- **Enhanced**. Explicit Policy Electronic Signatures Profiles (EPES)
- **Long Term Validation Profile (LTV)**. Signed printed documents are frequently kept in storage indefinitely, and retain their legal authority years or even decades after they were first executed. The Long Term Validation Profile provides the same service for electronically signed PDF files. With LTV, a user will be able to confirm a signature was valid at the time the electronic signature was applied to the PDF many years after the PDF file is created.

AddPAdESBasic.java

This sample program shows how to create a signed input PDF document with a signature that is compliant with the PAdES Basic standard.

Relevant APIs:

- com.adobe.pdfjt.pdf.digsig.PDFSignatureSubFilter
- com.adobe.pdfjt.services.digsig.SignatureOptionsCADES

See also these other samples:

- core.digitalSignatures.pades.PAdESEPES

AddPAdESEPES.java

This sample demonstrates how to create a signed PDF document with a signature compliant with PAdES Basic Electronic Signature (BES) or Explicit Policy Electronic Signature (EPES). The difference between BES and EPES is that EPES requires the policy identifier attribute, and BES does not.

Relevant APIs:

- com.adobe.pdfjt.pdf.digsig
- com.adobe.pdfjt.services.digsig.cryptoprovider

See also these other samples:

- core.digitalSignatures.pades.PAdESLTV

AddPAdESLTV.java

This sample adds Long Term Validation data that is compliant with PAdES-LTV specification. It expects that the document that is used as input already contains a PAdES basic, BES, or EPES compliant signature that was created using the PDFJT evaluation certificate and CRL.

This sample adds a DSS dictionary to the PDF document and a time and stamp signature.

The program applies a certificate and key to the document provided by Datalogics, drawn from the key and certificate files pdfjt-eval-key.der and pdfjt-eval-cert.der, both stored in the input/digsig directory.

The program also makes use of the pdfjt-eval-crl.der file, holding a Certificate Revocation List (CRL). A CRL is a list of serial numbers for certificates that have been revoked by certifying authorities. These revoked certificates should no longer be trusted.

DER refers to Distinguished Encoding Rules, a standard for encoding and decoding data structures.

Relevant APIs:

- com.adobe.pdfjt.pdf.digsig.PDFValidationRelatedInfo
- com.adobe.pdfjt.services.digsig.spi

See also these other samples:

- core.digitalSignatures.AddLongTermValidationInfo

PDF Document Features

AddJavaScript.java

This program shows how to add JavaScript code to a PDF Java Toolkit program. In this case the JavaScript provides an open action for a PDF document. When you run the program it generates a PDF output document. When you open this PDF document, you will hear a beep and see an error message, “Hello World.”

Relevant APIs:

- com.adobe.pdfjt.pdf.interactive.action.PDFActionJavaScript
- com.adobe.pdfjt.pdf.document.PDFOpenAction

See also these other samples:

- core.digitalSignatures.JSSigInfoExtensionDemo

AddWatermarksAndBackground.java

This sample program shows how to add a watermark and a text background to each page of a PDF document. It creates a watermark, and sets it at an angle, which is then displayed on every page.

The code also shows how to create a header that will appear on a PDF document when it is being viewed online but that does not print when the document is sent to a printer. A watermark becomes a part of a page and will always print; a background statement can be made to be visible when viewed on a screen but not printed.

Relevant APIs:

- com.adobe.pdfjt.services.xobjhandler
- com.adobe.pdfjt.services.rcg

See also these other samples:

- core.document.XObjectRichTextWatermark

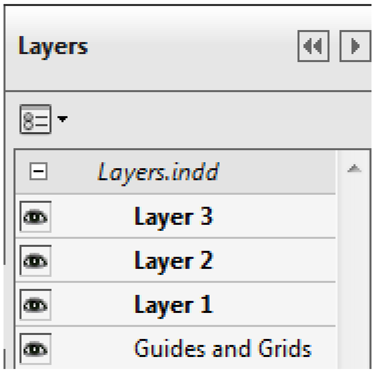
ChangeLayerConfiguration.java

This sample changes the On/Off value for a set of layers within a PDF document. This determines whether or not the layers are visible when the PDF document is opened in Adobe Acrobat or another viewer that can interpret layers.

The program starts with a PDF document with three layers, all turned on by default, so all three appear when the file is opened in Acrobat. Layer 1 is the background image, and layer 2 is a text header. Layer 3 includes a set of text that appears on top of the image on the page. From this input file the program generates an output file where layer three is turned off by default.

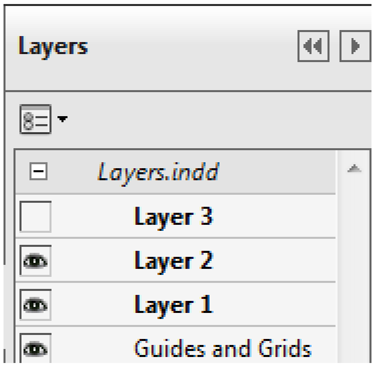
You can see the layers if you click the layer icon in Acrobat, .

For the input file, the layers default to this:



⊟ *Layers.indd*
👁 **Layer 3**
👁 **Layer 2**
👁 **Layer 1**
👁 Guides and Grids

The output file turns off layer 3:



⊟ *Layers.indd*
□ **Layer 3**
👁 **Layer 2**
👁 **Layer 1**
👁 Guides and Grids

Relevant APIs:

- com.adobe.pdfjt.pdf.graphics.optionalcontent.PDFOCConfig
- com.adobe.pdfjt.pdf.graphics.optionalcontent.PDFOCGroup

See also these other samples:

- core.document.LayerOptions

CopyImageSample.java

This sample shows how to insert raw compressed image data into a PDF document. The program draws a photograph from a newsletter called WatchDog.PDF, distinguishing the graphic from surrounding text, and saves this raw image (and only this image) to the first page of a PDF file called TestCopyImage.PDF. The program also saves the image to a graphics file called TestCopyImage.JPG.

Relevant APIs:

- com.adobe.pdfjt.pdf.graphics
- com.adobe.pdfjt.services.imageconversion

See also these other samples:

- core.document.RasterizationSample
- core.imageConversion.ImageExtractionSample

FindStructureContent.java

This sample demonstrates how to search through the structure tree of a PDF document to find Marked Content. Marked Content operators (or tags) allow a portion of a content stream within a PDF document to be grouped, for purposes like indicating structural elements such as headers, paragraphs, or graphical images.

The program opens a PDF document, scans for Marked Content objects, and then lists information about these objects at the command prompt or in the Eclipse console, like this:

```
Entry for page 0
Element 58/0: <</C/SC2508/K 0/P 55 0 R/S/Span/Pg 97 0 R>>
Contents: Object 106/0
MCIDs:
0
End of Entry
Entry for page 0
Element 60/0: <</C/SC2512/K 1/P 59 0 R/S/Span/Pg 97 0 R>>
Contents: Object 106/0
MCIDs:
1
End of Entry
```

Relevant APIs:

- com.adobe.pdfjt.pdf.interchange.structure.PDFStructureElement
- com.adobe.pdfjt.services.interchange.structure.StructureFinder

See also these other samples:

- core.document.removeContent

HelloWorldSample.java

This program is a simple introductory program. It demonstrates how to create a new PDF document and place the text “Hello World” using the Richtext Content Generator (RCG). The output file is called HelloWorld.pdf.

Relevant APIs:

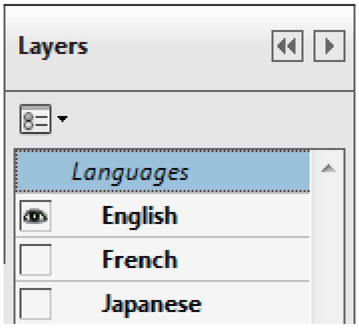
- com.adobe.pdfjt.services.rcg.RichTextContentGenerator
- com.adobe.pdfjt.pdf.graphics.xobject.PDFXObjectForm

LayerOptions.java

This sample program manages layers in a PDF document. LayerOptions.java takes an input PDF document with an image and text, on a single page, and creates a set of output PDF documents, each with an extra page inserted. The second page of each of the output files is a copy of the first page of the source file, but the program demonstrates the use of two different content options for working with PDF pages.

When you open the test PDF document in Acrobat, the image appears with the optional content as defined in the document's default config dictionary. The page also features optional content that has been turned off.

If you click the View Layers icon, , on the left side of the Acrobat window, a set of options appears, allowing you to change the language of the text shown on the page:



The source PDF document also features some optional text that is configured to be invisible, a blue banner that overlays the image shown. The sample output files show the two pages with no banner visible, one with the banner shown on the second page, and one with the banner shown on the first page.

Relevant APIs:

- com.adobe.pdfjt.services.manipulations.PMMOptions
- com.adobe.pdfjt.services.manipulations.PMMService

See also these other samples:

- core.document.changeLayerConfiguration

LinearizeDocument.java

This sample program shows how to produce a linearized PDF document. A linearized PDF is restructured in a way that allows the first page of the file to appear on a user's Web browser while the rest of the file is being downloaded. This type of PDF document can thus display more quickly on a web page; the user does not need to wait for the entire document appear before he or she can start reading it. This process is also known as byte-serving.

Relevant APIs:

- com.adobe.pdfjt.pdf.document.PDFSaveOptions
- com.adobe.pdfjt.pdf.document.PDFSaveLinearOptions

See also these other samples:

- core.document.sanitizationSample

ModifyDocument.java

This sample program shows how to change some of the entries in the Document Catalog for a PDF file. The program updates these entries:

1. **PageLayout**. This option describes how the pages will be displayed when the document is opened in a viewer. Examples include one page at a time (SinglePage) and display pages two at a time, with either the odd-numbered pages on the left (TwoPageLeft) or odd-numbered pages on the right (TwoPageRight).
2. **PageMode**. PageMode defines how a document is displayed when opened in a viewer. Some of the options include UseOutlines, or display the document outline, UseThumbs, or display thumbnail images of every page in the document, and FullScreen, or show the document without a menu bar or any window controls.
3. **OpenAction**. This value specifies a destination in the document that will be displayed when the document is opened, or an action that will be performed. If the OpenAction entry is blank, the PDF document will open at the top of the first page.
4. **PageLabels**. This is a number tree that allows for pages to have more sophisticated page numbering. For example, the preface of a document might use page numbers i, ii, and iii, and page numbers 1, 2, and 3 in the body of the text.

When the program runs, it displays a response at the command prompt or in the Eclipse console panel. In this case, it describes the Page Layout and Page Mode:

```
Layout was SinglePage
Mode was PagesOnly
Index is 0
```

Relevant APIs:

- com.adobe.pdfjt.pdf.document.PDFCatalog
- com.adobe.pdfjt.pdf.page.PDFPageLabel

See also these other samples:

- core.document.SetViewerPrefs

OpenDocument.java

This sample program extracts the version of the PDF document and the document-level XMP metadata from the input file AdobeCollections.PDF. The program sends these values at to the console.

XMP refers to the Extensible Metadata Platform, a standard created by Adobe Systems to guide the creation, processing, and exchange of metadata for a variety of digital resources.

Relevant APIs:

- com.adobe.pdfjt.pdf.document.PDFDocument
- com.adobe.pdfjt.pdf.document.PDFOpenOptions

See also these other samples:

- core.document.QueryDocument

PDFToolkitXMPSample.java

This sample extracts standard metadata from an input PDF document and displays it in the console. The data listed includes the authors of the file, the creator, the date and time stamps for when the document was created and changed, the producer, subject, and title.

The program then updates some of the metadata in the document and displays those changes.

Relevant APIs:

- com.adobe.pdfjt.services.xmp.XMPService
- com.adobe.pdfjt.services.xmp.Metadata

See also these other samples:

- core.document.QueryDocument

PermissionsQuerySample.java

This program shows how to query the permissions in a PDF document, to see if form flattening is permitted. If it is permitted, a message to that effect is sent to the console and the program creates a flattened version of the document.

Relevant APIs:

- com.adobe.pdfjt.core.permissionprovider.ObjectOperations
- com.adobe.pdfjt.services.permissions.PermissionsManager

See also these other samples:

- core.security.reEncryptDocument

QueryDocument.java

This sample program is similar to the OpenDocument.Java sample program. It extracts the version of the PDF document and the document-level XMP metadata from the input file AdobeCollections.PDF. The program displays these values at the command prompt or the Eclipse Console. It also counts and lists the number of pages in that PDF document as part of the output. XMP refers to the Extensible Metadata Platform, a standard created by Adobe to guide creating, processing, and exchanging metadata for a variety of digital resources.

