Developer Guide

Adobe® XML/PDF Access API for Java™

Version 1.0

Adobe® XML/PDF Access for Java™ Developer Guide
July 2004

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

Contents

List of Examples

Preface

This guide provides information about Adobe® XML/PDF Access API for Java™ (XPAAJ), one of the many products provided by Adobe.

What's in this guide?

This guide provides programming guidelines for developing custom applications that incorporate this product. It is a companion guide to the *API Reference*. Each chapter includes brief code examples. For brevity, most of the examples do not include `try-catch` blocks to handle exceptions. You should include such blocks in your own custom code.

Who should read this guide?

This guide is intended for Java developers who are responsible for developing applications that manipulate Adobe PDF files and their data through XML/PDF Access API for Java.

Related documentation

The resources listed in this table can help you learn more about the product:

For information about	See
Installing the library in a development environment	*Getting Started*
Product interfaces and classes	*API Reference*
Other Adobe services and products	www.adobe.com

1 Introduction

XML/PDF Access API for Java is a Java library that enables you to incorporate PDF documents into existing XML based workflows. This product provides document-level operations. You can import and export annotations, form data, XMP metadata, and file attachments. In addition, a text extraction capability is provided to support word searches and indexing.

About the XML/PDF Access API for Java SDK

The XML/PDF Access API for Java SDK works with the XML subassemblies of PDF documents or the XML Forms Architecture datasets associated with an XDP file. The XML subassemblies conform to the Adobe XML Forms Architecture specification, which can be viewed at http://partners.adobe.com/asn/tech/pdf/xmlformspec.jsp.

You can use XML/PDF Access API for Java to perform these tasks programmatically:

Access PDF documents and XDP files as XML streams You can access unencrypted PDF documents or XDP files and perform simple query operations, such as determining whether the document contains a form, the number of pages, and the PDF version. For more information, see "Invoking XML/PDF Access API for Java" on page 9.

Prepopulate forms and extract form data You can prepopulate interactive form fields and export form data. For more information, see "Importing and Exporting Form Data" on page 12.

Search for and extract text You can search for and extract text at the word level. For more information, see "Extracting Text" on page 46.

Convert the PDF representation of a form to its XDP counterpart You can convert a PDF document to an XDP file. For more information, see "Saving and Converting PDF Documents" on page 27.

Save changes to a modified XML stream After PDF data is exported as an XML data stream, you can modify the data and optionally save the changes in a file or back into the PDF document. For more information, see "Saving and Converting PDF Documents" on page 27.

Export and import document XMP metadata and image XMP metadata You can export and import information about a PDF document and its embedded images. For more information, see "Working with XMP Metadata" on page 36.

Manipulate annotations You can export, import, save, and delete annotations. For more information, see "Working with Annotations" on page 41.

Manipulate file attachments You can create, export, and delete file attachments. For more information, see "Working with File Attachments" on page 31.

Client applications can be delivered as Java applets or as stand-alone Java applications or, if web-based server-side processing is required, developers can implement Java servlets. A typical enterprise application would deploy such applications as Java 2 Platform, Enterprise Edition (J2EE) components. For more information, see "Deploying Custom Applications" on page 48.

Note: For information about XDP files, see the XML Data Package Specification at http://partners.adobe.com/asn/tech/pdf/xfa/xdp_2.0.pdf.

XML/PDF Access API for Java architecture

XML/PDF Access API for Java is implemented as a Java library based on the Java 2 Platform, Standard Edition (J2SE) 1.4.1 run-time environment. Applications can be deployed in a J2SE or J2EE environment.

Reading and writing operations are performed through a positionable XML stream. For simplicity, most of the methods for manipulating PDF/XDP files and their XML data are provided through the single public interface `PDFDocument`. The reading and writing operations support access to `PDFDocument` objects and other high-level objects.

Internally, XML/PDF Access API for Java contains three layers. Each layer contains classes that provide operations on that layer:

- The top layer is the PDF package, com.adobe.pdf, which contains the `PDFDocument` object; other classes internal to the PDF package provide support to implement the public operators on the `PDFDocument` object.
- The middle layer provides a representation of the PDF syntax, and provides the ability to parse a data stream and save objects that have been modified.
- The bottom layer is a utility layer that supports operations that depend on the run-time environment (there are no operating system dependencies).

For information about the public interfaces and classes that make up the com.adobe.pdf package, refer to the *API Reference*.

2 Invoking XML/PDF Access API for Java

XML/PDF Access API for Java is used to create custom applications capable of manipulating PDF documents. For example, using the XML/PDF Access API for Java library, you can create an application capable of extracting data from a PDF document and storing the data in an enterprise database. Before you can manipulate a PDF document, you must invoke XML/PDF Access API for Java.

The XML/PDF Access API for Java library is implemented in Java and has public methods that enable you to invoke it. Using a Java development environment, you can invoke XML/PDF Access API for Java by using a static `PDFFactory` object to create a `PDFDocument` object or a `FileAttachment` object. For information about these objects, see the *API Reference*.

In addition to using the XML/PDF Access API for Java library, you must also use standard Java classes. Some methods require a Java `InputStream` object as an argument. The examples in this chapter describe how to use the public methods and standard Java classes to invoke XML/PDF Access API for Java.

This chapter contains the following information:

Topic	Description	See
Including the library file	Describes how to include the XML/PDF Access API for Java library file in your Java project.	page 9
Using a PDFFactory object	Describes how to use a `PDFFactory` object.	page 9
Creating a PDFDocument object	Describes how to create a `PDFDocument` object.	page 10

Including the XML/PDF Access API for Java library file

The name of the library file is XPAAJ.jar. You need to include this file in your Java project to use this product. For information about installing the XPAAJ.jar file, see the *Getting Started* guide.

Adding the import statement

After you include the XPAAJ.jar file, add the following import statement to your Java project:

```
import com.adobe.pdf.*;
```

Once you add this import statement, you can start developing application logic. Your first task is to use a `PDFFactory` object to create a `PDFDocument` object.

Using a PDFFactory object

You use a `PDFFactory` object to create a `PDFDocument` or a `FileAttachment` object. A `PDFFactory` object is static and does not have to be created. This object contains the following methods:

openDocument creates a `PDFDocument` object.

newfileAttachment creates a `FileAttachment` object.

Note: For information about creating `FileAttachment` objects, see "Creating file attachments" on page 31.

Creating a PDFDocument object

`PDFDocument` is the predominant object that enables you to perform PDF document manipulation operations. Using a `PDFDocument` object, you can perform a variety of tasks, such as importing data into a PDF document, exporting data from a PDF document, and working with PDF document resources, such as file attachments.

You create a `PDFDocument` object by using a `PDFFactory` object and referencing an existing PDF document. You cannot create a new PDF document. To reference an existing PDF document, create a Java `InputStream` object, allocate memory to the `InputStream` object by using a `FileInputStream` constructor, and pass the location of the PDF document.

You can create a `PDFDocument` by calling the `PDFFactory` object's `openDocument` method and passing the Java `InputStream` object that references an existing PDF document. The following example creates a `PDFDocument` object that references a PDF document named Test.pdf located in the root of the C drive.

Example 2.1 ***Creating a PDFDocument object***

```
InputStream myPDFInput = new FileInputStream("C:\\Test.pdf");
PDFDocument pdfDocument = PDFFactory.openDocument(myPDFInput);
```

Performing PDF document query operations

After you create a `PDFDocument` object, you can manipulate a PDF document. This section examines how to perform the following PDF document query operations:

- Determining the form type.
- Determining the number of pages.
- Determining the PDF document version.

Note: Other parts of this guide examine how to perform more complex PDF document operations, such as importing data into a PDF document. For information, see "Importing and Exporting Form Data" on page 12.

Determining the form type

You can use a `PDFDocument` object to determine the form type on which a PDF document is based. A PDF document is based on one of the following form types:

Acrobat form is a PDF document created in Adobe Acrobat.

XML_FORM is a PDF document created in Adobe Designer.

NOT_A_FORM is a PDF document that does not contain form fields.

You determine the form type by calling the `PDFDocument` object's `getFormType` method. This method returns a `FormType` object. A `FormType` object contains the properties `ACROFORM`, `XML_FORM`, and `NOT_A_FORM`. The only methods that a `FormType` object has are those inherited from the Java `Object` class, such as `getClass`.

The following example determines the form type on which a PDF document is based.

Example 2.2 ***Determining the form type of a PDF document***

```
FormType ftype = pdfDocument.getFormType();
if (ftype == FormType.ACROFORM)
   System.out.println("This document is based on an ACROFORM.");
else if (ftype == FormType.XML_FORM)
   System.out.println("This document is based on an XML_FORM.");
else
   System.out.println("This document is not a form.");
```

Determining the PDF document version

You can use the `PDFDocument` object to determine the version of a PDF document. This value matches the version number in the Document Properties dialog box. For information about this dialog box, see "Getting information on PDF documents" in the Adobe Reader® Help.

You call the `PDFDocument` object's `getVersion` method to determine the version of a PDF document. This method returns a string value representing the version. The following example determines the version of a PDF document.

Example 2.3 ***Determining the version of a PDF document***

```
String docVersion = pdfDocument.getVersion();
System.out.println("The version of the PDF document is " +docVersion +".");
```

Determining the PDF document page count

You can use the `PDFDocument` object to determine the number of pages in a PDF document by calling the `PDFDocument` object's `getNumberOfPages` method. This method returns an integer value representing the number of pages. The following example determines the number of pages in a PDF document.

Example 2.4 ***Determining the number of pages in a PDF document***

```
int numPages = pdfDocOb.getNumberOfPages();
System.out.println("The PDF document contains "+ numPages +" pages(s).");
```

3 Importing and Exporting Form Data

You can use XML/PDF Access API for Java and standard Java classes to import and export interactive PDF form data. An interactive PDF form is a PDF document that contains one or more fields for collecting information from a user or displaying custom information. There are two types of PDF forms:

- An *Acrobat form* (created in Acrobat) is a PDF document that contains form fields.
- An *XML form* (created in Designer 6.0) is a PDF document that conforms to the Adobe PDF 1.5 specification and contains form field information that conforms to the XML Forms Architecture (XFA).

The `PDFDocument` interface contains methods that enable you to import and export form data. For example, using the `PDFDocument` interface's `exportFormData` method, you can export data from a PDF form. The data format that you can use for importing and exporting depends on the type of form on which you are performing the operation. This data can be in one of the following formats:

- An XFDF file which is an XML version of the Acrobat form data format.
- An XDP file which is an XML file that contains form field definitions. It may also contain form field data and an embedded PDF file. An XDP file generated by Adobe Designer 6.0 can only be used if it carries an embedded base-64-encoded PDF.
- An arbitrary XML file that contains name/value pairs.

