



# Java™ Print Service API User Guide

---

(Draft 0.1)

Alpha Draft

Sun Microsystems, Inc.  
901 San Antonio Road  
Palo Alto, CA 94303  
U.S.A. 650-960-1300

Part No. 8xx-xxxx-xx  
Month 2001, Revision 01

Send comments about this document to: [java-print@sun.com](mailto:java-print@sun.com)

<b>1. Introduction.....</b>	<b>5</b>
History of Printing on the Java Platform .....	5
What the Java Print Service API Offers.....	6
Java Print Service Architecture.....	6
The javax.print Package .....	6
Discover Print Services .....	7
Specify the Print Data Format .....	7
Create Print Jobs.....	7
The javax.print.event Package .....	8
How Applications Use the Java Print Service .....	8
A Basic Example.....	9
<b>2. Attributes.....</b>	<b>11</b>
Attribute Categories and Values .....	11
Attribute Roles .....	12
Attribute Sets.....	13
How to Specify Attributes.....	14
Standard Attributes .....	15
OrientationRequested .....	15
Copies.....	16
Media.....	16
MediaSize .....	17
MediaPrintableArea.....	17

Sun Microsystems, Inc.  
901 San Antonio Road  
Palo Alto, CA 94303  
U.S.A. 650-960-1300

	Destination .....	18
	SheetCollate .....	18
	Sides .....	19
	Fidelity .....	19
	Uses of JPS Attributes.....	19
<b>3.</b>	<b>Specifying Document Types.....</b>	<b>21</b>
	Client-Formatted Print Data.....	22
	MIME Types for Preformatted Data .....	22
	Text Data .....	23
	Page Description Language Documents.....	23
	Image Data.....	23
	Autosense Print Data .....	24
	Representation Classes.....	24
	Importance of Character Encoding .....	25
	Service-Formatted Print Data.....	25
	How to Use DocFlavor .....	26
<b>4.</b>	<b>Printing and Streaming Documents.....</b>	<b>28</b>
	StreamPrintService Versus PrintService.....	28
	Locating Services .....	29
	Discovering Print Services .....	29
	Discovering Stream Print Services.....	30
	Obtaining a Print Job.....	31

Sun Microsystems, Inc.  
901 San Antonio Road  
Palo Alto, CA 94303  
U.S.A. 650-960-1300

Creating a Doc .....	31
Registering for Events.....	32
Print Service Events .....	32
Print Job Events.....	33
PrintJobAttributeListener .....	33
PrintJobListener.....	33
Submitting the Print Job.....	34
Submitting the Print Job to the Printer .....	34
Submitting the Print Job to the Stream.....	35
Print Service Providers.....	36
<b>5. Printing and Streaming 2D Graphics .....</b>	<b>37</b>
Using PrinterJob to Print or Stream Graphics.....	37
Printing 2D Graphics.....	38
Streaming 2D Graphics .....	39
Using Service-Formatted Data.....	40
Printing the Service-Formatted Data.....	40
Streaming Service-Formatted Print Data .....	40
<b>A. Example: PrintPS.java .....</b>	<b>42</b>
<b>B. Example: PrintGIFtoStream.java .....</b>	<b>44</b>
<b>C. Example: Print2DPrinterJob.java .....</b>	<b>46</b>
<b>D. Example: Print2DGraphics.java .....</b>	<b>48</b>
<b>E. Example: Print2DtoStream.java .....</b>	<b>51</b>

Sun Microsystems, Inc.  
901 San Antonio Road  
Palo Alto, CA 94303  
U.S.A. 650-960-1300

<b>F. Example: PrintGIF.java.....</b>	<b>53</b>
<b>G. Java™ Print Service Glossary.....</b>	<b>56</b>

Sun Microsystems, Inc.  
901 San Antonio Road  
Palo Alto, CA 94303  
U.S.A. 650-960-1300

# Introduction

---

The Java™ Print Service (JPS) is a new Java Print API that is designed to support printing on all Java platforms, including platforms requiring a small footprint, but also supports the current Java™ 2 Print API. This unified Java Print API includes extensible print attributes based on the standard attributes specified in the Internet Printing Protocol (IPP) 1.1 from the IETF Specification, [RFC 2911](#). With the attributes, client and server applications can discover and select printers that have the capabilities specified by the attributes. In addition to the included `StreamPrintService`, which allows applications to transcode print data to different formats, a third party can dynamically install their own print services through the Service Provider Interface.

---

## History of Printing on the Java Platform

Basic printing support for the Java platform was first introduced in the Java Development Kit, version 1.1 in 1997. The JDK 1.1 printing API provided developers with a basic framework for printing the user-interface content from client applications. JDK 1.1 printing, also called the AWT Printing API, was designed around the `java.awt.PrintJob` class, which encapsulates a printing request. The `PrintJob` class creates a subclass of `Graphics`, which implements the rendering calls to image the page.

In 1998, the SDK 1.2 advanced printing on the Java platform with the `java.awt.print` package by allowing applications to print all Java 2D graphics, which includes 2D graphics, text, and images.

For the SDK version 1.3, the `JobAttributes` and `PageAttributes` classes were introduced to AWT printing so that client applications could specify the properties of a print job and the attributes of a page.

The two APIs primarily support page imaging, which is the rendering and formatting of a page and is one component of a printing subsystem. Although a client can choose a printer from a print dialog and can specify printing attributes using `JobAttributes` and

PageAttributes, neither the AWT nor Java 2D printing APIs support discovery of printers based on their capabilities, which is another key component of any printing subsystem. For more details on the history of printing on the Java platform refer to [Printing on the Java Platform \(http://java.sun.com/printing/whitepaper.html\)](http://java.sun.com/printing/whitepaper.html).

---

## What the Java Print Service API Offers

The Java Print Service API unifies and extends printing on the Java platform by addressing many of the common printing needs of both client and server applications that are not met by the current Java printing APIs.

In addition to supporting the current Java 2D printing features, the Java Print Service offers many improvements, including:

- Both client and server applications can discover and select printers based on their capabilities and specify the properties of a print job. Thus, the JPS provides the missing component in a printing subsystem: programmatic printer discovery.
  - Implementations of standard IPP attributes are included in the JPS API as first-class objects.
  - Applications can extend the attributes included with the JPS API.
  - Third parties can plug in their own print services with the Service Provider Interface.
- 

## Java Print Service Architecture

The Java Print Service API consists of four packages: `javax.print`, `javax.print.attribute`, `javax.print.attribute.standard`, and `javax.print.event`.

### The `javax.print` Package

The `javax.print` package is the main package of the API. It contains classes and interfaces that you use to:

- Discover Print Services
- Specify the print data format
- Create print jobs from a print service
- Send the print data to a printer or a stream

## Discover Print Services

An application invokes the static methods of the abstract class `PrintServiceLookup` to locate print services that have the capabilities to satisfy the application's print request. For example, to print a double-sided document, the application first needs to find printers that have the double-sided printing capability.

An implementation of the `PrintService` interface represents an actual printer that might be returned from an implementation of `PrintServiceLookup`. The `PrintService` can be queried to determine its capabilities by invoking one of the many methods that return the print service's supported attributes. The *Locating Services* section of the *Printing and Streaming Documents* chapter discusses using `PrintService` and `PrintServiceLookup`.

The Java 2 SDK includes `PrintServiceLookup` implementations that can locate the standard platform printers. To locate other types of printers, such as IPP printers or JINI printers, a print-service provider can write implementations of `PrintServiceLookup`. The print-service provider can dynamically install these `PrintServiceLookup` implementations using the SPI JAR file specification.

Rather than sending data to a printer, an application can also print the data to an output stream using a `StreamPrintService`, which allows the application to convert the data to different formats. To obtain a `StreamPrintService`, an application uses the lookup methods of the `StreamPrintServiceFactory` abstract class. This class is similar to that of the `PrintServiceLookup` class in that it also discovers print services. The chapter *Printing and Streaming Documents* describes using `StreamPrintServiceFactory` and `StreamPrintService`. A `StreamPrintService` can also be used to send 2D graphics to an output stream. See *Printing and Streaming 2D Graphics* for more details.

## Specify the Print Data Format

The `DocFlavor` class represents the print data format, such as JPEG or PostScript. A `DocFlavor` object consists of a MIME type describing the format, and a representation class name, such as `java.io.InputStream`, that indicates how the document will be delivered to the service. As a convenience, the JPS API also includes pre-defined `DocFlavor` objects representing common data formats. The *Specifying Document Types* chapter describes in more detail how to use `DocFlavor` objects.

## Create Print Jobs

When an appropriate print service is found, an application creates a print job from it. The print job is represented by the `DocPrintJob` class, which provides the actual `print` method. Before calling `print`, the application creates a `Doc` object to encapsulate the print data and the print request attributes, which are defined in the `javax.print.attribute` and `javax.print.attribute.standard` packages. The chapter *Printing and Streaming Documents* discusses using `Doc` and `DocPrintJob`. An application that is using the Java Print

Service to print 2D Graphics can use either the `PrinterJob` class or the `DocPrintJob` class. The chapter *Printing and Streaming 2D Graphics* discusses using the Java Print Service to print 2D Graphics.