Relevant APIs:

- com.adobe.pdfjt.pdf.document.PDFDocument
- com.adobe.pdfjt.pdf.page.PDFPageTree

See also these other samples:

- core.document.PermissionsQuerySample
- core.document.setViewerPrefs

RasterizationSample.java

This sample program shows how to render each page of a PDF Document to a bitmap image. The program opens an input file and exports the file pages to a series of PNG graphics files.

Relevant APIs:

- com.adobe.pdfjt.services.rasterizer.PageRasterizer;
- com.adobe.pdfjt.services.rasterizer.RasterizationOptions

ReadingOrderTextExtractionSample.java

This sample program searches through all of the words in a PDF document and provides information about these words, printing each one along with the associated coordinates for the Bounding Box for that word. The information is drawn from a given input file, and is output to a pair of text files, one Unicode and one standard text. The Bounding Box is a rectangle that can hold the content for a string of text or a page.

Relevant APIs:

- com.adobe.pdfjt.services.readingorder.ReadingOrderTextExtractor
- com.adobe.pdfjt.services.textextraction

See also these other samples:

- core.document.TextExtractionSample

RedactionSample.java

This sample program provides code used to redact content from a PDF document. The Redacting process removes information within a PDF document typically by replacing it with blank rectangles over the areas where the information was removed. Redaction is commonly used to remove sensitive or classified information when a document is released to the public. The process first applies an annotation to the document to cover sensitive content, and then removes the content found under the annotation, leaving only the rectangle to indicate where the information was removed.

Relevant APIs:

- com.adobe.pdfjt.services.redaction.RedactionService
- com.adobe.pdfjt.services.redaction.RedactionOptions

See also these other samples:

- core.document.SanitizationSample

RemoveContent.java

This program parses content streams on every page within a PDF files. The program searches for a pre-defined sequence of operators, and rewrites the content stream in the PDF file, removing any matches to that pre-defined sequence. In this example, the program removes a solid white triangle that threatens to obscure part of a watermark that is to be added to the page.

Relevant APIs:

- com.adobe.pdfjt.pdf.contentmodify.ContentWriter
- com.adobe.pdfjt.pdf.content

SanitizationSample.java

This program shows how to use code to sanitize a PDF document. This process involves removing hidden data from the document, such as metadata, annotations, form fields, attachments, and bookmarks. The resulting PDF file is smaller and optimized for better performance.

Relevant APIs:

- com.adobe.pdfjt.services.sanitization.SanitizationService;
- com.adobe.pdfjt.pdf.document.PDFSaveOptions

See also these other samples:

- core.document.LinearizeDocument
- core.document.RedactionSample

SetupFilterParams.java

This sample demonstrates how to set up a PDFFilterList representing the compression filters and decode parameters used by an image in a PDF document.

The PDFFilterList is a Java class that represents an ordered list of PDFFilter Objects. PDF documents normally compress graphics images that they contain, to reduce the size of the resulting file. PDF documents use these compression filters, or algorithms, to compress graphics objects:

- LZW, or Lempel-Ziv-Welch
- FLATE (ZIP)
- JPEG and JPEG2000
- CCITT
- JBIG2
- RLE, or Run Length Encoding

The program opens a sample input PDF file and analyzes the single graphic image it contains. Then, it responds with details about the Discrete Cosine Transform (DCT) image compression of the graphic at the command line or in the Eclipse console:

```
DCTDecode: <</HSamples[1 1 1 1]/Columns 2550/Rows 3300/QFactor 0/Blend
1/Colors 3/ColorTransform 1/VSamples[1 1 1 1]>>
```

Relevant APIs:

- com.adobe.pdfjt.pdf.filters.PDFFilterParams
- com.adobe.pdfjt.core.cos

See also these other samples:

- core.pdfa.ConvertPDFA1Document

SetViewerPrefs.java

The SetViewerPrrefs sample program shows how to write code to automatically change the viewer preferences within a PDF document. The program opens AdobeCollections.PDF, a product guide from Adobe Systems, updates the viewer preferences, and saves the results to a new PDF file called Creates ViewerPrefs.PDF. Specifically, the program opens AdobeCollections.PDF, opens the Page Thumbnails on left side of the window, and switches the Page Display setting from Single Page View to Two Page Scrolling.

Relevant APIs:

- com.adobe.pdfjt.pdf.interactive.PDFViewerPreference
- com.adobe.pdfjt.pdf.document.PDFCatalog
- com.adobe.pdfjt.pdf.page.PDFPageLayout

See also these other samples:

- core.document.QueryDocument

TextExtractionSample.java

TextExtractionSample extracts all of the words found in a given document, and prints these words at the command prompt or console. The program provides associated coordinates for the Bounding Box for each word.

Relevant APIs:

- com.adobe.pdfjt.services.textextraction

See also these other samples:

- core.document.ReadingOrderTextExtractionSample

TransformPage.java

The TransformPage sample program takes a document called ShalomWorld.PDF and transforms the first page by stretching the content horizontally and flipping the page on its side, and then saving the result to a new output file called ShalomWorld_trans.PDF. The program uses the PMMPageTransformations method to adjust the scale and rotation of the content on the page.

PMM refers to “Page Matrix Manipulation;” it allows you to rotate text objects, change their scale, and make other changes to how they appear on the page.

Relevant APIs:

- com.adobe.pdfjt.services.manipulations.PMMPageTransformations
- com.adobe.pdfjt.pdf.graphics.PDFRotation

See also these other samples:

- core.imageConversion.TransformedImageExtractionSample

UnembedFonts.java

Use this sample program to remove embedded fonts from a PDF document.

Generally embedding a font in a PDF documents is considered a best practice, but it is not required. PDF viewers substitute a system font if a font is not embedded in a PDF document. But that can cause problems if a PDF document that must rely on certain fonts being available on a local machine cannot find those fonts there. A PDF document with embedded fonts is more portable, and can be more safely archived, as with a PDF/A document (PDF Archive). But it is possible to remove embedded fonts if you want a file that is smaller, and that can draw fonts from a local machine.

Relevant APIs:

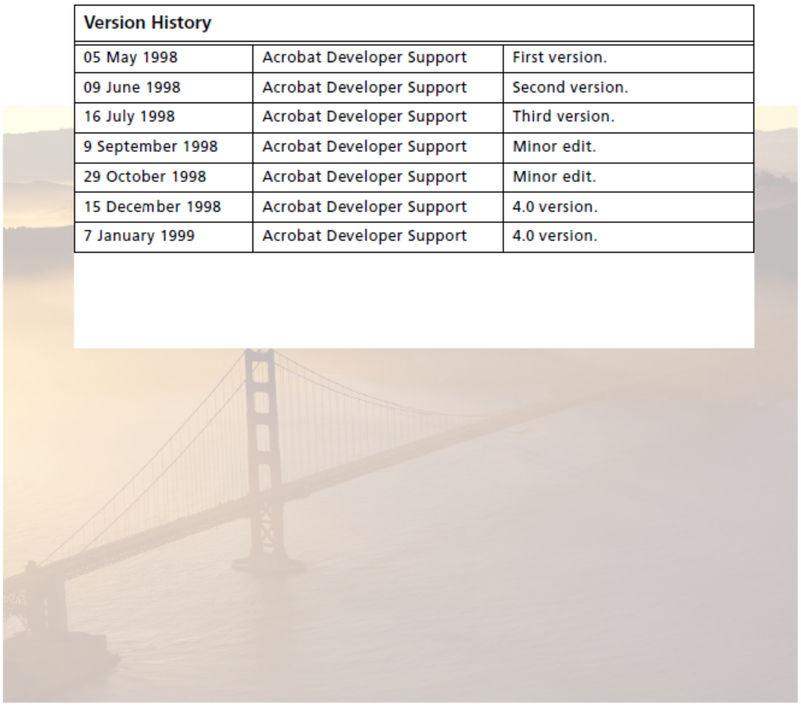
- com.adobe.pdfjt.pdf.graphics.font
- com.adobe.pdfjt.core.cos

See also these other samples:

- core.pdfa.ConvertPDFA1Document

XObjectImageWatermark.java

The sample program takes a JPEG graphics file called California.JPG (a photo of the Golden Gate Bridge) and adds the image as a watermark to the bottom of every page of the input PDF document. The output file, ImageWatermarked.PDF, features pages that look like this:



Version History		
05 May 1998	Acrobat Developer Support	First version.
09 June 1998	Acrobat Developer Support	Second version.
16 July 1998	Acrobat Developer Support	Third version.
9 September 1998	Acrobat Developer Support	Minor edit.
29 October 1998	Acrobat Developer Support	Minor edit.
15 December 1998	Acrobat Developer Support	4.0 version.
7 January 1999	Acrobat Developer Support	4.0 version.

Relevant APIs:

- com.adobe.pdfjt.services.imageconversion.ImageManager
- com.adobe.pdfjt.pdf.graphics.xobject.PDFXObjectForm

See also these other samples:

- core.document.HelloWorldSample
- core.imageconversion.ImageInsertionSample

XObjectRichTextWatermark.java

The sample program creates a Rich Text watermark and adds this text to every page of a PDF document. The watermark characteristics, including type size, font, color, position of the text, and other values are all specified using a Mark Up language defined in the PDF Reference or the XFA specification. The XFA specification is similar to HTML but referred to as Rich Text by Adobe Systems.

Relevant APIs:

- com.adobe.pdfjt.services.rcg.RichTextContentGenerator
- com.adobe.pdfjt.services.xobjhandler

See also these other samples:

- core.document.HelloWorldSample

PDF Electronic Forms

Some of these sample programs work with FDF files. FDF refers to Forms Data Format, a file format described in the PDF specification. FDF files use the same low-level syntax as PDF, but the FDF format is only used to describe PDF annotations and form fields.

FDF can be used in a variety of ways, such as submitting data from forms to a server application for processing, or to archive forms information. XFDF is an XML-based version of the FDF format. PDFJT can import and export both FDF and XFDF to and from PDF documents.

AddFieldUsingGivenFont.java

This sample program shows how to create text fields in a PDF form. The program demonstrates several examples of how to customize a text field, including applying a font to that field.

Relevant APIs:

- com.adobe.pdfjt.pdf.interactive.forms.PDFFieldText
- com.adobe.pdfjt.pdf.interactive.annotation
- com.adobe.pdfjt.services.ap

ExecuteJavaScriptInFieldCalculationOrder.java

This sample program opens an input PDF form document with six form fields and executes the JavaScript actions associated the fields shown. The sample program then updates the form appearance by flattening the form fields. This shows that it is the PDF Java Toolkit program that is manipulating the PDF document.

The flattening process removes the dynamic elements of the fields on the form, leaving the PDF document with the field content fixed on the page. Whenever a PDF document is opened in Adobe Acrobat or Adobe Reader the viewer automatically executes any scripts or calculations associated with that document. The sample program will execute the JavaScript and then flatten the form to demonstrate that it is Java Toolkit that is running the script and editing the document, not Adobe Acrobat.

Relevant APIs:

- com.adobe.pdfjt.services.javascript.JavaScriptHandler
- com.adobe.pdfjt.pdf.interactive.action.PDFJavaScriptEvent

See also these other samples:

- core.document.addJavascript
- core.forms.JSExtension

FDFFormExportDemo.java

FDFFormExportDemo demonstrates how to export data from Acrobat Form (Acroform) fields in an existing PDF file to an FDF file. This process could be used, for example, to archive these forms data values to a smaller file. The program takes the form data from a file called Sample2.PDF and creates an export file called FDFFormExportDemo_output.FDF.

Relevant APIs:

- com.adobe.pdfjt.services.fdf.FDFDocument
- com.adobe.pdfjt.services.fdf.FDFService

See also these other samples:

- core.Forms. FDFFormImportDemo
- core.Forms.XFDFFormImportDemo
- core.Forms.XFAFormImport.Demo

FDFFormImportDemo.java

FDFFormInportDemo demonstrates how to import form data into an existing PDF file. The program takes the form data from a FDF file called MySample2.FDF and adds it to a blank PDF form to create an export file called FDFFormImportDemo_output.PDF.