The following table shows which data formats you can use for each form type:

Form type	Import formats	Result	Export formats	Result
Acrobat form	XFDF	Successful	XFDF	Successful
Acrobat form	XDP/XML	Unsuccessful	XDP/XML	Unsuccessful
XML form	XDP/XML	Successful	XDP/XML	Successful
XML form	XFDF	Unsuccessful	XFDF	Unsuccessful

This chapter contains the following information:

Topic	Description	See
About form fields	Explains form fields.	page 13
Prepopulating form fields	Explains how to prepopulate PDF forms.	page 15
Extracting form data	Explains how to extract data from PDF forms.	page 24

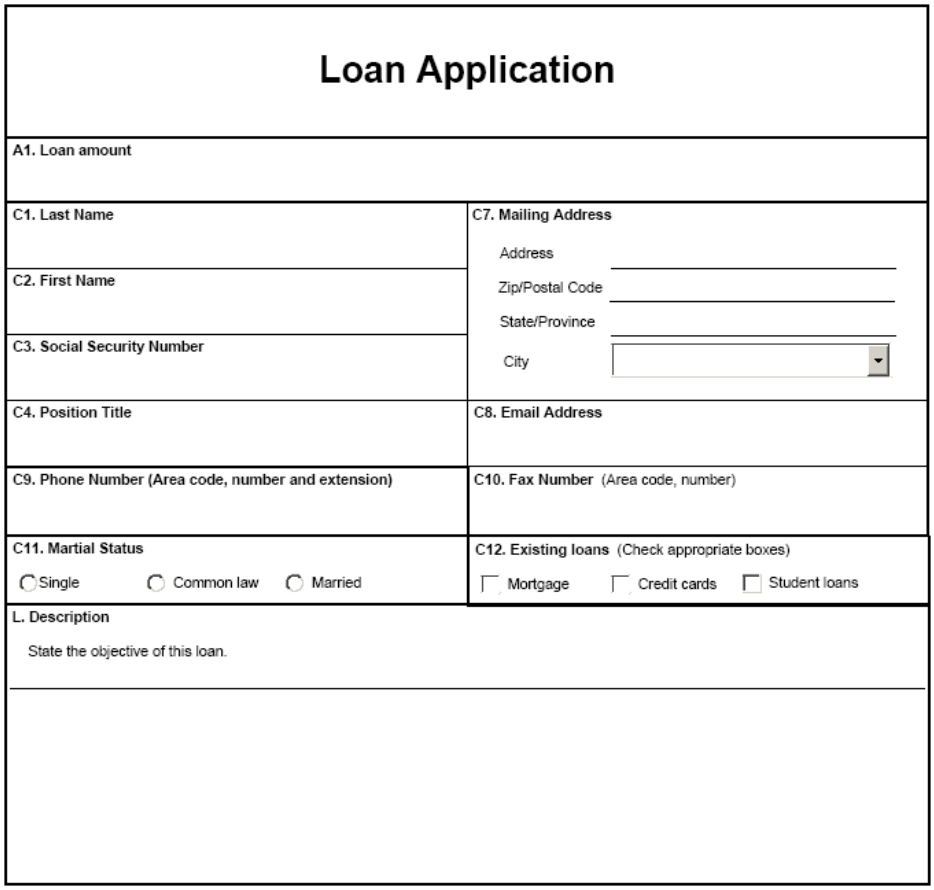
Note: The examples in this chapter use XML forms as the form type, and arbitrary XML as the data format.

About form fields

An XML form can be derived from a form design created in Designer (an Acrobat form is created using Acrobat). The form design specifies a set of layout, presentation, and data capture rules, including calculating values based on user input. The rules are applied whenever a form is filled with data.

Loan form

The following diagram shows an example of an XML form that was created in Designer:



This form represents an example of a form that financial institutions use to collect data when deciding whether to approve a loan. Notice that this form contains different types of fields. Most of the fields are text boxes; however, other types of fields, such as radio buttons, are used as well.

All form fields appear as nodes in the XML data file that is used for importing and exporting operations. The node values correspond to the field values. Because this form is used throughout this chapter, it is important that you understand the relationship between its fields and the corresponding XML data file. The following table describes the loan form fields:

Field	Name	Section
Loan amount text box	Amount	A1
Last name text box	LastName	C1
First name text box	FirstName	C2
SSN text box	SSN	C3

Field	Name	Section
Position title text box	PositionTitle	C4
Address text box	Address	C7
ZIP text box	ZipCode	C7
State text box	StateProv	C7
City drop down list box	DropDownList1	C7
Email text box	Email	C8
Phone text box	PhoneNum	C9
Fax text box	FaxNum	C10
Marital status radio buttons	RadioButtonList	C11
Mortgage check box	CheckBox1	C12
Credit cards check box	CheckBox2	C12
Student loans check box	CheckBox3	C12
Description text box	Description	L

Loan arbitrary XML data file

The loan arbitrary XML data file is used to import data into the XML loan form and extract data from it. Each form field is an XML element with a name matching the field name. An XML form can be exported as an arbitrary XML file by using the Acrobat menu (assuming you have the product installed). The following diagram shows the loan form after it is exported as an arbitrary XML file:

These XML elements correspond to the field names that appear on the loan form.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <form1>
  - <grantApplication>
    - <page1>
        <Amount />
        <LastName />
        <FirstName />
        <SSN />
        <PositionTitle />
        <Address />
        <StateProv />
        <ZipCode />
        <Email />
        <PhoneNum />
        <FaxNum />
        <Description />
        <PhoneNum />
        <RadioButtonList />
        <DropDownList1 />
        <FaxNum />
        <CheckBox1>0</CheckBox1>
        <CheckBox2>0</CheckBox2>
        <CheckBox3>0</CheckBox3>
      </page1>
    </grantApplication>
</form1>
```

Note: The code examples in this chapter use the loan arbitrary XML data file to import data into the loan form.

Loan XDP data file

The loan XDP data file can also be used to import data into the XML loan form. The difference between an arbitrary XML data file and an XDP file is that an XDP data file conforms to the XML Forms Architecture (XFA). An XML form can be exported as an XDP file by using the Acrobat menu (assuming you have the product installed). The following diagram shows the loan form after it is exported as an XDP file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<?xfa generator="XFA2_0" APIVersion="1.4.4047.0"?>
- <xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/">
  - <xfa:datasets xmlns="" xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/">
    - <xfa:data>
      - <form1>
        - <grantApplication>
          - <page1>
              <Amount />
              <LastName />
              <FirstName />
              <SSN />
              <PositionTitle />
              <Address />
              <StateProv />
              <ZipCode />
              <Email />
              <PhoneNum />
              <FaxNum />
              <Description />
              <PhoneNum />
              <RadioButtonList />
              <DropDownList1 />
              <FaxNum />
              <CheckBox1>0</CheckBox1>
              <CheckBox2>0</CheckBox2>
              <CheckBox3>0</CheckBox3>
            </page1>
          </grantApplication>
        </form1>
      </xfa:data>
    </xfa:datasets>
    <pdf href="Loan.pdf" xmlns="http://ns.adobe.com/xdp/pdf/" />
</xdp:xdp>
```

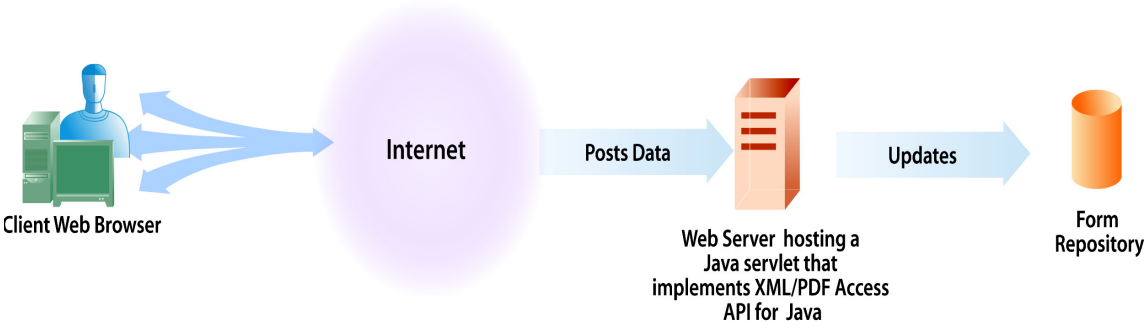
Notice that an XDP file contains additional elements that conforms to XFA syntax. For information about XFA, go to http://partners.adobe.com/asn/tech/pdf/xfa/xfapecification_2.1_draft2.pdf.

Prepopulating forms

You can create applications that are capable of prepopulating PDF documents. Prepopulating a PDF document involves inserting data into PDF form fields. The data can come from a variety of sources, such as a web application, an enterprise database, or another form. Prepopulating a form has several advantages:

- Automates the process of filling in a form.
- Enables the user to view custom data.
- Reduces the amount of typing.
- Ensures data integrity by placing valid data in specific fields.

An example of prepopulating a form involves using a web application. For example, assume that a web user enters data into a web page and posts the data to a web server hosting a Java servlet that uses XML/PDF Access API for Java. Once the data arrives, the Java servlet prepopulates a form, saves the form as a PDF document, and stores the form in a form repository. This diagram shows the process, which is explained in the table following the diagram:



1	A user fills in a web page and clicks the Submit button.
2	The data is posted to a web server hosting a Java servlet that uses XML/PDF Access API for Java.
3	The Java servlet prepopulates a form with the posted data and saves it as a PDF document.
4	The Java servlet sends the PDF document to a form repository, where it can be processed at a later time.

Creating application logic to prepopulate form fields

Call the `PDFDocument` object's `importFormData` method to prepopulate form fields. This method requires a Java `InputStream` object that represents an XML document containing elements that correspond to the form fields. If the form is an XML form, the XML document can be an arbitrary XML file or an XDP file. If the form is an Acrobat form, the XML document must be an XFDF file.

An XML element must exist for every form field you want to prepopulate. An XML element is ignored if it does not correspond to a form field. Matching the exact structure of the XML document is not necessary. For example, the order in which the XML elements are specified is not important.

Assume that a form containing 10 fields has data in 4 of the fields. Next, assume that you want to prepopulate the remaining 6 fields. In this situation, you must specify 10 XML elements in the XML data source used to prepopulate the form. If you specify only 6 elements, the original 4 fields will be empty.

The XML element value is used to prepopulate the corresponding form field. For example, the following XML syntax can be used to prepopulate the `Amount`, `FirstName`, and `LastName` fields in the Loan form:

```
<root>
   <Amount>3000</Amount>
   <FirstName>Jerry</FirstName>
   <LastName>Jones</LastName>
</root>
```

For information about these form fields, see "Loan form" on page 13.

This chapter describes the following ways in which you can use an arbitrary XML data source to prepopulate form fields:

- Converting an XML string to a byte stream.
- Referencing an existing XML document.
- Dynamically creating an in-memory XML data structure.

