Datalogics

Working with the Java and .NET Interfaces

Adobe PDF Library Java and .NET Programming Guide

Version 1.0

Updated 8/15/2013

For additional information, contact:

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

Table of Contents

Introduction

The Adobe PDF Library (APDFL) is an Application Programming Interface (API) designed to allow programmers to work with the Adobe PDF file format. The APDFL Software Development Kit (SDK) provides a method for software developers and vendors to build their own third-party systems that allow them to create, change, process, review, and otherwise work with PDF files. The tools in the library are based on the Adobe Acrobat software but will run independently, so that a copy of Adobe Acrobat does not need to be installed.

The Datalogics Extension (DLE) is a Java and .Net interface to the Adobe PDF Library. DLE offers a set of modules for the APDFL that allow programmers working in Java, or C# or other languages supported by Microsoft .Net to take advantage of the APDFL tools and resources. DLE encapsulates the original Adobe PDF Library; the DLE interface allows you to work with the original core library functions directly, and seamlessly, in Java or .Net.

Some scenarios for building programs in Java and .Net

You are a programmer, working for a firm that needs to create, manage, and work with PDF files. You need to add code to one of your internal applications or to a product that you plan to resell that will generate a set of PDF output files, or that draws data from a set of PDF files.

We offer some below some scenarios that describe ways to use the Java and .Net programming interfaces to the Adobe PDF Library.

Basic PDF functions

We start with some of the most common functions involved in working with PDF files. This will also allow us to look at some of the basic concepts in working with the Adobe PDF Library.

- Opening a file
- Saving a file
- Opening a PDF file with a password

Opening a file

To open a PDF file use the new Document constructor:

```
Document doc = new Document("samples\data\sample.pdf");
```

You can define the file name to open as a separate string value:

```
string filename = "samples\data\sample.pdf";
Document doc = new Document(filename);
```

In each case, however, the program defines the name of the file to open, and in this example, a partial path, in quotation marks. The sample PDF files used by the programs are stored in the directory:

C:\dotnet_and_java:\data\samples

For Java, opening a file is similar:

```
String filename = "Samples\Data\sample.pdf";

Document doc = new Document(filename);
      System.out.println("Opened a document " + filename);
```

Saving a file

Likewise, many of our sample programs feature C# and Java instructions to save data to a new PDF file, including Watermark, ChangeLayerConfiguration, and ListInfo.

The Watermark program prompts the user to enter the name of a PDF file holding the watermark graphic needed. It applies this graphic to the original document, and saves a new PDF file with the result.

The Document.Save method saves the content to a new PDF file.

```
doc.Save(SaveFlags.Full, filename);
```

Opening a file with a password

This code shows how to prompt for a password to open a PDF file, and then validate the password when entered. It is drawn from the sample program DisplayPDFForm.cs, called by DisplayPDF.

```
{
    DialogResult result = DialogResult.None;
    while (result == DialogResult.None)
    {   PasswordForm mypasswordform = new PasswordForm(this);
        result = mypasswordform.ShowDialog();
        if (result == DialogResult.OK)
        {
            if (PDFDoc.PermRequest(documentpassword, PermissionRequestOperation.Export) == true)
            {
                return true;
            }
            else
            {
                MessageBox.Show("Invalid Document Password");
                // Force this to loop until either the correct password is enter
                // or Cancel is clicked.
                result = DialogResult.None;
            }
        }
```

Printing a PDF file

To send a PDF file to a printer, use the code offered in the PrintPDF sample program. This program is designed to prompt a user to enter the name of a PDF file to send to a printer, but you could automate the process to select PDF files. For example, you could write code to select a series of PDF files from a server directory instead.