Relevant APIs:

- com.adobe.pdfjt.services.fdf.FDFDocument
- com.adobe.pdfjt.services.fdf.FDFService

See also these other samples:

- core.forms.FDFFormExportDemo

- core.forms.XFDFFormExportDemo
- core.forms.XFAFormExportDemo

FlattenForm.java

The FlattenForm sample program opens an input document with form fields and generates a flattened version of that document. This means that the input fields and widget annotations in the PDF Form, such as radio buttons and drop down menus, are removed, leaving only the appearances, or selections and entries, in place, permanently fixed.

Relevant APIs:

- com.adobe.pdfjt.services.formflattener.FormFlattener

See also these other samples:

- core.digitalSignatures.FlattenSignatures

FormFieldServiceSample.java

This sample demonstrates how to add new Acrobat Form (AcroForm) fields to a PDF document; it demonstrates the FormFieldService API, an API that makes creating forms easier (`com.adobe.pdfjt.services.forms.FormFieldService`).

The program opens a PDF document and adds two text fields, a list box with five options, a check box, four radio buttons, and two push buttons, before saving the result to an output PDF file.

Relevant APIs:

- com.adobe.pdfjt.services.forms.FormFieldService
- com.adobe.pdfjt.services.forms.FormFieldManager

See also these other samples:

- core.forms.AddFieldUsingGivenFont

FormTypeEvaluator.java

The FormTypeEvaluator sample program reviews a series of PDF files to determine the type of PDF form each one holds:

1. AcroForm
2. XFAStatic, shell or non-shell
3. XFADynamic, shell or non-shell

The source files are found in the PDFForms input file. The program evaluates each one in the following order and displays its findings:

1.	TestNoForm.PDF	Flat, or a form with no interactive capability
2.	TestAcroform.PDF	Acroform
3.	TestXFAStatic.PDF	StaticNonShellXFA
4.	TestXFADynamic.PDF	DynamicShellXFA

AcroForm, or Acrobat Forms, is a standard format that supports interactive forms, with items like radio buttons and drop down lists. XFA, or XML Forms Architecture, is a set of proprietary XML specifications for use with web forms. Since 2002 the XFA standard has been owned by Adobe Systems. XFA forms are saved internally in PDF files.

Relevant APIs:

- com.adobe.pdfjt.pdf.document.PDFDocument
- com.adobe.pdfjt.services.xfa.XFAService

See also these other samples:

- core.forms.PDFToolkitXFASample

JSExtension.java

This program demonstrates how PDF Java Toolkit can run JavaScript in PDF documents. A JavaScript embedded in a PDF document would normally run with the PDF document displayed in Adobe Acrobat. PDF Java Toolkit offers a utility called the JavaScript Handler that allows a JavaScript to *imitate* Adobe Acrobat. Thus the script can work with a PDF document without Acrobat being installed on the machine being used.

For this program the JavaScript Handler uses a callback interface ExtensionApp to provide support for setting a number of properties and methods in the Adobe Acrobat object, specifically related to Acrobat form fields within a PDF form document.

An Extension allows a system to extend the virtual functions available when the JavaScript Handler imitates utilities provided by Adobe Acrobat. The extension provides JavaScript that will replace an equivalent feature within Acrobat.

A callback is a method executed when an event is completed, or when called by a timer.

Relevant APIs:

- com.adobe.pdfjt.services.javascript.extension
- com.adobe.pdfjt.services.javascript.JavaScriptHandler

See also these other samples:

- core.digitalSignatures.JSSigInfoExtensionDemo

PDFToolkitXFASample.java

This sample program demonstrates how to work with the XFAService package. This Java package provides functions that allow clients to change, remove, or append XFA elements from PDF documents. XFA, or XML Forms Architecture, is a set of proprietary XML specifications for use with web forms. Since 2002 the XFA standard has been owned by Adobe Systems. XFA forms are saved internally in PDF files.

The program runs through a series of tests and then prints out the results at the command prompt or in the Eclipse console.

The tests are described below, with the response output. The tests apply to a pair of PDF input documents, XFACosArray.pdf and XFACosStream.pdf.

1. The program exports two standard XFA elements from this pair of PDF documents, the Datasets element and the Config Element. Then it attempts to export a custom element, “afe,” an arbitrary name for a custom element that the program cannot find:

   ```
   --- exportXFAElements()


   File = input\xfa\XFACosArray.pdf

   === Exported - datasets : http://www.xfa.org/schema/xfa-data  -  strip namespace version

   === Exported - config : http://www.xfa.org/schema/xci  -  strip namespace version

   === Element not found = afe : http://gibson.adobe.com/lessbasiclayout  -  dont' strip
   namespace version


   File = input\xfa\XFACosStream.pdf

   === Exported - datasets : http://www.xfa.org/schema/xfa-data  -  strip namespace version

   === Exported - config : http://www.xfa.org/schema/xci  -  strip namespace version

   === Element not found = afe : http://gibson.adobe.com/lessbasiclayout  -  dont' strip
   namespace version
   ```

2. Then, the program repeats the process of exporting the Datasets element from each of these two PDF files, and tries to import them back into the original PDF document. It also seeks to import the custom element into each PDF document. The import for the Dataset fails, but the custom import succeeds.

   ```
   ====================================

   --- importXFAElements()


   File = input\xfa\XFACosArray.pdf

   === Exported - datasets : http://www.xfa.org/schema/xfa-data  -  strip namespace version

   === Importing the DATASETS element failed as it should have.  The XFA content and the API
   parameters were mismatched.

   === Successfully imported our custom element.

   === Exported - hoser : com.sctv/show/greatwhitenorth  -  strip namespace version


   File = input\xfa\XFACosStream.pdf

   === Exported - datasets : http://www.xfa.org/schema/xfa-data  -  strip namespace version

   === Importing the DATASETS element succeeded when it maybe shouldn't have.  The XFA
   content and the API parameters were mismatched.

           If the PDF uses an XFA stream instead of an array then validation between the
   replacement XFA and the declared element name is not done.

   === Successfully imported our custom element.

   === Exported - hoser : com.sctv/show/greatwhitenorth  -  strip namespace version
   ```

3. The program exports the Datasets element from the XFA content for each of the PDF input documents, changes it, and then imports the Dataset back into the original file.

   ```
   ====================================

   --- modifyDatasets_data()
   ```

```
File = input\xfa\XFACosArray.pdf

=== Found and exported - datasets : http://www.xfa.org/schema/xfa-data  -  strip namespace
version

=== Imported the modified element - datasets : http://www.xfa.org/schema/xfa-data  -
strip namespace version

=== Exported modified element - datasets : http://www.xfa.org/schema/xfa-data  -  strip
namespace version


File = input\xfa\XFACosStream.pdf

=== Found and exported - datasets : http://www.xfa.org/schema/xfa-data  -  strip namespace
version

=== Imported the modified element - datasets : http://www.xfa.org/schema/xfa-data  -
strip namespace version

=== Exported modified element - datasets : http://www.xfa.org/schema/xfa-data  -  strip
namespace version
```

4. Finally, the program exports the complete XFA stream from each PDF document and shows how to chain your own XML processing into the process that extracts XFA from the PDF document. In this case, we chain in a simple custom processor that counts the number of XML elements in the document. The number of XWL elements found in each PDF document is displayed at the command prompt or in the Eclipse console.

```
====================================

--- chainExportFull()


File = input\xfa\XFACosArray.pdf

XFA Exported Successfully.  Found 309 XML elements.


File = input\xfa\XFACosStream.pdf

XFA Exported Successfully.  Found 1020 XML elements.
```

Note that it may be necessary to set the VM argument `org.xml.sax.driver` to point to the Simple API for XML (SAX) parser that you are using for XML. For example, if you are using the Apache Software Xerces Java parser for XML, you would need to edit the VM argument to look like this:

```
org.xml.sax.driver=org.apache.xerces.parsers.SAXParser.
```

This is necessary because of issues related to the SAX parser Jar file and the JVM SAX search algorithm.

Relevant APIs:

- com.adobe.pdfjt.services.xfa.XFAService
- com.adobe.pdfjt.services.xfa.XFAService.XFAElement

See also these other samples:

- core.forms.TextFieldContentReplacer

QueryForms.java

This sample program reviews the fields in a PDF form document and displays the field types and names, such as the Employee Name and the Social Security Number.

Relevant APIs:

- com.adobe.pdfjt.pdf.interactive.forms.PDFInteractiveForm
- com.adobe.pdfjt.pdf.interactive.forms.PDFField

See also these other samples:

- core.annotations.QueryAnnotations

TextFieldContentReplacer.java

This sample program finds a string of text in a text field in a PDF document and replaces that text with a new value. The text field can be either plain text or Rich Text, and the PDF document can be either a regular Acrobat Form or a static non-shell XFA Form. XFA, or XML Forms Architecture, is a set of proprietary XML specifications for use with web forms.

Relevant APIs:

- com.adobe.pdfjt.pdf.interactive.forms
- com.adobe.pdfjt.services.xfa.acroform

See also these other samples:

- core.forms.PDFToolkitXFASample

XFAFormExportDemo.java

XFAFormExportDemo demonstrates how to export data from the fields in a XFA form in a PDF document to an XML file. XFA, or XML Forms Architecture, is a set of proprietary XML specifications for use with web forms. Since 2002 the XFA standard has been owned by Adobe Systems. XFA forms are saved internally in PDF files.

This process could be used, for example, to archive these forms data values to a smaller file. The program takes the form data from a file called dynamicform.PDF and creates an export file:

```
output/XFAFormExportDemo/dynamicform.pdf.xml
```

Relevant APIs:

- com.adobe.pdfjt.pdf.document.PDFDocument
- com.adobe.pdfjt.services.xfa.XFAService

See also these other samples:

- core.forms. FDFFormImportDemo
- core.forms.XFDFFormImportDemo
- core.forms.XFAFormImport.Demo

XFAFormImportDemo.java

XFAFormInportDemo demonstrates how to import form data into an existing PDF file. The program takes the form data from a XML file called Check_Request_XFA_Data.xml and a PDF file called Check_Request_XFA.PDF. The program add these data to a blank PDF form to create an export file:

```
output/XFAFormImportDemo/Check_Request_XFA.pdf
```

Relevant APIs:

- com.adobe.pdfjt.pdf.document.PDFDocument
- com.adobe.pdfjt.services.xfa.XFAService

See also these other samples:

- core.forms.FDFFormExportDemo
- core.forms.XFDFFormExportDemo
- core.forms.XFAFormExportDemo

XFDFFormExportDemo.java

XFDFFormExportDemo demonstrates how to export data from Acrobat Form (Acroform) fields in an existing PDF file to an XML-based XFDF file. This process could be used, for example, to archive the values in the fields in a PDF form to a smaller file.

Relevant APIs:

- com.adobe.pdfjt.services.xfdf.XFDFService

See also these other samples:

- core.forms. XFDFFormImportDemo
- core.forms.FDFFormImportDemo
- core.forms.XFAFormImport.Demo

XFDFFormImportDemo.java

XFDFFormInportDemo demonstrates how to import form data into an existing PDF file. The program takes the form data from an XML-based XFDF file and inserts the data into the fields in a matched PDF form, creating a new PDF file.

Relevant APIs:

- com.adobe.pdfjt.services.xfdf.XFDFService

See also these other samples:

- core.forms.XFDFFormExportDemo
- core.forms.FDFFormExportDemo
- core.forms.XFAFormExportDemo

FormExportDemo.java

This sample program takes a PDF form input document, containing either an embedded AcroForm or XFA Form, and exports the data in the form fields to an output file for further processing. For an Acroform document, the output file is FDF or XFDF; for an XFA form, the output is saved to an XML file. Thus this sample can generate all three export file types, FDF, XFDF, or XML.

Suppose you have 700 PDF forms that were completed and saved by your customers. You could use this sample as a model to build a program that could extract all of the values entered in the form fields for these PDF documents and save them to a set of FDF files. Then, you could import the data from these files into a SQL Server database for processing and reporting.

AcroForm, or Acrobat Forms, is a standard format that supports interactive forms, with items like radio buttons and drop down lists. XFA, or XML Forms Architecture, is a set of proprietary XML specifications for use with web forms.

FDF refers to Forms Data Format, a file format described in the PDF specification. FDF files use the same low-level syntax as PDF, but the FDF format is only used to describe PDF annotations and form fields. FDF can be used in a variety of ways, such as submitting data from forms to a server application for processing, or to archive forms information. XFDF is an XML-based version of the FDF format.