Note: The examples in this section assume that a `PDFDocument` object exists. For information, see "Creating a PDFDocument object" on page 10.

Converting an XML string to a byte stream

You can prepopulate a form by using a Java string that resembles an XML schema. This technique is convenient when you need to prepopulate several fields as opposed to an entire form. After you create a Java string, convert it to a Java `InputStream` object. Pass the `InputStream` object to the `PDFDocument` object's `importFormData` method.

The following process describes how to prepopulate a form by converting a string to a byte stream:

1. Create a Java `String` object and assign it a value that resembles an XML schema that corresponds to the form fields you want to prepopulate.
2. Assign the Java `String` object to a byte array and call its `getBytes` method.
3. Create a Java `InputStream` object by using the `ByteArrayInputStream` constructor. Pass the byte array to the constructor.
4. Call the `PDFDocument` object's `importFormData` method and pass the `InputStream` object.
5. Save the PDF document. For information, see "Saving PDF documents" on page 27.

The following code example shows how to prepopulate form fields by converting a string variable to a byte stream.

Example 3.1 ***Prepopulating a form by converting a string variable to a byte stream***

```
//Assign an XML schema to a string variable
String formData ="<root><Amount>2500</Amount><FirstName>Jerry</FirstName>
<LastName>Jones</LastName></root>";

//Convert the string variable into a byte array
byte[] inputData = formData.getBytes("UTF8");

//Create an InputStream by calling the ByteArrayInputStream constructor
InputStream formInputStream = new ByteArrayInputStream(inputData);

//Call the PDFDocument object's importFormData method
PDFDoc.importFormData(formInputStream);
```

Referencing an existing XML document

You can prepopulate a form by referencing an existing XML document. The XML document must contain XML elements that match the XML schema on which the form design is based. After you prepopulate form fields, XML element values appear in the form.

The following diagram shows the loan XML document containing data:

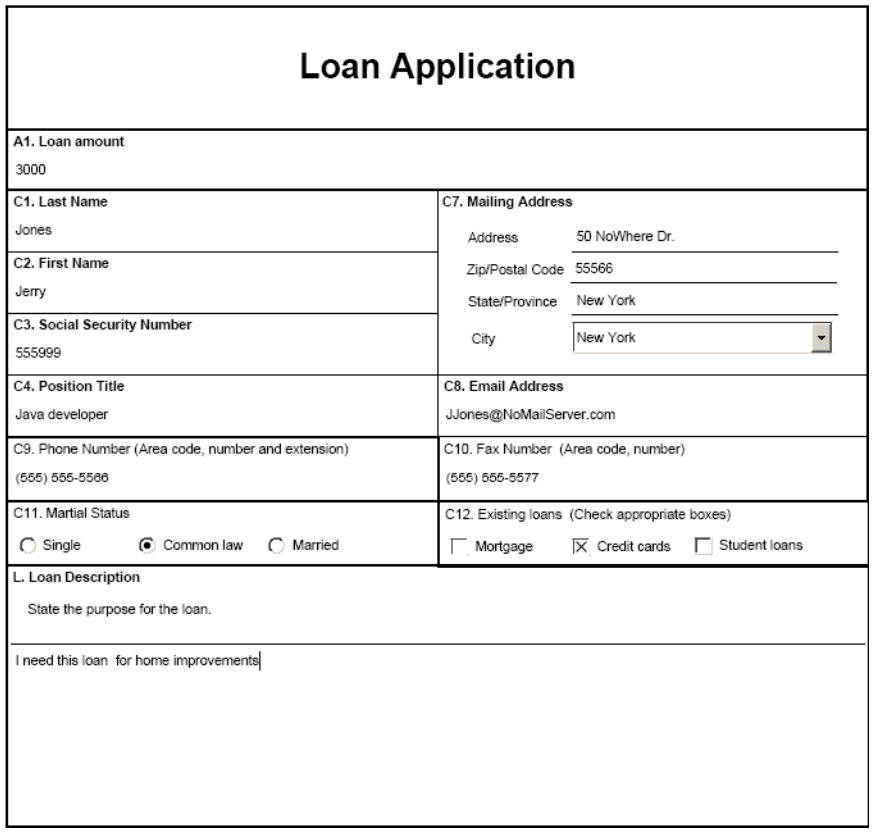
```
<?xml version="1.0" encoding="UTF-8" ?>
- <Form1>
    <Amount>3000</Amount>
    <LastName>Jones</LastName>
    <FirstName>Jerry</FirstName>
    <SSN>555999</SSN>
    <PositionTitle>Java developer</PositionTitle>
    <Address>50 NoWhere Dr.</Address>
    <StateProv>New York</StateProv>
    <ZipCode>55566</ZipCode>
    <Email>JJones@NoMailServer.com</Email>
    <PhoneNum>(555) 555-5566</PhoneNum>
    <FaxNum>(555) 555-5577</FaxNum>
    <Description>I need this loan for home improvements</Description>
    <RadioButtonList>2</RadioButtonList>
    <DropDownList1>New York</DropDownList1>
    <CheckBox1>0</CheckBox1>
    <CheckBox2>1</CheckBox2>
    <CheckBox3>0</CheckBox3>
  </Form1>
```

In the XML document, each element contains a data value. Most of these XML elements correspond to text box fields that appear on the form. The XML element value will appear in the corresponding text box. For example, the value `Jones` will appear in the `LastName` text box.

An XML element corresponding to a check box field has either a value of `0` or `1`. The value `1` results in the check box containing a check mark, and the value `0` results in the check box remaining unchecked. For example, because the value of the `CheckBox2` element is `1`, the credit cards check box contains a check mark.

Notice that the `RadioButtonList` element has a value of `2`. This value results in the second radio button being selected.

After the Loan form is prepopulated with the XML document shown in the previous diagram, data values that are located in the XML document appear in the form, as shown in the following diagram:



The following process describes how to prepopulate form fields by referencing an existing XML file:

1. Create a Java `InputStream` object by using the `FileInputStream` constructor. Pass the location of the XML file to the constructor.
2. Call the `PDFDocument` object's `importFormData` method and pass the `InputStream` object.
3. Save the PDF document. For information, see "Saving PDF documents" on page 27.

The following code example shows how to prepopulate a form by referencing an existing XML file.

Example 3.2 ***Prepopulating a form by referencing an existing XML file***

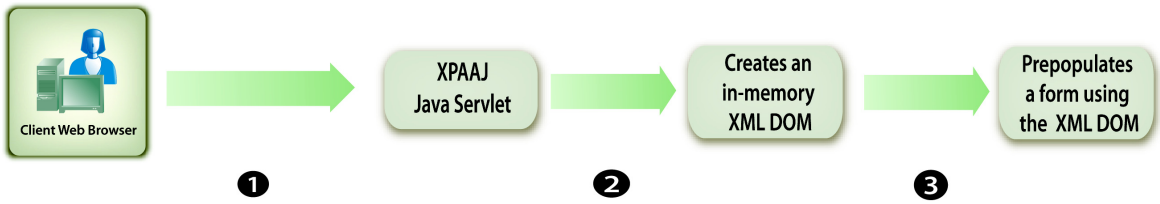
```
//Create an InputStream object by calling the FileInputStream constructor
InputStream formInputStream = new FileInputStream("C:\\loan.xml");

//Call the PDFDocument object's importFormData method
PDFDoc.importFormData(formInputStream);
```

Dynamically creating an in-memory XML data structure

You can prepopulate a form by dynamically creating an in-memory XML data structure. The advantage of creating a dynamic XML data structure is that you can place data that is obtained at run time directly into an XML data structure used to prepopulate a form.

Consider the example of prepopulating a form by using a web application. After a user fills in a web page and posts the data, it can be dynamically placed into an XML data structure. This diagram shows the process, which is explained in the table following the diagram:



1	A user fills in a web page and clicks the Submit button. Data is posted to a web server hosting a Java servlet that uses XML/PDF Access API for Java.
2	An in-memory XML data structure is created and data is placed into it.
3	The XML data structure is used to prepopulate a form.

You can use Java Document Object Model (DOM) classes to create an in-memory XML data structure. After you create and populate the XML data structure, you must convert it to a Java `InputStream` object that is required by the `PDFDocument` object's `importFormData` method. To accomplish this task, use Java XML transform classes. The in-memory XML data structure created in this section resembles the loan XML document. For information, see "Referencing an existing XML document" on page 18.

The following process describes how to prepopulate form fields by creating an in-memory XML data structure:

1. Create a Java `DocumentBuilderFactory` object by calling the `DocumentBuilderFactory` classes `newInstance` method.

2. Create a Java `DocumentBuilder` object by calling the `DocumentBuilderFactory` object's `newDocumentBuilder` method.

3. Create a new XML data structure by instantiating a Java `Document` object by calling the `DocumentBuilder` object's `newDocument` method.

4. Call the `Document` object's `createElement` method to create an `Element` object that represents the root element. Pass a string value representing the name of the element to the `createElement` method. Cast the return value to `Element`.

5. Create a child element that belongs to the root element by calling the `Document` object's `createElement` method, and pass a string value that represents the element's name. Cast the return value to `Element`. Next, set a value for the child element by calling it's `appendChild` method, and pass the `Document` object's `createTextNode` method as an argument. Specify a string value that appears as the child element's value. Finally, append the child element to the root element by calling the root element's `appendChild` method, and pass the child element object as an argument. The following lines of code shows this application logic:

   ```
   Element LastName = (Element)document.createElement("LastName");
   LastName.appendChild(document.createTextNode("Jones");
   root.appendChild(LastName);
   ```

6. Add all remaining elements to the XML data structure by repeating step 5 for each form field you want to prepopulate.

7. After you add all XML elements, convert the XML data structure to a byte array. One way to accomplish this task is to create a custom method that accepts the `Document` object as an argument and returns a byte array. In the code example that follows this list, this custom method is called `TransformDOM`. Java XML transform classes are used within the method.

8. Create a Java `InputStream` object by using the `ByteArrayInputStream` constructor and pass the byte array (created in step 7) to the constructor.

9. Call the `PDFDocument` object's `importFormData` method and pass the `InputStream` object.

10. Save the PDF document. For information, see "Saving PDF documents" on page 27.

The following code example shows how to prepopulate form fields by dynamically creating an XML data structure.