Open the file, and then select the print parameters:

```
PrintUserParams userParams = new PrintUserParams();
PrintParams printParams = userParams.PrintParams;
```

In Java, the parameter lines look like this:

```
// Get some parameters
PrintUserParams userParams = new PrintUserParams();
// These are the "other" print parameters that hang off the user parameters.
PrintParams printParams = userParams.getPrintParams();

printParams.setShrinkToFit(true);
```

Use pageRanges to print with specified page ranges, in C#:

```
IList<PageRange> pageRanges = new List<PageRange>();

pageRanges.Add(new PageRange(0, 1, PageSpec.AllPages));
pageRanges.Add(new PageRange(0, 1, PageSpec.EvenPagesOnly));
pageRanges.Add(new PageRange(0, 1, PageSpec.OddPagesOnly));

printParams.PageRanges = pageRanges;
```

Or in Java. In this case, we show the lines commented out, as they appear in the original code:

```
// Printing with specified page ranges
// Uncomment next code to allow DLE printing with specified page ranges

// List<PageRange> pageRanges = new ArrayList<PageRange>();

// pageRanges.add(new PageRange(0, 1, PageSpec.ALL PAGES));
// pageRanges.add(new PageRange(0, 1, PageSpec.EVEN PAGES ONLY));
// pageRanges.add(new PageRange(0, 1, PageSpec.ODD_PAGES_ONLY));

// printParams.setPageRanges(pageRanges);
```

You can print to a Postscript file, and send that file to a printer separately, in C#:

```
doc.PrintToFile(userParams, "PrintPDF_out.ps");
```

Or Java:

```
doc.printToFile(userParams, "PrintPDF_out.ps");
```

Or you can select the default printer for a machine. The versions in C# and Java are similar:

```
userParams.UseDefaultPrinter(doc);
doc.Print(userParams);

userParams.useDefaultPrinter(doc);
doc.print(userParams);
```

Exporting text from a set of PDF files

Consider an example. Your firm recently bought a subsidiary that has been maintaining detailed monthly product performance, sales, customer retention, and customer support & satisfaction reports for nearly 25 years. All of these reports were preserved as PDF files and are quite thorough. Now that you are managing the firm you would like to be able to convert the content in these PDF files to text so that you can apply it to an internal database. That would allow you to search the content and generate statistics for historical analysis and data mining. The problem, however, is that for the first eight years of these reports, the original spreadsheet files used to generate the PDF files have been lost. They were created by a consultant, who provided the PDF files, but the subsidiary neglected to ask for the source content. About ten years after the consultant decided to retire, and the reporting process was moved in house, one of the employees of this subsidiary sought to find the consultant. The employee wanted to ask him to provide copies of the original spreadsheet files that were used to generate these early PDF reports. At that point, however, the

employee learned that this consultant had died two years earlier. The source content could not be found.

So you have over 400 of these PDF files, compiled and written between 1988 and 1997 and converted into PDF files between 1994 and 1997. Before you can make use of the data in these files, you need to convert them to text. You can adapt one of the programs provided with APFL, TextExtract, to that end. TextExtract will be faster and more accurate than Optical Character Recognition (OCR) software, and anyway OCR is designed for rendering scanned pages as text. You already have the electronic content. Further, the content in the PDF reports is formatted in such a way that you can transfer it to a spreadsheet in a table format and then import it from there into a database. But you need a way to quickly and easily convert these PDF files to text files, and save these text files to a server directory.

You would need to create your own program based on TextExtract to look for all of the PDF files in a specific server directory you designate, and set up a selection structure to choose each of these files one by one to be converted to text.

The program should export the content of the PDF file to a text file. In TextExtract the text file is stored in the same server directory as the sample program itself. The C# code looks like this:

```
int nPages = doc.NumPages;
IList<Word> pageWords = null;

System.IO.StreamWriter logfile = new System.IO.StreamWriter("TextExtract-
 untagged-out.txt");

for (int i = 0; i < nPages; i++)
```

This is the same code in the Java sample program:

```
{
        int nPages = doc.getNumPages();
        List<Word> pageWords = null;

        FileOutputStream logfile = new FileOutputStream("TextExtract-tagged-out.txt");
        OutputStreamWriter logwriter = new OutputStreamWriter(logfile, "UTF-8");

        for (int i = 0; i < nPages; i++)
        {
```

You would need to save each text file to a different server directory, and add a command to save each text file separately, and with an incremented file name.

From there, a separate utility could run to pull the data from these spreadsheet files and load them into a database table.

Merging multiple PDF files into a single document

Your employer provides investment counseling and wealth management for a variety of foundations, colleges & universities, museums, and non-profit institutions in the United States and Europe. One of the services you offer to your customers is to provide each one with a custom monthly statement. You generate this as a PDF file; some of your customers prefer it to be printed and mailed, but for most of them you send it as an email attachment or offer secured download from your FTP site. The information comes from a variety of sources, including government regulators from several countries, financial data and charts from banks, exchanges, and underwriters, and market analysis and prediction reports from a variety of independent economists and financial analysts. You also add a cover letter to each report, specific to each customer, and each customer report receives a cover page with the current date.