When you run the sample program you can export the content to either FDF or XFDF if you are working with an AcroForm PDF document, with FDF being the default. You can add options that are available in the Command Line Interface to provide the path and file name of your own input file, or direct the program to save the output data to the file and path you specify. The options include:

```
-i input file path and file name
-o output file path and file name
-x output file and file name for XFDF for an AcroForm export
-h help
```

For example, if you wanted to import from a file called FruitForm_1_Acroform.pdf, you could enter a command like this:

```
$ java –cp
~/pdf-java-toolkit/libs/pdfjt.jar: ~/pdf-java-toolkit/libs/pdfjt-
support.jar:src:. pdfjt.webapi.FormExportDemo –i
input/workflows/FruitForm_1_AcroForm.pdf
```

By default, the input file is `input/fdf/Sample2.PDF`; the output file is `output/FormExport/FormExport.FDF`

Relevant APIs:

- com.adobe.pdfjt.pdf.document.PDFDocument;
- com.adobe.pdfjt.services.fdf.FDFDocument;
- com.adobe.pdfjt.services.fdf.FDFService;
- com.adobe.pdfjt.services.xfa.XFAService;

See also these other samples:

- core.forms.FDFFormExportDemo.java
- core.forms.XFDFFormExportDemo.java
- core.forms.XFAFormExportDemo.java

Converting Images

ColorSpaceConversionSample.java

This program shows how to pull a graphic image (XObject) from a PDF document into a buffered image in java.awt, and how to color convert that image to gray scale image. Finally, it shows how to save both exported images to a pair of PNG graphics files.

Relevant APIs:

- com.adobe.pdfjt.services.imageconversion
- java.awt.image

See also these other samples:

- core.imageConversion.ImageInsertionSample
- core.document.RasterizationSample

ImageExtractionSample.java

This sample shows how to extract an Image XObject from a PDF document and then convert the image to a JPG graphics file, using the Java Input/Output API. The program takes a photo of the Rocky Mountains from a PDF input document and exports and saves the image to a JPG output file.

Relevant APIs:

- com.adobe.pdfjt.services.imageconversion.BufferedImageWrapper
- com.adobe.pdfjt.pdf.graphics.xobject.PDFXObjectImageWithLocationMap

See also these other samples:

- core.imageConversion.TransformedImageExtractionSample
- core.imageConversion.ImageLocationDemo

ImageInsertionSample.java

This sample program shows how to take an image from a JPG graphic file and insert it into a blank page in a new PDF document. It demonstrates obtaining the image for import into the PDF document from a Buffered Image in java.awt.

Relevant APIs:

- com.adobe.pdfjt.services.imageconversion.ImageManager
- com.adobe.pdfjt.pdf.graphics.xobject.PDFXObjectImage

See also these other samples:

- core.document.XObjectImageWatermark
- core.document.copyImageSample

ImageLocationDemo.java

This sample program identifies the location of every image in a source PDF document, and displays the coordinates for each location to the command prompt or to an Eclipse console, like this:

```
location of the image is [0 0 500 546]
```

Then it extracts each image from the PDF document and saves it to an external JPG file.

Relevant APIs:

- com.adobe.pdfjt.pdf.graphics.xobject.PDFXObjectImageWithLocationMap

See also these other samples:

- core.imageConversion.ImageExtractionSample

TransformedImageExtractionSample.java

This sample shows how to extract transformed images to a series of graphics export files. The program uses the Current Transformation Matrix from the content stream to transform XObject images from the original PDF document. The exported JPG images that result match exactly, in size, shape, and orientation, the graphics as they appear in the original PDF document.

The Transformation Matrix is a graphics operator that maps the coordinates used within a PDF content stream to an output device. The matrix coordinates are specified as six numbers, usually in an array.

Relevant APIs:

- com.adobe.pdfjt.services.imageconversion
- com.adobe.pdfjt.pdf.graphics.xobject

See also these other samples:

- core.imageConversion.ImageExtractionSample

Manipulating PDF Documents

AddPageDecorations.java

This sample demonstrates how to add headers, footers and watermarks to the pages in a PDF document, drawing the placement and content for the headers and footers from an XML input file or series of XML files. The XML files you use must define how the headers and footers are formatted and placed on the PDF, or how to format and place a graphic to use for watermarks. The AddPageDecoratios.java program is found in the /webapi samples directory.

You can use the sample XML files provided with AddPageDecorations.java as a template for creating your own XML input files. The XML defines the font, size and color, the page margins for the target PDF document, the placement of the header or footer, and the header and footer text. You can also add a page number variable or a date variable (or both) to a header or footer to display the current page number and the current date. For watermarks, use the XML to specify the source of the image, either a graphic file (like a PNG) or a PDF document, as well as the placement and layout of the image on the PDF document pages.

The AddPageDecorations feature supports text in virtually any language you may want to use. You must provide the Unicode font set for that language to work with the PDF documents you seek to edit. Also, you can add headers and footers or watermarks to rotated pages in a PDF document, as you can with Adobe Acrobat. That is, if you provide a PDF document with a page that is rotated on its side, 90 degrees, the sample program assumes that the page is supposed to be rotated as shown. It will add a header to the top of this rotated page and a footer on the bottom, just as it would to every other page in the document, or display a watermark in a way that is consistent with the other pages.

If you want to add a header or footer or both as well as a watermark or watermarks to the pages in a PDF document, you will need to create at least two separate input XML files. Use at least one XML file for headers and footers, and at least one for watermarks.

You can create an XML file using Adobe Acrobat, or create one manually using an editor. Note that for Adobe Acrobat to read XML files properly the files must adhere strictly to XML formatting rules. If you build an XML file using an editor and format it with new-line whitespaces, you might not be able to work with the same file in Acrobat.

Creating an XML File using Adobe Acrobat

Open Adobe Acrobat version 10 or 11.

1 Choose Tools > Pages > Header & Footer > Add Header & Footer, or

Choose Tools > Pages > Watermark > Add Watermark.

If you can't find the Tools option, click View/Tool Sets/Default Tools.

2 Build your headers and footers or watermark using the user interface provided.

3 To save your changes, and create an XML export file, click Save Settings... . Enter a name that will be assigned to this XML file and click OK. Note that if you have a PDF document open in Acrobat, you can save your header and footer settings separately, to an external XML file. You don't have to save the PDF document as well.

4 Adobe Acrobat will save the XML file in the appropriate Preferences folder:

Acrobat 10

Windows:

AppData\Roaming\Adobe\Acrobat\10.0\Preferences\HeaderFooter

AppData\Roaming\Adobe\Acrobat\10.0\Preferences\Watermark

Mac:

/Users/USER/Library/Preferences/Adobe/Acrobat/10.0/HeaderFooter

/Users/USER/Library/Preferences/Adobe/Acrobat/10.0/Watermark

Acrobat 11

Windows:

AppData\Roaming\Adobe\Acrobat\11.0\Preferences\HeaderFooter

AppData\Roaming\Adobe\Acrobat\11.0\Preferences\Watermark

Mac:

/Users/USER/Library/Preferences/Adobe/Acrobat/11.0/HeaderFooter

/Users/USER/Library/Preferences/Adobe/Acrobat/11.0/Watermark

Creating an XML File Manually: General Features

This sample XML file adds a footer with the date “May 17, 2014,” to the bottom middle of the page, and a centered header with the text “ABC Software Systems”:

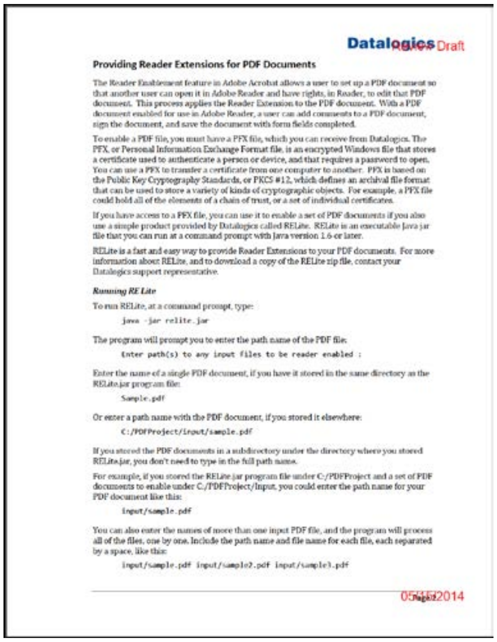
```
<?xml version = "1.0" encoding = "UTF-8" ?>

<HeaderFooterSettings version = "8.0">

<Font type="TrueType" size="12.0" name="Arial"/><Color r="0.0" b="1.0"
g="0.5"/>

<Margin right="72.0" left="72.0" bottom="36.0" top="36.0"/>

<PageRange start="-1"end="-1" even="1" odd="1"/>

<Header><Left></Left><Center>ABC Software
Systems/</Center><Right></Right></Header>

<Footer><Left></Left><Center>May 17,
2014</Center><Right></Right></Footer>

</HeaderFooterSettings>
```

You can use the Shrink to Fit feature, found in Adobe Acrobat, if you want to make more space within the page boundaries for your headers and footers, or for a watermark. This will make sure that your headers and footers or watermarks do not overwrite text on the pages of your PDF document. The Shrink to Fit feature will reduce the size of each page in the PDF document, leaving more space at the top and the bottom of each page.

Add “Shrink=”1” to a special appearance tag to the XML file:

```
<Appearance fixedprint="0" shrink="1"/>
```

The example on the left shows a PDF page with a header and footer added. Note that the new header and footer overlap existing text on the page. So we use Shrink To Fit in the next example, reducing the page size so that the new header and footer appear correctly:



Datalogics Review Draft

Providing Reader Extensions for PDF Documents

The Reader Enablement feature in Adobe Acrobat allows a user to set up a PDF document so that another user can open it in Adobe Reader and have rights, in Reader, to edit that PDF document. This process applies the Reader Extension to the PDF document. With a PDF document enabled for use in Adobe Reader, a user can add comments to a PDF document, sign the document, and save the document with form fields completed.

To enable a PDF file, you must have a PFX file, which you can receive from Datalogics. The PFX, or Personal Information Exchange Format file, is an encrypted Windows file that stores a certificate used to authenticate a person or device, and that requires a password to open. You can use a PFX to transfer a certificate from one computer to another. PFX is based on the Public Key Cryptography Standards, or PKCS #12, which defines an archival file format that can be used to store a variety of kinds of cryptographic objects. For example, a PFX file could hold all of the elements of a chain of trust, or a set of individual certificates.

If you have access to a PFX file, you can use it to enable a set of PDF documents if you also use a simple product provided by Datalogics called RELite. RELite is an executable Java jar file that you can run at a command prompt with Java version 1.6 or later.

RELite is a fast and easy way to provide Reader Extensions to your PDF documents. For more information about RELite, and to download a copy of the RELite zip file, contact your Datalogics support representative.

Running RE Lite

To run RELite, at a command prompt, type:

```
java -jar relite.jar
```

The program will prompt you to enter the path name of the PDF file:

```
Enter path(s) to any input files to be reader enabled :
```

Enter the name of a single PDF document, if you have it stored in the same directory as the RELite.jar program file:

```
Sample.pdf
```

Or enter a path name with the PDF document, if you stored it elsewhere:

```
C:/PDFProject/input/sample.pdf
```

If you stored the PDF documents in a subdirectory under the directory where you stored RELite.jar, you don't need to type in the full path name.

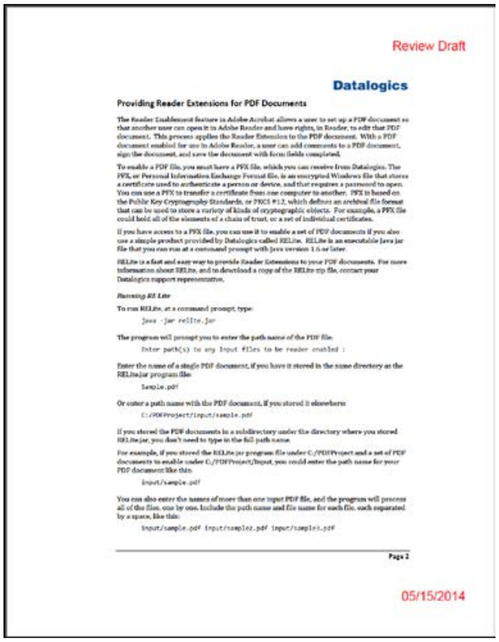
For example, if you stored the RELite.jar program file under C:/PDFProject and a set of PDF documents to enable under C:/PDFProject/Input, you could enter the path name for your PDF document like this:

```
input/sample.pdf
```

You can also enter the names of more than one input PDF file, and the program will process all of the files, one by one. Include the path name and file name for each file, each separated by a space, like this:

```
input/sample.pdf input/sample2.pdf input/sample3.pdf
```

05/15/2014 Page 2



Review Draft

Datalogics

Providing Reader Extensions for PDF Documents

The Reader Enablement feature in Adobe Acrobat allows a user to set up a PDF document so that another user can open it in Adobe Reader and have rights, in Reader, to edit that PDF document. This process applies the Reader Extension to the PDF document. With a PDF document enabled for use in Adobe Reader, a user can add comments to a PDF document, sign the document, and save the document with form fields completed.