### The Attribute Packages

The `javax.print.attribute` and `javax.print.attribute.standard` packages define print attributes, which describe a capability of a print service, a characteristic of a document, an instruction for processing a document or an entire print job, or the state of a print job or printer.

The `javax.print.attribute` package describes the types of attributes and how they can be collected into sets. The `Attribute` interface is the superinterface for all attributes. The `javax.print.attribute` package includes classes and interfaces defining the five different kinds of attributes, each of which describes the capabilities of one piece of the printing process. For example, the `PrintRequestAttribute` interface defines attributes that clients use to describe the characteristics of a print job, which might include the number of copies to print.

The `javax.print.attribute.standard` package enumerates all of the standard attributes supported by the API, most of which are implementations of attributes specified in the IPP specification. The Attributes chapter discusses the attribute API in more detail.

## The `javax.print.event` Package

The `javax.print.event` package contains classes that allow applications to register for events on print jobs and print services. Read the *Registering for Events* section for more information on registering for print job events.

---

# How Applications Use the Java Print Service

A typical application using the Java Print Service API performs these steps to process a print request:

1. Obtain a suitable `DocFlavor`, which is a class that defines the format of the print data.
2. Create and populate an `AttributeSet`, which encapsulates a set of attributes that describe the desired print service capabilities, such as the ability to print five copies, stapled, and double-sided.
3. Lookup a print service that can handle the print request as specified by the `DocFlavor` and the attribute set.
4. Create a print job from the print service.

5. Call the print job's print method.

The application performs these steps differently depending on what and how it intends to print. The application can either send print data to a printer or to an output stream. The print data can either be a document in the form of text or images, or a Java object encapsulating 2D Graphics. If the print data is 2D graphics, the print job can be represented by either a `DocPrintJob` or a `PrinterJob`. If the print data is a document then a `DocPrintJob` must be used.

The combinations of printing methods and print data formats yield a choice of six printing mechanisms:

- Print a document to a printer by using a `DocPrintJob` and an implementation of `PrintService`
- Stream a document to an output stream by using a `DocPrintJob` and a `StreamPrintService`
- Print 2D graphics to a printer by using a `DocPrintJob` and an implementation of `PrintService`
- Stream 2D graphics to an output stream by using a `DocPrintJob` and a `StreamPrintService`
- Print 2D graphics to a `PrintService` using `java.awt.print.PrinterJob`
- Stream 2D graphics to a `StreamPrintService` using `java.awt.print.PrinterJob`

The *Attributes* chapter and the *Specifying Document Types* chapter describe how to create an attribute set and specify document types for use with any of the print mechanisms. The *Printing and Streaming Documents* chapter discusses printing and streaming documents using `DocPrintJob`. The *Printing and Streaming 2D Graphics* chapter explains printing and streaming 2D graphics using both `DocPrintJob` and `PrinterJob`.

## A Basic Example

Most applications using the Java Print Service API will probably send a document directly to a printer, which the following code sample demonstrates:

```
// Input the file
FileInputStream textStream;
try {
    textstream = new FileInputStream("file.TXT");
} catch (FileNotFoundException ffne) {
}
if (textstream == null) {
    return;
}
// Set the document type
DocFlavor myFormat = DocFlavor.INPUT_STREAM.TEXT_PLAIN_ASCII;
// Create a Doc
```

```

Doc myDoc = new SimpleDoc(texttream, myFormat, null);
// Build a set of attributes
PrintRequestAttributeSet aset = new HashPrintRequestAttributeSet();
aset.add(new Copies(5));
aset.add(MediaSize.ISO_A4);
aset.add(Sides.DUPLEX);
// discover the printers that can print the format according to the
// instructions in the attribute set
PrintService[] services =
    PrintServiceLookup.lookupPrintServices(myFormat, aset);
// Create a print job from one of the print services
if (services.length > 0) {
    DocPrintJob job = services[0].createPrintJob();
    try {
        job.print(myDoc, aset);
    } catch (PrintException pe) {}
}

```

Although this sample only demonstrates one of the six ways to print, the other printing mechanisms work in a similar way. The rest of this guide discusses each piece of the printing process and all the printing mechanisms in more detail

## Attributes

---

When sending print data to a printer, a client also provides instructions on how to print the data, such as: what media (paper) to use and how many copies to print. The client specifies these processing instructions using the attribute definitions of the Java™ Print Service API.

The Java Print Service API includes two packages that define attributes: `javax.print.attribute` and `javax.print.attribute.standard`. The `javax.print.attribute` package contains the interfaces and classes that describe the types of attributes and how they are collected into sets. The `javax.print.attribute.standard` package enumerates all of the standard attributes in the API. Most of the attribute definitions in the JPS API are implementations of the standard and extensible attributes defined by the Internet Printing Protocol (IPP) Specification from the IETF.

This chapter describes the attribute definitions of the JPS API in more detail. The *Attribute Categories and Values* section explains what an attribute is according to the JPS. The *Attribute Roles* section describes the type of attributes. The *Attribute Sets* section describes how the attributes are collected into sets. The *Standard Attributes* section describes some of the more-commonly used attributes defined in the `javax.attribute.standard` package.

---

## Attribute Categories and Values

Every printer has a set of capabilities, such as the ability to print on different paper sizes or the ability to print more than one copy. Each of the capabilities has a range of values. For example, a printer's orientation capability might have this range of values: [landscape, portrait]. For each print request, the capability is set to one of these possible values. The Java Print Service API uses the term **attribute category** to refer to a printer capability, such as orientation, and the term **attribute value** to refer to the value of the capability, such as landscape.

In the Java Print Service API, an attribute category is represented by a Java class implementing the `Attribute` interface, and attribute values are instances of such a class or one of its subclasses. For example, to print 5 copies of a job, an application constructs an instance of the `Copies` class with the value of 5 and uses the `Copies` instance to specify the print job. The `Copies` class represents the attribute category, and the `Copies` instance represents the attribute value. Because the attribute value is encapsulated as a Java object, the attribute value implies its attribute category, and so an attribute value is usually referred to simply as an attribute. The category and value constitute a key-value pair, which you can add to a set along with other attributes. See *Attribute Sets* for more information

The `Attribute` interface defines just two methods: `getName` and `getCategory`. The `getName` method returns a `String` name for the attribute category. The `getCategory` method returns the class that first implements that category. To ensure that all implementations of a particular category report the same name and class, subclasses should not override the `getName` or `getCategory` methods.

To determine what attribute categories and values are supported by a printer, use the query methods defined in the `PrintService` interface. A print request does not need to specify attribute values for all of the target printer's supported attribute categories: each supported attribute category has a default value. For example, a printer might support printing up to 999 copies of a job, but if a print request does not specify the number of copies, the printer almost always prints a default of just 1 copy.

---

## Attribute Roles

When submitting a print job to a printer, the client provides the attributes describing the characteristics of the print data, such as the document name, and how the print data should be printed, such as double-sided, five copies. If a print job consists of multiple pieces of print data, different pieces might have different processing instructions, such as 8 x 11 inch media for the first document, and 11 x 17 inch media for another document.

Once the printer starts processing the print job, additional information about the job becomes available, which might include: the job state (such as completed or queued) and the number of pages printed so far. These pieces of information are also attributes. Attributes can also describe the printer itself, such as: the printer name, the printer location, and the number of jobs queued.

The Java Print Service API defines these different kinds of attribute roles with five subinterfaces of `Attribute`:

- `PrintRequestAttribute` is used by an application to represents a setting applied to an entire print job and to specify how the entire print job should be printed.
- `DocAttribute` is used by an application to specify a characteristic of a single document and the print job settings to be applied to the document.

- `PrintJobAttribute` is used by a print service to report how a job is being printed. These values will usually be identical to the requested attribute values. However, if the printer does not support the value of a print request attribute then the corresponding print job attribute contains a different value that is supported by the printer
- `PrintServiceAttribute` is used by a print service to report the status of the print service.
- `SupportedValuesAttribute` is used by a print service to indicate the range of values supported for a request attribute. For example, a printer might support only a certain range of copies. When a print service is queried for the supported range of copies, it returns this information in a `CopiesSupported` object, which implements `SupportedValuesAttribute`.

Many attributes can be members of more than one role. For example, the `Media` attribute belongs to the `doc`, `print job`, and `print request` roles because the `Media` attribute can describe the paper size, the paper tray, or paper type. In fact, most `doc` attributes are also request attributes, and all request attributes are also job attributes.

The next section describes how to collect attributes together into attribute sets, which also have roles.

---

## Attribute Sets

A client usually needs to provide more than one processing instruction when submitting a print job. For example, the client might need to specify an A4 media and a landscape orientation. To send more than one processing instruction to the printer, the client collects the attributes representing the instructions into an attribute set, which the Java Print Service API represents with the `AttributeSet` interface.