All of the content is in English but you typically end up with seven or eight different PDF files for each of your 150 customers, not including the cover page, and the formatting varies from one file to another. Some documents are in A4, others 8 ½ x 11 inch, and some are in landscape layout. First you need to sort the PDF files in the right order, identify pages in the files that you don't need, and delete those pages. Then, you need to adjust the format for these individual PDF files for each customer so that they are consistent and then merge the pages together, and in the right order.

What you need to do is to create a program that will format the PDF pages and then merge them into a new PDF file for each customer, one by one. The program will need to define a standard page size and layout to appear in the final PDF output file. Then, the program needs to be able to analyze the set of PDF files one by one and identify those where the page sizes must be adjusted. After the page sizes are corrected, the program must be able to merge these pages in the right order, creating a final PDF output file that you can print or deliver to the customer directly. The vendors and institutions that provide your original source PDF files create them each month on a pre-determined schedule and use the same format, layout, and page order each month, so it is possible to create a program that can automatically identify and delete pages that aren't needed after merging the PDF files into a single output file.

To complete these steps you would rely on the code from the sample program MergePDF. This program is designed to allow a user to enter the names of two different PDF files. The program joins them together in a new PDF output file. In this case you would use the sample code to create a process that would identify a series of PDF files and merge them together into a single output file.

But that is the end of the process. Before you can create the final merged files, you need to create a program or set of scripts that will complete the following steps:

1. Collect all of the source PDF files from vendors and suppliers into a single server directory. These will be delivered to you from various sources, and over several days.
2. Create a new monthly server directory for each customer. Each one will serve as a staging area for assembling the final reports to these customers.
3. Identify those PDF files that are specific to individual customers or groups of customers as they are received. Rename each file to add the customer account number to the file name.
4. Move these customer-specific PDF files to the appropriate customer subdirectories.
5. Make a copy of the generic PDF files and put one copy in each customer subdirectory. For example, if you have a general financial forecast that you want to add to each customer monthly statement, and you are creating 175 monthly statements, you would make 175 copies of this financial forecast PDF and place one copy in each of the customer subdirectories.
6. Now you have all of the content you need for each customer's monthly statement. Eliminate the pages that you don't need in each of the PDF files. Select each PDF file one by one, based on its name prefix, and use the DeletePage method in the C# Document Class.

To delete a single page in a PDF file:

```
Public void DeletePage (
   Int pageNumber
)
```

Or, for a series of pages:

```
public void DeletePages(
   int firstPageNumber,
   int lastPageNumber,
   string destinationFileName
)
```

7. Process the PDF files in each subdirectory so that the page sizes are all adjusted to the 8½ x 11 inch standard. Landscape diagrams can be left in the landscape layout.

   The European A4 standard is close to the American 8½ x 11 inch page size, so you might want to use the Page.Mediabox method to simply change the page size by setting the page dimensions.

   If you need to scale the contents of a page, especially if one page is considerably larger than the standard, you could import the content of the larger page into a Form object. Then, you could insert that form into a new destination page where you have set the dimensions to the size you want. For C# the code might look like this:

```
Form pageForm;
using (Page page = originalDoc.GetPage(i))
  {
       // Make a form that has the page's content
       pageForm = new Form(page.Content);
  }
  //creates the next page in the newDocument
  Page newPage = originalDoc.CreatePage(mergeAfterIndex, newPageMediaBox);

  // Put a copy of that form into the newPage
  newPage.Content.AddElement(pageForm.Clone());

  //update the content of the page that was just created
  newPage.UpdateContent();
```

8. When you have the final set of PDF files, with the unneeded pages removed, and the page formats set to match the 8½ x 11 inch standard, run code from the MergePDF program to create a single output PDF file for each customer.