Example 3.3 ***Prepopulating a form by dynamically creating an XML data structure***

```
//Assume the following code is located in a method named populateForm
// Create a Document factory object
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();

// Create a new Document object
Document document = builder.newDocument();

//Create the root element and append it to the XML DOM
Element root = (Element)document.createElement("Form1");
document.appendChild(root);

//Create the Amount element
Element Amount = (Element)document.createElement("Amount");
Amount.appendChild(document.createTextNode("3000");
root.appendChild(Amount);

//Create the LastName Element
Element LastName = (Element)document.createElement("LastName");
LastName.appendChild(document.createTextNode("Jones");
root.appendChild(LastName);

//Create the FirstName element
Element FirstName = (Element)document.createElement("FirstName");
FirstName.appendChild(document.createTextNode("Jerry");
root.appendChild(FirstName);

//Create the DropDownList element
Element DropDown1 = (Element)document.createElement("DropDownList1");
DropDown1.appendChild(document.createTextNode("New York");
root.appendChild(DropDown1);

//Create the SSN element
Element SSN= (Element)document.createElement("SSN");
SSN.appendChild(document.createTextNode("555999");
root.appendChild(SSN);

//Create the PositionTitle element
Element PositionTitle = (Element)document.createElement("PositionTitle");
```

```
PositionTitle.appendChild(document.createTextNode("Java developer");
root.appendChild(PositionTitle);

//Create the address element
Element address = (Element)document.createElement("address");
address.appendChild(document.createTextNode("50 NoWhere Dr");
root.appendChild(address);

//Create the StateProv Element
Element StateProv = (Element)document.createElement("StateProv");
StateProv.appendChild(document.createTextNode("New York");
root.appendChild(StateProv);

//Create the ZipCode Element
Element ZipCode = (Element)document.createElement("ZipCode");
ZipCode.appendChild(document.createTextNode("55566");
root.appendChild(ZipCode);

//Create the Email Element
Element Email = (Element)document.createElement("Email");
Email.appendChild(document.createTextNode("JJones@NoMailServer.com");
root.appendChild(Email);

//Create the PhoneNum Element
Element PhoneNum = (Element)document.createElement("PhoneNum");
PhoneNum.appendChild(document.createTextNode("(555) 555-5566");
root.appendChild(PhoneNum);

//Create the FaxNum Element
Element FaxNum = (Element)document.createElement("FaxNum");
FaxNum.appendChild(document.createTextNode("(555) 555-5577");
root.appendChild(FaxNum);

//Create the Description Element
Element Description = (Element)document.createElement("Description");
Description.appendChild(document.createTextNode("I need this loan for home
improvements");
root.appendChild(Description);

//Create the RadioButtonList Element
Element RadioButtonList = (Element)document.createElement("RadioButtonList");
RadioButtonList.appendChild(document.createTextNode("2");
root.appendChild(RadioButtonList);

//Create the CheckBox1 Element
Element CheckBox1 = (Element)document.createElement("CheckBox1");
CheckBox1.appendChild(document.createTextNode("0");
root.appendChild(CheckBox1);

//Create the CheckBox1 Element
Element CheckBox2 = (Element)document.createElement("CheckBox2");
CheckBox2.appendChild(document.createTextNode("1");
root.appendChild(CheckBox2);

//Create the CheckBox1 Element
```

```
Element CheckBox3 = (Element)document.createElement("CheckBox3");
CheckBox3.appendChild(document.createTextNode("0");
root.appendChild(CheckBox3); // End of the XML data structure

//Convert the DOM into a byte stream
byte[] DOMBytes = TransformDOM(document);

// Create an InputStream by calling the ByteArrayInputStream constructor!
InputStream formInputStream = new ByteArrayInputStream(DOMBytes);

//Call the PDFDocument object's importFormData method
PDFDoc.importFormData(formInputStream);
}//End of the populateForm method

//TranformDOM converts a Document object into an array of bytes
private byte[] TransformDOM(Document doc)
{
//Declare a byte array
byte[] mybytes = null ;
try
   {
   // Create a Java Transformer object
   TransformerFactory transFact = TransformerFactory.newInstance();
   Transformer transForm = transFact.newTransformer();

   // Create a Java ByteArrayOutputStream object
   ByteArrayOutputStream myOutStream = new ByteArrayOutputStream();

   //Create a Java Source object
   Source myInput = new DOMSource(doc);

   //Create a Java Result object
   Result myOutput = new StreamResult(myOutStream);

   //Populate the Java ByteArrayOutputStream object
   transForm.transform(myInput,myOutput);

   // Get the size of the ByteArrayOutputStream buffer
   int myByteSize = myOutStream.size();

   //Allocate myByteSize to the byte array
   mybytes = new byte[myByteSize];

   // Copy the content to the byte array
   mybytes = myOutStream.toByteArray();
}
catch (Exception e)
   {
   System.out.println(e) ;
   }
return mybytes ;
}
```

Note: In the code example, the element values are hard-coded for simplicity. Typically, these values are stored in variables at run time.

Java import statements

The following Java import statements must be added to your Java project to successfully compile the previous code example:

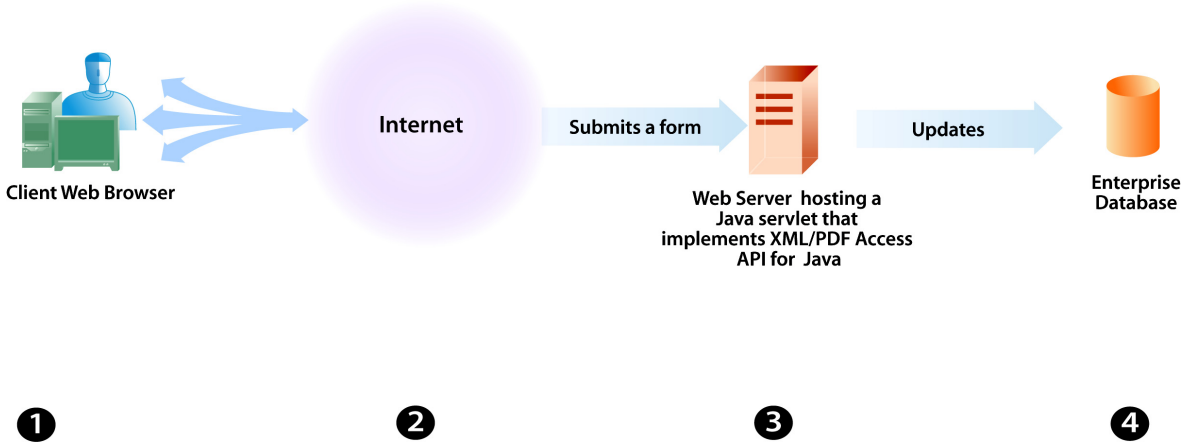
```
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import org.w3c.dom.Element;

import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
```

Note: These Java import statements are added in addition to the import com.adobe.pdf import statement. For information, see "Adding the import statement" on page 9.

Extracting form data

You can create applications that are capable of extracting data from PDF documents based on XML or Acrobat forms. Consider a client application that requires a user to fill in an online form and submit it to a server-based application that uses XML/PDF Access API for Java. After the server-based application receives the form, it can extract the data and process it in a variety of ways, such as storing it in a database, sending it to another application, merging it with another form, or displaying it in a web browser. This diagram shows the process, which is explained in the table following the diagram:



1	A user fills in an online form.
2	The form is sent to a web server that is hosting a Java servlet that uses XML/PDF Access API for Java.
3	The Java servlet extracts data from the form.
4	The Java servlet updates an enterprise database.

Note: The code example in this section assumes that a `PDFDocument` object exists. For information, see "Creating a PDFDocument object" on page 10.

Creating application logic to extract form data

Call the `PDFDocument` object's `exportFormData` method to extract data. This method returns a Java `InputStream` object that can be converted to an XML document containing elements that correspond to the form's fields. You can use Java DOM classes to parse the XML document and retrieve data. For information about the relationship between form fields and the corresponding XML document, see "About form fields" on page 13.

When calling `exportFormData`, you must pass one of the following values:

FormDataFormat.XFA represents a PDF form that is based on an XML form.

FormDataFormat.XFDF represents a PDF form that is based on an Acrobat form.

The following process describes the application logic to extract data from a PDF form that is based on an XML form:

1. Create a Java `InputStream` object and populate it by calling the `PDFDocument` object's `exportFormData` method. Pass `FormDataFormat.XFA` as an argument.
2. Create a Java `DocumentBuilderFactory` object by calling the `DocumentBuilderFactory` classes `newInstance` method.
3. Create a Java `DocumentBuilder` object by calling the `DocumentBuilderFactory` object's `newDocumentBuilder` method.
4. Create a `Document` object by calling the `DocumentBuilder` object's `parse` method, and pass the `InputStream` object that was returned from the `exportFormData` method.
5. Retrieve the value of each node within the XML document. One way to accomplish this task is to create a custom method that accepts two parameters: the `Document` object and the name of the node whose value you want to retrieve. This method returns a string value that represents the value of the node within the XML file. In the code example that follows this list, this custom method is called `getNodeText`.
6. Call the `getNodeText` method for each form field from which to extract data.

The following code example shows how to extract data from the loan form, which is based on an XML form. For information, see "Loan form" on page 13.

Example 3.4 ***Extracting data from a PDF form***

```
//Assume this is located in a method named RetrieveFormData
// Create an InputStream object by calling the exportFormData method
InputStream formDataStream = PDFDoc.exportFormData(FormDataFormat.XFA);

// Create a Document factory object
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();

// Create a new Document object by calling parse and
// passing the InputStream object
Document myDOM = builder.parse(formDataStream);

// Call getNodeText for each field in the loan form
String loanAmount = getNodeText("Amount", myDOM);
String lastName = getNodeText("LastName", myDOM);
```

```
String firstName = getNodeText("FirstName", myDOM);
String sinNum= getNodeText("SSN", myDOM);
String myTitle = getNodeText("PositionTitle", myDOM);
String addressVal = getNodeText("Address", myDOM);
String cityVal = getNodeText("DropDownList1", myDOM);
String stateProv = getNodeText("StateProv", myDOM);
String zipCode = getNodeText("ZipCode", myDOM);
String emailAdd = getNodeText("Email", myDOM);
String phoneNum = getNodeText("PhoneNum", myDOM);
String faxNum = getNodeText("FaxNum", myDOM);
String description= getNodeText("Description", myDOM);
String marrigeStatus= getNodeText("RadioButtonList", myDOM);
String mortgageValue = getNodeText("CheckBox2", myDOM);
String creditCards = getNodeText("CheckBox2", myDOM);
String studentLoan= getNodeText("CheckBox3", myDOM);
}//End of the RetrieveFormData method

// Create the getNodeText custom method
private String getNodeText(String nodeName, Document myDOM)
{
   //Get the node by name. NodeName is the name of the
   //node passed to this method
   NodeList oList = myDOM.getElementsByTagName(nodeName);
   Node myNode = oList.item(0);
   NodeList oChildNodes = myNode.getChildNodes();

   String sText = "";
   for (int i = 0; i < oChildNodes.getLength(); i++)
     {
      Node oItem = oChildNodes.item(i);
      if (oItem.getNodeType() == Node.TEXT_NODE)
          {
         sText = sText.concat(oItem.getNodeValue());
          }
     }
   return sText;

}//End of getNodeText
```

Java import statements

The following Java import statements must be added to your Java project to successfully compile the previous code example:

```
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import org.w3c.dom.Element;
```

Note: These Java import statements are added in addition to the import com.adobe.pdf import statement. For information, see "Adding the import statement" on page 9.