The `AttributeSet` interface is similar to the `java.util.Map` interface: it provides a map of keys to values, in which each key is unique and can contain no more than one value. However, the `AttributeSet` interface is designed to specifically support the needs of the Java Print Service API. An `AttributeSet` requires that:

- Each key in an `AttributeSet` corresponds to a category, and the value of the key can only be one of the attribute values that belong to the category represented by the key. Thus, unlike a `Map`, an `AttributeSet` restricts the possible values of a key: an attribute category cannot be set to an attribute value that does not belong to that category.
- Only attributes implementing the `Attribute` interface can be added to the attribute set.
- Only one attribute from a particular category can exist in an `AttributeSet`. For example, if an attribute set already contains a `Media` attribute, adding another `Media` attribute overwrites the first one.

Like a single attribute, an attribute set has a role. The roles are defined by the `AttributeSet` subinterfaces, which parallel the `Attribute` subinterfaces mentioned in the *Attribute Roles* section. The `AttributeSet` subinterfaces are: `PrintRequestAttributeSet`,

`PrintJobAttributeSet`, `DocAttributeSet`, and `PrintServiceAttributeSet`. The role of the attribute set determines what kind of attributes it contains: a print request attribute set can contain only print request attributes.

The Java Print Service API includes `HashSet` as a concrete implementation of the `AttributeSet` interface, but developers can provide their own implementation. Rather than using `HashSet` directly, applications will more often use one of the subclasses of `HashSet`, which are: `HashPrintRequestAttributeSet`, `HashPrintJobAttributeSet`, `HashDocumentAttributeSet`, and `HashPrintServiceAttributeSet`. These subclasses implement the corresponding `AttributeSet` subinterfaces. For example `PrintRequestAttributeSet` extends `AttributeSet`, and `HashPrintRequestAttributeSet` is the concrete implementation of `PrintRequestAttributeSet`. An attribute set created with `HashPrintRequestAttributeSet` or any implementation of `PrintRequestAttributeSet` can contain only request attributes.

Unlike its subclasses, a `HashSet` can contain any kind of attribute. An application uses a `HashSet` directly when it needs a set to contain more than one kind of attribute. One example is the set returned by the `getUnsupportedAttributes` method of `PrintService`. The unsupported attributes returned might belong to more than one attribute role.

As with single attributes, applications will more often create request attribute sets because specifying print requests is the printing application's primary function. To create a request attribute set, use one of the constructors from the `HashPrintRequestAttributeSet` class:

```
PrintRequestAttributeSet aset = new HashPrintRequestAttributeSet();
```

The next section describes some of the more commonly-used attributes that you can add to a set.

---

## How to Specify Attributes

Whether you are printing or streaming data, you specify attributes in the same way. You will usually use `PrintRequestAttribute` more often than the other kinds of attributes because print request attributes specify the settings for an entire print job. Because you will most likely want to specify more than one attribute for a particular job, you need to create an attribute set. This sample demonstrates creating a `PrintRequestAttributeSet` that instructs the service to use A4 media to print five copies of the job using both sides of the paper:

```
PrintRequestAttributeSet aset = new HashPrintRequestAttributeSet();
aset.add(new Copies(5));
aset.add(MediaSize.ISO_A4);
aset.add(Sides.DUPLEX);
```

After creating an attribute set, you will pass it to the print job's print method, along with the DocFlavor. The *Specifying Document Types* discusses using DocFlavor. The *Printing and Streaming Documents* chapter discusses how to print. The next section explains some of the more commonly-used attributes in more detail.

---

## Standard Attributes

The `javax.print.attribute.standard` package enumerates all of the standard attributes in the Java Print Service API. Most of these standard attributes are taken from the attributes defined in the IETF Internet Printing Protocol (IPP) 1.1 specification. This means that each IPP-compliant attribute class(category) defined in package `javax.print.attribute.standard` corresponds to an IPP attribute category, and the name (as returned by `getName`) is the actual IPP name for the category. The class names also usually reflect the IPP name as closely as the coding conventions of the Java programming language permit. Furthermore the values defined for a category are the same as the IPP values. IPP compatibility of each attribute category is documented in the API specification.

This section describes the attributes that developers will probably use most frequently. The more commonly-used attributes, including the ones listed here, implement `PrintRequestAttribute` because printing applications will usually specify how an entire print job should be printed, which is the role of a `PrintRequestAttribute`.

## OrientationRequested

The `OrientationRequested` attribute category allows you to specify the orientation of the imaging on the paper. The possible attribute values are: `PORTRAIT`, `LANDSCAPE`, `REVERSE_PORTRAIT`, and `REVERSE_LANDSCAPE`. `OrientationRequested.PORTRAIT` is usually the default value. This code snippet demonstrates adding an `OrientationRequested` attribute to a set:

```
aset.add(OrientationRequested.REVERSE_PORTRAIT);
```

The `OrientationRequested` object is a type-safe enumeration encapsulating `String` values, which correspond to the possible orientations. These values are the actual IPP keywords.

Some pre-formatted document types, such as "Postscript", might not be able to support this attribute category because pre-formatted document types embed printer language commands that are interpreted by the printer, and these commands take precedence over a client request.

Clients can discover the supported orientation values for a particular print service by calling:

```
PrintService.getSupportedAttributeValues(OrientationRequested.class, ...).
```

This method returns an array of type `OrientationRequested` enumerating the supported values.

## Copies

The `Copies` attribute category allows you to specify the number of copies to print. The `Copies` class encapsulates an integer representing the number of copies requested. This code snippet demonstrates adding a `Copies` attribute, set to five copies, to an attribute set:

```
aset.add(Copies(5));
```

Clients can discover the range of copies that a print service supports by calling :

```
PrintService.getSupportedAttributeValues(Copies.class, ...)
```

This method returns a `CopiesSupported` object, which encapsulates a range of integer values representing the range of copies that the service can handle. Calling `getSupportedAttributeValues` with `CopiesSupported` instead of `Copies` always returns `null` because the `CopiesSupported` object does not implement the `PrintRequestAttribute` interface, and so a client cannot specify a `CopiesSupported` attribute in a print request.

This code sample demonstrates discovering if a print service supports printing 5 copies and adding a `Copies` attribute with a value of 5 copies to an attribute set:

```
CopiesSupported copSupp =  
    (CopiesSupported) service.getSupportedAttributeValues(Copies.class, null,  
                                                         null);  
  
if (copSupp != null && copSupp.contains(5)) {  
    requestAttrSet.add(new Copies(5));  
} else { ...  
}
```

## Media

Media is the IPP attribute that identifies the medium on which to print. The `Media` attribute is an important attribute to understand, but is relatively complex.

The Java Print Service API defines three subclasses of the abstract class `Media` to reflect the overloaded `Media` attribute in the IPP specification: `MediaSizeName`, `MediaName` and `MediaTray`. All the `Media` subclasses have the `Media` category, for which each subclass defines different standard attribute values. For example, a `MediaTray` object can specify a value of `MANUAL` for the `Media` attribute to indicate that the document must be printed on paper from the printer's manual tray. This code snippet demonstrates adding a `Media` attribute to a set:

```
aset.add(MediaTray.MANUAL);
```

The value of the `Media` attribute is always a `String`, but because the attribute is overloaded, its value determines the type of media to which the attribute refers. For example, the IPP pre-defined set of attribute values include the values “a4” and “top-tray”. If `Media` is set to the value “a4” then the `Media` attribute refers to the size of paper, but if `Media` is set to “top-tray” then the `Media` attribute refers to the paper source. Because the `String` attribute value can refer to such diverse types of media, an application can extend the attribute set to include values such as “company-letterhead” or “yellow letter paper”. Of course, to extend the `Media` attribute in this way, an application must discover a print service that is configured to print with this media.

In most cases, applications will use either `MediaSizeName` or `MediaTray`. The `MediaSizeName` class enumerates the media by size. The `MediaTray` class enumerates the paper trays on a printer, which usually include a main tray and a manual feed tray. The IPP 1.1 specification does not provide for specifying both the media size and the media tray at the same time, which means, for example, that an application cannot request size A4 paper from the manual tray. A future revision of the IPP specification might provide for a way to request more than one type of media at a time, in which case the JPS API will most likely be enhanced to implement this change.

The JPS API also includes two additional media-related `Attribute` classes that are not IPP attributes: `MediaSize` and `MediaPrintableArea`.

## MediaSize

`MediaSize` is not a request attribute; it is an enumeration of paper dimensions and a mapping to `MediaSizeName` instances. Each `MediaSizeName` instance usually has a `MediaSize` object associated with it so that clients can obtain the dimensions of the paper that the `MediaSizeName` instance defines. To determine the dimensions of the `MediaSizeName` instance, call:

```
MediaSize size = MediaSizeName.getMediaSizeForName(paper);
```

## MediaPrintableArea

`MediaPrintableArea` is used in a print request in conjunction with a compatible `Media` to specify the area of the paper on which to print. Printer hardware usually defines the printable area of a page, which is rarely the entire page. For this reason, an application needs to determine what printable area a printer defines for a particular size media to ensure that the print data can fit within this area.