   The MergePDF program defines a new file for each original PDF document, in C#:

```
Document doc1 = new Document(filename);
```

   And Java:

```
Document doc1 = new Document(filename);
```

   And then insert pages, attaching the second file to the end of the first, for C#:

```
doc1.InsertPages(Document.LastPage, doc2, 0, Document.AllPages,
PageInsertFlags.All);
```

or for Java:

```
doc1.insertPages(Document.LAST_PAGE, doc2, 0, Document.ALL_PAGES,
EnumSet.of(PageInsertFlags.ALL));
```

When this process is complete, you will have one server subdirectory for each customer, with all of the individual PDF files needed for each customer to create the next monthly report.

Delete all of the unneeded files in each subdirectory, leaving the actual report file for each customer. Then, send the PDF file to the customer as an email attachment or send it to a printer.

Finding and editing file content

Besides exporting text or graphics from a PDF file, the Adobe PDF Library offers several Java and C# sample programs that can find, present, or edit information *about* a PDF file.

Automatically applying keywords to a group of PDF files

A global manufacturing concern has a group of subsidiaries that build electronic tools and instruments intended for the health care field, academic, governmental and scientific laboratories, and regulators that monitor the construction trade. These products tend to be highly specialized, so besides having small sales volumes, the products are hard to market. You don't create TV commercials or place ads in *Vogue* for a device that measures the soundness of 15-year-old concrete in a bridge deck, or use Facebook to help with an effort to sell a background radiation detector intended for use in hospitals. At the same time, the firm faces stiff competition from several other manufacturers in Asia, and the firm's customers are generally small institutions that are broadly distributed around the world. Besides sales reps and trade shows, the firm needs to help its potential customers find its products by making effective use of Internet search tools.

Part of this effort involves effectively placing PDF files so that potential customers can find them. The firm provides a great deal of technical information about its products online, in the form of product descriptions, technical marketing content, specifications, and performance reports. All of these items are updated at least once every six months. Some are generated as frequently as once a month, and the firm has nearly 1000 of these documents available to support over one hundred individual products. All of these materials are provided as PDF files, as PDF is better suited than HTML for presenting long documents with diagrams and technical content. Further, the firm has automated tools in place to generate and post these updated files.

But PDF files do not lend themselves as well to online searching. PDF files lack the tag structure found in HTML files that the Google search engine favors. Rather, metadata needs to be added to the Properties page for each PDF file, including the firm's name, the name of each product, and a set of general keywords related to the firm's products.

When the firm generates a new PDF file the product name is included in the name of the PDF file, and this name also appears in the Title field in the Properties page. But the firm wants to create a utility that will automatically add the Firm name in the Author field, and a series of keywords in the Keywords field. The keywords need to be specific to the type of product. So the documents would need to be sorted into three categories first, health care, laboratory, and construction, and then a set of relevant keywords applied separately to each group. All of the health care products would receive the same set of keywords, added to the Keyword field in the Properties window, and the same is true for the laboratory and

construction instruments. Further, the firm wants to be able to apply new keywords to existing PDF files, rather than always have to generate new PDF files with the keywords added and then save the updated PDF files. This takes a lot less time and system overhead, and allows the marketing group to update the files regularly whenever new keywords are introduced.

1. Generate the PDF files for each instrument, with the name of the product included in the file name for each file.
2. Sort the files by product type, health care, laboratory equipment, and construction.
3. Copy the PDF files to the appropriate server directories.
4. Run a program that will apply keywords for each type of product. The program will draw keywords from an external text file.
5. Run the utility for each batch of files to add the keywords to the PDF files.
6. Save the PDF files back to the original directories and then post them to the appropriate web pages.

The ListInfo program prompts a user to enter the name of a PDF file and then displays the Title, Subject, Author, Keywords, Creator, and Producer. Then the system invites the user to enter new values for these items. But for the purpose of this firm the user interface is not needed. A developer could use ListInfo code as the basis for a program that would automatically select a PDF file and automatically add metadata values to the Properties window before saving that file. The program would need to access the PDF files from a server directory, as described above, and feature a loop so that it would review them one by one. It would also need to be able to access the keywords from an external text file to apply to the PDF files, in place of a user entering them at a command line prompt, and apply them in a group to the Keywords field in the Properties Screen. And it would need to automatically save each PDF file back to its original directory, overwriting the original version of the file.

