Developer's Guide

v2.3.0 – April 22nd, 2009

PDFTextStream Developer's Guide.

Snowtide Informatics Systems, Inc.
243 King Street, Suite 248
Northampton, MA 01060

http://www.snowtide.com

Produced and printed in the United States.

Welcome!

Thank you for using PDFTextStream. We have worked very hard to ensure that the PDFTextStream library meets the highest quality and performance standards, while still being easy to use and easy to integrate into your applications. Although we're confident that you will find PDFTextStream pleasant to work with, we have designed this Developer's Guide to anticipate how you might use the library and what potential snags you might run into.

If you find that this Guide does not answer your questions, please do not hesitate to contact customer support through our website, http://www.snowtide.com.

Again, thank you for using PDFTextStream.

Chas Emerick
Founder & President
Snowtide Informatics Systems

INTRODUCTION

PDFTextStream is a Java class library that enables applications to access text and metadata content in PDF documents quickly, easily, and accurately. While there are many excellent tools and Java libraries available for generating PDF documents, PDFTextStream is the first and only library to focus on the extraction of text and metadata from PDF files. As such, it is the fastest, most feature-complete PDF text extraction library available on the market today.

Using .NET or Python?
PDFTextStream can be used by .NET or Python applications without compromising speed or functionality. See page 45 for details.

Here is a summary of the main features of PDFTextStream:

- Support for versions 1.0 – 1.6 of the PDF document specification (current through Acrobat 7)
- Pure Unicode output, including Chinese, Japanese, and Korean (CJK) support
- 40- and 128-bit document decryption
- Support for extracting bookmarks, annotations, and interactive form data
- Provides access to all document metadata contained in a PDF file, including Adobe XMP metadata streams
- Subclasses `java.io.Reader`, which ensures a simple, familiar interface, and easy integration opportunities with existing components expecting a `java.io.Reader` instance
- Easy integration with Apache Lucene, the most popular pure-Java indexing and search engine

Given PDFTextStream's capabilities and its focus on performance, it is well suited for use in a number of different development environments, including:

- High-volume enterprise environments that need to extract text from large numbers of PDF files

- Content management systems (CMS's) that need access to the text of PDF files for categorization or summarization purposes
- Full-text indexing and search systems that wish to add comprehensive support for searching PDF documents
- Data conversion processes, especially those that aim to selectively extract and convert unstructured PDF content into structured data elements.
- Alternative content delivery systems that need to provide access to PDF document content to devices that cannot readily open and view PDF content (i.e. mobile phones, PDA's, etc)

The following sections will provide you with the reference and tutorial information you need to successfully integrate PDFTextStream into your application.

Setting Up PDFTextStream

Requirements

PDFTextStream is compatible with version 1.4 and higher of the Sun Java Runtime Environment. Other Java virtual machines may be used and should provide a suitable environment for PDFTextStream to operate in, but note that support is provided only for users using the Sun JVM.

PDFTextStream uses temporary files at various points in its operation. This requires that PDFTextStream must be used on a system where there is a suitable temporary directory or mount point established, with the path to that temporary directory properly set in the `java.io.tmpdir` system property. Properly setting the `java.io.tmpdir` system property is something that is typically taken care of by the JRE.

External Dependencies

PDFTextStream's core functionality does not require any external libraries.

If PDFTextStream's Lucene integration features are used, the libraries required by Lucene 1.2 or higher are required.

NOTE: Previous versions of PDFTextStream depended upon the Apache Jakarta commons-logging library to facilitate the automatic integration of PDFTextStream into the logging infrastructure already in place in your deployment environment. Starting with v1.3, PDFTextStream has such integration functionality built-in, which eliminates the dependency upon the commons-logging library. Please see page 37 for details on how PDFTextStream can integrate with your logging infrastructure.

Classpath

An application's classpath must be changed in order for PDFTextStream's classes to be available from the application's code. For example, if an application's current classpath is:

```
.:lib/:lib/yourapplication.jar:ext/resources/
```

and the PDFTextStream jar is in the lib directory, then you should modify the application's classpath to be:

```
.:lib/:lib/yourapplication.jar:ext/resources/:lib/pdftextstream.jar
```

These classpath examples are suitable for a Unix/Linux/BSD system. The appropriate classpath for a Windows system would be the same as above, except using semicolons (;) instead of colons, and using backslashes (\) instead of forward slashes.

Licensing

If you have purchased PDFTextStream, you should have received a license file named *pdftextstream.license*. Deploying this file along with PDFTextSTream will remove all evaluation limitations:

- Approximately half of all digits (0-9) in text extracts will be randomized.
- The `title` field of half of all `com.snowtide.pdf.Bookmark` objects will be slightly modified.
- The `contents` field of half of all `com.snowtide.pdf.annot.Annotation` objects will be slightly modified
- The `richContent` field of half of all `com.snowtide.pdf.annot.FreeTextAnnotation` objects will be slightly modified
- The `uri` field of half of all `com.snowtide.pdf.annot.LinkAnnotation` objects will be slightly modified
- The values held by half of all form field objects (those objects whose classes implement the `com.snowtide.pdf.forms.FormField` interface) will be slightly modified.

Deploying a PDFTextStream license file can be accomplished in any of four ways:

- Simply place the `pdftextstream.license` file in your application's current directory, and PDFTextStream will load it automatically.
- Set the `pdfts_license_path` environment variable to the full path where the `pdftextstream.license` file may be found. For example, if you copy the `pdftextstream.license` file to the `C:\WINDOWS` directory, this can from the command line by issuing this command: `set pdfts_license_path="C:\WINDOWS\pdftextstream.license"`
- Set the `pdfts_license_path` system property to the full path where the `pdftextstream.license` file may be found. Using PDFTextStream.Java or PDFTextStream.Python, this can be done either by setting the system property before using PDFTextStream by calling the `java.lang.System.setProperty(String, String)` function, or by specifying the full path of the license file's location using a `–D` argument to the JVM, such as `–Dpdfts_license_path=~/pdftextstream.license`. PDFTextStream.NET users can set this system property in an application's App.config file, like so:

```
<appSettings>
  <add key="ikvm:pdfts_license_path"
      value="C:\WINDOWS\pdftextstream.license"/>
</appSettings>
```

- The location of the license file may also be specified at runtime, rather than at application startup or through configuration. This may be done by calling the `com.snowtide.pdf.PDFTextStream.loadLicense(String)` function. That function may be called at any time to load and verify a license file at the specified path.
- Place the `pdftextstream.license` file at a root of the JRE's classpath. For example, if an application's classpath is defined as shown in the second example in the 'Classpath' section above, then the license file should be placed either in `lib` or `ext/resources/`. PDFTextStream.NET users can specify the classpath in their application's `App.config` file, like so:

```
<appSettings>
  <add key="ikvm:java.class.path" value="c:\resources\"/>
</appSettings>
```

Configuration

PDFTextStream offers a number of configuration options, all of which are available in the `com.snowtide.pdf.PDFTextStreamConfig` class. Please see that class' API documentation for details of the available configuration options.

Usage and Examples

While performance and text extraction accuracy are critical concerns of PDFTextStream, it also aims to be extraordinarily easy to use within your applications. It subclasses `java.io.Reader`, and therefore can be dropped into modules that already work with the `java.io.Reader` interface without any modification or configuration.

Extracting Text

Beyond this simple 'drop-in' usage scenario, it is very common to use PDFTextStream directly. Even in this case, the code needed to read the text out of a PDF file using PDFTextStream is trivial, as this simple utility method shows:

```
public StringBuffer getPDFText (File pdfFile) throws IOException {
  PDFTextStream stream = new PDFTextStream(pdfFile);

  StringBuffer sb = new StringBuffer(1024);
  //  get OutputTarget configured to pipe text to the provided StringBuffer
  OutputTarget tgt = OutputTarget.forBuffer(sb);
  stream.pipe(tgt);

  stream.close();
  return sb;
}
```

Here, a `com.snowtide.pdf.OutputTarget` instance is used instead of the `java.io.Reader` interface. When a PDFTextStream instance is provided with an `OutputTarget` instance via its `pipe(OutputHandler)` function, it sends all of its output to that `OutputHandler` (`OutputTarget` is the default implementation of `OutputHandler` – see page 17 for information on why and how you might create your own `OutputHandler` implementations). This eliminates some internal buffering that would otherwise be needed to support the `java.io.Reader` interface, and prevents you from having to create a temporary buffer to take advantage of PDFTextStream's `java.io.Reader` implementation. Additionally, it further simplifies the code that is necessary to use PDFTextStream.

This is certainly cleaner than using the `java.io.Reader` interface if all you need to do is read all of the text out of a PDF document. `OutputTarget` instances can also be used to send a PDFTextStream's output directly to a local file:

```
public void savePDFText (File pdfFile, File textFile) throws IOException {
  PDFTextStream stream = new PDFTextStream(pdfFile);

  BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(
         new FileOutputStream(textFile)));

  //  get OutputTarget configured to pipe text to the provided file path
  OutputTarget tgt = new OutputTarget(writer);
  stream.pipe(tgt);

  writer.flush();
  writer.close();

  stream.close();
}
```

This is a significant improvement compared to the repetitive code that you would otherwise need to write to gather PDFTextStream's output into a buffer, and then write that buffer out to a file.

Things that should be noted at this point:

- If PDFTextStream's `java.io.Reader` interface is used, then the output of PDFTextStream should not be buffered using a `java.io.BufferedReader`. The PDFTextStream instance already reads the text out of the provided PDF file one page at a time, performing any necessary buffering automatically. This is done because generic buffering strategies are not effective when reading content out of a PDF -- the structure of PDF files is very complex, requiring specialized buffering techniques for optimum performance. Any additional buffering would diminish performance and increase memory consumption.
- PDFTextStream instances need to be closed when they are no longer needed. This ensures that various system-level resources are released after all of the text is read out of the PDF file.

Extracting Text, Page by Page

PDFTextStream also provides access to the text content of individual pages in PDF documents. For example, these functions extract only the fifth page of text from the given file, and the first 3 pages of text:

```
public String getFifthPageText (File pdf) throws IOException {
  PDFTextStream stream = new PDFTextStream(pdf);
  StringBuffer sb = new StringBuffer(1024);

  OutputTarget tgt = new OutputTarget(sb);
  Page fifthPage = stream.getPage(4); // page numbers are 0-based
  fifthPage.pipe(tgt);

  stream.close();

  return sb.toString();
}

public String getFirstThreePagesText (File pdf) throws IOException {
  PDFTextStream stream = new PDFTextStream(pdf);
  StringBuffer sb = new StringBuffer(1024);

  OutputTarget tgt = OutputTarget.forBuffer(sb);
  for (int pagenum = 0; pagenum < 3; pagenum++) {
      Page page = stream.getPage(pagenum);
      page.pipe(tgt);
  }

  stream.close();

  return sb.toString();
}
```

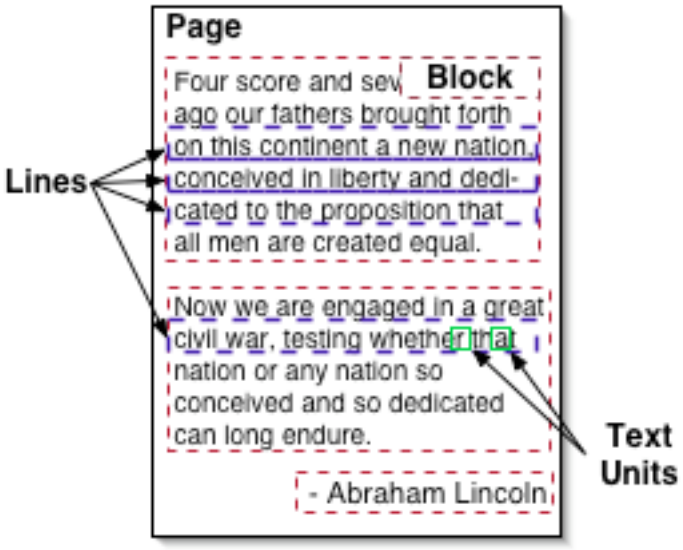
Pages in PDF documents are represented by `com.snowtide.pdf.Page` instances. In addition to providing a `pipe(OutputTarget)` function (which works just like the `PDFTextStream.pipe(OutputTarget)` function, but only within the context of a single page), `com.snowtide.pdf.Page` instances provide a set of functions for accessing a range of page-related attributes. These attributes include page height, width, rotation, and more (see the PDFTextStream javadoc for details).