For example, to determine the supported printable area for 5" x 7" paper, the application needs to choose a media size attribute value that corresponds to this size paper and then query the print service with the media size:

```
PrintRequestAttributeSet aset = new HashPrintRequestAttributeSet();
```

```
aset.add(MediaSizeName.NA_5X7);
MediaPrintableArea printableArea =
(MediaPrintableArea)service.
    getSupportedAttributeValues(MediaPrintableArea.class, null, aset);
```

The returned value indicates the largest printable area that can be supported by that printer for that paper size.

## Destination

The Destination attribute allows you to redirect your print data to a file rather than sending it to a printer device. The “print-to-file” option is very common in user dialogs, but the spooled data is not always usable because it might be a device-specific raster that can only be interpreted by the device from which it was redirected. For this reason, the Java Print Service API requires that the client query the print service to determine if it can redirect the output to a file. The service might not support the category at all, or it might support only particular values. For example, since the JPS API can be used in a network environment, in which the formatting of print data does not occur on the host computer, specifying a local file for output might not be possible because the service formatting the data might not have access to the local filesystems of the client. The Destination attribute takes a URL as the value of the destination, which allows a network printer to use a protocol such as ftp to upload formatted print data. However, most printers that support this attribute will run as part of a local environment and can accept the “file:” protocol URL. This code snippet redirects the output to a file called out.prn on the c: drive:

```
aset.add(new Destination("file:c:\out.prn"));
```

## SheetCollate

The SheetCollate attribute allows you to specify whether or not your print job is collated when you are printing more than one copy of a multi-page document. For example, the pages of a 3-page, 2-copy collated job will print as (1,2,3,1,2,3), but the pages of the same document submitted in a 2-copy uncollated job will be printed as (1,1,2,2,3,3). This attribute is not in version 1.1 of the IPP specification, but it is very useful and most printers support it. This code snippet demonstrates specifying a collated job:

```
aset.add(SheetCollate.COLLATED);
```

## Sides

Some printers, particularly high-end printers, can print on both sides of the paper. The Sides attribute allows applications to specify two-sided printing instead of the usual default of single-sided printing. Two-sided printing is sometimes referred to as "duplex" or "tumble" printing. These two values are differentiated by the orientation of the output. The Java Print Service API refers to duplex as "two sided long edge" and tumble as "two sided short edge". Read the API specification for Sides for further explanation. This code snippet specifies that a job print the documents two-sided:

```
aset.add(Sides.DUPLEX);
```

## Fidelity

The Fidelity attribute is an IPP boolean attribute that represents whether or not a print job should be rejected if the print service does not support any attribute specified in the print request. Fidelity is not an attribute many developers will need to consider, but it is an important attribute in the context of the JPS API. The default value is FIDELITY\_FALSE, which indicates that a print job should not be rejected if the print service does not support an attribute specified in the print request. For example, if an application specifies an orientation of reverse landscape, but the printer does not support reverse landscape, the job is rejected if fidelity is true, but if fidelity is false then the printer might substitute a reasonable alternative, such as the landscape orientation. The Fidelity attribute allows applications to decide whether to only print the document exactly as specified or to print it even though the printer might not support all attributes. This code snippet specifies that a job should be rejected if the printer does not support the requested attributes:

```
aset.add(Fidelity.FIDELITY_TRUE);
```

For the cases in which fidelity is important, the Java Print Service API provides many tools for applications to query exactly what can be supported for a particular print request. See the various query methods on the PrintService interface.

---

## Uses of JPS Attributes

Unlike attributes of previous Java printing APIs, the Java Print Service attributes can be used in both the program and from the user dialog. Applications can specify as little as needed about how a job should be printed and can inspect the selections made by end users in user dialogs.

The JPS attributes are designed to be extensible. An AttributeSet can contain anything that properly implements the Attribute interfaces, which means that the set of standard attributes can evolve and still be compatible with future SDK versions, and implementors of services can extend attributes in two ways:

- Using their own subclasses of a standard attribute category to support additional values of that category.
- Creating new attribute categories.

Demonstrating attribute usage from the user dialog requires providing a user interface, which is beyond the scope of this user guide. Likewise, extending JPS attributes is a task for service providers, and is not included as an example in this version of the user guide. However, the Printing a Document chapter explains a simple printing application, which covers the basics of using the JPS attributes as described in this chapter.

## Specifying Document Types

---

From the user's perspective, a document can take many different forms, including: a PDF file, an image from a digital camera, an email, a word processor document, or a web page. Before printing a document of a particular format, the client needs to ensure that the printer can understand the format. Sometimes a printer can directly print documents of a given format: photo printers can directly print images of various formats, and PostScript™ printers can directly print a PostScript™ document. However, there are few printers that can directly print a wide range of formats; most printers require some higher-level software support to translate the source document into a format they can print.

A printing API needs to provide a way to describe document types so that:

- The printer can report what formats it can print.
- The client can describe the format of the data it wants to print.
- The client can describe the encoding of text data.

The Java™ Print Service API describes document types using the `DocFlavor` class. A `DocFlavor` is comprised of:

- A MIME type that tells the printer how to interpret the data.
- A representation class name indicating the Java class that describes how the data is sent to the printer.

To describe an HTML page to a print service, a client might choose to use a `DocFlavor` with a MIME type string of "text/html; charset=utf-16" and a representation class name of "java.io.InputStream". The client can obtain this `DocFlavor` in one of two ways:

- Construct a `DocFlavor`:  

```
DocFlavor htmlStreamFlavor = new DocFlavor("text/html; charset=utf-16",  
"java.io.InputStream");
```
- Use the pre-defined instance that represents this type of `DocFlavor`:  
`DocFlavor.INPUT_STREAM TEXT_HTML_UTF_16`. The Java Print Service API provides a set of pre-defined instances for common doc flavors as a convenience.

Because the HTML page contains text data, the MIME type String includes the text encoding, which is `charset=utf-16` in this example. The client is responsible for accurately describing the print data to the print service. The section, *Client-Formatted Print Data*,

explains how to properly construct a `DocFlavor` to accomplish this. If the text encoding is not included in the MIME type, unexpected results can occur, as explained in the section, *Importance of Character Encoding*. The client can allow the service to determine the format of data that the client supplies as a Java object. The *Service-Formatted Print Data* section describes using a `DocFlavor` to represent service-formatted print data.

Keep in mind that, just because the `DocFlavor` API has a pre-declared doc flavor, this doesn't mean that an implementation of the particular flavor is available. For example, even if you use the pre-defined `DocFlavor` representing HTML text in UTF-16, you won't be able to print the HTML unless you have a print service that supports printing HTML. Again, it is the user's responsibility to ensure that a printer supports a particular format.

---

## Client-Formatted Print Data

The client uses the `DocFlavor` to describe the print data format and to indicate how the data will be delivered to the print service. The MIME type specifies the data format. The representation class name specifies how the data will be delivered to the printer.

The `DocFlavor` class has an inner class for each of the common representation class names. Each inner class contains a set of `DocFlavor` object constants representing formats that can be delivered using the representation class. For example, an input stream can stream many different kinds of print data formats, including GIF, represented by `DocFlavor.INPUT_STREAM.GIF`, and PostScript<sup>TM</sup>, represented by `DocFlavor.INPUT_STREAM.POSTSCRIPT`. If one of the `DocFlavor` object constants accurately describes the format of a given piece of print data, the client can use this constant instead of constructing a `DocFlavor`.

If the appropriate `DocFlavor` is not already defined in the JPS API, a client can create it with the `DocFlavor` constructor. The client can use the MIME types and representation class names explained in the next two sections to construct the `DocFlavor`.

## MIME Types for Preformatted Data

The four most common types of preformatted data are: text data, page description language documents, image data, and autosense print data. This section describes each one of these data types and lists their corresponding MIME types.