The ListInfo program is written to display the current metadata value in a selected file, and then prompts the user to enter a new value to replace it, shown here in C#:

```
Console.WriteLine("Document Subject " + doc.Subject);
Console.WriteLine("Change document subject to: ");
string newsubject = Console.ReadLine();

Console.WriteLine("Document Author " + doc.Author);
Console.WriteLine("Change document author to: ");
string newauthor = Console.ReadLine();
```

The same code appears in Java:

```
System.out.println("Document Subject " + doc.getSubject());
System.out.println("Change document subject to: ");
String subject = stdin.readLine();
doc.setSubject(subject);

System.out.println("Document Author " + doc.getAuthor());
System.out.println("Change document author to: ");
String author = stdin.readLine();
doc.setAuthor(author);
```

After the user provides the value, the program replaces the original value with the new value provided by the user, shown here in C#:

```
doc.Subject = newsubject;
doc.Author = newauthor;
```

In Java, the matching line of code, `doc.setSubject(subject);`, appears above, in the prompt shown.

And then the program saves the values to a new PDF file, using the Document.Save method. The application described above would need to replace the user input for the keywords by accessing the keyword values in an external text file. The file could be updated at will as long as the name and location of the file remain the same. The application would also need to be written to select a PDF file, update the keywords in that file, and then save the file with the new values provided.

Adding and editing text in a PDF file

The Adobe PDF Library provides several sample programs that you can use to add text to a PDF file, in the form of glyphs or Unicode characters.

Suppose your firm has a computer system that regularly generates PDF files that you post to a series of web pages. These are marketing documents for your firm's products; periodically you want to add a note to the top of one of these PDF files announcing special offers, sales, or news items. These notes are commonly 15 to 20 words long; your customers have learned to expect them and look for them. To make the messages distinctive, you use a special font that your firm created for that purpose. But that means that you need to render the text on each PDF file as a series of glyphs. In PDF files, each character in a font shown is rendered as a glyph.

The process would include these steps to update a PDF file with a special marketing message:

1. A user types a brief statement in a user interface, or at a command prompt.
2. The interface or program saves the message to a text file and then translates it into a series of glyph characters.
3. Another program adds the glyphs to the upper left corner of the appropriate PDF file.

This program would run once for each PDF file to be updated.

You could consider the AddGlyphs sample program in APDFL. This program creates a new PDF file and adds a series of glyphs to that file:

```
Document doc = new Document();

Rect pageRect = new Rect(0, 0, 612, 792);
Page docpage = doc.CreatePage(Document.BeforeFirstPage, pageRect);
Console.WriteLine("Created page.");

Font font = new Font("Arial");

List<Char> glyphIDs = new List<Char>();
glyphIDs.Add('\u002b');
glyphIDs.Add('\u0028');
glyphIDs.Add('\u002f');
glyphIDs.Add('\u002f');
glyphIDs.Add('\u0032');
```

Then, it names and saves this PDF file.

The same code appears in our Java sample code:

```
Document doc = new Document();
Rect pageRect = new Rect(0, 0, 612, 792);
Page docpage = doc.createPage(Document.BEFORE_FIRST_PAGE, pageRect);
System.out.println("Created page.");

Font font = new Font("Arial");

List<Character> glyphIDs = new ArrayList<Character>();
glyphIDs.add(new Character('\u002b'));
glyphIDs.add(new Character('\u0028'));
glyphIDs.add(new Character('\u002f'));
glyphIDs.add(new Character('\u002f'));
glyphIDs.add(new Character('\u0032'));
```

You could use the command prompt interface provided with several of our other sample programs to create a simple way for a user to enter text, like this one found in ListInfo, here in C:

```
Console.WriteLine("Opened a document.");

Console.WriteLine("Document Title " + doc.Title);
Console.WriteLine("Change document title to: ");
string newtitle = Console.ReadLine();
```

and in Java:

```
System.out.println("Opened a document.");

 BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));

 System.out.println("Document Title " + doc.getTitle());
 System.out.println("Change document title to: ");
 String title = stdin.readLine();
 doc.setTitle(title);
```

To convert the text provided by the user into glyphs, you would need to set up a translation table that would map upper and lower case letters and standard punctuation characters for a typical font, like Times Roman or Arial, into the corresponding glyphs for your unique font. The table would need to map about 60 characters to glyphs.

Then, the program would need to add them to the PDF file, one glyph at a time.