The PDFTextStream Document Model

PDF documents specify their text content one character at a time, without any indication of physical structure (such as lines, paragraphs, columns,

etc). Therefore, PDFTextStream must employ advanced document understanding processes to derive the structure of each PDF document page it is asked to extract. These processes gather characters into lines, lines into blocks, blocks into columns, and so on. The document structure that these entities represent and the API that PDFTextStream exposes for developers to work with them collectively forms the PDFTextStream document model.

The document model is necessarily hierarchical, and its API mirrors that hierarchy:



Pages (`com.snowtide.pdf.Page`) contain Blocks (`com.snowtide.pdf.layout.Block`). Blocks may contain other Blocks or Lines (`com.snowtide.pdf.layout.Line`) (but not both). Lines contain TextUnits (`com.snowtide.pdf.layout.TextUnit`), which roughly represent single characters.

This structure is rooted at the page-level by an object that implements the `com.snowtide.pdf.layout.BlockParent` interface, available via the `Page.getTextContent()` function. Objects that implement the `BlockParent` interface contain an ordered set of `com.snowtide.pdf.layout.Block` instances, each of which represent a block of text presented on a page within a PDF document. In most circumstances, each `Block` instance corresponds to a paragraph of text.

`Block`s, `Line`s, and `TextUnit`s all implement the `com.snowtide.pdf.layout.Region` interface, which allows for the retrieval of their positioning on the page (x- and y-coordinates, height, width, etc.). Additionally, `Block`s also implement the `BlockParent` interface, which means that `Block`s can contain other `Block`s. This is necessitated by document structures such as tables, where distinct blocks of content must be grouped and ordered together. Use the PDFTextStream javadoc to find the particulars of how to traverse the document model – each document model entity presents a very simple list-like API that should be essentially self-explanatory.

TextUnit Details

To anyone who does not know the inner workings of PDF documents and how fonts and encodings work in the PDF document specification, "text unit" might seem to be a strange diversion from the straightforward names given to the other parts of the PDFTextStream document model ("page", "block", "line", and so on). It is worth exploring why these entities are called "TextUnits" and not simply "Characters".

A quick look at the javadoc of `com.snowtide.pdf.layout.TextUnit` reveals a few interesting functions: `TextUnit.getCharacterSequence()` and `TextUnit.getCharCode()`. Text in PDF documents is encoded as a series of character codes (available via `TextUnit.getCharCode()`), which are then mapped to a concrete sequence of Unicode characters (available via `TextUnit.getCharacterSequence()`) based on the encoding that is specified by a PDF document. That sounds straightforward enough until one realizes that PDF documents can encode more than one Unicode character for each individual raw character code.

For example, a PDF document might specify that the character code 188 should be mapped to the Unicode (and ASCII) character 'f'. However, it could specify instead that the character code 189 should be mapped to a sequence of Unicode characters, such as "fi" or "ae". Therefore, each `TextUnit` instance can represent an indeterminate number of Unicode characters.

Please note that the font in effect when each `TextUnit` is outputted is available via `TextUnit.getFont()`.

OutputHandlers: Text Extraction using Document Model Events

In many applications, simple text extraction as shown in the 'Extracting Text' section of this chapter is not enough to meet requirements. Sometimes a PDF document is so large that extracting all of its text would strain your application's available resources. Perhaps your application needs to produce something other than plain text, such as an HTML version of the PDF document. The best way to meet such requirements is to utilize the `com.snowtide.pdf.OutputHandler` interface.

The `OutputHandler` interface is directly analogous to the lightweight SAX XML `ContentHandler` interface. Just like XML, PDFTextStream defines a document model that can be traversed systematically using a random-access interface (which is called DOM in the XML world). But also just like XML, PDFTextStream also provides a way to process document content in a lightweight, serial fashion.

The `OutputHandler` interface represents this second option. It defines a range of functions that can be selectively implemented to, for example, only be notified of character-level data. An `OutputHandler` subclass that does this by overriding the `OutputHandler.textUnit(TextUnit)` function will receive one event (in the form of a `TextUnit` object) for every `TextUnit` object in a given PDF document page or block. The `OutputHandler` subclass can then take whatever action is necessary given its purpose – write the `TextUnit`'s content to disk, send it over a network connection, make note of where the `TextUnit` is located on the page for display purposes, and so on.

Each PDFTextStream, `com.snowtide.pdf.Page`, and `com.snowtide.pdf.layout.Block` instance provides a `pipe(OutputHandler)` function. Invoking this function on any instance of these classes will cause the appropriate PDF document model events to be sent to the provided `OutputHandler` object in the natural order that they occur. For example, just before starting the events associated with a block

of content, the `OutputHandler`'s `startBlock(Block)` function will be called with that `Block` instance as a parameter; when all of the child entities of that `Block` have been delivered (its child blocks, its child lines, its child `TextUnit`s, etc), the `OutputHandler`'s `endBlock(Block)` function will be called. Events bookend content like this for all of the containers in the PDFTextStream document model: the PDF document itself (`startPDF(String, File)` and `endPDF(String, File)`), pages (`startPage(Page)` and `endPage(Page)`), blocks (as has already been discussed), and lines (`startLine(Line)` and `endLine(Line)`).

Source code for example `OutputHandler` implementations are included with your PDFTextStream distribution. The `pdfts.examples.GoogleHTMLOutputHandler` sample will produce an XHTML document that roughly duplicates the spirit of the "view as text" page that Google provides for PDF search results. Another `OutputHandler` example, `pdfts.examples.XMLOutputTarget`, writes an XML document directly to a provided `StringBuffer` that includes structural document information, as well as indications of text formatting (e.g. bolding, underlining, strikethroughs, italics, etc.).

Retrieving Document Metadata

PDFTextStream allows your applications to access both varieties of document-level metadata that might be available in a PDF file: DocumentInfo name/value mappings, and Adobe XMP data.

DocumentInfo Name / Value Metadata

Sometimes referred to as "classic" metadata, DocumentInfo name/value pairs typically include creation and modification dates, the PDF document's author's name, and other potentially interesting metadata attributes. Retrieving the document metadata attributes contained in a PDF file is a no-brainer, as shown in this code segment:

```
PDFTextStream stream = new PDFTextStream(pdfFile);

// get collection of all document attribute names
Set attributeKeys = stream.getAttributeKeys();

// print the values of all document attributes to System.out
Iterator iter = attributeKeys.iterator();
String attrKey;
while (iter.hasNext()) {
   attrKey = (String)iter.next();
   System.out.println(attrKey + "=" + stream.getAttribute(attrKey));
}

// print the value of the Author attribute to System.out
String authorName = (String)stream.getAttribute(PDFTextStream.ATTR_AUTHOR);
System.out.println("Author: " + authorName);
```

A few notes about this code:

- PDFTextStream will read a PDF file's document attributes automatically -- no special method call or configuration is necessary.
- If a PDFTextStream constructor returns successfully, then it can be assumed that all of the document attributes in the PDF file have been read and are available for retrieval. The attributes can be accessed from a PDFTextStream instance even if the instance has been closed.
- A number of methods are available for retrieving all of a PDF file's document attributes, just the keys of the attributes, and even a method for retrieving a copy the `java.util.Map` instance in which the document attributes are held. Check the PDFTextStream javadocs for details.
- The names of many standard document attributes are held as static final Strings by PDFTextStream; such Strings' variable names all begin with 'ATTR' to identify them as such.
- `PDFTextStream.getAttribute(String)` can technically return any Object. However, in most circumstances, the value of document attributes are `String` objects; this is true for all standard PDF attributes, but PDF file generators are allowed to add non-String attributes. The allowable data types in PDF attribute values include `Strings`, `Numbers` (either `Integer` or `Float` instances depending on the type of number), `Booleans`, and `Object[]` arrays that contain any of the other possible attribute value types.

- Date attributes are stored in PDF files as specially-encoded Strings. Such attributes can be converted into `java.util.Date` objects by calling the `PDFDateParser.parseDateString(String)` method with the String date attribute as the only parameter. The only standard date attributes contained in PDF files are the creation and modification dates, associated with the `ATTR_CREATION_DATE` and `ATTR_MOD_DATE` keys, respectively.
- `PDFTextStream.getAttribute(String)` will return null if an attribute key is provided that is not defined in the PDF file that was read.

Adobe XMP Data

A PDF document may also contain metadata in the form of an Adobe XMP (Extensible Metadata Platform) stream. XMP streams are XML documents that adhere to the XMP metadata schema as defined by Adobe. XMP streams typically contain the same set of metadata attributes that are available through the "classic" metadata attribute accessors, described above. However, some specialized PDF generators and workflows do add metadata constructs to a document's XMP stream that does not fit within the simple name / value pair structure of "classic" metadata.

PDFTextStream allows your application to access XMP streams very easily, as shown in this example:

```
PDFTextStream stream = new PDFTextStream(pdfFile);

// [. . . read PDF text . . .]

// get XMP data stream
byte[] xmlMetadata = stream.getXmlMetadata();

// close PDFTextStream instance **AFTER** retrieving XMP metadata stream
stream.close();

// handle metadata from XMP stream in application-specific manner
processXMPMetadata(pdfFile, xmlMetadata);
```

Things to consider when retrieving XMP data from a PDFTextStream instance:

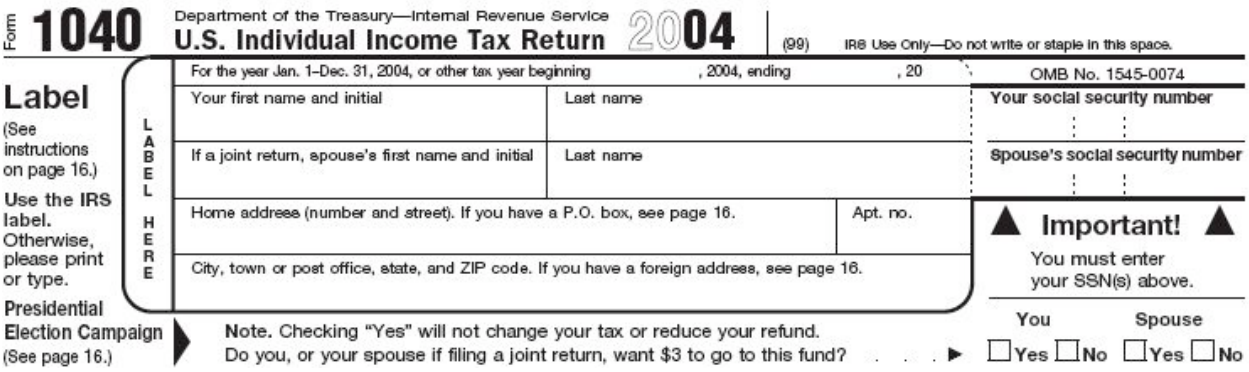
- The `PDFTextStream.getXmlMetadata()` method may not be called after the PDFTextStream instance is closed, and it should be called either before or after any text is read from the instance.
- The byte array returned by `PDFTextStream.getXmlMetadata()` is XML data pulled directly from the PDF file being read; PDFTextStream does not process this data at all. Its format is defined by the Adobe XMP specification (see http://www.adobe.com/products/xmp/main.html for details).
- `PDFTextStream.getXmlMetadata()` will return null if the PDF file being read does not contain any document-level XMP data.

Accessing Interactive PDF Forms

Forms are used in virtually every industry and environment to efficiently collect data from individuals, but paper forms have frequently represented the worst of modern institutions – bureaucracy, unresponsiveness, and inflexibility.