4 Saving and Converting PDF Documents

After you modify a PDF document, you can either save it or convert it to an XDP file. When saving a PDF file, you can also determine its kilobyte size, which may be required for certain application logic. For example, if a PDF file exceeds a specific size, you may not be able to send it over a network as a file attachment.

This chapter contains the following information:

Topic	Description	See
Saving a PDF document	Covers the methods you can use to save a PDF document.	page 27
Converting a PDF document to an XDP file	Covers the methods you can use to convert a PDF document to an XDP file.	page 29

Saving PDF documents

You use XML/PDF Access API for Java methods and standard Java classes to save a PDF document to a specific directory. To save a PDF document, call the `PDFDocument` object's `Save` method. This method returns the content of a PDF document to a Java `InputStream` object. For information about the `Save` method, see the *API Reference*.

You then create a Java `File` object and write the contents of the `InputStream` object to the `File` object. Ensure that you specify .pdf as the file name extension.

You can save a PDF document to a specific directory by performing the following programmatic tasks within a Java project:

1. Create a `PDFDocument` object. For information, see "Creating a PDFDocument object" on page 10.
2. Call the `PDFDocument` object's `Save` method. This method returns a Java `InputStream` object that contains the PDF document as a stream of bytes.
3. Get the byte size of the `InputStream` object by calling its `available` method. This method returns a integer value that represents the byte size of the `InputStream` object.
4. Create a byte array and allocate the byte size of the `InputStream` object (use the `available` method's return value).
5. Get the `InputStream` object's data content by calling its `read` method. Pass the byte array as the `read` method's argument. (This populates the byte array created in the previous step.)
6. Create a Java `File` object by using the `File` constructor. Because this file represents the saved PDF document, ensure that the file name extension is .pdf.
7. Create a `FileOutputStream` object by using its constructor. This object is used to write the data content of the byte array to the PDF file.
8. Call the `FileOutputStream` object's `write` method and pass the byte array as an argument.
9. Close the `FileOutputStream` object by calling its `close` method.

The following code example shows how to save a PDF document named Test.pdf to the root of C.

Example 4.1 ***Saving a PDF document***

```
//Call the PDFDocument object's save method
InputStream PDFIS = pdfDocument.save();

//Get the byte size of the InputStream object
int numBytes = PDFIS.available();

//Create an array of bytes. Allocate numBytes of memory
byte [] PDFBytes = new byte[numBytes];

//Populate the byte array by calling the InputStream object's read method
PDFIS.read(PDFBytes);

//Create a PDF file named Test.pdf and place it in the root of C:\
File myFile = new File("C:\\Test.pdf");

//Create a FileOutputStream object.
FileOutputStream myFileW = new FileOutputStream(myFile);

//Call the FileOutputStream object's write method and pass the pdf data
myFileW.write(PDFBytes);

//Close the FileOutputStream object
myFileW.close();
```

Determining the size of PDF documents

You can dynamically determine a PDF document's kilobyte size. After you create a `PDFDocument` object, call its `save` method which returns the PDF document as an `InputStream` object.

Once you have a `InputStream` object, you can call its `available` method to get the byte size. Use the division operator (/) and divide by 1024 to convert the value from bytes to kilobytes. The following example, which assumes you have created a `PDFDocument` object, shows how to get the kilobyte size of a PDF document.

Example 4.2 ***Getting the kilobyte size of a PDF document***

```
//Call the PDFDocument object's save method
InputStream PDFIS = pdfDocument.save();

//Get the byte size of the InputStream object
int numBytes = PDFIS.available();

//Convert from bytes to kilobytes
long pdfKilobyteSize = numBytes /1024;
System.out.println("The kilobyte size of the PDF file is " +pdfKilobyteSize);
```

Converting PDF documents to XDP files

You use XML/PDF Access API for Java methods and standard Java classes to convert an existing PDF document to an XDP file. You call the `PDFDocument` object's `saveAsXDP` method to convert an existing PDF document to an XDP file. You can convert an existing PDF document to an XDP file by performing the following programmatic tasks within a Java project:

1. Create a `PDFDocument` object. For information, see "Invoking the PDF Manipulation service" on page 17.
2. Call the `PDFDocument` object's `saveAsXDP` method. This method returns a Java `InputStream` object that contains the PDF document in XDP format.
3. Get the byte size of the `InputStream` object by calling its `available` method. This method returns an integer value that represents the size, in bytes, of the `InputStream` object.
4. Create a byte array and allocate the size, in bytes, of the `InputStream` object (use the `available` method's return value).
5. Get the `InputStream` object's data content by calling its `read` method. Pass the byte array as the `read` method's argument. (This populates the byte array created in the previous step.)
6. Create a Java `File` object by using the `File` constructor. Because this file represents the saved XDP file, ensure that the file name extension is .xdp.
7. Create a `FileOutputStream` object by using its constructor. This object is used to write the data content of the byte array to the XDP file.
8. Call the `FileOutputStream` object's `write` method and pass the byte array as an argument.
9. Close the `FileOutputStream` object by calling its `close` method.

The following example shows how to convert a PDF document to an XDP file.

Example 4.3 ***Converting a PDF document to an XDP file***

```
//Call the PDFDocument object's saveAsXDP method
InputStream XDPIS = pdfDocument.saveAsXDP();

//Get the byte size of the InputStream object
int numBytes = XDPIS.available();

//Create an array of bytes and allocate numBytes of memory
byte [] XDPBytes = new byte[numBytes];

//Populate the byte array by calling the InputStream object's read method
XDPIS.read(XDPBytes);

//Create an XDP file named Test.xdp and place it in the root of C
File myFile = new File("C:\\Test.xdp");

//Create a FileOutputStream object
FileOutputStream myFileW = new FileOutputStream(myFile);

//Call the FileOutputStream object's write method
myFileW.write(XDPBytes);

//Close the FileOutputStream object
myFileW.close();
```

Note: XDP files that are created by calling the `saveAsXDP` method cannot be opened in Designer. However, you can open these XDP files in Acrobat if you have the product installed.

5 Working with File Attachments

A PDF document can reference embedded file streams that can be archived or transmitted along with a PDF document. An *embedded file stream* is a file attachment that users can open and view while the PDF document is open in Adobe Reader (or Acrobat). For example, if the file attachment is a graphic file, users can view the file attachment in an application such as Adobe Photoshop®.

Using XML/PDF Access API for Java, you can work with file attachments. For example, you can create a new file attachment and embed it into a PDF document. After you perform a file attachment operation, you can save the PDF document. For information, see "Saving PDF documents" on page 27.

This chapter contains the following information:

Topic	Description	See
Creating file attachments	Describes how to create a new file attachment.	page 31
Importing file attachments	Describes how to import a file attachment into a PDF document.	page 32
Determining file attachment names	Describes how to determine file attachment names.	page 33
Exporting file attachments	Describes how to export a file attachment from a PDF document.	page 33
Deleting file attachments	Describes how to delete file attachments.	page 35

Note: In this chapter, it is assumed that a `PDFDocument` object exists. For information, see "Creating a PDFDocument object" on page 10.

Creating file attachments

A `FileAttachment` object represents a file attachment. You create a `FileAttachment` object by calling the `PDFFactory` object's `newfileAttachment` method and specifying the following arguments:

- A Java `InputStream` object that represents the file to attach to the PDF document.
- The name of the file that is attached.
- The MIME type of the file attachment. For example, if the file attachment is a bitmap file (*.bmp), you specify `image/bmp`.

To reference an existing file to use as a file attachment, create a Java `InputStream` object, allocate memory to it by using a `FileInputStream` constructor, and pass the location of the file. The following example creates a `FileAttachment` object by calling the `PDFFactory` object's `newFileAttachment` method. A Java `InputStream` object is created by referencing a bitmap file named water.bmp located in the root of the C drive.

Example 5.1 ***Creating a FileAttachment object***

```
InputStream fileAttachIS = new FileInputStream("C:\\water.bmp");
String fileAttName = "water.bmp";
String JavaMime = "image/bmp";
FileAttachment newFileAttach =
PDFFactory.newFileAttachment(fileAttachIS,fileAttName,JavaMime);
```

Note: For information, see "Using a PDFFactory object" on page 9

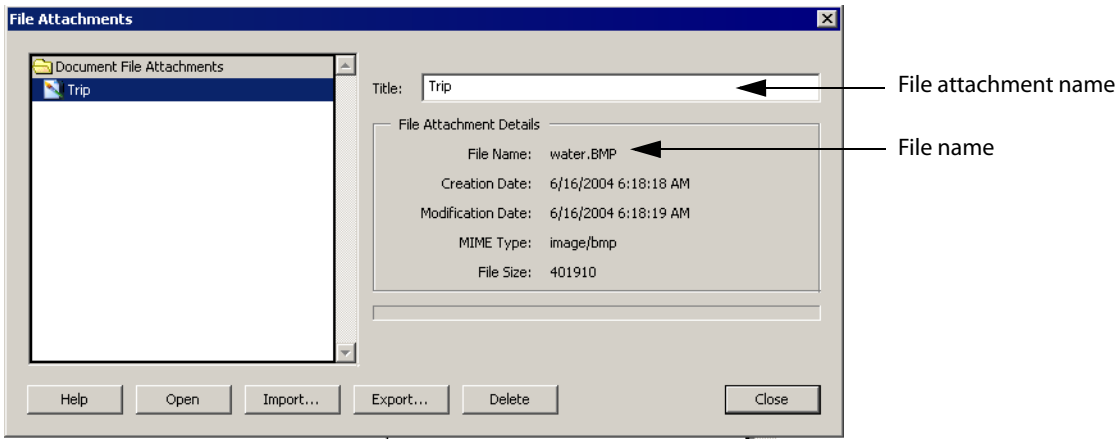
Importing file attachments

You import a file attachment into a PDF document by using the `PDFDocument` object's `importFileAttachment` method. Before you can import a file attachment, create a `FileAttachment` object by using the `PDFFactory` object's `newfileAttachment` method. For information, see "Creating file attachments" on page 31.

The `importFileAttachment` method requires the following arguments:

- A byte array that represents the file attachment name.
- A `FileAttachment` object that represents the file attachment.

A file attachment name is different from the name of the file located in the file attachment. For example, the file attachment can be named Trip while the embedded file is named water.bmp. The file name is defined when a `FileAttachment` object is created. The file attachment name is defined when the `importFileAttachment` method is called. The following illustration, which is the File Attachment dialog box in Acrobat, shows the difference between a file attachment name and the file name:



The following example imports a `FileAttachment` object named `newFileAttach`. The file attachment name is Trip.