## Text Data

Preformatted text data is usually provided in a character-oriented representation class, such as a character array, `String`, or `Reader`, or in a byte-oriented representation class, such as a byte array, input stream, or `URL`. Plain text and HTML are two common forms of preformatted text data. You can use these MIME type strings to represent the format of the data when constructing a `DocFlavor`:

---

<code>"text/plain"</code>	Plain text in the default character set US-ASCII
<code>"text/plain; charset=xxx"</code>	Plain text in character set xxx
<code>"text/html"</code>	HyperText Markup Language in the default character set (US-ASCII)
<code>"text/html; charset=xxx"</code>	HyperText Markup Language in the character set xxx

---

## Page Description Language Documents

Preformatted page description language (PDL) documents are usually provided in a byte-oriented representation class, such as byte array, `InputStream`, or `URL`. You can use these MIME type strings to represent the format of the data when constructing a `DocFlavor`:

---

<code>"application/pdf"</code>	Portable Document Format document
<code>"application/postscript"</code>	PostScript <sup>TM</sup> document
<code>"application/vnd.hp-PCL"</code>	Printer Control Language document

---

## Image Data

Preformatted image data is provided in a byte-oriented representation class: byte array, `InputStream`, or `URL`. You can use these MIME type strings to represent the format of the data when constructing a `DocFlavor`:

---

<code>"image/gif"</code>	Graphics Interchange Format image
<code>"image/jpeg"</code>	Joint Photographic Experts Group image
<code>"image/png"</code>	Portable Network Graphics image

---

## Autosense Print Data

Preformatted autosense print data allows the printer to decide how to interpret the print data. This type of data is usually provided in a byte-oriented representation class. You can use this MIME type string to represent the format of the data when constructing a DocFlavor:

```
"application/octet-stream"
```

## Representation Classes

For client-formatted print data, the print data representation class is usually one of the following:

---

Character array ( <code>char [ ]</code> )	The print data consists of the Unicode characters in the array. This representation class can provide text data and PDL data.
<code>java.lang.String</code>	The print data consists of the Unicode characters in the string. This representation class can provide text data.
A Character stream represented by <code>java.io.Reader</code>	The print data consists of the Unicode characters read from the stream up to the end-of-stream. This representation class can provide text data.
Byte array ( <code>byte [ ]</code> )	The print data consists of the bytes in the array. The bytes are encoded in the character set specified by the doc flavor's MIME type. If the MIME type does not specify a character set, the default character set is US-ASCII. This representation class can provide text data, PDL documents, and image data.
A Byte stream represented by <code>java.io.InputStream</code>	The print data consists of the bytes read from the stream up to the end-of-stream. The bytes are encoded in the character set specified by the doc flavor's MIME type. If the MIME type does not specify a character set, the default character set is US-ASCII. This representation class can provide text data, PDL documents, and image data.
Uniform Resource Locator, <code>java.net.URL</code>	The print data consists of the bytes read from the URL location. The bytes are encoded in the character set specified by the doc flavor's MIME type. If the MIME type does not specify a character set, the default character set is US-ASCII. This representation class can provide text data, PDL documents, and image data. To print documents to a network print service that might not have access to the URL, open an input stream on the URL and use an input stream data flavor instead.

---

---

## Importance of Character Encoding

If a MIME type for byte print data does not include a charset parameter, indicating the encoding, the Java Print Service assumes a character set of US-ASCII. This behavior is different than that of the Java runtime, which always assumes the default encoding for the user's locale on the underlying operating system, which might be different than US-ASCII.

Every instance of the Java<sup>TM</sup> Virtual Machine<sup>1</sup> has a default character encoding determined during virtual-machine startup that usually depends upon the locale and charset used by the underlying operating system. In a distributed environment, two JVM's might not share the same default encoding. Thus clients who want to stream platform-encoded text data from the host platform to a Java Print Service instance must explicitly declare the charset and not rely on defaults.

For these reasons, applications that stream text data should always specify the charset in the MIME type. To specify a MIME type, an application needs to determine the host platform's encoding, which it does by calling the `DocFlavor.hostEncoding` method. The MIME type returned from this method is guaranteed to be understood by the current JVM.

See [character encodings](#) for more information on the character encodings supported on the Java platform.

---

## Service-Formatted Print Data

Rather than the client describing the print data format with a MIME type, the client can supply a Java object from which a print service determines the print data format. For example, the Java object can encapsulate a PostScript<sup>TM</sup> document. Instead of the client explicitly describing the document as PostScript<sup>TM</sup> with a MIME type, the client can wrap a reference to the document in a Java object, which the print service inspects to determine that the format is PostScript<sup>TM</sup>. Because the print data is delivered to the print service through the Java object, the class that the Java object implements is the representation class.

The `DocFlavor` class has an inner class called `DocFlavor.SERVICE_FORMATTED` that contains `DocFlavor` object constants representing service-formatted print data. Each of these constants has a MIME type of "application/x-java-jvm-local-objectref", which indicates that the client will supply a reference to a Java object implementing the interface named as the representation class.

The three `DocFlavor` object constants contained in `DocFlavor.SERVICE_FORMATTED` each represent a common representation class name used with service-formatted print data:

1. As used in this guide, the terms "Java Virtual Machine" or "JVM" mean a virtual machine for the Java platform

- `DocFlavor.SERVICE_FORMATTED.PAGEABLE`:  
The client supplies an object that implements the `java.awt.print.Pageable` interface, which represents a set of pages to be printed. The printer calls methods in that interface to obtain the pages to be printed, one by one. For each page, the printer supplies a graphics context and prints whatever the client draws in that graphics context.
- `DocFlavor.SERVICE_FORMATTED.PRINTABLE`:  
The client supplies an object that implements the `java.awt.print.Printable` interface, which is responsible for drawing the contents of each page. The printer calls methods in that interface to obtain the pages to be printed, one by one. For each page, the printer supplies a graphics context and prints whatever the client draws in that graphics context.
- `DocFlavor.SERVICE_FORMATTED.RENDERABLE_IMAGE`:  
The client supplies an object that implements interface `java.awt.image.renderable.RenderableImage`, which represents an image that can be manipulated in a rendering-independent manner and can be rendered to various contexts, such as a printer, without compromising quality. The printer calls methods in the interface to obtain the image to be printed.

As stated in the *Introduction*, applications that wrap 2D graphics into a Java object can use the service-formatted print data `DocFlavor` constants. See *Printing and Streaming 2D Graphics* for more information.

---

## How to Use DocFlavor

These steps illustrate the role of the `DocFlavor` in a typical application:

1. A client determines the format of the print data and decides how to present the data to the printer. The client then creates or obtains a `DocFlavor` representing the format:

```
DocFlavor flavor = DocFlavor.INPUT_STREAM.GIF;
```

2. The client uses the `DocFlavor` to find printers that can understand the format specified by the `DocFlavor` and has the capabilities specified in the client's attribute set:

```
PrintService[] services =  
    PrintServiceLookup.lookupPrintServices(flavor, aset);
```

3. From a print service, the client creates a print job, represented by a `DocPrintJob` object:

```
DocPrintJob printJob = services[0].createPrintJob();
```

4. The client creates a `Doc` representing the document to be printed:

```
Doc doc = new InputStreamDoc("duke.gif", flavor);
```

5. The client prints the document by calling the print method of the DocPrintJob object:

```
printJob.print(doc, aset);
```

When the client passes the Doc object to the print method, the DocPrintJob object obtains the print data from the Doc object and determines the doc flavor that the client can supply. The doc flavor's representation class is a conduit for the DocPrintJob to obtain a sequence of characters or bytes from the client.

The *Printing and Streaming Documents* chapter demonstrates a complete printing application.

## Printing and Streaming Documents

---

Before submitting a print job, an application needs to locate print services that have the capability to handle the print data. Once the appropriate service is located, the application obtains a print job from it and submits the print job to the service.

This chapter demonstrates how to print and stream documents by:

- Locating appropriate services using the elements presented in the two previous chapters: Attributes and Document Flavors
- Creating a print job
- Submitting the print job to the printer or service
- Registering for events on the print job or service.

---

### StreamPrintService Versus PrintService

The Java™ Print Service API includes a `PrintService` class and a `StreamPrintService` class. A `StreamPrintService` extends `PrintService`, and so a `StreamPrintService` can be used anywhere a `PrintService` can be used. However, `PrintService` and `StreamPrintService` are used for different purposes. A `PrintService` is used to direct output to a printer; a `StreamPrintService` is used to export formatted print data to a stream, usually to a different format. When locating a `StreamPrintService`, you specify the required output format in the form of a MIME type argument and provide an `OutputStream` to receive the data. You do not provide a representation class when locating a `StreamPrintService`, as you do when locating a `PrintService`, because the output is always delivered to an `OutputStream`.

After locating a service, you obtain a print job and submit the print job to the service in the same way whether you are using a `PrintService` or a `StreamPrintService`. The major differences between `StreamPrintService` and `PrintService` are in the way they are located. The next section discusses locating both print services and stream print services.

---

# Locating Services

An application locates a print service in a slightly differently way from locating a stream print service. The *Discovering Print Services* section explains locating print services. The *Discovering Stream Print Services* section explains locating stream print services.

## Discovering Print Services

Before sending a print job to a printer, the application needs to find printers that have the capabilities to handle the print job. To print a double-sided document, the application first needs to find printers that have the double-sided printing capability.

The `javax.print.PrintServiceLookup` class included in the Java Print Service API provides static methods that applications use to locate printers.