The interactive form features offered by PDF document technology are helping to ease the handling of forms and form data by eliminating the need for paper forms, enabling user-friendly entry of form data and information, and providing for the efficient extraction of that data and information after a form is submitted. PDFTextStream supports both the extraction of form data from PDF documents as well as the generation of PDF documents with updated form field data.

Take, for example, a form that is ubiquitous and known to all within the United States, the dreaded IRS Form 1040:



Form **1040** Department of the Treasury—Internal Revenue Service **U.S. Individual Income Tax Return** **2004** (99) IRS Use Only—Do not write or staple in this space.

Label (See instructions on page 16.) Use the IRS label. Otherwise, please print or type. Presidential Election Campaign (See page 16.)

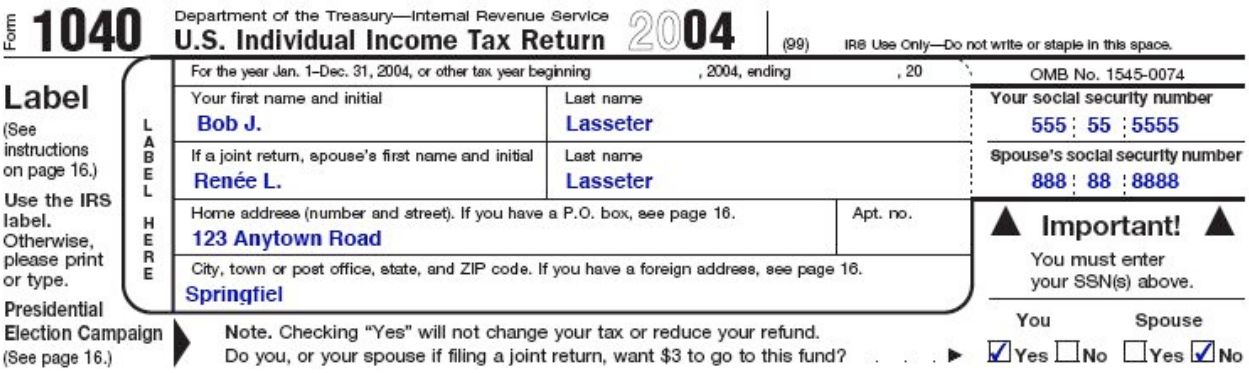
LABEL HERE

For the year Jan. 1–Dec. 31, 2004, or other tax year beginning , 2004, ending , 20 — OMB No. 1545-0074

Your first name and initial	Last name	Your social security number
If a joint return, spouse's first name and initial	Last name	Spouse's social security number
Home address (number and street). If you have a P.O. box, see page 16.	Apt. no.	▲ Important! ▲ You must enter your SSN(s) above.
City, town or post office, state, and ZIP code. If you have a foreign address, see page 16.		

Note. Checking "Yes" will not change your tax or reduce your refund. Do you, or your spouse if filing a joint return, want $3 to go to this fund? . . . ▶ You ☐Yes ☐No Spouse ☐Yes ☐No

The image above is taken from the PDF version of the 1040, which faithfully reproduces the appearance of the 1040 as the U.S. Government prints it each year. It could be printed, filled in by hand, and submitted by mail. However, if opened by a PDF viewer (like Adobe Acrobat) that is forms-capable, then each field within the form becomes available to user input, like so:



Form **1040** Department of the Treasury—Internal Revenue Service **U.S. Individual Income Tax Return** **2004** (99) IRS Use Only—Do not write or staple in this space.

Label (See instructions on page 16.) Use the IRS label. Otherwise, please print or type. Presidential Election Campaign (See page 16.)

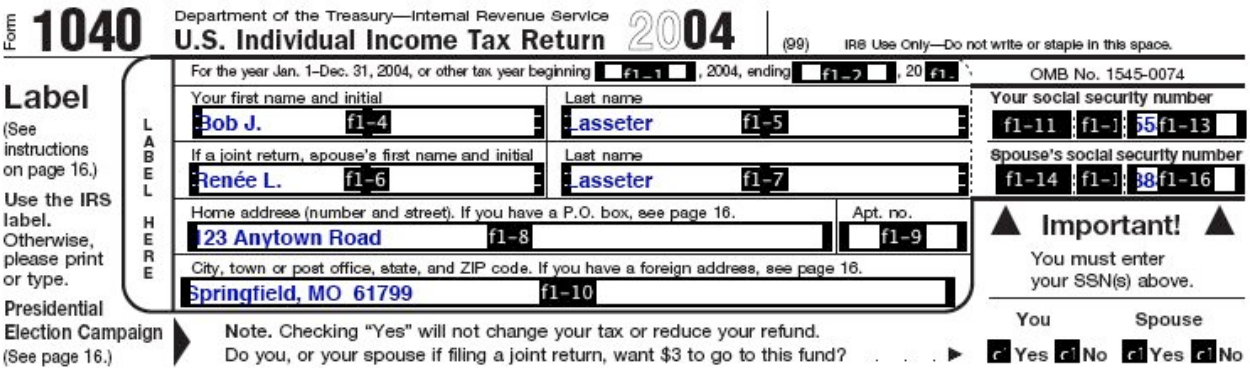
LABEL HERE

For the year Jan. 1–Dec. 31, 2004, or other tax year beginning , 2004, ending , 20 — OMB No. 1545-0074

Your first name and initial	Last name	Your social security number
Bob J.	Lasseter	555 : 55 : 5555
If a joint return, spouse's first name and initial	Last name	Spouse's social security number
Renée L.	Lasseter	888 : 88 : 8888
Home address (number and street). If you have a P.O. box, see page 16.	Apt. no.	▲ Important! ▲ You must enter your SSN(s) above.
123 Anytown Road		
City, town or post office, state, and ZIP code. If you have a foreign address, see page 16.		
Springfiel		

Note. Checking "Yes" will not change your tax or reduce your refund. Do you, or your spouse if filing a joint return, want $3 to go to this fund? . . . ▶ You ☑Yes ☐No Spouse ☐Yes ☑No

So, once the form is filled out within the PDF viewer, it could be submitted electronically to the IRS, which could then extract all of the data from the form programmatically instead of employing thousands to tediously enter such data by hand from paper copies of the form. PDFTextStream could be used for this extraction task; here's what it would "see" when presented with the 1040 form:


Form 1040 Department of the Treasury—Internal Revenue Service U.S. Individual Income Tax Return 2004

(99) IRS Use Only—Do not write or staple in this space.

For the year Jan. 1–Dec. 31, 2004, or other tax year beginning f1-1, 2004, ending f1-2, 20 f1- OMB No. 1545-0074

Label (See instructions on page 16.) Use the IRS label. Otherwise, please print or type.

Your first name and initial: Bob J. f1-4 | Last name: Lasseter f1-5 | Your social security number: f1-11 f1-1 55 f1-13

If a joint return, spouse's first name and initial: Renée L. f1-6 | Last name: Lasseter f1-7 | Spouse's social security number: f1-14 f1-1 38 f1-16

Home address (number and street). If you have a P.O. box, see page 16. 123 Anytown Road f1-8 | Apt. no. f1-9

City, town or post office, state, and ZIP code. If you have a foreign address, see page 16. Springfield, MO 61799 f1-10

Important! You must enter your SSN(s) above.

Presidential Election Campaign (See page 16.) Note. Checking "Yes" will not change your tax or reduce your refund. Do you, or your spouse if filing a joint return, want $3 to go to this fund? ▶

You: c1 Yes c1 No Spouse: c1 Yes c1 No

Notice that each form field has a name – for example, the city/town/state/zip field has the name `f1-10`. Each field's name is unique within the form, making it very easy to access only particular form field elements and values. Here's a code sample using PDFTextStream where the main name and address information is extracted from the 1040 form and associated with application-specific names:

```
public static Map get1040Data (PDFTextStream pdfts_1040) throws IOException {
   com.snowtide.pdf.forms.Form form = pdfts_1040.getFormData();
   HashMap data = new HashMap();

   com.snowtide.pdf.forms.FormField field = form.getField("f1-4");
   data.put("first_name", field.getValue());

   field = form.getField("f1-5");
   data.put("last_name", field.getValue());

   field = form.getField("f1-8");
   data.put("address", field.getValue());

   field = form.getField("f1-10");
   data.put("city_state_zip", field.getValue());

   return data;
}
```

A `com.snowtide.pdf.forms.Form` object contains references to all of the form field elements included in the PDF form, mapped to each form fields' full, unique name. Specifically, that `Form` object is a `com.snowtide.pdf.forms.AcroForm` instance; the `AcroForm` subinterface guarantees that all fields it contains implement the `com.snowtide.pdf.forms.AcroFormField` interface. All `Form` objects provide functions for iterating over all of the available form fields

(`iterator()`), getting an `Enumeration` of all the names of a form's `FormFields` (`getFieldNames()`), and getting a particular `FormField` instance using its unique name (`getField(String)`).

The forms extraction API fundamentally presents a simple name/value mapping, and is therefore conceptually very similar to the document metadata extraction API. This is especially true with regard to text-based form fields, represented by `com.snowtide.pdf.forms.AcroTextField` instances, whose `getValue()` function will always return a String object that describes literally the retained contents of the form field.

Export and Display Values

Nontext form fields such as button fields (represented by `com.snowtide.pdf.forms.AcroButtonField` objects) and choice fields (represented by `com.snowtide.pdf.forms.AcroChoiceField` objects) have slightly more complex aspects.

`AcroButtonFields` have a variety of subtypes – principally, checkboxes (`com.snowtide.pdf.forms.AcroCheckboxField`) and radio button groups (`com.snowtide.pdf.forms.AcroRadioButtonGroupField`). These kinds of widgets are quite familiar to users of web browsers, which have analogous form entry elements. However, since these form fields are primarily visual in nature, their retained values are visually-oriented as well – the `getValue()` function of all `AcroButtonFields` will return a String code indicating how a PDF viewer should display the field's widget.

In most cases, this code will have no meaning to an extracting application, so many PDF document forms will specify export values that correspond to each potential display code, and likely describe the field's selected widget. All export values known for a particular field are available via the `getExportValues()` function; the single export value associated with a field's current value (display code) is available via the `getExportValue()` function.

`AcroChoiceFields` have a different design, which is similar to how dropdown choice widgets and their values are described in HTML documents. Each

choice available in an `AcroChoiceField` is a pairing of values: one is an export value, which is typically used in programmatic extraction and/or submission of form data, and the other is an associated display value that is shown to the user when inputting or viewing form data.

When an `AcroChoiceField` allows only one selection (as indicated by the `allowsMultipleChoices()` function), the `getValue()` function provided by `AcroChoiceFields` will return a field's export value. The corresponding display value is available via the `getDisplayValue(String)` function. When multiple selections are allowed in an `AcroChoiceField`, the `getValue()` function can return an `Object[]` containing String export values.

Finally, in some cases, an `AcroChoiceField`'s value may be arbitrarily set by the user. If this is possible, the field's `isEditable()` function will return true, and the `String` returned by the `getValue()` function may not yield any associated display value via the `getDisplayValue(String)` function.

Updating Form Field Values

PDFTextStream also supports the generation of PDF documents containing updated interactive form field values. This is supported for text, checkbox, radio button group, and choice form fields. This feature may be used to support a user-centric forms update process, as well as to drive an automated forms generation system, where (for example) template PDF form documents are customized with customers' specific information prior to being delivered or archived.

The actual update process is very simple:

1. Retrieve the form fields to be updated
2. Set new values on each form field (typically using `AcroFormField.setValue(String)` – although some form fields have specialized value setters, such as `AcroCheckboxField`)
3. Finally, call `AcroForm.writeUpdatedDocument(File)` (or `AcroForm.writeUpdatedDocument(OutputStream)`) to write out a copy of the open PDF document that contains the updated form field data.