To enable a PDF file, you must have a PFX file, which you can receive from Datalogics. The PFX, or Personal Information Exchange Format file, is an encrypted Windows file that stores a certificate used to authenticate a person or device, and that requires a password to open. You can use a PFX to transfer a certificate from one computer to another. PFX is based on the Public Key Cryptography Standards, or PKCS #12, which defines an archival file format that can be used to store a variety of kinds of cryptographic objects. For example, a PFX file could hold all of the elements of a chain of trust, or a set of individual certificates.

If you have access to a PFX file, you can use it to enable a set of PDF documents if you also use a simple product provided by Datalogics called RELite. RELite is an executable Java jar file that you can run at a command prompt with Java version 1.6 or later.

RELite is a fast and easy way to provide Reader Extensions to your PDF documents. For more information about RELite, and to download a copy of the RELite zip file, contact your Datalogics support representative.

Running RE Lite

To run RELite, at a command prompt, type:

java -jar relite.jar

The program will prompt you to enter the path name of the PDF file:

Enter path(s) to any input files to be reader enabled :

Enter the name of a single PDF document, if you have it stored in the same directory as the RELite.jar program file:

Sample.pdf

Or enter a path name with the PDF document, if you stored it elsewhere:

C:/PDFProject/input/sample.pdf

If you stored the PDF documents in a subdirectory under the directory where you stored RELite.jar, you don't need to type in the full path name.

For example, if you stored the RELite.jar program file under C:/PDFProject and a set of PDF documents to enable under C:/PDFProject/Input, you could enter the path name for your PDF document like this:

input/sample.pdf

You can also enter the names of more than one input PDF file, and the program will process all of the files, one by one. Include the path name and file name for each file, each separated by a space, like this:

input/sample.pdf input/sample2.pdf input/sample3.pdf

Page 2

05/15/2014

Disregard the “fixedprint=’0’” statement that appears in the Appearance tag.

You can also use the underline command to underline all of the text in every header and footer in a PDF document. If you want to underline text, add the “Underline=true” tag to the font statement.

If you create a footer with the date and company name, the underlined result looks like this:

May 17, 2014   ABC Software, Inc.

Both tags are optional.

The edited XML file, with the shrink to fit and underline tags, would look like this:

```
<?xml version = "1.0" encoding = "UTF-8" ?>

<HeaderFooterSettings version = "8.0">

<Font type="TrueType" size="12.0" name="Arial" underline="true"/><Color
r="0.0" b="1.0" g="0.5"/>

<Margin right="72.0" left="72.0" bottom="36.0" top="36.0"/>

<Appearance fixedprint="0" shrink="1" />

<PageRange start="-1"end="-1" even="1" odd="1"/>

<Header><Left></Left><Center>ABC Software
Systems/</Center><Right></Right></Header>

<Footer><Left></Left><Center>May 17,
2014</Center><Right></Right></Footer>

</HeaderFooterSettings>
```

Creating an XML File Manually: Date Formats

In this sample, the footer includes a date variable in the middle of the page. So if you generate the PDF document on May 17, “5/17/2014” appears on the bottom of the page. If you were to generate the document the next day, “5/18/2014” would appear instead:

```
<?xml version = "1.0" encoding = "UTF-8" ?>

<HeaderFooterSettings version = "8.0">

<Font type="TrueType" size="12.0" name="Arial"/><Color r="0.0" b="1.0"
g="0.5"/>

<Margin right="72.0" left="72.0" bottom="36.0" top="36.0"/>

<PageRange start="-1"end="-1" even="1" odd="1"/>

<Header><Left></Left><Center>ABC Software
Systems/</Center><Right></Right></Header>
```

```
<Footer><Left></Left><Center><Date><Month format=”2”/>/<Day
format=”2”/>/<Year format=”4”/></Center><Right></Right></Footer>

</HeaderFooterSettings>
```

The format numbers for the month, day, and year, define the number of digits for each value. So if day=2, month=2, and year=4, the date will be presented as MM/DD/YYYY, as shown in the example above.

You can format the date a variety of ways, such as "day, month, year" or "month, day, year," and separate the values with slashes (/), periods (.), or dashes. You can include 1 or 2 digits for the day or month and 2 or 4 digits for the year; you can also enter zero ("0") for a value to leave the value out.

Here are a few examples of date formats you could add to an XML input file:

```
<Date><Month format="2"/>/<Year format="2"/><Day format="0"/></Date>

mm/yy


<Date><Day format="1"/>/<Month format="1"/>/<Year format="4"/></Date>

d/m/yyyy


<Date><Month format="2"/>.<Year format="4"/><Day format="0"/></Date>

mm.yyyy


<Date><Day format="2"/>.<Month format="2"/>.<Year format="4"/></Date>

dd.mm.yyyy
```

Creating an XML File Manually: Working with Page Numbers

In the sample below, the footer includes both a date variable and a page number. The format provided will be "Page x of n," so that that total number of pages in the document appears with the current page number, as in "Page 5 of 22."

```
<?xml version = "1.0" encoding = "UTF-8" ?>

<HeaderFooterSettings version = "8.0">

<Font type="TrueType" size="12.0" name="Arial"/><Color r="0.0" b="1.0"
g="0.5"/>

<Margin right="72.0" left="72.0" bottom="36.0" top="36.0"/>

<PageRange start="-1"end="-1" even="1" odd="1"/>

<Header><Left></Left><Center>ABC Software
Systems/</Center><Right></Right></Header>
```

```
<Footer><Left></Left><Center><Date><Month format=”2”/>/<Day
format=”2”/>/<Year format=”4”/>/<Page offset = "0"><PageIndex
format="1"/>of<PageTotalNum format="n"/></Page>

</Center><Right></Right></Footer>

</HeaderFooterSettings>
```

The Page offset defines the number to use to start the page numbering.

Normally the Page offset value will equal “0” because you will want to start the page numbering with “Page 1.”

If you wanted to start your numbering with “Page 10,” you would set the Page offset value equal to 9 (Page offset=”9”).

Disregard the PageIndexFormat attribute. It will always = 1.

PageTotalNum is the number of pages that you would display in the footer or header with a page numbering statement, as in “Page 1 of 45.” It is not necessarily the total number of pages in the document; you might want to set up your XML file so that the page numbers start on the fifth page of the PDF document, skipping a table of contents and a cover sheet.

But you can disregard the PageTotalNum value. This will always be set equal to “n”:

```
<PageTotalNum format="n"/>
```

You can control the page in the document where the page numbering starts, and the number of pages in the document that display page numbers. Suppose you have a PDF document that is 50 pages long, but that document includes a cover page, a page for copyright statements, and three pages of a table of contents. You want to start the page numbering after those pages, and continue the page numbering until the end. So the first page number value provided would be “Page 1 of 45” on the sixth page of the document.

To define these page numbering values, you need to edit the PageRange values at the beginning of the XML file. The PageRange settings label every page in the document.

These are the default PageRange values:

```
<PageRange start="-1"end="-1" even="1" odd="1"/>
```

You also need to edit the Page offset value in the page number statement in the header or footer:

```
<Page offset = "0"><PageIndex format="1"/>of<PageTotalNum
format="n"/></Page>
```

The PageRange start value is used to determine the page in the document where the page numbering should begin. The above statement means that that process will include both odd and even pages (even=”1” odd=”1”) and every page within the document (start=”-1” end “-1”).

But in our example we have a PDF document with a cover page, a copyright sheet, and a table of contents. You want to start numbering pages after those pages, so you would edit the PageRange value like this:

```
<PageRange start="5"end="-1" even="1" odd="1"/>
```

With the Page offset defined as equal to 0, and the PageRange start value set equal to 5, the page numbering will skip the first five pages of the PDF document and start with "Page 1" (or "Page 1 of n") on the sixth page of the document.

Also, note that this will cause the PDF to include page numbers on every page until the last page of the document. If you wanted to print page numbers for only ten pages and then stop, you could set the PageRange end value as well, like this:

```
<PageRange start="5"end="14" even="1" odd="1"/>
```

The value n in the PageTotalNum format is determined by adding the Page offset and the total number of pages selected in the document to label with a page number.

n = (total # of pages in document selected for page #s) + Page offset

Suppose you have a 50 page document. You want to include a page number on every page of that document, so you are starting with the first page, and the first page number will be page 1. So the Page offset =0, and that means that the n value will be:

n = 50

On the bottom of the first page, you will see a label, "Page 1 of 50."

Now suppose you want to skip the first five pages of the document. The document has a cover page, a second page with copyright information, and three pages for the table of contents, and you want to start the page numbering on the sixth page. Thus the document has 45 pages to label with page numbers. So the total number of pages in the document is 45:

n = 45

Now, one more step.

Suppose you are working with a series of four PDF documents. These are scanned copies of a scientific journal, and the original print documents were published as four quarterly issues, but were intended to be bound (for storage on a library shelf) as a single volume.

Because these four issues represent a single volume, the pages are numbered consecutively, from one issue to the next. The first issue, or document, has pages 1 through 20. The second issue starts at page 21 and continues to page 65.

You are working with this second PDF document. And this second document, while it is numbered from page 21 to page 65, also includes a cover sheet, copyright page, and three pages of a table of contents.

So, in this case, you want to start numbering the pages on the sixth page of this 50 page document, as described above. The page number labeling includes the last 45 pages of the document. But you also want to start the numbering with "Page 21," not "Page 1."

That means, then, that you need to set the PageRange start value = 5, to skip the first five pages of the document, and set the Page offset = 20. As a result, the n value will be calculated like this:

n = 45 + 20 = 65

That is, page 6 would be the first page where a page number would appear, in a 45 page document, but you are rolling the page numbering forward 20 pages. So the first page number value shown, on page 6, would be:

"Page 21 of 65"

Creating an XML File Manually: Page Number Formats

Adobe Acrobat supports these five page number formats:

```
<Page offset = "0"><PageIndex format="1"/></Page>
```

1

```
<Page offset = "0"><PageIndex format="1"/>/<PageTotalNum
format="n"/></Page>
```

1/15

```
<Page offset = "0"><PageIndex format="1"/>of<PageTotalNum
format="n"/></Page>
```

1 of 15

```
<Page offset = "0">Page<PageIndex format="1"/></Page>
```

Page 1

```
<Page offset = "0">Page<PageIndex format="1"/>of<PageTotalNum
format="n"/></Page>
```

Page 1 of 15

Creating an XML File Manually: Adding a WaterMark

This sample XML file adds a watermark graphic to each page:

```
<?xml version = "1.0" encoding = "UTF-8" ?>

<WatermarkSettings version = "8.0">

<SourceFile name="Test.pdf" type="" page="4" path="/Macintosh
HD/Users/tsmith/pdf-java-toolkit/samples/input/decorations/"/>

<Scale value="-0.5"/><Rotation value="0"/><Opacity
value="1.0"/><Location ontop="1"/>

<Color r="0.0" b="0.0" g="0.0"/>

<Alignment vertalign="1" horizalign="1" vertvalue="0.0" horizvalue="0.0"
unit="1"/>

<Appearance onscreen="1" onprint="1" fixedprint="0"/>

<PageRange end="-1" start="-1" even="1" odd="1"/>

</WatermarkSettings>
```

Importing a Graphic for Use as a Watermark

You can draw a graphic from a page in a PDF document:

```
<SourceFile name="Test.pdf" type="" page="4" path="/Macintosh
HD/Users/tsmith/pdf-java-toolkit/samples/input/decorations/"/>
```

Or provide a graphic file:

```
<SourceFile name="Test.jpg" type="" page="0" path="/Macintosh
HD/Users/tsmith/pdf-java-toolkit/samples/input/decorations/"/>
```

In either case you need to provide the path with the file name so that the AddPageDecorations process can find it.