Example 5.2 ***Importing a file attachment into a PDF document***

```
byte[] fileAttBytes = "Trip".getBytes();
PDFDoc.importFileAttachment(fBytes, newFileAttach);
```

Note: The `newFileAttach` object (which is an instance of the `FileAttachment` class) contains a file named water.bmp. For information about this object, see "Creating a FileAttachment object" on page 32.

Determining file attachment names

You use the `PDFDocument` object's `getFileAttachmentNames` method to determine the names of file attachments that are embedded in a PDF document. For example, assume that a PDF document contains a Trip file attachment and a Work file attachment. Using the `getFileAttachmentNames` method, you can determine the name of these file attachments.

The `getFileAttachment` method returns a two-dimensional byte array. The following code example shows how to use the `getFileAttachmentNames` method to determine the file attachment names that are embedded in a PDF document.

Example 5.3 ***Determining a file attachment name embedded in a PDF document***

```
//Determine the number of file attachments
byte[] names[] = doc.getFileAttachmentNames();
int num = names.length ;
System.out.println( "There are "+num +" file attachment(s) embedded in the PDF
document");

//Iterate through the names
for (int i= 0; i< names.length; ++){
   String fileAtt = new String (names[i], "UTF-16");
   System.out.println( ">The name of the file attachment is "+fileAtt );
};
```

Exporting file attachments

You can export a file attachment that is embedded in a PDF document by using the `PDFDocument` object's `exportFileAttachment` method. This method requires a byte array that represents the name of the file attachment and returns a `FileAttachment` object that represents the file attachment.

Using a `FileAttachment` object, you can perform the following tasks:

- Retrieve the data contained in the file attachment.
- Retrieve the file name.
- Retrieve the file attachment's MIME type.

Retrieving data contained in a file attachment

You can retrieve data contained in a file attachment by using the `FileAttachment` object's `getData` method. This method returns a Java `InputStream` object that represents the data contained in the file attachment. You can then perform a standard Java stream operation, such as writing the contents of the `InputStream` object to a file.

You can retrieve the file contained in a file attachment by performing the following programmatic tasks within a Java project:

1. Call the `PDFDocument` object's `exportFileAttachment` method and pass a byte array specifying the name of the file attachment whose data you want to retrieve. This method returns a `FileAttachment` object.

2. Call the `FileAttachment` object's `getData` method. This method returns a Java `InputStream` object that contains the data located in the file attachment.

3. Get the byte size of the `InputStream` object by calling its `available` method. This method returns an integer value that represents the size, in bytes, of the `InputStream` object.
4. Create a byte array and allocate the size, in bytes, of the `InputStream` object (use the `available` method's return value).
5. Get the `InputStream` object's data content by calling its `read` method. Pass the byte array as the `read` method's argument. (This populates the byte array created in the previous step.)
6. Create a Java `File` object by using the `File` constructor.
7. Create a `FileOutputStream` object by using its constructor. This object is used to write the data content of the byte array to the `File` object.
8. Call the `FileOutputStream` object's `write` method and pass the byte array as an argument.
9. Close the `FileOutputStream` object by calling its `close` method.

The following example shows how to retrieve data contained in the file attachment, write the data to a .jpg file and store it in a folder named myPics.

Example 5.4 ***Retrieving data contained in a file attachment***

```
//Export a file attachment named Trip
FileAttachment fileAtt = PDFDoc.exportFileAttachment("Trip".getBytes());

// Get the data located in the file attachment
InputStream fileData = fileAtt.getData();

//Get the byte size of the InputStream object
int buf = fileData.available();

//Create an array of bytes and allocate buf of memory
byte [] fileBytes = new byte[buf];

//Populate the byte array by calling the InputStream object's read method
fileData.read(fileBytes);

//Create an JPG file named holiday.jpg and place it in C:\myPics
File myFile = new File("C:\\myPics\\holiday.jpg");

//Create a FileOutputStream object
FileOutputStream myFileW = new FileOutputStream(myFile);

//Call the FileOutputStream object's write method
myFileW.write(fileBytes);

//Close the FileOutputStream object
myFileW.close();
```

Retrieving the file name

You can retrieve the name of a file that is located in a file attachment by calling the `FileAttachment` object's `getFilename` method. This method returns a string value representing the file name. The following code example shows how to retrieve the name of a file located in a file attachment.

Example 5.5 ***Retrieving a file name located in a file attachment***

```
//Export a file attachment named Trip
FileAttachment fileAtt = PDFDoc.exportFileAttachment("Trip".getBytes());

// Get the name of the file located in the file attachment
String fileName = fileAtt.getFilename();
System.out.println("The file name is "+fileName);
```

Retrieving the file attachment MIME type

You can retrieve the MIME type of a file that is located in a file attachment by calling the `FileAttachment` object's `getMimetype` method. This method returns a string value representing the MIME type of the file attachment. The following code example shows how to retrieve the MIME type of a file located in a file attachment.

Example 5.6 ***Retrieving the MIME type of a file attachment***

```
//Export a file attachment named Trip
FileAttachment fileAtt = PDFDoc.exportFileAttachment("Trip".getBytes());

// Get the mime type of the file
String mimeType = fileAtt.getMimetype();
System.out.println("The mime type of the file is "+mimeType);
```

Deleting file attachments

You can delete a file attachment that is embedded in a PDF document by using the `PDFDocument` object's `deleteFileAttachment` method. This method requires a byte array that represents the name of the file attachment to delete and doesn't have a return value. The following code example shows how to delete a file attachment.

Example 5.7 ***Deleting a file attachment embedded in a PDF document***

```
DFDoc.deleteFileAttachment("Trip".getBytes());
```

6 Working with XMP Metadata

PDF documents contain metadata, which is information about the file (not file content, such as text and images). The Adobe Extensible Metadata Platform (XMP) is a standard for handling metadata.

Using XML/PDF Access API for Java, you can obtain and replace the XMP metadata associated with the PDF document as a whole and the XMP metadata of any image embedded in the PDF document. XML/PDF Access API for Java provides methods for working with the document and image XMP metadata in PDF documents.

This chapter contains the following information:

About XMP metadata

XMP provides a standard format for creating, processing, and interchanging metadata for a wide variety of applications. XMP provides a model by which metadata is represented. XMP metadata is XML-formatted text that implements the W3C Resource Description Language (RDF) and is stored in a PDF document as UTF-8 encoded data.

XMP metadata consists of a set of file properties, including the file author, title, and modification date. Properties can also be associated with the components of a file (for example, embedded images). All metadata properties have names and values. The names must be legal XML names, and the values may be simple values (such as numbers and strings), structures that have named fields, or arrays.

The Adobe XMP Specification describes XMP in detail and is available on the Adobe Solutions Network website at www.adobe.com/products/xmp/pdfs/xmpspec.pdf.

Exporting XMP metadata

You call the `PDFDocument` object's `exportXMP` method to export XMP metadata. If required, any image XMP metadata (if available) can be exported separately. (See "Working with image XMP metadata" on page 38.) The `exportXMP` method returns a Java `InputStream` object that references the XMP metadata. For information about the `PDFDocument` object's `exportXMP` method, see the *API Reference*.

After the data is exported, you could call the `InputStream` object's `read` method to access the XMP metadata. The metadata can be manipulated afterward through the Adobe XMP Toolkit. The XMP Toolkit

describes a C++ programming interface for XMP. You can obtain a copy of the Adobe XMP Toolkit from http://partners.adobe.com/asn/developer/xmp/download/docs/MetadataToolkit.pdf.

Tip: If you choose to write the exported data to a file instead, you could create a Java `File` object and write the contents of the `InputStream` object to the `File` object. To reference the output file, you must create a Java `OutputStream` object, allocate memory to the `OutputStream` object by using a `FileOutputStream` constructor, and pass in the location of the output file.

Follow this process within a Java project to export and access XMP metadata:

1. Create a `PDFDocument` object that references the PDF document. For information, see "Creating a PDFDocument object" on page 10.
2. Call the `PDFDocument` object's `exportXMP` method. This method returns a Java `InputStream` object that contains the metadata as a stream of bytes.
3. Get the byte size of the `InputStream` object by calling its `available` method. This method returns an integer value that represents the byte size of the `InputStream` object.
4. Create a byte array and allocate the byte size of the `InputStream` object (use the `available` method's return value).
5. Get the `InputStream` object's data content by calling its `read` method. Pass the byte array as the `read` method's argument. (This step populates the byte array created in the previous step.)

The following code example shows how to extract metadata from a `PDFDocument` object.

Note: The example assumes that you have already obtained a `PDFDocument` object called `pdfDocument`. For information, see "Creating a PDFDocument object" on page 10.

Example 6.1 ***Exporting XMP metadata from a PDF document***

```
//Call the PDFDocument object's exportXMP method.
InputStream myXMPStream = pdfDocument.exportXMP();

//Get the byte size of the InputStream object.
int numBytes = myXMPStream.available();

//Create an array of bytes. Allocate numBytes of memory.
byte [] MDBytes = new byte[numBytes];

//Read the XMP metadata by calling the InputStream object's read method.
myXMPStream.read(MDBytes);
```

Importing XMP metadata

You call the `PDFDocument` object's `importXMP` method to import metadata from an XMP data source. The data is imported into the PDF document that is referenced by a previously obtained Java `InputStream` object. If image XMP metadata is available, that data can be imported separately. (See "Working with image XMP metadata" on page 38.)

The `importXMP` method verifies that the input stream represents valid XML and uses it to replace the metadata referenced by the PDF document. The input must be well formed; otherwise, an exception is thrown. For more information about the `PDFDocument` object's `importXMP` method, see the *API Reference*.

After the XMP metadata is imported, you can save the changes to the PDF document by calling the `PDFDocument` object's `Save` method. For information about the `Save` method, see the *API Reference*.

Follow this process within a Java project to import XMP metadata into a specific PDF document:

1. Create a `PDFDocument` object that references the PDF document. For information, see "Creating a PDFDocument object" on page 10.
2. Create a Java `InputStream` object to reference the XMP data source.
3. Call the `PDFDocument` object's `importXMP` method and pass in the input.
4. Optionally, save the PDF document by using the `PDFDocument` object's `Save` method. For information, see "Saving PDF documents" on page 27.

The following example shows how to import XMP metadata into a PDF document from the XMP data source XMPdata.txt.

Note: The example assumes that you have already obtained a `PDFDocument` object called `pdfDocument`. For information, see "Creating a PDFDocument object" on page 10.

Example 6.2 ***Importing XMP metadata into a PDF document***

```
//Create a Java InputStream object to reference the XMP data source.
InputStream myXMPData = new FileInputStream("XMPdata.txt");

//Call the PDFDocument object's importXMP method and pass in the input.
pdfDocument.importXMP(myXMPData);
```

Working with image XMP metadata

After you create the `PDFDocument` object for a PDF document, you can use the `PDFDocument` object's `getImages` method to obtain a list of the images embedded in the PDF document. The method returns a Java `List` object that references a sequence of `PDFImage` objects and implements the Java `Iterator` interface. For information about the `getImages` method, see the *API Reference*.