An instance of this procedure is shown below, continuing with our use of the IRS Form 1040 as an example:

```
public static void update1040Data (PDFTextStream pdfts_1040,
      String firstName, String lastName, String address,
      String city_state_zip, File updatePath) throws IOException {
  AcroForm form = (AcroForm)pdfts_1040.getFormData();

  AcroTextField field = (AcroTextField)form.getField("f1-4");
  field.setValue(firstName);

  field = (AcroTextField)form.getField("f1-5");
  field.setValue(lastName);

  field = (AcroTextField)form.getField("f1-8");
  field.setValue(address);

  field = (AcroTextField)form.getField("f1-10");
  field.setValue(city_state_zip);

  form.writeUpdatedDocument(updatePath);
}
```

Accessing XFA PDF Forms

In addition to the (now "legacy") interactive PDF forms, the PDF specification now includes support for XFA PDF forms. XFA is a way to represent forms data using XML, which makes it very easy to support form data interchange.

PDFTextStream allows you to access the XML documents that comprise a PDF document's XFA forms, which you can then query or process to meet your specific application requirements. Doing this is very simple, and builds upon PDFTextStream's existing interactive form data API. In the example below, we'll retrieve the XML document (as a byte array) that contains the XFA form's current values:

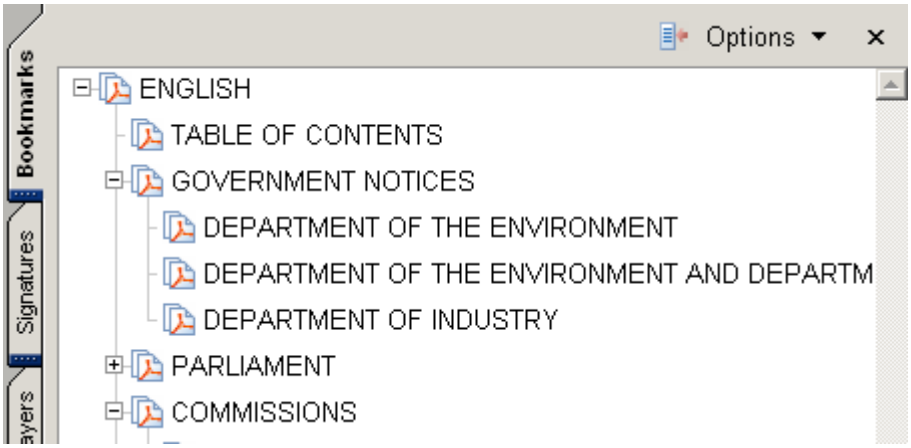
```
public static byte[] getXFADatasets (PDFTextStream stream) throws IOException {
    AcroForm form = (AcroForm)stream.getFormData();
    return form.getXFAPacketContents("datasets");
}
```

Further, we can access the full set of XFA form data in a PDF document using the getXFAContents() method on AcroForm. These values can be fed

into any existing XML libraries or tools to support XFA form data extraction, mapping of the form data to databases, or whatever else your application requires.

Accessing PDF Bookmarks

Some PDF documents contain bookmarks (which are sometimes referred to collectively as a *document outline*) that refer to significant document sections. If a document contains bookmarks, they appear in the 'Bookmarks' panel in Adobe Acrobat:



PDFTextStream allows you to access the bookmarks contained in PDF documents and all of the attributes associated with those bookmarks.

Bookmark Structure and Attributes

PDF bookmarks are organized into a tree structure with a single root node. If a PDF document contains bookmarks, that root node is returned by the `PDFTextStream.getBookmarks()` function as a `com.snowtide.pdf.Bookmark` instance. Each bookmark may contain child bookmarks, accessible using the `Bookmark.getChildCnt()` and `Bookmark.getChild(int)` functions; entire branches of the bookmark tree can also be easily retrieved using the `Bookmark.getAllDescendants()` and `Bookmark.getAllDescendants(java.util.List)` functions.

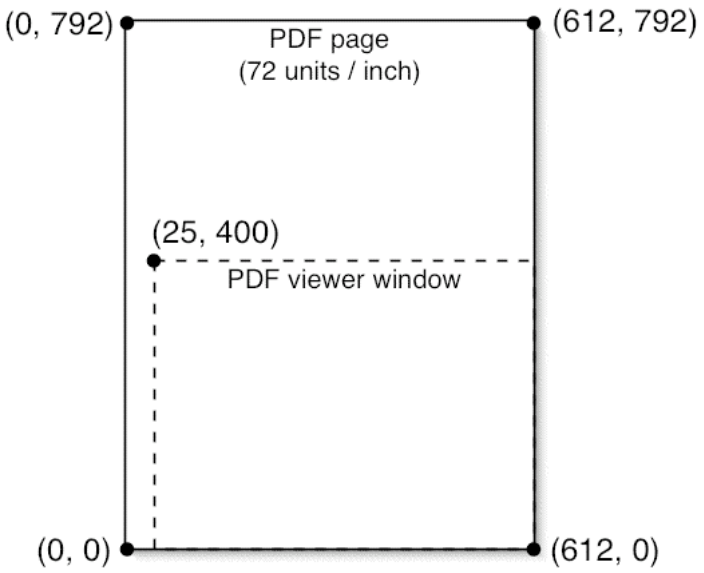
Bookmarks have two main attributes: a title (the text that describes the section to which the bookmark refers) and a page number. These attributes are accessible using the `Bookmark.getTitle()` and the

`Bookmark.getPageNumber()` functions, respectively. All leaf nodes in the bookmark tree should have a page number defined, and many branch nodes may specify a page number as well. It is common for the root node of the bookmark tree to define neither a page number or title. In that case, the `Bookmark.getTitle()` function will return null, and the `Bookmark.getPageNumber()` function with return -1.

Precise Bookmark Positioning

In addition to the page number, some bookmarks will provide specific spatial coordinates, defining where on the target page a PDF viewer should position its viewing window when a user activates a bookmark. These functions (`Bookmark.getTopBound()`, `Bookmark.getLeftBound()`, `Bookmark.getRightBound()`, and `Bookmark.getBottomBound()`) return such coordinates. Many bookmarks will specify only some coordinates, in which case a PDF viewer would orient its viewing window along the defined coordinates, and simply show all of the remaining portions of the target page.

For example, a bookmark referring to page 12 might specify a top bound of 400, a left bound of 25, and undefined right and bottom bounds (values of -1). A PDF viewer would therefore position its viewing window like so:



Having this level of precision available can be very useful, especially when requirements specify the extraction of text from only particular sections of a document. See Appendix B (page 65) for a sample code listing that will extract only a particular section of text from a document based on the precise coordinate bounds specified by a PDF document's bookmarks.

Accessing PDF Annotations

Some PDF documents contain annotations, bits of data that are associated with specific regions of a PDF document's pages. Annotations include:

- Text notes ("stickies")
- Styled text attachments
- Links (referring to a position within the PDF document or to local or network resources)
- File, audio, and video attachments
- Drawings and in-line graphics
- More…

All annotations share a base set of possible attributes; functions for accessing these base attributes are established by the `com.snowtide.pdf.annot.Annotation` interface. These attributes include an annotation's name (typically unique within the page on which the annotation is found), text contents (also used as a description field when the annotation's primary content is non-text, as in a file attachment), and the region on the document's page where the annotation is placed.

PDFTextStream provides richer implementations for three types of annotations: text notes (via the `com.snowtide.pdf.annot.TextAnnotation` class), styled text attachments (via the `com.snowtide.pdf.annot.FreeTextAnnotation` class), and links (via the `com.snowtide.pdf.annot.LinkAnnotation` class). The additional attributes provided by these functions are well documented in the class javadoc; here is a code sample where all of the link annotations are retrieved from a PDF document, and their URI's are printed to standard out:

```
public void printURILinks (PDFTextStream stream) throws IOException {
    Annotation annot;
    LinkAnnotation link;
    String uri;

    for (int i = 0, len = stream.getPageCnt(); i < len; i++) {
        List annots = stream.getAnnotations(i);
        for (int c = 0, clen = annots.size(); c < clen; c++) {
            annot = (Annotation)annots.get(c);
            if (annot instanceof LinkAnnotation) {
                link = (LinkAnnotation)annot;
                if (link.getLinkActionName().equals("URI")) {
                    uri = link.getURI();
                    if (uri != null) {
                        System.out.println("URL link found on page " +
                                (i + 1) + ": " + uri);
                    }
                }
            }
        }
    }
}
```

Note that `LinkAnnotation` instances may also refer to a position within the PDF document using a page number and precise bounding coordinates as some bookmarks do; see page 28.

Reading Encrypted PDF Files

PDFTextStream includes support for decrypting PDF files encrypted with 40- or 128-bit encryption. Using PDFTextStream with such files is almost as easy as using it with unencrypted PDF files.

If it is known that a PDF file is encrypted ahead of time, reading it with PDFTextStream is as simple as providing the file's password (as a byte array) to the appropriate PDFTextStream constructor:

```
public void readPdfFile (File pdfFile, String passwordStr) {

   // convert the password into a byte array
   byte[] password = passwordStr.getBytes();

   // provide the password to PDFTextStream upon creation
   PDFTextStream stream = new PDFTextStream(pdfFile, password);

   // [... use PDFTextStream instance as usual ...]
}
```

Once a PDFTextStream instance has been successfully created using a given password, it can be used normally, without regard to the fact that the file being read is encrypted.

Note that in the case of encrypted PDF files, PDFTextStream's constructors can throw an `EncryptedPDFException` (a subclass of `IOException`). There are a number of reasons why an `EncryptedPDFException` might be thrown by a PDFTextStream constructor; most of them are related to some error in decrypting data contained in a PDF file. However, one reason why such an exception might be thrown is if an incorrect password (or no password) is provided to a PDFTextStream constructor. In this case, an `EncryptedPDFException` with an error type of `EncryptedPDFException.ERROR_BAD_PASSWORD` is thrown.

This is very important in an interactive environment, where the application doesn't necessarily know that a PDF is encrypted, and is relying upon a user to enter the password for any encrypted PDF files it does encounter. In this case, the application should attempt to open each PDF file assuming it is unencrypted, watch for an `EncryptedPDFException` with an error type of `EncryptedPDFException.ERROR_BAD_PASSWORD`, and then prompt the user in an appropriate manner for the password. This code shows an example of this technique:

```
public String readPdfText (File pdfFile, String password)
      throws IOException {
  try {
      PDFTextStream stream;
      if (password == null) {
          // no password, assume the file is unencrypted
          stream = new PDFTextStream(pdfFile);
      } else {
          stream = new PDFTextStream(pdfFile, password.getBytes());
      }

      // [... read PDF text, return resulting string ...]

  } catch (EncryptedPDFException e) {
      if (e.getErrorType() == EncryptedPDFException.ERROR_BAD_PASSWORD) {
          // return null to indicate that a different password is needed
          return null;
      } else {
          // some error in the decryption process
          // treat just like a regular IOException
          throw e;
      }
  }
}
```

Notice that if an `EncryptedPDFException` with an error type of `EncryptedPDFException.ERROR_BAD_PASSWORD` is thrown, then the method returns null. The module calling this method could then appropriately prompt the user for a different password, and then call the method with the new password.

Notice that for other types of `EncryptedPDFException`, the method just rethrows the exception. Those other error types indicate an encryption problem that cannot readily be solved at runtime, including a corrupted or invalid encryption method being used in a PDF file, or the failure of one of the security mechanisms in the JRE that PDFTextStream depends upon in its decryption process.

Error Handling

PDFTextStream is designed to only throw `java.io.IOException` exceptions. This is true when invoking any of PDFTextStream's constructors or other

functions. This is convenient in that, in the simplest cases, you only need to worry about catching `IOException` instances.

However, in a few special cases, PDFTextStream will throw other kinds of exceptions in order to indicate that particular kinds of errors have occurred. Thankfully, each of these exception types subclass IOException, which is helpful in keeping prototyping code simple and clean.

EncryptedPDFException

We saw in the last section that PDFTextStream's constructors can throw `EncryptedPDFExceptions` when an encryption-related error occurs. Please refer to the examples and explanation in the previous section for details on this exception type.