An application invokes the `lookupPrintServices` method of `PrintServiceLookup` with a `DocFlavor` and an `AttributeSet`:

```
DocFlavor flavor = DocFlavor.INPUT_STREAM.POSTSCRIPT;
PrintRequestAttributeSet aset = new HashPrintRequestAttributeSet();
aset.add(MediaSizeName.IOS_A4);
aset.add(new Copies(2));
PrintService[] service =
    PrintServiceLookup.lookupPrintServices(flavor,
                                           aset);
```

This method returns an array of print services representing printers that have the capabilities specified in the attribute set and can print the data format specified in the doc flavor. See the *Attributes* chapter and the *Specifying Document Types* chapter for more help in choosing a `DocFlavor` and creating an `AttributeSet`.

The base set of printers returned from the `lookupPrintServices` method are the same as the set of printers returned by the platform. For example, when using Windows NT, the set of returned printers is the same as the set of printers visible in the Windows Printer Control Panel. Likewise, when using Solaris, the returned printers are the same as those enumerated by the System V Unix "lpstat" command. However, since third parties can augment these sets, additional printers, such as JINI printers, can be returned.

After obtaining a suitable `PrintService`, the application can access its many query methods to determine what values are supported for attribute categories. The *Obtaining a Print Job* section explains how to get a print job from the `PrintService`.

## Discovering Stream Print Services

An application uses stream print services to convert print data to different formats. This is useful, for example, if you do not have a printer that can print the format of a particular piece of print data. This section demonstrates how to convert a GIF document into Postscript using a `StreamPrintService`.

The `StreamPrintServiceFactory` class has a `lookupStreamPrintServiceFactories` method for locating stream print services. This method, like the `lookupPrintServices` method in `PrintServiceLookup`, takes a `DocFlavor` that represents the type of the input document—in this case, `DocFlavor.INPUT_STREAM.GIF`. Unlike `lookupPrintServices`, this method also takes a MIME type. This MIME type represents the format of the output. Since this example converts GIF to postscript, the MIME type is “application/postscript”. The `DocFlavor` class has the `getMimeType` method for returning the MIME type from a `DocFlavor`. The `lookupStreamPrintServicesFactories` method returns an array of `StreamPrintServiceFactory` objects, which are factories for `StreamPrintService` instances. This code sample demonstrates obtaining an array of `StreamPrintServiceFactory` objects that can return `StreamPrintService` objects able to convert a GIF image into PostScript:

```
DocFlavor flavor = DocFlavor.INPUT_STREAM.GIF;
String psMimeType = DocFlavor.BYTE_ARRAY.POSTSCRIPT.getMimeType();
StreamPrintServiceFactory[] psfactories =
    StreamPrintServiceFactory.lookupStreamPrintServiceFactories(
        flavor, psMimeType);
```

The `StreamPrintServiceFactory` object has an instance method called `getPrintService` that takes an `OutputStream` parameter and creates a `StreamPrintService` instance that writes to that stream:

```
FileOutputStream fos = new FileOutputStream(filename);
StreamPrintService psService = psfactories[0].getPrintService(fos);
```

The Java 2 SE SDK V1.4 includes one stream print service that can export Postscript from graphics calls, such as through the `Pageable` and `Printable` interfaces. To verify the availability of this service, use the `StreamPrintServiceFactory` class to try to locate it. The *Printing and Streaming 2D Graphics* chapter discusses streaming 2D graphics.

`StreamPrintService` implements `PrintService`, which means you can use a `StreamPrintService` wherever you can use a `PrintService`. The application is responsible for closing the output stream after a job has printed to the stream. Once the stream is closed, the `StreamPrintService` instance can no longer be used.

---

## Obtaining a Print Job

A print job is a submitted print request and includes one or more pieces of print data and a set of processing instructions. The Java Print Service represents a print job with the `DocPrintJob` object. Whether you are sending print data to a stream or to a printer, you create a print job in the same way: by calling `createPrintJob` on the service:

```
DocPrintJob pj = pservices[0].createPrintJob();
```

An application obtains a print job from a service because the service only creates print jobs that are capable of handling data that the particular service can accept.

The `DocPrintJob` interface provides the `print` method, which takes a `PrintRequestAttributeSet` parameter and a `Doc` encapsulating the print data and the doc flavor:

```
pj.print(doc, aset);
```

The next section discusses creating a `Doc`.

---

## Creating a Doc

To create a `Doc` you must provide an implementation of the `Doc` interface. The Java Print Service API provides a convenient implementation of `Doc` called `SimpleDoc`. An application is not required to use the `SimpleDoc` implementation, but to ensure compliance with `Doc`, any `Doc` implementation must observe the same required semantics that `SimpleDoc` implements, which are:

- Every `Doc` implementation must implement all five methods of the `Doc` interface.
- The `Doc` implementation must allow multiple threads to access the `Doc` object.
- Every time a `Doc` method is called, it returns the same object. This means you do not return a new stream. Since there is only one input stream there can only be one consumer of the `Doc`.
- The `Doc` returns a stream for the service if requested
- The `Doc` checks if the data type matches the doc flavor
- The attributes returned from `getAttributes` always override those passed in the `print` method.

Before creating a `Doc`, you need to load your document from the file. The representation class of the `DocFlavor` determines how you load the document from the file. In this case, the representation class is an `InputStream`:

```
FileInputStream fis = new FileInputStream("java2dlogo.gif");
```

Once you have the stream, pass it to `SimpleDoc` with the `DocFlavor` and a `DocAttributeSet`, if you have one. If you don't have a `DocAttributeSet`, pass in `null` instead:

```
Doc doc = new SimpleDoc(fis, flavor, null);
```

See *Example: PrintGIF.java* for an example of a custom `Doc` implementation.

The next section demonstrates how to register for events on your print job or service.

---

## Registering for Events

The Java Print Service API allows services to report two types of events to applications: printer status updates and print job progress updates. The events API, which includes the `javax.print.event` package and methods to register listeners on a service and `DocPrintJob`, follows the familiar listener model used in AWT.

### Print Service Events

Print Service event listeners monitor a service's changes in status and report events as changes in the values of print service attributes. An application can monitor events on a print service by implementing the `javax.print.event.PrintServiceAttributeListener` interface and installing itself as a listener on a `PrintService` as shown in this example:

```
public class PrintPS implements PrintServiceAttributeListener {
    ...
    pservices[0].addPrintServiceAttributeListener(this);
    ...
    public void attributeUpdate(PrintServiceAttributeEvent e){
        // Do something if an attribute is updated
    }
    ...
}
```

The `PrintServiceAttributeListener.attributeUpdate()` method is called when print service attributes change. The service uses the `PrintServiceAttributeListener` interface to decide which events it supports.

An application can discover which print service attributes a service supports by using the same query methods it uses to discover which request attributes a service supports. For example, to discover if the service supports the `QueuedJobCount` attribute an application calls:

```
service.isAttributeCategorySupported(QueuedJobCount.class);
```

The service determines how frequently it reports updates on attributes. If many attributes are supported, the service might batch events, which means an application isn't guaranteed to receive a particular event. The delivered event contains only attributes that have changed in value, which means that static service attributes, such as the printer model, will never be reported in an event.

## Print Job Events

Most printing clients are more interested in monitoring a print job than monitoring a service's status. A client can install two different kinds of listeners on a `DocPrintJob`: `PrintJobAttributeListener` and `PrintJobListener`.

### PrintJobAttributeListener

`PrintJobAttributeListener` is similar to the service attribute listener: the job reports changes in attributes that implement the `PrintJobAttribute` interface. Usually these attributes are also print request attributes and are fixed over the lifetime of a print job. Only a few attributes, such as `JobMediaSheetsCompleted`, are likely to change. Since few clients are interested in this granularity of detail, and even fewer services support this capability, clients will most likely use `PrintJobListener` to monitor a job's progress.

### PrintJobListener

`PrintJobListener` is easier to use than `PrintJobAttributeListener` because it delivers simple messages, such as `printJobCompleted` or `printJobFailed`. The interface has only six methods, each of which reports a significant but simple piece of information as an event. As a convenience, the adapter class, `PrintJobAdapter`, provides default implementations of these six methods.

One message in particular: `printJobNoMoreEvents` is unusual but useful. A client is often interested in knowing if a job has finished or failed. If possible, a service should deliver such "terminal" events, but sometimes the service cannot be sure if the job finished or failed, and a "completed" message is misleading in this case. For example, a job might be spooled to a network print service that has a queue that's not visible. In this case, the "no more events" message is not sufficient to say that the job has succeeded, but the client can at least infer that it is not known to have failed. The following example demonstrates adding a listener that monitors `printJobNoMoreEvents` messages:

```
public class PrintPS extends PrintJobAdapter{
    ...
    pj.addPrintJobListener(this);
    ...
    public void printJobNoMoreEvents(PrintJobEvent e){
```

```
        // Do something here
    }
    ...

```

---

## Submitting the Print Job

Once you have the `Doc` and `DocPrintJob`, you can call the `DocPrintJob` object's `print` method to submit the document to the service. The *Submitting the Print Job to the Printer* section completes the printing example. The *Submitting the Print Job to the Stream* section completes the streaming example.