You also need to provide the page number where the graphic is found in the input file. If the file has only one page, the page number will be zero. This is the default value:

```
<SourceFile name="Test.jpg" type="" page="0" path="/Macintosh
HD/Users/tsmith/pdf-java-toolkit/samples/input/"/>
```

If you want to draw a graphic from another PDF document that has multiple pages, you need to define the page number where that watermark graphic appears. If the image is on the fifth page of the PDF input document , you would set the page value equal to 4:

```
<SourceFile name="Test.pdf" type="" page="4" path="/Macintosh
HD/Users/tsmith/pdf-java-toolkit/samples/input/decorations/"/>
```

Disregard the “type” value. The type of graphic file used is defined by the file name, as in test.jpg or test.pdf.

Using Text as a Watermark

If you decide to use text as your watermark, rather than an image, you need to define the font type, size, and color, and provide the text:

```
<Font name="Arial" type="TrueType" size="24.0"/>Confidential Draft<Color
r="0.0" b="0.0" g="0.0"/>
```

The font color defaults to black:

```
<Color r="0.0" b="0.0" g="0.0"/>
```

The color only applies if you are using text for your watermark. If you are using a graphic for the watermark, disregard the color setting.

To define a font color, enter the numbers for red, blue, and green from the RGB color space table, between 0 and 255 for each of these three colors. To create a light reddish brown, for example, the values would be:

Red = 200, Blue = 45, Green = 100

But in the XML file, the number must be between 0 and 1, so divide each number by 255 to convert it into a ratio. The Red number, 200, divided by 255, equals .784:

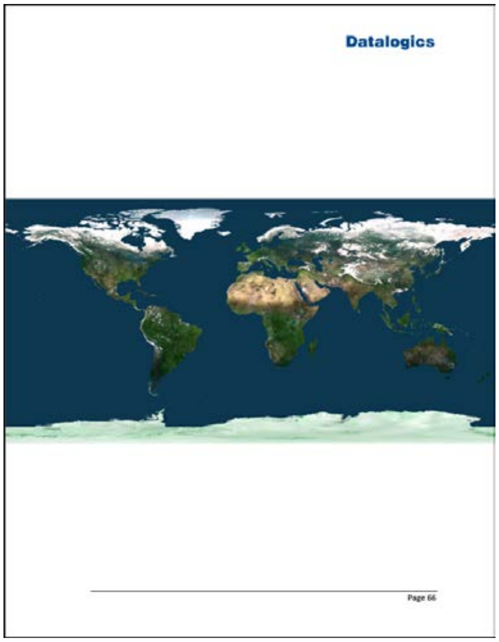
```
<Color r="0.784317" g="0.392151" b="0.176468"/>
```

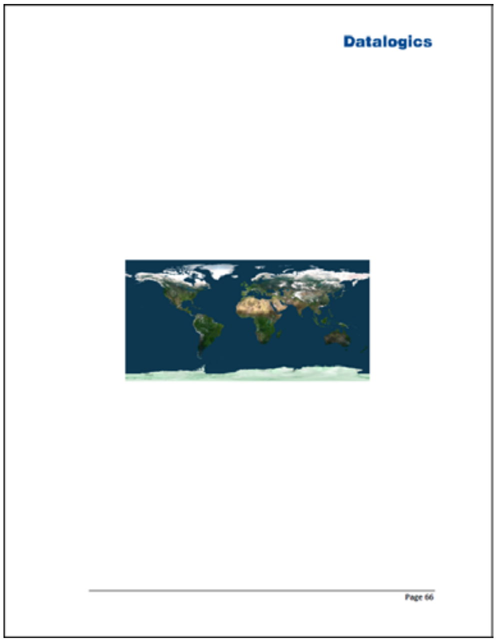
Defining the Scale of the Graphic

You also need to define the scale and rotation of the graphic on each page:

```
<Scale value="-0.5"/><Rotation value="0"/><Opacity
value="1.0"/><Location ontop="1"/>
```

The Scale defines the size of the image used as a watermark as a ratio of its original size. If Scale=1, the watermark as it appears will be the same size as the original graphic, as shown on the left. If the Scale =**-0.5**, the image will appear at half the size of the original:





If you want to use an image that fills most or all of the page in the PDF document, we recommend that you start with a JPG or PNG or similar graphic file that is large enough and has a large enough resolution to serve. Then, use the scaling function to reduce it to fit. If you are starting with a small graphic image, and you want it to appear considerably larger as a watermark, you might want to try to replace it with a compatible image file that is larger, or use an editing tool such as Adobe PhotoShop to increase the size and resolution. Then, try using the imported file for the watermark with AddPageDecoration.java.

Opacity, Rotation, and Location

Next, define the level of opacity, the rotation, and location of the graphic on the page:

```
<Scale value="-0.5"/><Rotation value="0"/><Opacity Value="1.0"/>
<Location ontop="1"/>
```

The rotation allows you to tip the watermark graphic at an angle, to the right or left, such as 45 degrees. The default value is zero. Enter the Rotation value as a number between 0 and 360. You can also use a negative number to rotate to the left:

```
<Scale value="-0.5"/><Rotation value="45”/><Opacity
value="1.0"/><Location ontop="1"/>
```

The location describes the relationship of the watermark graphic to the text on each page. If the location is on top:

```
 <Scale value="-0.5"/><Rotation value="0"/><Opacity Value="1.0"/>
<Location ontop="1"/>
```

It means that the watermark will cover over text on the page. Set the value equal to zero if you want the graphic to appear behind the page.

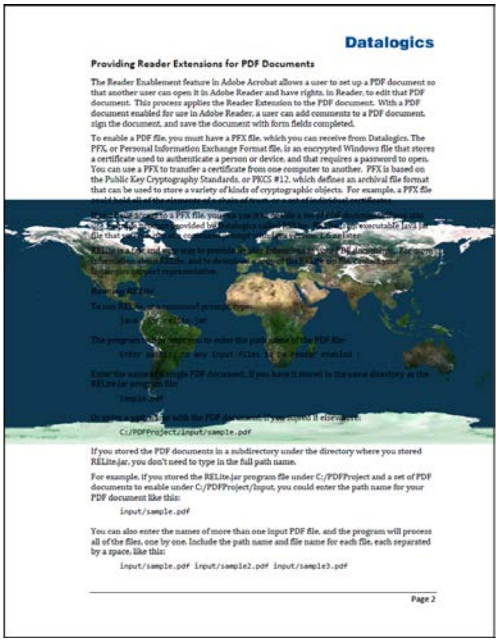
The Opacity setting defines how well you will be able to see through the watermark. The default value equals 1, or the graphic is completely opaque.

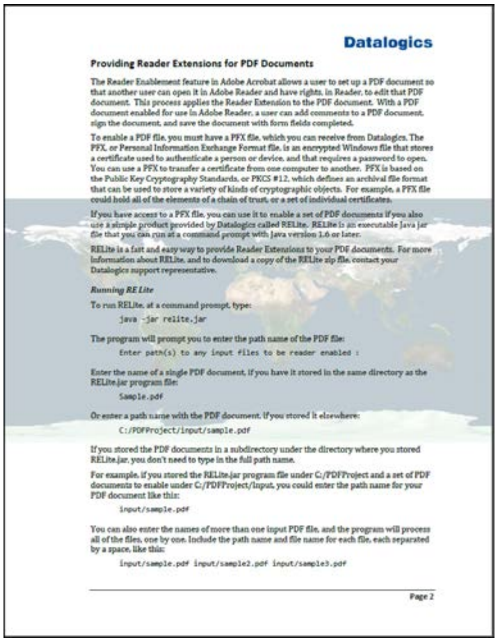
Let's set up the watermark so that it rests behind the text on each page (location on top = 0) and the Opacity is 25% of the original, so you can see through the image.

The XML code would look like this:

```
<Scale value="-0.5"/><Rotation value="0"/><Opacity Value=".25"/>
<Location ontop="0"/>
```

The page on the left shows the watermark so that it is not on top of the page. The page on the right also features the watermark with an opacity of 25% (.25):





Datalogics

Providing Reader Extensions for PDF Documents

The Reader Enablement feature in Adobe Acrobat allows a user to set up a PDF document so that another user can open it in Adobe Reader and have rights, in Reader, to edit that PDF document. This process applies the Reader Extension to the PDF document. With a PDF document enabled for use in Adobe Reader, a user can add comments to a PDF document, sign the document, and save the document with form fields completed.

To enable a PDF file, you must have a PFX file, which you can receive from Datalogics. The PFX, or Personal Information Exchange Format file, is an encrypted Windows file that stores a certificate used to authenticate a person or device, and that requires a password to open. You can use a PFX to transfer a certificate from one computer to another. PFX is based on the Public Key Cryptography Standards, or PKCS #12, which defines an archival file format that can be used to store a variety of kinds of cryptographic objects. For example, a PFX file could hold all of the elements of a chain of trust, or a set of individual certificates.

If you have access to a PFX file, you can use it to enable a set of PDF documents if you also use a simple product provided by Datalogics called RELite. RELite is an executable Java jar file that you can run at a command prompt with Java version 1.6 or later.

RELite is a fast and easy way to provide Reader Extensions to your PDF documents. For more information about RELite, and to download a copy of the RELite zip file, contact your Datalogics support representative.

Running RE Lite

To run RELite, at a command prompt, type:

```
java -jar relite.jar
```

The program will prompt you to enter the path name of the PDF file:

```
Enter path(s) to any input files to be reader enabled :
```

Enter the name of a single PDF document, if you have it stored in the same directory as the RELite.jar program file:

```
Sample.pdf
```

Or enter a path name with the PDF document, if you stored it elsewhere:

```
C:/PDFProject/input/sample.pdf
```

If you stored the PDF documents in a subdirectory under the directory where you stored RELite.jar, you don't need to type in the full path name.

For example, if you stored the RELite.jar program file under C:/PDFProject and a set of PDF documents to enable under C:/PDFProject/Input, you could enter the path name for your PDF document like this:

```
input/sample.pdf
```

You can also enter the names of more than one input PDF file, and the program will process all of the files, one by one. Include the path name and file name for each file, each separated by a space, like this:

```
input/sample.pdf input/sample2.pdf input/sample3.pdf
```

Page 2

Alignment

The Alignment values determine the placement of the watermark on each page:

```
<Alignment vertalign="1" horizalign="1" vertvalue="0.0" horizvalue="0.0"
unit="1"/>
```

The vertalign and horizalign values range from zero to 2 to describe the position on the page relative to the page center.

Center vertical, center horizontal	1/1
Top vertical, left horizontal	0/0
Bottom vertical, right horizontal	2/2

You can also use fractional values to adjust the placement of the watermark, such as .75/1.25 to place a watermark slightly left and below the center of the page.

The vertvalue and horizvalue parameters describe moving the watermark left or right or up or down relative to the alignment you define using the vertalign and horizalign values. The vertvalue and horizvalue can be negative numbers, for moving down and to the left. For example, if you set the alignment to the center of the page, using 1 for vertalign and 1 for horizalign, you can adjust the placement of the graphic slightly from that position using the vertvalue and horizvalue, entering points for each of these parameters, like this:

```
<Alignment vertalign="1" horizalign="1" vertvalue="0.3" horizvalue="0.3"
unit="1"/>
```

The alignment unit defaults to inches:

Inches	Unit=1
Percent	Unit=10
Centimeters	Unit=3
Millimeters	Unit=2
Picas	Unit=4
Points	Unit=0

All of the values are actually shown in points. For a percent, you would enter a decimal to show the percentage of the size of the page, as in .25 for 25%.

You can set up the watermark to appear on screen or on page when the document is printed, or both. The default value is 1 for each setting. If onscreen and onprint are set equal to 1, the watermark will appear on the screen when the PDF document is opened, and on the page when printed.

```
<Appearance onscreen="1" onprint="1" fixedprint="0"/>
```

Set either value equal to zero if you want to hide the watermark on screen, or not show it on the pages in the document when printed.

Disregard the "fixedprint='0'" statement that appears in the Appearance tag.

Selecting the Pages for the Watermark to Appear in the PDF Document

Finally, the page range works the same way for watermarks as it does for headers and footers.

```
<PageRange end="-1" start="-1" even="1" odd="1"/>
```

The PageRange start value is used to determine the page in the document where the watermarks should begin. The default values in the above statement means that that process will include both odd and even pages (even="1" odd="1") and every page within the document (start="-1" end "-1").