FaultyPDFException

PDFTextStream is also capable of throwing `com.snowtide.pdf.FaultyPDFException` from its contstructors, as well as from most of its other functions that access PDF data. This exception type is thrown when PDFTextStream encounters file data that it doesn't understand. This indicates one of the following:

1. The file in question is not a PDF document
2. The file is a PDF document, but is corrupted or otherwise unusable, and PDFTextStream cannot repair it

Exception Handling Patterns

In production environments, especially when PDFTextStream is being used to extract content from PDF documents sourced from untrusted parties (such as indexing PDF documents found on the internet), handling these exceptions properly is important for proper monitoring of the results of your PDF content extraction efforts.

Below is a typical pattern that is ideal for such environments – it illustrates the pattern that should be used for properly handling each of the three types of exceptions most commonly seen when working with PDFTextStream.

```
public static String extractPDFText (File pdfFile) {
  try {
      PDFTextStream stream = new PDFTextStream(pdfFile);
      StringBuffer sb = new StringBuffer(1024);
      OutputTarget tgt = new OutputTarget(sb);
      stream.pipe(tgt);
      stream.close();
      return sb.toString();
  } catch (EncryptedPDFException e) {
      System.out.println("PDF document (" + pdfFile.getAbsolutePath() +
          ") is encrypted...");
  } catch (FaultyPDFException e) {
      System.out.println("PDF document (" + pdfFile.getAbsolutePath() +
          ") cannot be read because: " + e.getMessage());
  } catch (IOException e) {
      System.out.println("PDF document (" + pdfFile.getAbsolutePath() +
          ") caused general IO error: " + e.getMessage());
  }

  return null;
}
```

Obviously, logging these errors to `System.out` isn't what one would do in production, but the pattern is the same – just insert the appropriate logging or other application-specific routines for handling each type of exception.

Command-Line Operation

PDFTextStream also provides a useful main(String[]) method, allowing it to be used from the command line. The main method can be used with a command of this form:

```
java –cp [classpath] com.snowtide.pdf.PDFTextStream [pdfFile] [optional
outputpath]
```

[classpath]

The classpath needed to reference all of the jars required by PDFTextStream should be provided here. For more information on how to properly construct the classpath, please refer to the classpath portion of the section Setting Up PDFTextStream.

[pdfFile]

This parameter should be a path to the PDF file to be read. It can be a relative path (i.e. ../pdfs/someDocument.pdf) or an absolute path (i.e.

C:\pdfs\someDocument.pdf). If the file path that is provided is found by PDFTextStream to not correspond to an existing file that is readable, then it will output a message to standard out indicating this fact, and exit with an exit code of 1.

[optional outputpath]
This parameter should be a path to which the text read out of the input PDF file should be written. Any file that exists at the given path will be overwritten. If PDFTextStream cannot write to the given path, it will report an error to standard out, and exit with an return code of 1.

If no output path is provided, then the text read out of the input PDF file will be written to standard out directly.

PDF Merging

While PDFTextStream's primary mission has always been (and will continue to be) extraction of PDF content, customer demand persuaded us to add a PDF merge utility. This merge utility will concatenate the content of each of a series of PDF files into a new single PDF file.

Using the merge utility is very simple:

```
public static void mergePDFFiles (File src1, File src2, File destination)
            throws IOException {
   com.snowtide.pdf.util.MergeUtil.mergeDocuments(
       new File[] { src1, src2 }, destination);
}
```

This simple function accepts `java.io.File` references to two source PDF documents, and a `File` reference to where the resulting merged PDF document should be saved.

The merged PDF document content can also be captured in-memory. An example of this is shown here:

```
public static byte[] mergePDFFiles (File src1, File src2) throws IOException {
   java.io.ByteArrayOutputStream dst = new java.io.ByteArrayOutputStream(1024);
   com.snowtide.pdf.util.MergeUtil.mergeDocuments(
       new File[] { src1, src2 }, dst);

   return dst.toByteArray();
}
```

Three things to note:

- `MergeUtil` currently does not include metadata, form data, bookmarks, or annotations found in the source PDF documents when creating the merged destination PDF document.
- Any kind of `java.io.OutputStream` may be passed to the `MergeUtil.mergeDocuments(File[], OutputStream)` function
- Any number of source PDF files can be used as inputs to the merging process (within the limits of your memory resources)

Logging

PDFTextStream is designed to integrate seamlessly into your application's infrastructure, including whatever logging apparatus your application depends upon.

Without any special configuration, PDFTextStream automatically detects and configures itself if one of the following logging toolkits is available:

- JDK v1.4+ `java.util.logging` package
- Log4J

When one of these logging toolkits is used, which levels of log messages are outputted, their format, and their destination are all determined by the logging toolkit configuration. If none of the logging toolkits listed above are available, error information will be written to standard out.

PDFTextStream uses the `java.util.logging` package by default. To force it to use Log4J, simply set the `pdfts.loggingtype` system variable to "log4j" before using PDFTextStream.

Custom Logging Toolkit API

If you use a custom logging toolkit, or a toolkit that is not natively supported, you can register it with PDFTextStream with relative ease. To do so, simply follow this process:

1. Create a class that implements the `com.snowtide.util.logging.LogFactory` interface. This class dispenses `com.snowtide.util.logging.Log` instances to which PDFTextStream sends logging messages.
2. Register your LogFactory implementation with `com.snowtide.util.logging.LoggingRegistry`. This can be done in one of two ways:

- *Before using PDFTextStream*, pass an instance of your `LogFactory` implementation to the `LoggingRegistry.registerFactory(LogFactory)` method.
  OR
- Set the system property `pdfts.logfactory` to the full classname of your `LogFactory` implementation. This can be done when starting your java application using the `-D` switch, or by editing the appropriate settings in your application server configuration.

PDFTextStream creates and gathers all of the `com.snowtide.util.logging.Log` instances it will need at class initialization time, so your `LogFactory` implementation must be registered before then in order for it to be used.

Customizing Logging

Please refer to the documentation related to the logging toolkit your application uses to determine how best to configure logging for the `com.snowtide.pdf` package.

It should be noted that while PDFTextStream presents you (the application developer) with a very simple and clean interface, there are many, many more classes involved in the reading of text content aside from `com.snowtide.pdf.PDFTextStream`. As such, some logging information does occasionally come from those classes, so any changes to logging configuration settings should be made at the package level in order to affect all classes in the PDFTextStream library. (In particular, it is recommended that logging for the `com.snowtide.pdf` package be set at INFO or above, as PDFTextStream and its associated classes log significant diagnostic information at the DEBUG level.)

For example, to change the logging level applied to messages emitted by PDFTextStream regardless of the logging level applied to the rest of your application, you would add these lines to your log4j.properties file (this example is provided for users of Log4J; please check your logging toolkit's documentation on how best to achieve similar effects):

```
log4j.logger.com.snowtide.pdf=ERROR
```

Unicode Text and Character Sets

PDFTextStream supports the extraction of Unicode text from all PDF files, with very few restrictions or caveats with regard to language or character set.

Single-byte character sets, such as those used in conjunction with Roman text (i.e. English, French, Spanish, Italian, German, Dutch, etc.) are fully supported.

Double-byte character sets, such as those used in conjunction with Chinese, Japanese, and Korean (CJK) text is also fully supported (when using the standard PDFTextStream JAR file). Both horizontal and vertical writing modes are recognized and translated into appropriate text extracts.

PDFTextStream usage is unchanged regardless of the type of text being extracted – all character encoding issues are handled automatically.

Controlling CJK Capabilities

PDFTextStream for Java and Python ship with two versions of the PDFTextStream JAR file. One (typically named `PDFTextStream-x.x.jar`, where x.x is the version) contains all of the resources necessary for extracting CJK text. The other (typically named `PDFTextStream-x.x-NOCJK.jar`) does not includes those resources, but is also around 2.5MB smaller than the "full" PDFTextStream JAR file. This can be helpful at times when your deployment environment benefits from minimizing the size of external dependencies.

PDFTextStream actively caches the resources it uses when extracting CJK text – this significant improves performance. However, caching does lead to increased memory consumption. To prevent this, you can turn off PDFTextStream's CJK text extraction capabilities by either using the NOCJK version of the JAR file, or by setting the `pdfts.cjk.enable` system property before using PDFTextStream. Please see page 71 for details.

Future Plans

PDFTextStream does not yet support the extraction of text that uses a right-to-left or bidirectional writing mode. This principally includes Arabic and Hebrew. Support for the extraction of such text is planned for a future PDFTextStream release.

Apache Lucene Integration

Apache Lucene (http://lucene.apache.org) is a full-text search engine written in Java. It is a perfect choice for applications that need 'built-in' search functionality: it's fast, works well with any kind of document structure, and is relatively painless to build around.

Lucene is focused on text indexing, and as such, it does not natively handle popular document formats such as Word, PDF, HTML, etc. Rather, it requires the use of external tools or libraries to convert any such documents into collections of text fields, which can then be easily indexed.

The PDFTextStream includes an easy to use API for integrating it with Lucene, versions 1.2 and later. After setting some configuration parameters, you can easily generate a collection of text fields that Lucene needs for indexing purposes given a PDF file.

The classes relevant to PDFTextStream/Lucene integration are in the `com.snowtide.pdf.lucene` package. Please note that before using any of these classes, a Lucene library jar must be added to the your application's classpath.

PDFDocumentFactory

This class provides a number of static methods that do the real work of bridging the gap between PDFTextStream and Lucene:

```
public static Document buildPDFDocument (File pdfFile) throws IOException;
public static Document buildPDFDocument (File pdfFile,
      DocumentFactoryConfig config) throws IOException;
public static Document buildPDFDocument (InputStream pdfData,
      String pdfName) throws IOException;
public static Document buildPDFDocument (InputStream pdfData,
      String pdfName, DocumentFactoryConfig config) throws IOException;
public static Document buildPDFDocument (ByteBuffer pdfData,
      String pdfName) throws IOException;
public static Document buildPDFDocument (ByteBuffer pdfData,
      String pdfName, DocumentFactoryConfig config) throws IOException;
public static Document buildPDFDocument (PDFTextStream stream,
      DocumentFactoryConfig config) throws IOException;
```

Each method returns an instance of the Lucene Document class, which encapsulates all of the text fields needed by Lucene to index the contents of a document. There is a method corresponding to each constructor in the PDFTextStream class to enable the translation of PDF files into Lucene Documents whether the PDF document is available as a local file or as data waiting to be read from an `InputStream` or `ByteBuffer`.

In general, these methods create a Lucene Document instance by:

- reading all the text out of the PDF file, and setting that content in a named field in the Lucene `Document` instance
- reading all metadata attributes of the PDF file, and building a separate field for each attribute in the Lucene document

Unless a `DocumentFactoryConfig` instance is provided in the call to one of the `buildPDFDocument()` methods (see below for details about `DocumentFactoryConfig`), the fields that are created in the Lucene document take on the defaults provided by the PDF file. For example, the default name of the creation date attribute included in the metadata of some PDF files is 'CreationDate', so that will be the name assigned to the field in the Lucene Document that contains the value of that attribute. The actual text content of a PDF file will be added to the Lucene Document as a field with the name defined in `DocumentFactoryConfig.DEFAULT_MAIN_TEXT_FIELD_NAME` (see the `DocumentFactoryConfig` javadoc for what the literal String value of this constant is).

Allowing these default names to be used for the fields in each Lucene `Document` is convenient, but is probably not what is required; few Lucene indexes will have used those defaults when being built. In order to seamlessly integrate PDFTextStream into your Lucene installation, you will want to customize how the `Document` instances are built. For this, you should use `DocumentFactoryConfig`.

DocumentFactoryConfig

This class is used to optionally control how the static methods of `PDFDocumentFactory` build Lucene Document instances. Typically, a single

`DocumentFactoryConfig` instance will be created and configured for each Lucene index that PDF content needs to be added to -- it would be very wasteful to build up and configure a new `DocumentFactoryConfig` instance for every PDF file that needs to be converted into a Lucene Document.

DocumentFactoryConfig provides a great deal of control over how PDFDocumentFactory operates.

Main Text Field Name (default: DocumentFactoryConfig.DEFAULT_MAIN_TEXT_FIELD_NAME)
The main body of text contained in a PDF file is stored in a Lucene `Document` object as just another named field. This name is set either via the `DocumentFactoryConfig` constructor, or by a setter method on a `DocumentFactoryConfig` instance.