You could do without the conversion table if you wanted to post a stock message to a PDF file. If you don't need to change the message, you can simply hard code the glyph characters into the program, as appears in the AddGlyphs sample program in APDFL.

Adding a watermark to PDF file

Your firm distributes PDF files to your customers in the form of monthly financial statements. One of the pages for this financial statement is a page from your outside auditor, verifying that the information that you provide on these reports has been audited and verified by an objective third party. You add this auditor's page to the back of each quarterly statement; the auditor's statement varies depending on the financial product offered, and you have 18 different product lines, so 18 different PDF files once a month from your auditor, to be added to each customer statement.

The auditor, of course, is a separate company that provides the monthly report on its own letterhead and using its own format. But you want to include your company logo on this statement in the upper left corner. And you want to automate this process. You want to be able to:

1. Generate the PDF files for your 1400 customers and sort them by product line into a set of server directories, sorted in order by account number
2. Accept the 18 different auditor statements each month and store them in a server directory
3. Apply the watermark to each of these 18 PDF files automatically
4. Copy the 18 auditor statements to the appropriate server directories where the customer statements are stored for each product
5. Add the auditor statement page for each product to the end of every other PDF file in the directory, and save those files
6. Delete the auditor statement

Then, you can send the PDF files out as email attachments, or print and mail them by US Mail.

To do this you can consider the APDFL Watermark sample program, to apply a graphic to a PDF file.

The Watermark sample program prompts a user to enter the name of the PDF file to apply the watermark, and then the name of the file holding the watermark graphic.

The program defines the size and placement of the watermark:

```
WatermarkParams watermarkParams = new WatermarkParams();
watermarkParams.Opacity = 0.8f;
watermarkParams.Rotation = 45.3f;
watermarkParams.Scale = 0.5f;
```

It defines the pages where the watermark will appear, in this case, every other page:

```
watermarkParams.TargetRange.PageSpec = PageSpec.EvenPagesOnly;
```

The code is similar in Java:

```
WatermarkParams watermarkParams = new WatermarkParams();
watermarkParams.setOpacity(0.8f);
watermarkParams.setRotation(45.3f);
watermarkParams.setScale(0.5f);

watermarkParams.getTargetRange().setPageSpec(PageSpec.EVEN_PAGES_ONLY);
```

The program then defines the page where a different watermark will appear, on odd pages:

```
doc.Watermark(watermarkDoc.GetPage(0), watermarkParams);

watermarkParams.TargetRange.PageSpec = PageSpec.OddPagesOnly;
```

For this watermark, the program uses text rather than a graphic, with the text hard-coded into the program itself, "Multiline\watermark." The program defines the color for the text, the font, the point size, and the text alignment:

```
WatermarkTextParams watermarkTextParams = new WatermarkTextParams();
 Color color = new Color(109.0f/255.0f, 15.0f/255.0f, 161.0f/255.0f);
 watermarkTextParams.Color = color;
```

```
watermarkTextParams.Text = "Multiline\nWatermark";

Datalogics.PDFL.Font f = new Datalogics.PDFL.Font("Courier",
   FontCreateFlags.Embedded | FontCreateFlags.Subset);
watermarkTextParams.Font = f;
watermarkTextParams.TextAlign = HorizontalAlignment.Center;

doc.Watermark(watermarkTextParams, watermarkParams);

doc.EmbedFonts();
```

You might decide to use text rather than a graphic image as your watermark. The sample program provides both options.

This section of code in Java looks like this:

```
doc.watermark(watermarkDoc.getPage(0), watermarkParams);

watermarkParams.getTargetRange().setPageSpec(PageSpec.ODD_PAGES_ONLY);

WatermarkTextParams watermarkTextParams = new WatermarkTextParams();
watermarkTextParams.setText("Multiline\nWatermark");

Font f = new Font("Courier", EnumSet.of(FontCreateFlags.EMBEDDED,
   FontCreateFlags.SUBSET));
watermarkTextParams.setFont(f);
watermarkTextParams.setTextAlign(HorizontalAlignment.CENTER);

Color c = new Color(109.0f/255.0f, 15.0f/255.0f, 161.0f/255.0f);
watermarkTextParams.setColor(c);

doc.watermark(watermarkTextParams, watermarkParams);

doc.embedFonts();
```

Working with graphics in PDF files

Adding and editing annotations

A user can edit a PDF file with Adobe Acrobat or Adobe Reader. With these viewing tools a user can add comments, highlight, underline, or cross out text, and insert images. You can also add links to move within a PDF file, open a web page, or launch a video. These changes and additions are called annotations. The Adobe PDF Library Java and .Net interfaces provide several sample programs that show how to write code to automatically add annotations to PDF files.