### Submitting the Print Job to the Printer

This section completes the printing application explained in this chapter. This example prints five copies of a PostScript document, double-sided on A4 paper, and stapled.

```
DocFlavor psFlavor = DocFlavor.INPUT_STREAM.POSTSCRIPT;
PrintRequestAttributeSet aset = new HashPrintRequestAttributeSet();
aset.add(new Copies(2));
aset.add(MediaSizeName.ISO_A4);
aset.add(Sides.TWO_SIDED_LONG_EDGE);
aset.add(Finishings.STAPLE);
PrintService[] pservices = PrintServiceLookup.lookupPrintServices(psFlavor,
                                                                    aset);
if (services.length > 0) {
    DocPrintJob pj = pservices[0].createPrintJob();
    try {
        FileInputStream fis = new FileInputStream("example.ps");
        Doc doc = new SimpleDoc(fis, psFlavor, null);
        pj.print(doc, aset);
    } catch (IOException e) {
        System.err.println(e);
    } catch (PrintException e) {
        System.err.println(e);
    }
}

```

See *Example: PrintPS.java* for the complete application.

## Submitting the Print Job to the Stream

This section completes the streaming example explained in this chapter. This example converts a GIF document to PostScript and embeds the specified printing attributes into the PostScript document.

```
DocFlavor flavor = DocFlavor.INPUT_STREAM.GIF;
String psMimeType = DocFlavor.BYTE_ARRAY.POSTSCRIPT.getMimeType();
PrintRequestAttributeSet aset = new HashPrintRequestAttributeSet();
aset.add(new Copies(2));
aset.add(MediaSizeName.A4);
aset.add(Sides.TWO_SIDED_LONG_EDGE);
aset.add(Finishings.STAPLE);
StreamPrintServiceFactory[] factories =
    StreamPrintServiceFactory.lookupStreamPrintServiceFactories(
        flavor, psMimeType);
if(factories.length==0) {
    System.err.println("No suitable factories");
    System.exit(0);
}
try {
    FileInputStream fis = new FileInputStream("java2dlogo.gif");
    String filename = "newfile.ps";
    FileOutputStream fos = new FileOutputStream(filename);
    StreamPrintService sps= factories[0].getPrintService(fos);
    DocPrintJob pj = sps.createPrintJob();
    Doc doc = new SimpleDoc(fos, psFlavor, aset);
    pj.print(doc, aset);
} catch (IOException e) {
    System.err.println(e);
} catch (PrintException e) {
    System.err.println(e);
}
}
```

See *Example: PrintGIFtoStream.java* for the complete application.

---

# Print Service Providers

Print Service Providers are third parties that write an implementation of the abstract class `PrintServiceLookup` and install it according to the SPI JAR file specification.

This class provides a standardized API for looking up instances of print services. For example an IPP print service provider, a JINI lookup service, or a JNDI-based directory service may each be implementations of this class which can return `PrintServices`.

The API is reasonably simple and generally requires implementation of the `PrintServiceLookup` class, and the `PrintService` interface, and `DocPrintJob`. The `PrintService` should also support the events defined by the Java Print Service API.

How much more code there is depends on the document types that are supported. If a printer wants to be able to print a JPEG image and the printer already has such capability in hardware, its little more than a case of properly spooling the data. However, to print 2D graphics, a class that can turn the graphics calls into a printer-formatted raster is a significant body of work.

## Printing and Streaming 2D Graphics

---

The Java 2D™ printing API is the `java.awt.print` package that is part of the Java™ 2 SE, version 1.2 and later. The Java 2D printing API provides for creating a `PrinterJob`, displaying a printer dialog to the user, and printing paginated graphics using the same `java.awt.Graphics` and `java.awt.Graphics2D` classes that are used to draw to the screen.

Many of the features that are new in the Java Print Service, such as printer discovery and specification of printing attributes, are also very important to users of the Java 2D printing API. To make these features available to users of Java 2D printing, the `java.awt.print` package has been updated for version 1.4 of the Java™ 2 SE to allow access to the Java™ Print Service from the Java 2D printing API.

Developers of Java 2D printing applications have four ways of using the Java Print Service with the Java 2D API:

- Print 2D graphics using `PrinterJob`.
- Stream 2D graphics using `PrinterJob`
- Print 2D graphics using using `DocPrintJob` and a service-formatted `DocFlavor`
- Stream 2D graphics using `DocPrintJob` and a service-formatted `DocFlavor`

The *Using PrinterJob to Print or Stream Graphics* section explains the first two methods. The *Using Service-Formatted Data* section explains the other two methods.

---

## Using PrinterJob to Print or Stream Graphics

The new API in `java.awt.print` consists of these new `PrinterJob` methods:

- Static convenience methods to look up print services that can image 2D graphics, which are returned as an array of `PrintService` or `StreamPrintServiceFactory` objects depending on the method.
- Methods to set and get a `PrintService` on a `PrinterJob`.
- A `pageDialog` method that takes a `PrintRequestAttributeSet` parameter.
- A `printDialog` method that takes a `PrintRequestAttributeSet` parameter.

- A print method that takes a `PrintRequestAttributeSet` parameter.

Because the new `printDialog` and `pageDialog` methods take an attribute set, users can edit the initial attribute settings from the dialogs.

Applications can use `PrinterJob` to print 2D graphics to a printer or to an output stream. The `lookupPrintServices` method returns an array of `PrintService` objects, each of which represents a printer that can print 2D graphics. The `lookupStreamPrintServices` method returns an array of `StreamPrintServiceFactory` objects, each of which can return a `StreamPrintService`. An application uses the `StreamPrintService` to send print data to an output stream. As with printing documents, applications can use a `StreamPrintService` to transcode 2D graphics to other formats. This section discusses using `PrinterJob` to submit 2D graphics to a printer and to an output stream.

## Printing 2D Graphics

The new `pageDialog`, `printDialog`, and `print` methods allow an application to initialize print settings and pass these settings to a dialog so that a user can update the settings before submitting the print request, as demonstrated by this code sample:

```
// Step 1: Set up initial print settings.
PrintRequestAttributeSet aset = new HashPrintRequestAttributeSet();
// Step 2: Obtain a print job.
PrinterJob pj = PrinterJob.getPrinterJob();
// Step 3: Find print services.
PrintService []services = PrinterJob.lookupPrintServices();
if (services.length > 0) {
    System.out.println("selected printer: " + services[0]);
    try {
        pj.setPrintService(service[0]);
        // Step 2: Pass the settings to a page dialog and print dialog.
        pj.pageDialog(aset);
        if (pj.printDialog(aset)) {
            // Step 4: Update the settings made by the user in the dialogs.
            // Step 5: Pass the final settings into the print request.
            pj.print(aset);
        }
    } catch (PrinterException(pe)) {
        System.err.println(pe);
    }
}
```

See *Example: Print2DPrinterJob.java* for the complete application.

Note that Step 4 in this code sample does not seem to correspond to any particular line of code. In fact, the user updates the print settings in the dialogs, and the updated settings are saved in the `PrintRequestAttributeSet`, `aset`.

One problem with using Java 2D and the Java Print Service together is that some attributes, such as *number of copies*, are defined in both APIs. If such an attribute is specified in a `PrintRequestAttributeSet`, it takes precedence over the same attribute specified in the `PrinterJob`. Note that if a user updates the number of copies in a print dialog, the `PrinterJob` is automatically updated to reflect that, which reconfirms the existing behavior.

The `PageFormat` specification also overlaps with the Java Print Service `Media`, `MediaPrintableArea`, and `OrientationRequested` attributes. If an application uses the `Printable` interface and the `print(PrintRequestAttributeSet)` method, the `media`, `orientation`, and `imageable area` attributes contained in the attribute set are added to a new `PageFormat`, which is passed to the `print` method of the `Printable` object. If an application uses the `Pageable` interface, the `PageFormat` does not change.

## Streaming 2D Graphics

An application can also use a `PrinterJob` and a `StreamPrintService` to send print data to an output stream. This example is similar to the example in the previous section, except a `StreamPrintService` is used in place of a `PrintService`:

```
PrinterJob job = PrinterJob.getPrinterJob();
String psMimeType = "application/postscript";
FileOutputStream outstream;
StreamPrintService psPrinter;
StreamPrintServiceFactory []spsFactories =
    PrinterJob.lookupStreamPrintServices(psMimeType);
if (factories.length > 0) {
    try {
        outstream = new File("out.ps");
        psPrinter = factories[0].getPrintService(fos);
        // psPrinter can now be set as the service on a PrinterJob
    } catch (FileNotFoundException e) { }
}
job.setPrintService(service[0]); // if app wants to specify this printer.
PrintRequestAttributeSet aset = new HashPrintRequestAttributeSet();
aset.add(new Copies(2));
job.print(aset);
}
```

---

## Using Service-Formatted Data

You can print or stream 2D graphics encapsulated in a `Pageable` or `Printable` object using a `DocPrintJob` and a service-formatted `DocFlavor`. A `DocFlavor` can represent any kind of data, including Java objects. An implementation of the `Pageable` or `Printable` interface is a Java object. As discussed in the *Specifying Document Types* chapter, the Java Print Service API includes pre-defined `DocFlavor` object constants for print data in the form of a Java object. An application can look up print services or stream print services supporting this type of data, encapsulate the object in a `Doc` implementation and submit it to the service in a `DocPrintJob`. The *Printing the Service-Formatted Data* section demonstrates printing the data. The *Streaming Service-Formatted Print Data* section demonstrates streaming the data. Registering for events on 2D graphics printing applications using `DocPrintJob` is done the same way as for document printing applications using `DocPrintJob`. See *Registering for Events* for more information.