Copy All PDF Attributes (default: true)
By default, all of the attributes found in PDF files processed by `PDFDocumentFactory` will be copied into fields in the resulting Lucene Documents. However, in many circumstances, only a subset of the attributes contained in a PDF file will be relevant to the index to which its content will be added. In this case, you can change this property to false, allowing only those PDF file attributes that have been explicitly mapped via `DocumentFactoryConfig.setFieldName(String, String)` to be added to the Lucene Document instances.

Custom Field Name Mappings
PDF document attributes are simple name/value pairs. By default, `PDFDocumentFactory` will add these name/value pairs directly to the Lucene Documents it generates. For example, a PDF file might contain these attributes:

Attribute Name	Attribute Value
Creator	Microsoft Word
Author	Kate Burneson
CreationDate	Mar 30, 2002 08:12:44 AM -0800

This is a problem if this example PDF file's content is to be added to a Lucene index that has document author fields named 'authored_by' and creation time/date stamps named 'create_dt'. Fortunately, this is easily fixed by using this code with a `DocumentFactoryConfig` instance:

```
DocumentFactoryConfig config = new DocumentFactoryConfig();
config.setFieldName(PDFTextStream.ATTR_AUTHOR, "authored_by");
config.setFieldName(PDFTextStream.ATTR_CREATION_DATE, "create_dt");
```

This will cause any invocation of a `PDFDocumentFactory.buildPDFDocument()` method that includes the 'config' object to build a Lucene `Document` instance that uses the name 'authored_by' for any 'Author' PDF metadata attribute, and 'create_dt' for any 'CreationDate' attribute. Note that the most common PDF document attributes have standardized names, which are fixed as static final constants in the PDFTextStream class. All such constant fields in the PDFTextStream class have an 'ATTR' prefix to identify them as standard document attribute names.

Storing vs. Indexing vs. Tokenizing

Fields in every Lucene document have three attributes associated with them, typically referred to as 'store', 'index', and 'token'. These attributes control how the Lucene indexing engine processes each field when it is added to an index as a part of a `Document` instance (a full discussion of these attributes and how they impact Lucene indexing and searching is beyond the scope of this guide; please refer to Lucene's documentation for more information).

The values to be used for 'store', 'index', and 'token' when creating named fields in Lucene Documents can be set for PDF document attributes via `DocumentFactoryConfig.setPDFAttrSettings(boolean, boolean, boolean)`. The values provided to this method are used for all fields created for PDF document attributes. The defaults for these settings are true, true, true.

The values for 'store', 'index', and 'token' for the main body of text read out of PDF files can be set via `DocumentFactoryConfig.setTextSettings(boolean, boolean, boolean)`.

The defaults for these settings are `false`, `true`, `true`.

PDFTextStream for .NET and Python

Starting with v2.0, PDFTextStream is available for .NET and Python versions in addition to its standard Java distribution. Although PDFTextStream is written and maintained in Java, a number of highly innovative tools are now available that enable Java libraries like PDFTextStream to be used in .NET and Python environments with no loss of functionality, reliability, or performance. We make sure of this by subjecting PDFTextStream.NET and PDFTextStream.Python to the same level of intense, abusive testing that made PDFTextStream for Java into what it is today.

PDFTextStream.NET and PDFTextStream.Python match the functionality offered by PDFTextStream for Java blow for blow. Support for extracting bookmarks, annotations and hyperlinks, interactive form data, document properties, and of course text are all available on Java, .NET, and Python.

The following sections detail how to configure and use PDFTextStream for .NET and Python. First, a few common notes:

- The PDFTextStream API is the same across all platforms. Given PDFTextStream's Java roots, the PDFTextStream javadoc is the authoritative API reference. This can be thought of as similar to C/C++ libraries that have bindings for different higher-level environments (such as wxWidgets, for example), but the C/C++ documentation is the primary reference for all platforms.
- All PDFTextStream text extracts use Unix-style line breaks (using a single linefeed character `\n`), regardless of operating system or application platform.

PDFTextStream.NET

Background

Whereas PDFTextStream.Python requires you to manage a secondary JVM instance within the Python virtual machine, PDFTextStream.NET is a .NET assembly through and through, and does not entail running a separate environment within the .NET process. This is possible because .NET and Java are very, very similar architecturally, and with regard to their respective object models. This makes translating a Java JAR file into a .NET assembly a straightforward process, at least conceptually.

Requirements and Architecture

PDFTextStream.NET is produced by translating the standard PDFTextStream JAR file into a pure .NET assembly using IKVM (http://www.ikvm.net). IKVM is an open source toolkit that makes it possible to run Java applications and libraries within the .NET environment. In PDFTextStream's case, IKVM translates the compiled Java bytecode into .NET IL bytecode. This preserves PDFTextStream's Java API's, architecture, functionality, and performance, and delivers it for the .NET platform.

(IKVM and the included GNU Classpath library both use a license that makes it possible to redistribute them with commercial products. Therefore, licensing PDFTextStream.NET for inclusion in your own product on an OEM basis is perfectly straightforward and poses no threat to your product's license.)

PDFTextStream.NET requires v1.1 or higher of the .NET framework. The library itself consists of three DLL files, all of which are found in the lib directory of the PDFTextStream.NET distribution:

- PDFTextStream.dll
- IKVM.GNU.Classpath.dll
- IKVM.Runtime.dll

The IKVM DLL files are PDFTextStream.NET's only dependencies. They provide the implementation of Java's standard library in .NET, as well as

some runtime components that are required by any Java JAR that has been translated into a .NET assembly. No configuration or special initialization of these DLL files are necessary.

Installation

Using PDFTextStream.NET within your .NET project is as simple as adding references to each of the three DLL files indicated in the previous section.

Typical Usage

Using PDFTextStream.NET is very straightforward, and mirrors typical PDFTextStream for Java usage. Here's a sample text extraction function in C#:

```
using com.snowtide.pdf;

namespace DotNetExampleFunction
{
  class ExampleFunction
  {
      static string extractPDFText (string pdfFilePath)
      {
       java.lang.StringBuffer sb = new java.lang.StringBuffer(1024);
       OutputTarget tgt = new OutputTarget(sb);

       PDFTextStream stream = new PDFTextStream(java.io.File(pdfFilePath));
       stream.pipe(tgt);
       stream.close();

       return sb.toString();
      }
  }

}
```

All of the PDFTextStream API is available in .NET, including support for extracting bookmarks, annotations and hyperlinks, interactive form data, and document properties. As was mentioned previously, the PDFTextStream javadoc is the authoritative reference for PDFTextStream.NET usage.

Notes and Limitations

There are no limitations to speak of. PDFTextStream.NET is a pure .NET assembly, through and through, and it acts like it.

One significant advantage of this is that one may write OutputHandler implementations in .NET (PDFTextStream.Python does not support this). Here is a trivial example for illustration that will count the number of characters extracted from a PDF:

```
namespace SubclassingExample
{
    class CharCountingTarget : com.snowtide.pdf.OutputTarget
    {
        private int cnt = 0;

        public CharCountingTarget (java.lang.StringBuffer sb) : base(sb)
        {
        }

        public override void textUnit (com.snowtide.pdf.layout.TextUnit tu)
        {
            base.textUnit(tu);
            cnt++;
        }

        public int getCount ()
        {
            int _cnt = cnt;
            cnt = 0;
            return cnt;
        }
    }
}
```

An `OutputHandler` (or `OutputTarget`, in this case) subclass like this can be used in conjunction with any `pipe(OutputHandler)` function, found on instances of the `com.snowtide.pdf.PDFTextStream`, `com.snowtide.pdf.Page`, and `com.snowtide.pdf.layout.Block` classes.

PDFTextStream.Python

Background

Python is architecturally quite different than Java in many ways. Therefore, using a Java library like PDFTextStream from within a Python application requires some bridge-building. While many "compatibility layers" have often disappointed in the past (especially with regard to performance), the integration approach embodied in PDFTextStream.Python results in Python developers being able to use PDFTextStream.Python as if it were a native Python module. Therefore, it is comprehensive, high-fidelity, and comes without significant developer effort, performance compromises, or API complications.

Requirements and Architecture

Using PDFTextStream.Python requires Python 2.3 or higher, and the JPype Python module (http://jpype.sourceforge.net). JPype is an open source module that allows a Python process to transparently utilize Java components by embedding a Java virtual machine (JVM) instance within the host Python process. (JPype is licensed under the liberal Apache License v2.0, making it possible to redistribute it with commercial products. Therefore, licensing PDFTextStream.Python for inclusion in your own product on an OEM basis is perfectly straightforward and poses no threat to your product's license.)

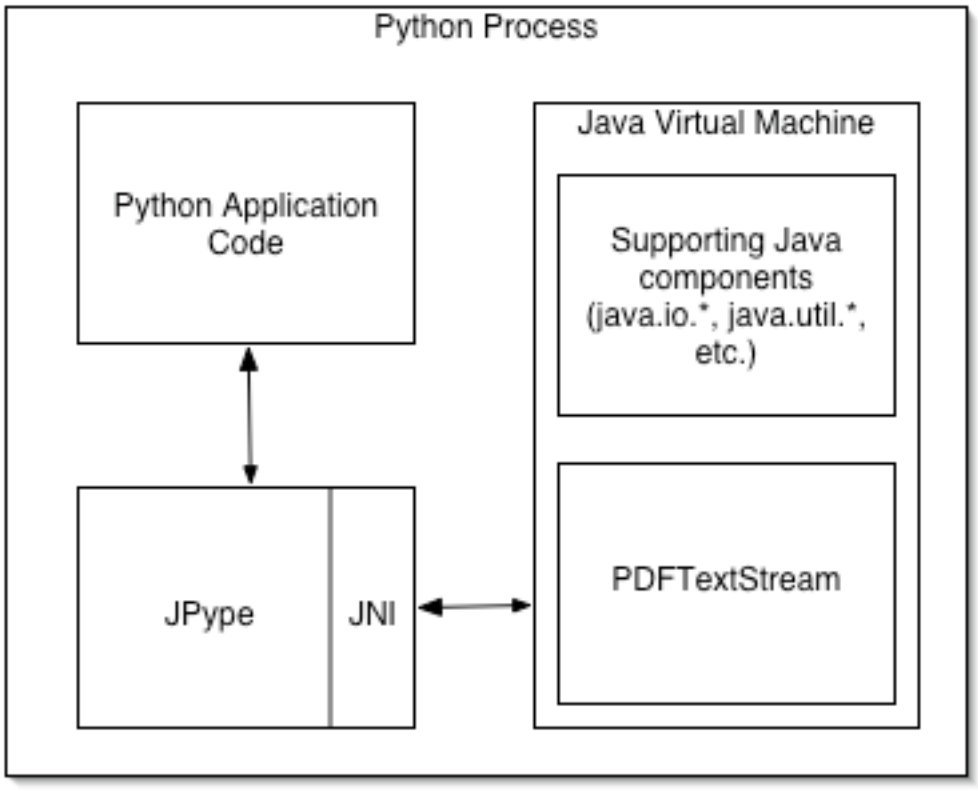


Figure 1: PDFTextStream.Python integration architecture.

Figure 1 depicts the architecture involved when utilizing PDFTextStream.Python. Conceptually, this architecture is very straightforward:

- Your Python application code requests JPype to initialize a JVM, which uses JNI to do so
- Python application code then imports and uses classes and functions available from the JVM, including PDFTextStream as well as core library components
- At all times, classes, function calls, and return values are transparently converted to native Java invocations and back by JPype and JNI

Installation

All PDFTextStream.Python distributions include the supported version of JPype (usually named JPype-x.x.x.x.zip, and available in the `lib` directory of your PDFTextStream.Python distribution). The JPype module must be

compiled and installed into your python environment before it can be used. To do so, simply navigate to the JPype root directory (after expanding its zip archive), and execute the install script:

```
python setup.py install
```

This should be all you need to do for Linux and Mac OS X, and most Windows environments. Some Windows systems may need an additional step; see page 57 for details.