Your firm generates PDF files as advertising sheets for several national retail chains, including grocery stores, bakeries, pharmacies, and hardware stores. You use PDF for the flyers because PDF makes it easy for you to use a template with a standard color palette, font, layout, and design for each of these clients. Also, many of your customers prefer to print the flyers so that they can deliver them door to door or provide them as newspaper inserts. But some of your retail clients post the PDF advertising circulars to their web pages or send them out as email attachments. For those shoppers who look at the flyers online, you want the PDF files to be interactive as well.

As your clients are national chains, the same small set of advertisements tend to apply across hundreds of individual retail locations across the country. But each store posts local content on its own web page, so you would generate a PDF advertising flyer that might apply to

several dozen grocery stores in a region, and then make custom changes to these files, adding an annotation that will allow a user to click on a link within the PDF file when reading it online. The link would open the store's individual web page.

The Action sample program embeds a hyperlink within a PDF file that links to a web address. It also creates a rectangle where the hyperlink appears, and provides the coordinates that defines the size and placement of that rectangle.

1. Generate the monthly advertising flyer to be used for 65 grocery stores in Illinois, Wisconsin, Indiana, and Michigan.
2. Run a script that makes 65 copies of this PDF file, and assigns each one a unique name, incrementally, including a code number in each file name that identifies the individual store where the PDF will be sent. The PDF files are sorted in the server directory by this code number.
3. Run the process that adds the URL addresses for the store web page to the PDF files, one by one. The URL addresses for the web pages for each of these 65 stores are kept in a text file, and in the same order as the list of PDF files. The function selects the URL addresses from this text file and adds them, one by one, to the appropriate PDF files. The process also adds a rectangle to the PDF file and places the hyperlink in the middle of this rectangle. The PDF file will include the text "Click here to visit the store's web page" under the rectangle.

   The Action sample program includes the code to create an annotation with the web page address www.datalogics.com\products\pdfl\pdflibrary.asp, and insert it into a rectangle on a PDF file. This is the C# program:

```
LinkAnnotation newLink = (LinkAnnotation)docpage.CreateAnnotation("Link", new
Rect(1.0, 2.0, 3.0, 4.0));
        Console.WriteLine(newLink.ToString());

        doc.BaseURI = "http:\\www.datalogics.com";
        URIAction uri = new URIAction("products\pdfl\pdflibrary.asp", false);
        Console.WriteLine("Action data: " + uri.ToString());

        newLink.Action = uri;
        docpage.AddAnnotation(-2, newLink);
```

   And the matching code in Java:

```
Document doc = new Document();

        Rect pageRect = new Rect(0, 0, 100, 100);
        Page docpage = doc.createPage(Document.BEFORE_FIRST_PAGE, pageRect);
        System.out.println("Created page.");

        // Create our first link with a URI action
        LinkAnnotation newLink = new LinkAnnotation(docpage, new Rect(1.0,
        2.0, 3.0, 4.0));
        System.out.println(newLink.toString());

        doc.setBaseURI("http:\\www.datalogics.com");
        URIAction uri = new URIAction("\products\pdfl\pdflibrary.asp", false);
        System.out.println("Action data: " + uri.toString());
```

Exporting images from a set of PDF files

Suppose you have a series of very long PDF files with dozens of embedded photographs and diagrams. You would like to quickly organize and catalog the graphics images in each of these PDF files. You have the original JPG and PNG files that you added to these PDF files, but the individual PDF files vary widely. The only way to find out which graphics are in

each file would be to open each PDF file one by one, scroll through it, record the images that appear on each page of each file.

The Adobe PDF Library provides a faster way to complete that process. The sample program ImageExport is designed to review a PDF file, identify images that appear, and export them to a series of external graphics files. You can select the format you like for these files, TIF, JPG, PNG, GIF, or BMP. After extracting the images to a separate directory you could roll through these images quickly using a tool like the Microsoft Picture Manager.