If you only want to add a watermark to odd pages, set odd=1 and even=0.

Suppose we have a PDF document with a cover page, a copyright sheet, and a table of contents. You want to start adding the watermark after those pages, so you would edit the PageRange value like this:

```
<PageRange start="5"end="-1" even="1" odd="1"/>
```

With the PageRange start value set equal to 5, the watermark would not appear on the first five pages of the PDF. Also, note that this will cause the PDF to add a watermark on every page until the last page of the document. If you wanted to include the watermark for only ten pages and then stop, you could set the PageRange end value as well, like this:

```
<PageRange start="5"end="14" even="1" odd="1"/>
```

Building the PDF documents

The program adds the header and footer content and watermarks using Form XObjects. Three XObjects are provided for footers and three for headers: left side, center, and right side. The content is added to the Piece Information (PieceInfo) dictionary within the PDF document page.

You can use a series of XML files with AddPageDecorations.java to update PDF documents, and you can apply changes to specific pages within a PDF document. For example, you can have one XML file that only applies certain headers and footers to even pages, and the other XML file, to odd pages. Or you could have one XML file for headers and footers on even pages, one for headers and footers on odd pages, and a third that places a watermark on every odd page. The input files are stored in:

```
input/decorations
```

Four sample XML files are provided, Footers.xml, HeaderFooterBasic.xml, AllEvenPages.xml, and AllOddPages.xml.

The output files are copied to:

```
output/AddPageDecorations
```

Relevant APIs:

- com.adobe.pdfjt.core.fontset.PDFFontSet;
- com.adobe.pdfjt.pdf.document.PDFDocument;
- com.adobe.pdfjt.services.manipulations.PageDecorator;

See also these other samples:

- core.document.AddWatermarksAndBackground.java

AddBatesNumber.java

This sample program demonstrates how to add a Bates Number stamp to each page of a given PDF document. A Bates Number is a unique serial number attached to every page of an electronic file. Bates Numbers are commonly used to identify legal, medical, and business documents, and often feature a date and time stamp.

The program applies the Bates Number content to the PDF file in an XObject form specifically labeled for that purpose. That means that Adobe Acrobat can recognize the Object as containing a Bates Number, and so a user can edit or remove this Object using Adobe Acrobat.

Relevant APIs:

- com.adobe.pdfjt.pdf.graphics.xobject.PDFXObjectForm
- com.adobe.pdfjt.pdf.page.PDFPage
- com.adobe.pdfjt.services.rcg.RichTextContentGenerator;

See also these other samples:

- core.document.XObjectRichTextWatermark

ApplyMultipleXobjects.java

This sample demonstrates how to generate an array of XObjects which share some common resources. This serves to limit the redundancy in the resulting PDF output document, and thus optimizes the document, making it smaller. For example, if the XObjects all share the same set of fonts, each font only needs to be added to the array one time.

The program adds page numbers (as XObjects) to every page of the PDF output document.

Relevant APIs:

- com.adobe.pdfjt.services.xobjhandler
- com.adobe.pdfjt.pdf.graphics.xobject.PDFXObjectForm

See also these other samples:

- core.manipulation.AddBatesNumber

ExtractandInsertPages.java

This sample demonstrates how to extract pages from one PDF file and insert them into another. The program pulls two pages from one source PDF document and one page from a second PDF document and saves them to a third PDF document.

Relevant APIs:

- com.adobe.pdfjt.services.manipulations.PMMService
- com.adobe.pdfjt.services.manipulations.PMMOptions

See also these other samples:

- core.manipulation.SubstitutePages

PDFWatermark.java

PDFWatermark shows how to take a page from a PDF document and use it as a watermark stamp, adding it to the page of a second PDF document. This sample simply appends a Watermark Form XObject to the content of each page, rather than using a Watermark annotation.

The program draws an image from one PDF document and applies it as a watermark image to the first page of another, and then saves the result to the output file Watermarked.pdf.

Relevant APIs:

- com.adobe.pdfjt.services.xobjhandler

See also these other samples:

- core.document.AddWatermarksAndBackground
- core.document.XObjectImageWatermark
- core.document.XobjectRichTextWatermark

SubstitutePages.java

This sample deletes all of the pages in a document and adds the pages from another document, while preserving the Forms (AcroForm) dictionary.

This is useful for updating PDFs that contain XFA forms. XFA, or XML Forms Architecture, is a set of proprietary XML specifications for use with web forms. Since 2002 the XFA standard has been owned by Adobe Systems. XFA forms are saved internally in PDF files.

XFA content, stored in the AcroForm dictionary, cannot be rendered by many third-party viewers. These viewers render the "traditional" PDF content contained in a document. This sample can be used to update the traditional content with a flattened version of the XFA content, if available. PDFJT can only flatten Static XFA content, but it is also possible to use flattened Dynamic XFA content obtained from an external source. This sample demonstrates the latter. In this sample, dynamicform.pdf is a Dynamic XFA form. flattened.pdf is a flattened version of the same form. We delete the pages out of dynamicform.pdf, and insert the pages from flattened.pdf. Along the way, we make sure to preserve the XFA data and to reset the NeedsRendering flag if it exists.

Relevant APIs:

- com.adobe.pdfjt.services.manipulations.PMMService
- com.adobe.pdfjt.pdf.interactive.forms.PDFInteractiveForm

See also these other samples:

- core.document.TransformPage
- core.forms.QueryForms

Working with PDF Archive Documents (PDF/A)

PDF/A is a subset of the PDF format (PDF 1.4) designed to be used for documents that need to be archived and stored for long periods. PDF/A-1 was based on PDF version 1.4. Later releases of PDF/A have added features introduced in updates to the PDF format. The most notable difference between ordinary PDF and PDF/A documents is that PDF/A documents must embed the fonts used in the content of the PDF document. For ordinary PDF documents, embedding fonts is merely a best practice. PDF viewers generally substitute a system font if a font is not embedded in a PDF document. But a document that must rely on certain fonts being available on a local machine is not suited for archival use.

ConvertPDFA1Document.java

ConvertPDFA1Document shows you how to convert a PDF document into a PDF Archive document, PDF/A-1b, currently the most commonly used version of PDF/A. It was introduced in October of 2005. The sample program uses PDFAService, and demonstrates how to use the PDFAConversionHandler.

Relevant APIs:

- com.adobe.pdfjt.services.pdfa
- com.adobe.pdfjt.pdf.graphics
- com.adobe.pdfjt.pdf.interactive

See also these other samples:

- core.pdfa.ConvertPDFA2Document
- core.pdfa.ValidateDocument

ConvertPDFA2Document.java

ConvertPDFA2Document shows how the APIs in PDF Java Toolkit can convert a PDF document into a PDF Archive document, PDFA/2-b. PDFA/2-b is a very recent standard, introduced in June of 2011. Note that this program, when run, prompts a user to enter the path and file name for the PDF document to be converted.

Relevant APIs:

- com.adobe.pdfjt.services.pdfa2

See also these other samples:

- core.pdfa.ConvertPDFA1Document
- core.pdfa.PDFA2DocumentValidation

PDFA2DocumentValidation.java

This sample demonstrates how to set up the PDF/A service to validate whether a PDF file is PDF/A-2b compliant or not. The program will review two PDF files, one valid, the other invalid, and responds with comments at the command line or in the Eclipse console for each one. This sample is closely related to PDFA2 Validation Summary; for this sample the comments provide a detailed discussion of any errors.

Relevant APIs:

- com.adobe.pdfjt.services.pdfa2.PDFA2ValidationHandler
- com.adobe.pdfjt.services.pdfa2.EmbeddedFilePDFA2ValidationHandler

See also these other samples:

- core.pdfa.PDFA2ValidationSummary
- core.pdfa.ValidateDocument

PDFA2ValidationSummary.java

This sample demonstrates how to set up the PDF/A service to provide summary statements that describe the validation errors that are found while attempting to validate a PDF file as PDF/A-2b compliant. The program will validate a given file. If no file is provided, the program will review two default PDF files, one valid, the other invalid, just like the PDFA2DocumentValidation sample. It will also use the same default input files.

Relevant APIs:

- com.adobe.pdfjt.services.pdfa2.PDFA2ValidationHandler
- com.adobe.pdfjt.services.pdfa2.EmbeddedFilePDFA2ValidationHandler

See also these other samples:

- core.pdfa.PDFA2DocumentValidation
- core.pdfa.ValidateDocument

ValidateDocument.java

This sample illustrates how to open a document and use the PDFAService class to validate that the file is PDF/A compliant. The program will review two PDF files, one valid, the other not, and responds with comments at the command line or in the Eclipse console for each one.

This program demonstrates how PDFAService and PDFAValidationHanlder work together. The PDFAService is a Java class that attempts to convert a PDF document to PDF/A. As the PDFAService encounters errors it cannot repair, it reports the errors to the PDFAValidationHandler method

Relevant APIs:

- com.adobe.pdfjt.services.pdfa.PDFAValidationHandler
- com.adobe.pdfjt.services.pdfa

See also these other samples:

- core.pdfa.PDFA2DocumentValidation

Working with a Portfolio

PDF files can hold embedded or attached files, rather like email messages. These files, referred to as PDF Portfolios, can be used as a means to store a group of related files. The Portfolio is effectively a container or coversheet, where each of the embedded files retains its own identity. Each PDF attachment within a PDF portfolio works as its own PDF file, with its own security settings, digital signatures, and so on. You can open a PDF Portfolio just as you would a regular PDF document, but the collection does not have its own pages, and you will see a different layout.

ExportNavigator.java

This sample demonstrates how to export the navigator (.nav) file from a PDF portfolio input document. The program generates a separate .nav output file.

A Navigator (.nav) file holds the Universal Container Format (UCF) for the PDF portfolio. The UCF is a compressed collection of related files in a single container, making it easier to move, store, and access the collection. It works much as a zip file would, though it can also offer digital signatures and encryption.

Relevant APIs:

- com.adobe.pdfjt.services.portfolio.PortfolioWriter
- com.adobe.pdfjt.pdf.interactive.navigation.collection.PDFPortableCollection

See also these other samples:

- core.collection.CopyPDFCollection

The PDF Auditor

AuditPDF.java

This sample program reviews a PDF input document and lists the size, in bytes, of every component in that document. AuditPDF.java runs the file through the PDFAuditor in the PDFJT method called PDFAuditor.auditPDF().

A similar feature is provided in Adobe Acrobat, the Audit Space Usage tool. With a PDF file open in Acrobat, click *File*, and then *Save As Other* and *Optimized PDF*.

Then, click . A window appears, listing the size in bytes of the various features of the PDF document, such as images and bookmarks, and the percentage of each of those features of the total size of that document. If the images in a PDF document take up 400K of a 1 MB PDF document, the images will be shown as being 40% of the document.

The AuditPDF sample program in PDF Java Toolkit does not provide a window. Rather, when you run the program it will list out the sizes of each element within the PDF document you are reviewing at the Eclipse viewer pane or at your command prompt, like this:

```
File Space Usage

Acrobat Forms*: 128
Color Spaces*: 580
Content Streams: 112163
Cross Reference Table: 39820
Document Overhead*: 1055
Extended Graphic States: 160
Fonts: 64785
Images: 396080
Structure Info*:162461
Thumbnails: 7885
XObject Forms: 2897

Total file Size: 788014
```

The components that the PDFAuditor provides sizes for include:

- Acrobat Forms
- Bookmarks
- Comments and 3D Content
- Content Streams
- Color Spaces
- Cross Reference Table
- Document Overhead
- Embedded Files
- Extended Graphic States
- Fonts
- Images
- Link Annotations
- Named Destinations

- Piece Info
- Shading Info
- Structure Info
- Thumbnails
- XObject Forms
- Pattern Info
- Web Capture Info

If the PDF document does not include one of the objects listed above, it will not be included in the list of output values.

The total size of the file appears at the end of the output list.

The Object Compression factor appears if the input PDF document includes compressed object streams. The compression is shown as a single negative number, to show how much space has been saved for the PDF document by compressing the objects embedded in that document.

Suppose you start with a PDF document that is 1.7 MB. You use Object Compression to reduce the size of the images, XObject Forms, and Content Streams in that document. In the process the file size is reduced to 800 KB. The compression has saved a total of 900 KB, so the Object Compression value would be -900,000.