Typical Usage

Using JPype to access PDFTextStream is very straightforward. This section will step through the example python script `getting_started.py` to demonstrate a typical usage pattern (note that some specific pointers related to implementation details are included in code comments in the example script, located in the `examples` directory in the PDFTextStream.Python distribution).

```
from jpype import java, startJVM, JPackage, getDefaultJVMPath, nio
import sys, os, codecs
```

These are some basic imports needed to get things started. The JPype imports are of special interest:

- `startJVM`: a function that initializes the JVM, and binds it to the current python process; its usage is detailed below
- `getDefaultJVMPath`: a function that aids in the location of the current environment's JVM installation
- `JPackage`: JPype's proxy class for java packages
- `jpype.java`: a `JPackage` instance representing the `java` package. JPype creates `JPackage` proxies for the `java` and `javax` packages automatically.
- nio: JPype module for exposing python strings as java `ByteBuffers` (allows for fully in-memory operation of PDFTextStream)

```
if len(sys.argv) < 3:
    sys.exit('Usage: python pdfts_demo.py path_to_pdfts_lib pdf_file
output_path')

pdfts_lib = sys.argv[1]
infile = sys.argv[2]
outfile = sys.argv[3]
```

Some simple testing and unpacking of command-line arguments for clarity, including the path to a PDFTextStream jar file, which will be used as the classpath when starting the Java VM.

```
classpath = '-Djava.class.path=' + pdfts_lib + os.pathsep + '.'
startJVM(getDefaultJVMPath(), '-Xmx512m', classpath)
```

This is where the JVM creation magic occurs. `startJVM` takes 1 or more arguments. The first is the absolute path to your system's JVM. The JPype function `getDefaultJVMPath()` can usually determine this automatically on Mac OS X and Windows. If for some reason it is unsuccessful, or if you want to explicitly specify which JVM to use, or you are using some flavor of Linux, then the following paths would be appropriate:

- On Windows: the path to the JVM's jvm.dll file, i.e. `C:\tools\j2sdk\jre\bin\client\jvm.dll`
- On Linux: the path to the JVM's libjvm.so file, i.e. `/usr/java/j2sdk1.4.2_04/jre/lib/i386/server/libjvm.so`
- On Mac OS X: the path to the JavaVM file (usually `/System/Library/Frameworks/JavaVM.framework/JavaVM`)

Note that on Linux, your `LD_LIBRARY_PATH` environment variable must include the paths that contain the JVM's shared libraries. For example, if your Linux Java installation has placed the `libjvm.so` file at `/usr/java/j2sdk1.4.2_04/jre/lib/i386/server/libjvm.so`, your `LD_LIBRARY_PATH` environment variable would include `/usr/java/j2sdk1.4.2_04/jre/lib/i386/:/usr/java/j2sdk1.4.2_04/jre/lib/i386/server/`. Windows and Mac OS X environments typically require no library path modifications.

Additional parameters provided to `startJVM` are taken as arguments into the new JVM. In the example above, a maximum heap size directive (`-Xmx512m`) and classpath are provided. Note that JPype does not support defining multipart JVM arguments (such as '`-classpath /some/path/here`') – this is why the classpath in this example is defined using the `java.class.path` system property directive. Also note that PDFTextStream's license file's parent directory must be included in the classpath, or PDFTextStream will operate with significant limitations.

```
pdfts = JPackage('com.snowtide.pdf')
```

This binds a `JPackage` instance representing the root of the `com.snowtide.pdf` package to the name `pdfts`. All of PDFTextStream's core classes reside in the `com.snowtide.pdf` package, and will be readily accessible from this `JPackage` instance.

```
sb = java.lang.StringBuffer(1024)
tgt = pdfts.OutputTarget(sb)
```

These lines sets up the `com.snowtide.pdf.OutputTarget` instance that will be used to pipe extracted text content from PDFTextStream to the created `java.lang.StringBuffer`.

```
stream = pdfts.PDFTextStream(java.io.File(infile))
stream.pipe(tgt)
stream.close()
```

This code does the actual work of creating a `com.snowtide.pdf.PDFTextStream` instance to process the file specified on the command line, and piping the extracted text to the `OutputTarget` that was created earlier.

```
txt = sb.toString()
```

This line brings the text extracted from the input PDF document (and then deposited into the `StringBuffer` by the `OutputTarget`) into the python

environment; the `StringBuffer.toString()` call will result in a python `unicode` string.

Notes and Limitations

No (easy) Default Package Access

Normally, gaining access to a reference to a Java class through JPype is very straightforward:

```
pdftsPackage = JPackage('com.snowtide.pdf')
PDFTextStream = pdftsPackage.PDFTextStream
```

This loads the `com.snowtide.pdf` package, and then sets a reference to the `com.snowtide.pdf.PDFTextStream` class.

However, accessing Java classes that reside in the default package is not as straightforward (all classes that do not have a package specification as the first first line in their source file are placed into the default, unnamed package). Here is how one would gain a reference to a Java class called 'MyTestClass' in the default package:

```
from jpype import java
MyTestClass = java.lang.Class.forName('MyTestClass', True,
    java.lang.ClassLoader.getSystemClassLoader())
```

This code first brings the `JPackage` instance that refers to the top-level java package into the local namespace. Then it uses Java's reflection to load the `MyTestClass` class using the default system classloader (which loads classes from the classpath specified when the JVM was started).

This approach works fine, but is far from elegant. Therefore, it is recommended that any Java classes that need to be referenced (such as custom `com.snowtide.pdf.OutputHandler` implementations) should be defined within a named package.

Python Thread Attachment

For threads to utilize a loaded JVM through JPype, they must call the `jpype.attachThreadToJVM()` function before attempting to call into the JVM. Here is some code that exemplifies the appropriate idiom to use; this checks for a running JVM, and checks that the current thread is not yet attached to the JVM before attaching the current thread:

```
if jpype.isJVMStarted() and not jpype.isThreadAttachedToJVM():
    jpype.attachThreadToJVM()
```

Not attaching each python thread to the JVM before calling into Java will result in an error. The thread that first creates the JVM does not need to be attached to it.

Cannot Subclass Java Classes

JPype does not yet support the creation of python classes that subclass Java classes or implement Java interfaces. This means that, for example, custom `com.snowtide.pdf.OutputHandler` implementations must be written in Java, and then referred to from within python in the same way that PDFTextStream.Python itself is accessed.

It is also reasonable to rewrite python text-processing code in Java when performance is a concern; this is analogous to the standard practice of rewriting performance-sensitive python modules in C. For example, if your application requires that a conversion process be performed on text extracted from PDF documents, you could write an initial solution in python. If that solution proves to be too slow for some reason, you can rewrite it in Java, and use the new Java-based conversion process from python in the same way that PDFTextStream.Python is accessed.

Troubleshooting

Windows

Attempting to use JPype raises a python ImportError (usually at the first import, i.e. `from jpype import *`), and displays a dialog reading "This application has failed to

start because MSVCP71.dll was not found. Re-installing the application may fix this problem."
Most Windows systems have this C++ runtime library installed already; unfortunately, neither the standard python installer nor the jpype distribution install it.

If you are presented with this error, simply copy the `MSVCP71.dll` file found in the lib directory in the PDFTextStream.Python distribution to one of the following locations:

- the `\Windows\system32` directory
- your current directory (where you are starting the python interpreter)

Python will then find the DLL, and JPype will function normally.

Appendix A – The Art of Reading PDF Text

This appendix is provided solely for those interested in some of the technical issues involved in reading text out of PDF files.

The Portable Document Format, invented by Adobe in 1993, is a document file format derived from Postscript, a control language used by most laser printers today. The PDF specification was created primarily to allow a single document to be reliably displayed or printed using a shared and consistent set of graphical instructions.

The PDF format has proven to be a successful and useful technology, given the ease with which professional-grade documents may be exchanged among computing platforms and the rapid pace at which the PDF file format has been accepted by the marketplace.

PDF files are structured to encode display-oriented information, such as where a particular image should be placed, or how wide the letter 'a' should be, etc. This makes for a very robust and powerful publishing and display technology, but it makes it very difficult to reliably read text out of PDFs.

For example, consider this sample text that could exist in any PDF document:

Hello there

Hello there

First, notice that in the line of larger text, the space between the words is around double the size of the space between the words in the smaller line or text. If those spaces were encoded in the PDF file explicitly, then reading that text out of the document would be very simple. However, many PDF files are written with instructions like these:

[(Hello)-1650(there)]TJ
[(Hello)-3500(there)]TJ

which roughly translates to these operations:

<print "Hello"><move .25" to the right><print "there">
<print "Hello"><move .55" to the right><print "there">

Notice that there is no actual space character in these instructions – there's just an instruction indicating that after the word 'Hello' is printed, the word 'there' should appear some distance to the right (.25" in one case, .55" in the other). The question is, how many spaces should be included in a text representation of that content? Maybe .25" is one space, and .55" is two. Perhaps .25" is really two spaces, and .55" is four. Just maybe, the first line's font size is much smaller than the second, so the differing amounts of space between the words really represents only one space for both lines (this is the case in our example here).

A related example that is even more difficult to process concerns justified text, as shown below:

> We hold these truths to be self-evident, that all men are created equal, that they are endowed by their Creator with certain unalienable Rights, that among these are Life, Liberty and the pursuit of Happiness.

Here, the problem of determining what is and is not a "real" space is compounded by two issues:

- The spacing between words in the one line has no relationship between how wide a space should be between words in that font, at that font size.
- There is no relationship between the width of an actual space in one line and the width of an actual space in another line, even though both lines use the same font, at the same font size.

It should be obvious that making a determination of where spaces should go and how many should be outputted when converting such content to plain text is very difficult.

One final example will illustrate even more difficult obstacles in converting a PDF document into text. Consider this example page layout:

Main text, column 1

Main text, column 2

Footnote section

There are a couple of approaches available for extracting the text out of a PDF document that has this kind of common layout.

The first is to output text in a way that matches the layout exactly. This would result in (for example) the first line of each column being on the same line of plain text, the seconds lines in each column being on the same second line of plain text, etc. This would be a disaster for any application that processed the resulting text in an automated way (for indexing, searching, or summarization purposes, for example). This is because such automated processing would have no way to know that content from completely different sections of the document is present on the same lines of text. This could lead to a range of problems, including nonsensical summaries and indexes keying on phrases that never really existed in the original document.

The second approach is to output the columns in order, resulting in the outputted text losing all visual correspondence with the original document's layout. This is an improvement (at least with regard to the viability of the outputted text in an automated processing environment), in that the text of a column is uninterrupted by the text that actually belongs in other columns. However, there is still an issue to overcome: how would such a method know that the footnote section at the bottom-left of the page shouldn't interrupt the main body text in columns one and two? After all, the footnote section is aligned in the second column, and without the benefit of human intelligence, a text extraction process might assume that the footnote section *is* part of the first column.

This is just a small taste of the complexities involved in reading useful, accurate text out of PDF files. This is compounded by innumerable variations between PDF files (for example, some may encode text backwards; a text extraction library that doesn't handle such things properly may end up outputting 'there Hello'). PDFTextStream employs hundreds of specialized, intelligent processes to figure out how best to handle these difficulties, and provide a high-quality, accurate text extract of whatever PDF files your application needs to process. By no means are these processes

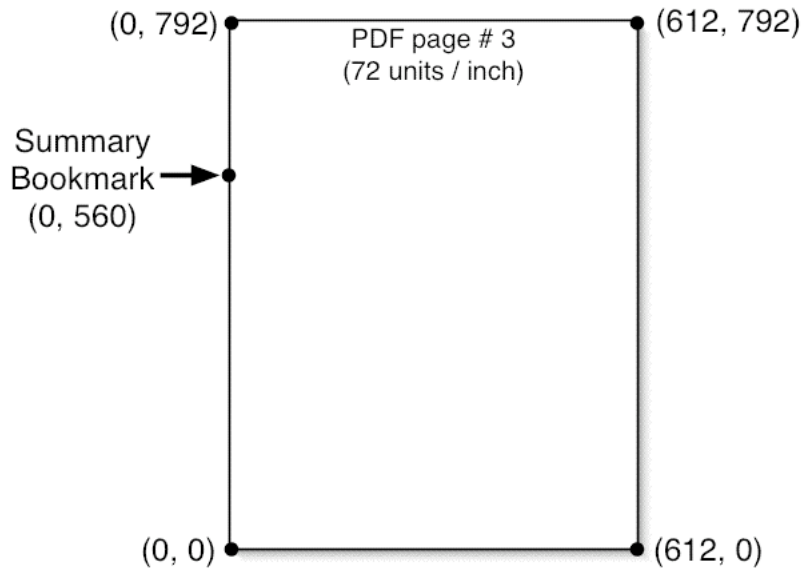
perfect. However, we are confident that they are the best available, and that we will continue to improve and perfect them to provide your applications with the best text output possible.