The list of images identifies each image in an existing PDF document. You can use the `hasNext` method of the Java `Iterator` interface to determine if the list contains elements and to perform XML/PDF Access API for Java operations on a specific `PDFImage` object.

A `PDFImage` object has two methods: `exportXMP` and `importXMP`. You use the `exportXMP` method to export the XMP metadata associated with an image. The `exportXMP` method converts the metadata into XML data and returns the output to a Java `InputStream` object. After the data is exported, your custom application could modify the XMP metadata in some way. For information about the `PDFImage` object's `exportXMP` method, see the *API Reference*.

The `importXMP` method imports XMP metadata for an image. The `importXMP` method parses the input stream and uses the input to replace the XMP metadata associated with the referenced image. For information about the `PDFImage` object's `importXMP` method, see the *API Reference*.

Follow this process within a Java project to export and import image XMP metadata:

1. Create a `PDFDocument` object that references the PDF document. For information, see "Creating a PDFDocument object" on page 10.

2. Call the `PDFDocument` object's `getImages` method. This method returns a Java `List` object that references a number of `PDFImage` objects and implements the Java `Iterator` interface.

3. Call the `List` object's `iterator` method. This method returns a Java `Iterator` object, which provides an iterator over the list of `PDFImage` objects.

4. In a `while` loop, call the `hasNext` method of the `Iterator` object to invoke processing (steps 5 through 8) only when an unprocessed element exists in the list of images.

5. Create a `PDFImage` object for the next element in the list.

6. Get that element's XMP metadata by calling the `PDFImage` object's `exportXMP` method for the current element. The method returns a Java `InputStream` object that references the associated XMP metadata.

7. Assuming that your custom application will modify and replace the data, complete these tasks:
   - Verify that XMP metadata actually exists (that is, the data referenced by the `InputStream` object is not NULL).
   - Modify the data programmatically, as required by your custom application, and reference the result through a new Java `InputStream` object.
   - Call the `PDFImage` object's `importXMP` method. Pass in the replacement input stream as the `importXMP` method's argument.

8. Save the PDF document by using the `PDFDocument` object's `Save` method. For information, see "Saving PDF documents" on page 27.

The following code example shows how to get a list of images from a PDF document, obtain an iterator for the collection of images, export the XMP metadata associated with each of the images and, after modifications, replace the image XMP metadata in the PDF document.

Note: The example assumes that you have already obtained a `PDFDocument` object called `pdfDocument`. It also assumes that a custom class named `MyXMPModifier` (having a `modifyXMP` method) is written to modify the exported XMP metadata.

Example 6.3 ***Exporting and importing image XMP metadata***

```
//Get the list of images.
List pdfImages = pdfDocument.getImages();

//Get the iterator.
Iterator imageIterator = pdfImages.iterator();

//Iterate over the list of images.
while(imageIterator.hasNext()) {

   //Get the next image.
   PDFImage pdfImage = (PDFImage)( imageIterator.next() );

   //Get the XMP metadata InputStream.
   InputStream myXMPStream = pdfImage.exportXMP();

   //Check for the existence of XMP metadata.
   if(myXMPStream != null) {

      //In some way, modify the XMP metadata by using a custom class/method.
      InputStream newXMPStream = MyXMPModifier.modifyXMP(myXMPStream);

      //After modifications, import the modified XMP metadata.
      pdfImage.importXMP(newXMPStream);

   }

}

//Save the modified PDF document.
InputStream newPDFStream = pdfDocument.save();
```

7 Working with Annotations

An annotation associates an object, such as a comment, with a location on a PDF document page. Users can use annotations to add information to existing PDF documents much like they can use a pen to mark up a paper copy of a document. Annotations provide more flexibility in the type of information that can be attached to a document, such as sounds and video clips.

Annotations are displayed in an open state or a closed state. When in a closed state, an annotation appears on a page in a distinctive form, such as an icon or a rubber stamp. When a user activates the annotation, the annotation exhibits its associated object or performs an action, such as playing a video clip. For example, you can click on a comment to read its contents.

Using XML/PDF Access API for Java, you can import, export, and delete annotations. Annotation-related methods belong to the `PDFDocument` interface. For information about this interface, see the *API Reference*.

This chapter contains the following information:

About annotations

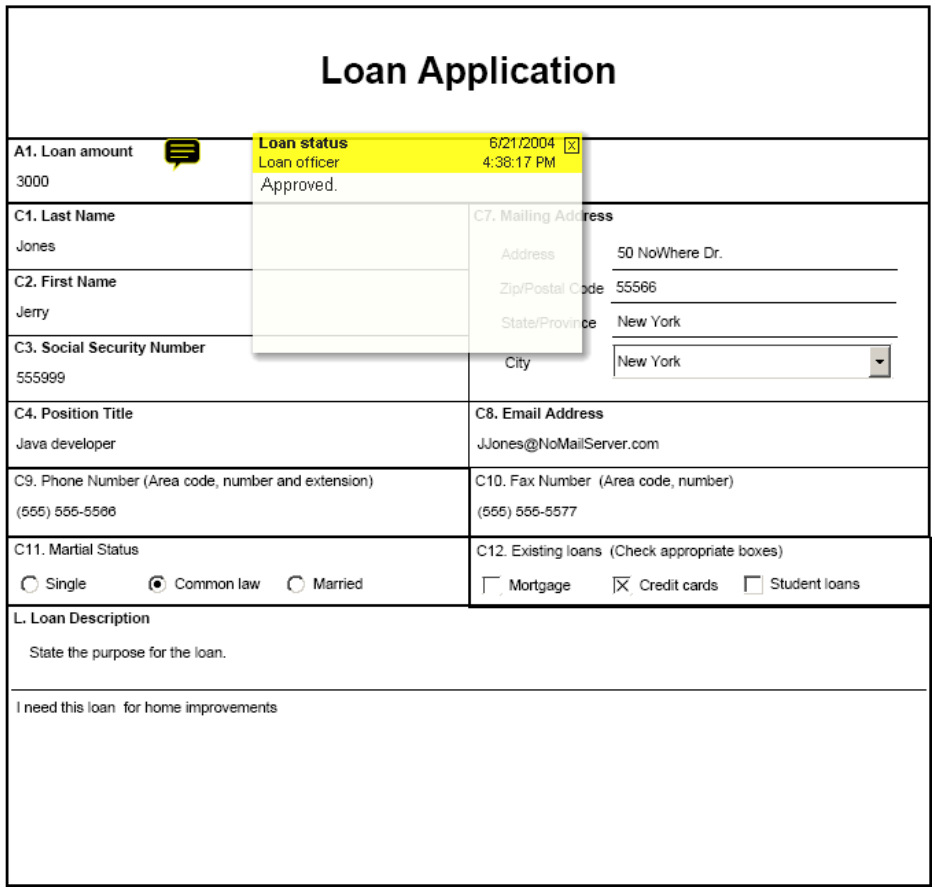
The following table lists and describes some of the annotation types that you can use.

Annotation type	Description
Note	Represents a post-it note attached to a point on a page. When closed, it appears as an icon; when open, an associated Popup annotation displays the text of the annotation.
Strikeout	When open, an annotation may contain text explaining why information is crossed out.
Highlight	When opened, an annotation displays text explaining why information is highlighted.
Underline	When opened, an associated Popup annotation displays text explaining why information is underlined.
FreeText	Displays text directly on the page. This type of annotation does not have an open or closed state or an associated Popup annotation because the text is always visible.
Line	Displays a single, straight line on a page. When opened, an associated Popup annotation displays text explaining what the line represents.
Square	Displays a rectangle on the page, usually around a portion of text. When opened, an associated Popup annotation displays text explaining why the square is used.

Annotation type	Description
Circle	Displays an ellipse on the page. When opened, an associated Popup annotation displays text explaining the circle.
Ink	A freehand scribble composed of one or more disjointed paths.
Popup	Displays text in a popup window for text entry, editing, and viewing. It usually does not appear alone but is associated with a parent markup annotation, such as a highlight or note.
Stamp	Displays text or graphics intended to appear on a page as though they were stamped with a rubber stamp.
Link	Represents either a hypertext link to a destination elsewhere in the document or an action to perform.

Annotation example

A note annotation enables users to place comments directly into a PDF document. For example, consider the loan form introduced in an earlier chapter. An annotation can be placed into this document to indicate that the loan was approved, as the following diagram shows:



Note: For information about the loan form, see "Loan form" on page 13.

Exporting annotations

You can export all annotations that are embedded in a PDF document by using the `PDFDocument` object's `exportAnnotations` method. This method returns a Java `InputStream` object that represents an XFDF XML data stream containing annotations. You can use the `InputStream` object to perform standard Java stream operations, such as writing the XFDF XML data to a file. This guide does not explain XFDF XML; however, if you are interested in learning more, go to the website http://partners.adobe.com/asn/tech/pdf/xfa/xfdf_2.0_draft.pdf.

The following diagram shows XFDF XML containing the text annotation displayed in the loan form:

```
<?xml version="1.0" encoding="UTF-8" ?>
- <xfdf xmlns="http://ns.adobe.com/xfdf/" xml:space="preserve">
  - <annots>
    - <text page="0" color="#FFFF00" date="D:20040621165541-04'00'" title="Loan officer" name="XA0lRD19RbHuhwN0pvOIhD" rect="140.740509,
      648.07428, 160.740509, 666.07428" flags="print,nozoom,norotate" creationdate="D:20040621131710-04'00'" subject="Loan status"
      icon="Comment">
      - <contents-richtext>
        - <body xmlns="http://www.w3.org/1999/xhtml" xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/"
            xfa:APIVersion="Acrobat:6.0.1" xfa:spec="2.0.2">
          - <p>
              <span style="font-size:10.0pt">Approved</span>
            </p>
          </body>
        </contents-richtext>
        <popup rect="225.925537, 552.556335, 405.924713, 672.555786" flags="print,nozoom,norotate" open="yes" />
      </text>

    </annots>
</xfdf>
```

The following process describes how to export annotations from a PDF document to an XML file:

1. Create a Java `InputStream` object by calling the `PDFDocument` object's `exportAnnotations` method.
2. Get the byte size of the `InputStream` object by calling its `available` method. This method returns a integer value that represents the byte size of the `InputStream` object.
3. Create a byte array and allocate the byte size of the `InputStream` object (use the `available` method's return value).
4. Get the `InputStream` object's data content by calling its `read` method. Pass the byte array as the `read` method's argument.
5. Create a Java `File` object by using the `File` constructor. Because this file represents the XML file, ensure that the file name extension is .xml.
6. Create a `FileOutputStream` object by using its constructor. This object is used to write the data content of the byte array to the XML file.
7. Call the `FileOutputStream` object's `write` method and pass the byte array as an argument.
8. Close the `FileOutputStream` object by calling its `close` method.