The input file is:

```
input/AdobeCollections.PDF
```

You can use the Command Line Interface to enter a path and file name for your own PDF document instead of this default value.

The program does not generate any output files.

Relevant APIs:

- com.adobe.pdfjt.pdf.document.PDFDocument;
- com.adobe.pdfjt.services.auditor.PDFAuditor;

See also these other samples:

- none

PDF Security

ReEncryptDocument.java

This sample shows how to open a PDF file protected by both a user (document) password and an owner (permissions) password. The program changes the passwords and a few document permissions, and re-encrypts the document with a specified encryption method.

Relevant APIs:

- com.adobe.pdfjt.services.permissions
- com.adobe.pdfjt.services.security

See also these other samples:

- core.document.PermissionsQuerySample
- core.digitalSignatures.QueryPermissionsSignatures

Filling in PDF Forms with Imported Data

PDF Java Toolkit provides a means to import data into a PDF form document, manipulate that data, and then save the PDF form for later use.

Preliminary requirements

Apache JAR File

To use the Fill Forms sample program, we include a JAR file called commons-cli1.2.JAR, the Apache Commons Command Line Interface (CLI). This file, provided by the Apache Software Foundation, is in addition to the JAR library file that we provide with the PDF Java Toolkit.

This JAR file provides a standard Command Line Interface that should be familiar to most developers who work with Java systems, especially those with UNIX experience. Some samples will not compile in the Command Line Interface without this JAR file.

FDF, XFDF, or XML Import File

To use FillForm, you will need to export a database record that contains form data to an *FDF, XFDF,* or *XML* file. FDF refers to Forms Data Format, a file format described in the PDF specification. FDF files use the same low-level syntax as PDF, but the FDF format is only used to describe the content found in PDF annotations and form fields. XFDF is an XML-based version of the FDF format. The XML file that you can export data into will be used for XFA forms only. FDF, XFDF, and XML data can be used in a variety of ways, such as giving it to a server application for processing, or archiving form information. Commonly FDF and XFDF files are used to hold extracted information from a PDF AcroForm so that it can be stored in a database. XML files, in the case of FillForm, are used to hold extracted information from XFA forms stored in PDFs. XFA, or XML Forms Architecture, is a set of proprietary XML specifications for use with web forms.

Scenario

Suppose you have a database of 1500 customer records, including the following information for each one:

- Name, address, phone numbers, and email address
- Date of birth
- Linked in, Facebook, and personal blog or web site addresses
- Account and purchase history
- Records of past contacts by email or phone

You would like to survey your customer base, to get an idea of their view of your firm and your products. You would also like to ask them to confirm their contact information.

You can easily create a PDF form for the survey, and include fields for the customer name, address, phone number, email address, and personal web links. But rather than asking your customers to fill out these fields, you want to populate them with the information you already have in your database. Each customer can confirm his or her personal information and then complete the rest of the survey.

So you want to export the data from your database into a stock PDF form, and then create 1500 unique copies of this document, one for each customer.

The Fill Form sample program with PDF Java Toolkit makes this possible. After you export the customer account information you need from your database, you can use this program to import these values into the appropriate fields in a PDF form document, and create one PDF form document for each customer.

FillForm.java

This sample program uses FDF, XFDF, or XML files as a source of import data into a PDF form. FillForm can accept FDF or XFDF files for AcroForm documents, and XML files for XFA documents. The program can also run calculations on the new data after it is imported, and generate appearances for the data, which is a way of formatting how the field data looks. Finally, you can use this program to flatten the final PDF form document, keeping the form data but removing the interactive form fields.

Note that calculation, appearance generation, and flattening only work with AcroForms (Acrobat Forms), and not with XFA forms, or XML Forms Architecture.

The "generating appearances" process refers to formatting data displayed in PDF annotations or form field widgets. For example, assume one of the form fields is a Date field. A user can enter the date in the MM/DD/YYYY format into the field, and appearance generation could be used to make the date appear instead in "Month Day, Year" format. That is, a date entered as "04/11/2014" would be rendered "April 11, 2014."

A PDF Form document needs to have appearance streams re-generated when the data changes in that document. Otherwise, the fields in that form will either be blank, out of date, or incorrectly formatted.

The program is written to automatically calculate values in the form fields, and will also automatically generate appearances, unless you turn these functions off in the code. However, the program is also set by default to not flatten the form fields after the data is imported from the FDF file, calculated, and formatted. If you want the program to flatten the form data, you need to turn that feature on.

Relevant APIs:

- `com.adobe.pdfjt.pdf.interactive.forms.PDFInteractiveForm;`
- `com.adobe.pdfjt.services.ap.AppearanceService;`
- `com.adobe.pdfjt.services.fdf.FDFDocument;`
- `com.adobe.pdfjt.services.fdf.FDFService;`
- `com.adobe.pdfjt.services.formflattener.FormFlattener;`
- `com.adobe.pdfjt.services.javascript.JavaScriptHandler;`
- `com.adobe.pdfjt.services.xfa.XFAService;`
- `com.adobe.pdfjt.services.xfdf.XFDFService;`

See also these other samples:

- core.forms. FDFFormImportDemo
- core.forms.ExecuteJavaScriptInFieldCalculationOrder
- core.digitalSignatures.JSSigInfoExtensionDemo

Importing and Exporting Data to and from PDF Forms

PDF Java Toolkit offers two programs that provide a workflow that simulate populating fields in a PDF form, editing that content, and then exporting the field records from a PDF form file to an FDF, XFDF, or XML file so that the records can be posted to a database.

Two related programs are provided, one that works with Acroform, and the other with XFA.

AcroFormSample.java

This sample demonstrates how to import data from the fields in an FDF or XFDF file to a blank PDF form, based on Adobe Acrobat Form (AcroForm). AcroForm is a standard format that supports interactive forms, with items like radio buttons and drop down lists.

FDF refers to Forms Data Format, a file format described in the PDF specification. FDF files use the same low-level syntax as PDF, but the FDF format is only used to describe PDF annotations and form fields.

FDF can be used in a variety of ways, such as submitting data from forms to a server application for processing, or to archive forms information. XFDF is an XML-based version of the FDF format.

The sample program is designed to pull data from a sample input FDF or XFDF file into the AcroForm stored in a sample PDF form called FruitForm_1.PDF. The program then manipulates that imported forms data.

The AcroForm sample program completes the following steps.

1. The program starts by importing field data from the FDF or XFDF sample input file into the fields in the AcroForm in the PDF file FruitForm_1_AcroForm.PDF.
2. It then saves FruitForm_1_AcroForm.PDF with this imported data to the output directory. So two copies of this PDF file are available, one blank, the other with imported records.
3. AcroFormSample exports the form data from the output version of FruitForm_1_AcroForm.PDF to an FDF (or XFDF) file, FruitForm_1_AcroForm.FDF, and saves it to the output directory. The program then edits the values in several of the fields of this FDF or XFDF file, and saves this export file with the changes, overwriting the content originally found in this file.

   Note that the file format remains consistent throughout the process. If you provide an FDF import file, the sample program will also produce an FDF export file.
4. Then, the sample program imports this data in this FDF or XFDF output file back into the output copy of the PDF file, FruitForm_1_AcroForm.PDF and saves it. This secondFi import process overwrites the original data in several of the fields on the form. If you were to open the input version of FruitForm_1_AcroForm.PDF and the output version of this file, also called FruitForm_1_AcroForm.PDF, you would see how the fields have been updated. The input version of the PDF file will be blank, and the output version will show edited values in the form fields.

   This process of exporting the data, editing it, and then re-importing it is intended to simulate a user opening the PDF form with the fields already populated and manually entering changes in some of those fields. In this example, the program imports values into the Global Fruit Order Form, including the name, address, city, state, zip code, phone number, and email address for the person placing the order. A series of checkboxes are completed for the types of fruit to include in the order, and the form then calculates the cost for the order and displays that cost, including delivery. The sample program changes the values in the Name field for an FDF import; for XFDF, it updates the Name field and the City and Zip fields as well.

5. AcroFormSample then exports the data a second time from the output version of FruitForm_1_AcroForm .PDF to an FDF or XFDF file that already exists in the output directory, and overwrites the information stored in that file. This step simulates pulling the data records from the completed PDF form and storing them in an FDF file, so that the data can be later posted to a database table.
6. The program generates appearances for the saved FruitForm_1_AcroForm.PDF file in the output folder.
7. The program flattens the form file, and saves the final version of this PDF document.
8. The program takes the PDF form that has been flattened and saves it using a format for long-term archival storage, PDF/A-1B.

   As a result three PDF files appear in the output directory:

   a. The completed FruitForm_1_Acroform.PDF form file
   b. The flattened PDF form file, Flattened_FruitForm_1_Acroform.PDF
   c. The flattened PDF form, converted to the PDF/A-1B archive format, called Archived_FruitForm_1_Acroform.PDF

The input files are stored in:

```
input/workflows
```

Output files are copied to:

```
output/AcroFormSample
```

Relevant APIs:

- com.adobe.pdfjt.services.fdf.FDFService;
- com.adobe.pdfjt.services.formflattener.FormFlattener;
- com.adobe.pdfjt.services.xfdf.XFDFService;

See also these other samples:

- workflow.XFASample

XFASample.java

The XFASample.java program imports data from an XML file into a blank, static XFA form embedded in a PDF document and manipulates that imported data. XFA, or XML Forms Architecture, is a set of proprietary XML specifications for use with web forms. Since 2002 the XFA standard has been owned by Adobe Systems. XFA forms are saved internally in PDF files.

The XFAForm sample program completes the following steps:

1. The program starts by importing field data from the XML sample input file into the fields in the XFA in the PDF file Check_Request_XFA.PDF.

2. It then saves Check_Request_XFA.PDF with this imported data to the output directory. So two copies of this PDF file are available, one blank, the other with imported records.
3. XFASample exports the FDF data from the output version Check_Request_XFA.PDF to an XML file, Check_Request_XFA_data.XML, and saves it to the output directory as well. The program then edits the values in several of the fields of this output version of the XML file, and saves this file with the changes, overwriting the content found in the original version of the file.
4. Then, the sample program imports this data in this XML output file back into the output copy of the PDF file, Check_Request_XFA.PDF, and saves the file. This second import process overwrites the original data in several of the fields on the form. If you were to open the input version of Check_Request_XFA.PDF and the output version of this file, also called Check_Request_XFA.PDF, you would see how the fields have been updated. The input version of the PDF file will be blank, and the output version will show edited values in the form fields.

   This process of exporting the data, editing it, and then re-importing it is intended to simulate a user opening the PDF form with the fields already populated and manually entering changes. In this example, the program imports values into the Check Request Form, including the name, title, department, phone number, date, date needed, reason/account, payee, amount, delivery instructions, and comments. The sample program updates the values in the Name and Reason/Account fields.
5. XFASample then exports the data a second time from the output version of Check_Request_XFA .PDF to an XML file that already exists in the output directory, and overwrites the information stored in that file. This step simulates pulling the data records from the completed PDF form and storing them in an XML file, so that the data can be later posted to a database table.
6. The program calls the XFADOMService.getXFADOM() service for the FruitForm_1_XFA.PDF file. XFADOM refers to XML Document Object Model, an interface that allows programs and scripts to access and update content, structure, and style of documents. The XFADOM service builds the PDF form structure for the data to match the XFA structure so that the resulting file can be managed like a typical PDF file. The XFASample program completes this step as an alternative to generating appearances for the final PDF form document.
7. The program flattens the form file, and saves the final version of this PDF document.
8. The program takes the PDF form that has been flattened and saves it using a format for long-term archival storage, PDF/A-1B.

   As a result three PDF files appear in the output directory:

   a. The completed Check_Request_XFA.PDF form file
   b. The flattened PDF form file, Flattened_Check_Request_XFA.PDF
   c. The flattened PDF form, converted to the PDF/A-1B archive format, called Archived_Check_Request_XFA.PDF

The input files are stored in:

```
input/workflows
```

Output files are copied to:

```
output/XFASample
```

Relevant APIs:

- com.adobe.pdfjt.services.formflattener.FormFlattener
- com.adobe.pdfjt.services.xfa.XFADOMService
- com.adobe.pdfjt.services.xfa.XFAProcessingOptions
- com.adobe.pdfjt.services.xfa.XFAService
- com.adobe.pdfjt.services.xfa.XFAService.XFAElement

See also these other samples:

- workflow.AcroFormSample