Appendix B – Selective Text Extraction Based on Bookmark Coordinates

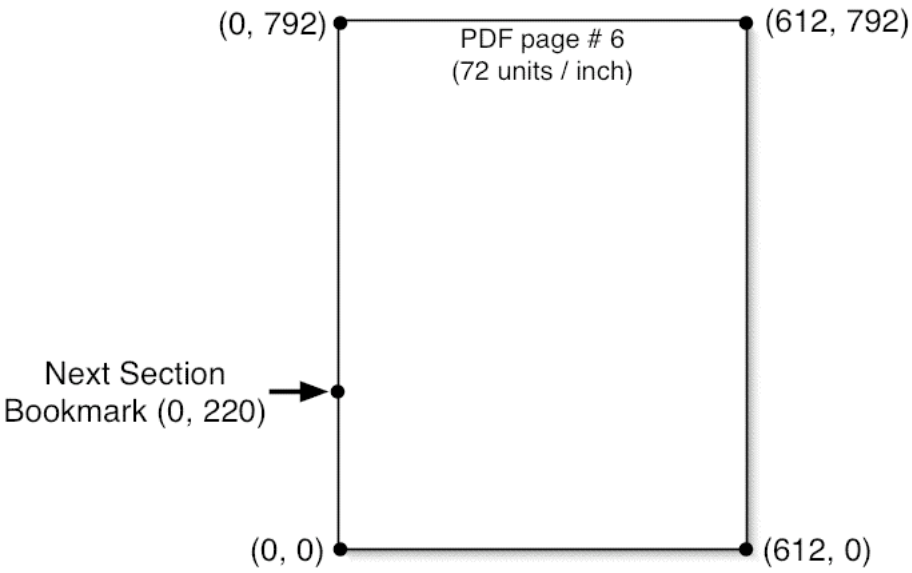
This code sample uses PDFTextStream's bookmark capabilities to selectively extract text from PDF documents using specific spatial coordinates provided by the documents' bookmarks.

Scenario: consider a collection of thousands of PDF documents, all following a particular format. For some reason, you only want to extract the text of a particular section – perhaps the summary, which would make a good set of inputs for indexing. The problem is that the summary does not start on the same page in every document, and it is of varying lengths in every document.

However, all of the documents do have bookmarks, and through experimentation on a few of them, you have found that their bookmarks specify accurate top bound coordinates (the vertical coordinate where the bookmark is positioned, indicating the start of the corresponding section). For example, one of the documents has the bookmark for its summary section referring to the third page of the document, with a top bound (accessible using the `Bookmark.getTopBound()` function) of 560:



That neatly gives us the location of the start of the summary section, but doesn't help with where the section ends. For that, we simply look at the bookmark that follows the summary bookmark; our example document has the next bookmark referring to page 6, with a top bound of 220:



So, now we know that the summary section for this particular PDF document runs from page 3, y-coordinate 560, to page 6, y-coordinate 220. Extracting only the text from those pages and between those coordinates is pretty easy:

1. Create a PDFTextStream instance for a document.
2. Extract all of the bookmarks in the document.
3. Order those bookmarks according to the positions they refer to in the document.
4. Find the bookmark corresponding to the start of the section of interest. (The sample code uses the bookmark title to determine this. The method that your applications uses might be different, such as the position of a bookmark in the bookmark tree hierarchy if a particular document type is very consistent in its structure from edition to edition).
5. Find the next bookmark, which will indicate the end of the section of interest.

6. Access each of the pages between the page numbers indicated by the bookmarks being used (inclusive).
7. For the first page in that range, remove all of the `com.snowtide.pdf.layout.Block` instances above the top bound specified by the first bookmark; this eliminates the text before the start of the section.
8. For the last page in the range, remove all of the `Block` instances below the top bound specified by the second bookmark; this eliminates the text after the end of the section.
9. For each of the pages in the range (and after removing blocks from the first and last pages as necessary), use the `Page.pipe(OutputTarget)` function to extract the text of the section.

The code sample below implements this approach, with some minor embellishments to handle boundary cases, such as if there is no bookmark following the section of interest.

```
import com.snowtide.pdf.Bookmark;
import com.snowtide.pdf.OutputTarget;
import com.snowtide.pdf.PDFTextStream;
import com.snowtide.pdf.Page;
import com.snowtide.pdf.layout.Block;
import com.snowtide.pdf.layout.BlockParent;

import java.io.*;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;


public class ExtractBookmarkedSection {

    /**
     * Extracts from the given PDF file only the text from the section that
     * is delimited by a PDF Bookmark with the given section title.
     */
    public static String extractSectionText (File pdffile, String sectionTitle)
                    throws IOException {
        PDFTextStream stream = new PDFTextStream(pdffile);
        Bookmark root = stream.getBookmarks();

        List allbookmarks = root.getAllDescendants();
        Collections.sort(allbookmarks, new DocumentOrderBookmarkComparator());

        Bookmark bm;
```

```
        int startpage, endpage;
        float starttop, endtop;
        starttop = endtop = startpage = endpage = -1;

        for (int i = 0, len = allbookmarks.size(); i < len; i++) {
            bm = (Bookmark)allbookmarks.get(i);
            if (bm.getTitle().equals(sectionTitle)) {
                startpage = bm.getPageNumber();
                starttop = bm.getTopBound();

                if (i + 1 < len) {
                    bm = (Bookmark)allbookmarks.get(i + 1);
                    endpage = bm.getPageNumber();
                    endtop = bm.getTopBound();
                }

                break;
            }
        }

        // couldn't find section start from title
        if (startpage == -1) return null;

        // handle when we're extracting the last bookmarked section
        if (endpage == -1) endpage = stream.getPageCnt() - 1;

        Page page;
        StringBuffer sb = new StringBuffer(1024);
        OutputTarget tgt = OutputTarget.forBuffer(sb);

        for (int i = startpage; i <= endpage; i++) {
            page = stream.getPage(i);
            if (i == startpage && starttop != -1) {
                // remove all blocks above bookmark,
                // if bookmark bound is defined
                removeBlocksAbove(page.getTextContent(), starttop);
            } else if (i == endpage && endtop != -1) {
                // remove all blocks below end bookmark,
                // if bookmark bound is defined
                removeBlocksBelow(page.getTextContent(), endtop);
            }

            page.pipe(tgt);
        }

        stream.close();

        return sb.toString();
    }

    /**
     * Removes all of the child blocks within the given BlockParent instance
```

```
     * that are positioned above the given y-coordinate position.
     */
    private static void removeBlocksAbove (BlockParent blocks, float pos) {
        Block b;
        for (int i = blocks.getChildCnt() - 1; i > -1; i--) {
            b = blocks.getChild(i);
            if (b.ypos() >= pos) {
                blocks.removeChild(i);
            } else {
                removeBlocksAbove(b, pos);
            }
        }
    }

    /**
     * Removes all of the child blocks within the given BlockParent instance
     * that are positioned below the given y-coordinate position.
     */
    private static void removeBlocksBelow (BlockParent blocks, float pos) {
        Block b;
        for (int i = blocks.getChildCnt() - 1; i > -1; i--) {
            b = blocks.getChild(i);
            if (b.endypos() <= pos) {
                blocks.removeChild(i);
            } else {
                removeBlocksAbove(b, pos);
            }
        }
    }

    /**
     * Orders the Bookmarks within a List according to where they refer within
     * the document (technically, bookmarks can refer to any page, any
     * location, and not necessarily be in a typical reading order within the
     * tree).
     */
    private static class DocumentOrderBookmarkComparator implements Comparator
{
        private Bookmark b1, b2;

        public int compare (Object o1, Object o2) {
            b1 = (Bookmark)o1;
            b2 = (Bookmark)o2;

            if (b1.getPageNumber() < b2.getPageNumber()) {
                return -1;
            } else if (b1.getPageNumber() > b2.getPageNumber()) {
                return 1;
            } else {
                if (b1.getTopBound() < b2.getTopBound()) {
                    return -1;
                } else if (b1.getTopBound() == b2.getTopBound()) {
```

```
                    return 0;
                } else {
                    return 1;
                }
            }
        }
    }
}
```

Appendix C – PDFTextStream System Properties

PDFTextStream uses system properties to handle certain configuration tasks. Each of the following system properties must be set before referencing PDFTextStream in any way, as the properties are checked and their values (if any) are acted upon when PDFTextStream is statically initialized. Therefore, the safest way to use these configuration-related system properties is to set them when starting your application:

```
java –cp [classpath] –Dpdfts.config.property=value your.main.classname
```

You can also set system properties in your code as long as you do so before your first usage of PDFTextStream. In Java:

```
System.setProperty("pdfts.config.property", "config_value");
PDFTextStream stream = new PDFTextStream(new File("c:\some\path.pdf"));
```

In python:

```
from jpype import java, JPackage
java.lang.System.setProperty("pdfts.config.property", "config_value")
pdfts = JPackage('com.snowtide.pdf')
stream = pdfts.PDFTextStream(java.io.File('/some/path'))
```

In .NET:

```
using com.snowtide.pdf;
java.lang.System.setProperty("pdfts.config.property", "config_value");
PDFTextStream stream = new PDFTextStream(new java.io.File("c:\some\path.pdf"));
```

PDFTextStream.NET users can also set these properties the app.config file, which is equivalent to the Java convention of specifying system properties on the command line using the `-D` options (note the `ikvm:` prefix, which exposes the property to the Java namespaces):

```
<?xml version="1.0"?>
<configuration>
  <appSettings>
    <add key="ikvm:pdfts.config.property" value="config_value" />
  </appSettings>
</configuration>
```

pdfts.cjk.enable

Setting this system property to "N" will disable PDFTextStream's ability to extract Chinese, Japanese, or Korean (CJK) text. This may be desirable if memory utilization is a concern – CJK character maps are very large, and can consume significant amounts of memory. As always, application profiling is recommended to determine the actual source(s) of memory consumption.

pdfts.mmap.enable

Due to an unfortunate bug in Java's implementation of memory-mapped files in Windows environments (see http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4724038), it is possible that a PDF file opened and processed by PDFTextStream will remain locked even after the PDFTextStream instance's close() function has been called, and PDFTextStream has released all of the filesystem handles it has allocated. This locking behaviour (which is known to occur only on Windows) will prevent the PDF file from being deleted or moved until Java's garbage collector eliminates certain JDK-internal objects that are used to track and manage the previously memory-mapped PDF file.

The solution in this case is to force PDFTextStream to not memory map source PDF files. This is done by setting the `pdfts.mmap.enable` system property to "N".

pdfts.logfactory

PDFTextStream defaults to using `java.util.logging` or Log4J for logging informational and error messages. However, many environments demand customized logging frameworks. Therefore, PDFTextStream provides a pluggable logging architecture that enables you to hook your custom logging

framework into PDFTextStream. To do so, simply implement the `com.snowtide.util.logging.LogFactory` interface, and set the `pdfts.logfactory` system property to the full classname of your implementation.

pdfts.loggingtype

PDFTextStream normally defaults to using the `java.util.logging` logging framework (available in Java v1.4+). To force PDFTextStream to default to using Log4J, set the `pdfts.loggingtype` system property to "log4j".