The following code example shows how to export annotations from a PDF document to an XML file:

Example 7.1 ***Exporting annotations from a PDF document***

```
//Call the PDFDocument object's exportAnnotations method
InputStream AnnIS = pdfDocument.exportAnnotations();

//Get the byte size of the InputStream object
int numBytes = AnnIS.available();

//Create an array of bytes. Allocate numBytes of memory
byte [] PDFBytes = new byte[numBytes];

//Populate the byte array by calling the InputStream object's read method
AnnIS.read(PDFBytes);

//Create a PDF file named Test.xml and place it in the root of C:\
File myFile = new File("C:\\Test.xml");

//Create a FileOutputStream object.
FileOutputStream myFileW = new FileOutputStream(myFile);

//Call the FileOutputStream object's write method and pass the pdf data
myFileW.write(PDFBytes);

//Close the FileOutputStream object
myFileW.close();
```

Importing annotations

You can import annotations into a PDF document by using the `PDFDocument` object's `importAnnotations` method. This void method requires a Java `InputStream` object that represents an XFDF XML data file containing annotations. Assume that the XFDF XML data file shown earlier in this chapter is used to import annotations into a PDF document. Next, assume that the text value is changed from `Approved` to `Decline`. In this situation, the text annotation shown in the loan form displays `Decline`.

You can create a Java `InputStream` to pass to the `importAnnotations` method by using the `FileInputStream` constructor and passing the location of an XML file that contains XFDF XML data. The following example shows how to import annotations into a PDF document.

Example 7.2 ***Importing annotations into a PDF document***

```
//Create an InputStream object containing annotations
InputStream AnnIS = new FileInputStream("C:\\Annotations.xml");

//Import the annotations into the PDF document
doc.importAnnotations(AnnIS);
```

Deleting annotations

You can delete all annotations that are in a PDF document by using the `PDFDocument` object's `deleteAnnotations` method. Only those annotations listed in this chapter are deleted. For information, see "About annotations" on page 41.

After you call this method, all listed annotations are deleted. You cannot delete individual annotations and leave others intact. The `deleteAnnotations` method doesn't have a return value and doesn't require any arguments.

The following example shows how to delete annotations from a PDF document.

Example 7.3 ***Deleting annotations from a PDF document***

```
//Delete all annotations
doc.deleteAnnotations();
```

8 Extracting Text

You can use XML/PDF Access API for Java to extract text from a PDF document. Once you extract the text, you can process it in a variety of ways, such as storing it in a text file, sending it to another application, or displaying it in a web browser. Extracting text from a PDF document is different from extracting data from form fields in a PDF document. For information, see "Extracting form data" on page 24.

Extracting text from PDF documents

You extract text from a PDF document by calling the `PDFDocument` object's `getWords` method. This method returns a Java `ListIterator` collection where each element is a `PDFWord` object. For information about a `PDFWord` object, see the *API Reference*.

Using a `PDFWord` object, you can perform the following tasks:

- Get a Unicode string representing the word.
- Get the page number where the word is located.
- Get a list of bounding quads for the word.

The following code example extracts text from a PDF document and creates a separate text file for each page.

Example 8.1 ***Extracting text from a PDF document***

```
ListIterator wordIterator = doc.getWords();
int pN = 1;
int cPageN = 1;
String text = "";
String outTextFilename
while(wordIterator.hasNext()){
   //Get a PDFWord object
   PDFWord word = (PDFWord) wordIterator.next();
   pN = word.getPageNumber();

   // append text if in the same page
   if(cPageN == pN) {
      //Determine if the text variable is empty
      if (text.length() ==0)
         text = word.getString();
      else
         text = text+" "+word.getString();
   }

   // write text to a file for each page
```

```
   else {
      outTextFilename = "C:\\Page"+cPageN+".txt";
      saveStringToFile(text, outTextFilename);
      // Clear the text string because a new page is started
      text = "";
      text = word.getString();
      cPageN = pN;
   }
}

//Out of the loop and there are no more words
//Write the last page to a file named LastPage.txt
outTextFilename = "C:\\LastPage.txt";
saveStringToFile(text, outTextFilename);

// save text string to a file.
public void saveStringToFile(String text, String filePath)
{
try {
     BufferedWriter outTxtFile = new BufferedWriter(new FileWriter(filePath));
     outTxtFile.write(text, 0, text.length());
     outTxtFile.close();
     }
catch (IOException e) {
     System.out.println("Error in saving text file.");
   }
}
```

Note: Punctuation such as periods and commas are not extracted with the text.

9 Deploying Custom Applications

This chapter outlines the most common ways to deploy custom J2SE or J2EE applications that use the XML/PDF Access API for Java class library. It is assumed that the class library will be delivered as an extension to your application. The format of your custom application determines the required deployment strategy.

This chapter contains the following information:

Topic	Description	See
Client-side deployments	Provides an overview of how to deploy two types of client applications.	page 48
Server-side deployments	Provides an overview of how to deploy two types of server applications.	page 49
Deploying a custom application	Explains how to deploy the class library as part of a stand-alone application.	page 50
Deploying a custom web application	Explains how to deploy the class library as part of a web application.	page 50

Client-side deployments

A *client* can be any program that requests services from a server application. For client-side deployments, you can implement an applet or a stand-alone application:

- If your application runs within the JVM environment provided by a web browser, you can download the application to client computers as a custom applet at run time.
- If your application runs as a stand-alone application in the JVM environment provided by a user's computer, you will most likely distribute the application with an installation utility on CD or through web downloads.

Applets

A Java applet can be downloaded and run on the user's computer automatically when a web page is displayed or when a user makes a selection that invokes the applet. The byte-code is stored in a file separate from the web page.

An XHTML object element similar to the following example can be embedded in an HTML page to run a Java applet as soon as the web page is loaded:

```
<object archive="URIList" data="data_URL" type="mime_type">
<param name="param_name" value="param_value" />
Alternate text.
</object>
```

In the example, `archive` provides a comma-separated list of URIs pointing to the Java archive (JAR) files that will be preloaded when the HTML page is displayed, `data` specifies the location of any object data to

be processed, `type` is the MIME type associated with the object data, and `param` supplies any information that the applet needs to perform its intended operations. Many applets require more than one `<param />` tag.

Alternatively, you can declare an object element, which causes the applet to be activated only when instantiated through the selection of a hyperlink. For more information about object elements, visit the W3C website at www.w3.org.

To support browsers that do not recognize HTML 4.0 or XHTML 1.0, you must supply an `<applet>` tag instead. To specify the use of the XPAAJ.jar file with an applet in this case, enter an applet tag similar to the following example:

```
<applet code="classname.class" archive="XPAAJ.jar" width="80"
  height="50" alt="This applet inserts information into a PDF form.">
<param name="param_name" value="param_value" />
Alternate text.
</applet>
```

In the example, `code` indicates which class contains the entry point of the applet, and `archive` specifies the relative path to the JAR file that contains *`classname`*`.class`.

Stand-alone applications

Stand-alone Java applications run independently of any browser and are portable across many platforms. Your application may have a graphical user interface or a command-line interface. In addition, the application may run on a stand-alone computer or be installed in a client-server environment.

For people who don't have continuous access to the Web, stand-alone applications are more useful than applets. Full-fledged applications have many advantages compared to applets, including higher performance and access to trusted methods. In addition, applications have complete control over the entire environment in which the application runs (no sandbox).

The best way to deliver a stand-alone application may involve bundling the installation instructions (or installation utility) and your .class, resource, and data files into a single JAR file. If you won't be redistributing the Java 2 run-time environment, be sure to include instructions for how users can obtain the software from Sun™ Microsystems™, Inc.

Server-side deployments

A *server* can be any program that responds to client requests. For server-side deployments, you could implement JSP technology or a servlet:

- If your application responds to client requests through a web server, deployment may involve creating custom JavaServer pages.
- If your application runs as part of a network service, deployment could involve adding a custom servlet to an existing server installation. Many application server vendors supply the associated deployment tools for J2EE servers.

If your application extends the capabilities of a web server, the web components must be supported by the services of a web container, such as Apache Tomcat, at run time. You will need to include a web application deployment descriptor in an XML file. The deployment descriptor, all web components and resources, and any server-side utility classes will need to be packaged in a web archive file.

JSP technology

JSP technology can be used to create platform-independent, web-based server applications. For example, a web application can be built by using J2EE technology. J2EE technology supports the development of dynamic web pages and provides a platform for integrating enterprise applications.

JavaServer pages are ideally suited to situations where text-based markup such as HTML and XML needs to be generated. A JavaServer page, which combines HTML text and Java source code in the same document, executes within the server application environment as a Java servlet. Both are compiled to a servlet prior to execution by the servlet container virtual machine.

For example, the user may click a button on a JavaServer page, which sends a request to a web server. The web server responds by running a JSP-converted servlet, which makes calls to XML/PDF Access API for Java. The JSP-converted servlet contains programming logic that determines the operations to be performed on the specified PDF or XDP file and the appropriate response to return.

For more information about JSP technology, see the Sun™ JavaServer pages technology website at http://java.sun.com/products/jsp/.

Servlets

Servlets are often used to extend the functionality of web servers and may perform additional service-oriented functions where required to support enterprise workflows. For example, suppose you wrote a custom servlet that responds to HTTP requests, extracts data from a submitted HTML form, and saves that data in a PDF file. To submit input from users and run the servlet, the HTML form would contain a tag similar to the following example:

```
<form method="post" action="servlet_URL" >
...
</form>
```

In the example, `method` specifies the `post` argument, and `action` provides the URL of the servlet that processes the form input. Input elements may be hidden in the form to send certain data items to the servlet without the user seeing those elements. For more information about servlets, see the Sun servlet website at http://java.sun.com/products/servlet/.

Deploying custom applications

To deploy the XPAAJ.jar file with a custom stand-alone application, update the `CLASSPATH` environment variable to specify the location of the JAR file.

Deploying a custom web application

You can deploy the XPAAJ.jar file in an applet or with a custom JSP or servlet that runs on an application server or in a servlet container:

- To deploy a web application, include the XPAAJ.jar file in the application's lib directory.
- To deploy a WAR file, add the XPAAJ.jar file to the appropriate location:
  - If the application is not yet deployed, add the XPAAJ.jar file to the lib directory of the WAR file.
  - If the application is already deployed, add the file to the application's WEB-INF/lib directory.
- To deploy a web application to a Tomcat JSP/servlet container, add the XPAAJ.jar file to the shared/lib directory of the Tomcat installation.

Index

I

J

L

M

N

O

P

Q

R

S

T

U

W

X