The ImageExport program defines the types of graphics files that can be used for the export:

```
public void export_doc_images( Document doc)
 {
     export_doc_images_type(doc, ImageType.TIFF);
     export_doc_images_type(doc, ImageType.JPEG);
     export_doc_images_type(doc, ImageType.PNG);
     export_doc_images_type(doc, ImageType.GIF);
     export_doc_images_type(doc, ImageType.BMP);
 }
```

In Java, ImageExport.java calls the program ExportDocumentImages.java, where the values are defined:

```
public void export_doc_images( Document doc)
{
    export_doc_images_type(doc, ImageType.TIFF);
    export_doc_images_type(doc, ImageType.JPEG);
    export_doc_images_type(doc, ImageType.PNG);
    export_doc_images_type(doc, ImageType.GIF);
    export_doc_images_type(doc, ImageType.BMP);
}
```

And you can select the file type to use for export. This sample code would export images to JPG and PNG files, but you could write your program to only use one file type:

```
if (exporttype == ImageType.JPEG)
      {
     isp = new ImageSaveParams();
     isp.JPEGQuality = 80;
     img.Save("ImageExport-out" + next + ".jpg", exporttype, isp);
 }

 if (exporttype == ImageType.PNG)
 {
     img.Save("ImageExport-out" + next + ".png", exporttype);
 }
```

The Java code is in ExportDocumentImages.java:

```
if ( exporttype == ImageType.JPEG)
{
    isp = new ImageSaveParams();
    isp.setJPEGQuality(80);
     img.save("ImageExport-out" + next + ".jpg", exporttype, isp);
          }

if ( exporttype == ImageType.PNG )
          {
   img.save("ImageExport-out" + next + ".png", exporttype);
}
```

The program is designed to ask a user to enter a file name at a command prompt:

```
Console.Write("Enter filename: ");
try
{
    filename = Console.ReadLine();
}
```

The same prompt appears in ImageExport.java:

```
    System.out.print("Enter filename: ");
    BufferedReader stdin = new BufferedReader(new
InputStreamReader(System.in));
    try
            {
        filename = stdin.readLine();
            }
```

You could use this sort of prompt to allow a user to run your custom program manually. Or you might want to write a program that finds a set of PDF files in a server directory and cycles through them one by one. In that case you would probably want to include in the export statement a command that would copy the export graphics files for each PDF to a separate server directory.

Using Optical Character Recognition (OCR) technology to convert a PDF file to text

You receive a fax from a customer or vendor, and you would like to be able to turn the pages into text. You have an Optical Character Recognition (OCR) tool that can review PNG image files, convert images into text, and export that text to a separate TXT file. But before you can run the OCR utility, you need to be able to provide PNG files. You can take the fax and save it as a PDF file, rather than printing it. But you need to be able to convert the PDF file to a series of PNG files, one PNG for each page.

The DoctoImages sample program can convert each page of a PDF file to the graphics image file type that you select. This program does not find and extract images from the PDF; rather, each page is converted, as it appears, into a single graphics file.

The program sets the output format to use, in this case, JPG. This is the C# code:

```
public bool checkformatcompressionjpg()
 {
     if (compress == CompressionCode.DCT)
         return (outputformat == ImageType.JPEG);
     else
         return true;
 }
```

And Java:

```
            if (arg.startsWith("-format="))
            {
                if (opt.equals("jpg"))
                {
                    options.setoutputformat(ImageType.JPEG);
                }
```

Note that in the code snippets above, the C# program refers to the Compression code (DCT), and in Java, the code refers to the format option. These are both options that a user could enter with the DoctoImages command when running the program at a command prompt.

You could use the program code for DoctoImages and complete the conversion of a single PDF file to a series of JPG pages if you like. You could also write a new program that draws

from DoctoImages that selects a PDF file from a server directory automatically, and by default converts the pages of that file to JPG or PNG images.

The sample program saves each exported page from the PDF file to a new JPG file, and assigns it a name, in C#:

```
                case    ImageType.JPEG:
                    outputfile = filename + ".jpg";
                    break;
```

And in Java:

```
        else if ( imagetype == ImageType.JPEG)
        {
            outputfile = filename + ".jpg";
```