## Printing the Service-Formatted Data

To locate print services that can handle the service-formatted data, pass the appropriate service-formatted `DocFlavor` object constant to the `lookupPrintServices` method:

```
PrintService []services =
    PrintServiceLookup.lookupPrintServices(DocFlavor.SERVICE_FORMATTED.PRINTABLE,
                                           null);
```

The printing application implements the `Printable` interface. To create the `Doc`, use `SimpleDoc`, passing this in for the `printData`, the service-formatted `DocFlavor` constant for the `DocFlavor`, and an optional attribute set:

```
Doc doc = new SimpleDoc(this, DocFlavor.SERVICE_FORMATTED.PRINTABLE, null);
```

Create the `DocPrintJob`, and submit it to the service:

```
DocPrintJob pj = service[0].createPrintJob();
pj.print(doc);
```

See *Example: Print2DGraphics.java* for the complete application.

## Streaming Service-Formatted Print Data

A stream print service can be used to export 2D graphics encapsulated in a Java object to another format. This example exports graphics in a `Printable` to Postscript:

```

DocFlavor flavor = DocFlavor.SERVICE_FORMATTED.PRINTABLE
StreamPrintServiceFactory []factories =
    StreamPrintServiceFactory.lookupStreamPrintServiceFactories(flavor,
        "application/postscript");
if (factories.length == 0) {
    System.err.println("No suitable factories");
    System.exit(0);
}
try{
    FileOutputStream fos = new FileOutputStream("out.ps");
    StreamPrintService sps = factories[0].getPrintService(fos);
}

Doc doc = new SimpleDoc(this, flavor, null);
sps.createPrintJob().print(doc);

```

See *Example: Print2DtoStream.java* for the complete application

## Example: PrintPS.java

---

```
/*
 * Copyright 2001 Sun Microsystems, Inc. All Rights Reserved.
 *
 * This software is the proprietary information of
 * Sun Microsystems, Inc.
 * Use is subject to license terms.
 */
import java.io.*;
import javax.print.*;
import javax.print.attribute.*;
import javax.print.attribute.standard.*;

public class PrintPS {
    public static void main(String args[]) {
        PrintPS ps = new PrintPS();
    }
    public PrintPS() {
        /* Construct the print request specification.
         * The print data is Postscript which will be
         * supplied as a stream. The media size
         * required is A4, and 2 copies are to be printed
         */
        DocFlavor flavor = DocFlavor.INPUT_STREAM.POSTSCRIPT;
        PrintRequestAttributeSet aset
            = new HashPrintRequestAttributeSet();
        aset.add(MediaSizeName.ISO_A4);
        aset.add(new Copies(2));
        aset.add(Sides.TWO_SIDED_LONG_EDGE);
        aset.add(Finishings.STAPLE);
        /* locate a print service that can handle it
         */
        PrintService[] pservices =
            PrintServiceLookup.lookupPrintServices(flavor, aset);
        if (pservices.length > 0) {
            System.out.println("selected printer " +
                pservices[0].getName());
        }
    }
}
```

```

/* create a print job for the chosen service
*/
DocPrintJob pj = pservices[0].createPrintJob();
try {
    /* * Create a Doc object to hold the print data.
    * Since the data is postscript located in a disk file,
    * an input stream needs to be obtained
    * BasicDoc is a useful implementation that will if
    * requested close the stream when printing is completed.
    */
    FileInputStream fis = new FileInputStream("example.ps");
    Doc doc = new SimpleDoc(fis, flavor, null);
    /* print the doc as specified
    */
    pj.print(doc, aset);
} catch (IOException ie) {
    System.err.println(ie);
} catch (PrintException e) {
    System.err.println(e);
}
}
}
}

```

## Example: PrintGIFtoStream.java

---

```
/*
 * Copyright 2001 Sun Microsystems, Inc. All Rights Reserved.
 *
 * This software is the proprietary information of
 * Sun Microsystems, Inc.
 * Use is subject to license terms.
 */

import java.io.*;
import javax.print.*;
import javax.print.attribute.*;
import javax.print.attribute.standard.*;

/*
 * Use the Java(TM) Print Service API to locate a service which can
 * export a GIF-encoded image to a stream as Postscript. This may be
 * spooled to a Postscript printer, or used in a postscript viewer.
 */
public class PrintGIFtoStream {

    public static void main(String args[]) {

        /* Use the pre-defined flavor for a GIF from an InputStream */
        DocFlavor flavor = DocFlavor.INPUT_STREAM.GIF;

        /* Specify the type of the output stream */
        String psMimeType =
            DocFlavor.BYTE_ARRAY.POSTSCRIPT.getMimeType();

        /* Locate factory which can export a GIF image stream as
           Postscript */
        StreamPrintServiceFactory[] factories =
            StreamPrintServiceFactory.lookupStreamPrintServiceFactories(
                flavor, psMimeType);
        if (factories.length == 0) {
            System.err.println("No suitable factories");
            System.exit(0);
        }
    }
}
```

```

}

try {
    /* Load the file */
    FileInputStream fis =
        new FileInputStream("java2dlogo.gif");
    /* Create a file for the exported postscript */
    String filename = "newfile.ps";
    FileOutputStream fos = new FileOutputStream(filename);

    /* Create a Stream printer for Postscript */
    StreamPrintService sps = factories[0].getPrintService(fos);

    /* Create and call a Print Job for the GIF image */
    DocPrintJob pj = sps.createPrintJob();
    PrintRequestAttributeSet aset = new
        HashPrintRequestAttributeSet();
    aset.add(new Copies(2));
    aset.add(MediaSizeName.ISO_A4);
    aset.add(Sides.TWO_SIDED_LONG_EDGE);
    aset.add(Finishings.STAPLE);

    Doc doc = new SimpleDoc(fis, flavor, null);

    pj.print(doc, aset);
    fos.close();

} catch (PrintException pe) {
    System.err.println(pe);
} catch (IOException ie) {
    System.err.println(ie);
}
}
}

```

## Example: Print2DPrinterJob.java

---

```
/*
 * Copyright 2001 Sun Microsystems, Inc. All Rights Reserved.
 *
 * This software is the proprietary information of
 * Sun Microsystems, Inc.
 * Use is subject to license terms.
 */
import java.io.*;
import java.awt.*;
import java.net.*;
import java.awt.image.*;
import java.awt.print.*;
import javax.print.*;
import javax.print.attribute.*;
import javax.print.attribute.standard.*;

public class Print2DPrinterJob implements Printable {

    public Print2DPrinterJob() {

        /* Construct the print request specification.
         * The print data is a Printable object.
         * the request additionally specifies a job name, 2 copies, and
         * landscape orientation of the media.
         */
        PrintRequestAttributeSet aset = new
            HashPrintRequestAttributeSet();
        aset.add(OrientationRequested.LANDSCAPE);
        aset.add(new Copies(2));
        aset.add(new JobName("My job", null));

        /* Create a print job */
        PrinterJob pj = PrinterJob.getPrinterJob();
        pj.setPrintable(this);
        /* locate a print service that can handle the request */
        PrintService[] services =
```

```

        PrinterJob.lookupPrintServices();

    if (services.length > 0) {
        System.out.println("selected printer " +
            services[0].getName());
        try {
            pj.setPrintService(services[0]);
            pj.pageDialog(aset);
            if(pj.printDialog(aset)) {
                pj.print(aset);
            }
        } catch (PrinterException pe) {
            System.err.println(pe);
        }
    }
}

public int print(Graphics g,PageFormat pf,int pageIndex) {

    if (pageIndex == 0) {
        Graphics2D g2d= (Graphics2D)g;
        g2d.translate(pf.getImageableX(), pf.getImageableY());
        g2d.setColor(Color.black);
        g2d.drawString("example string", 250, 250);
        g2d.fillRect(0, 0, 200, 200);
        return Printable.PAGE_EXISTS;
    } else {
        return Printable.NO_SUCH_PAGE;
    }
}

public static void main(String arg[]) {

    Print2DPrinterJob sp = new Print2DPrinterJob();
}
}

```

## Example: Print2DGraphics.java

---

```
/*
 * Copyright 2001 Sun Microsystems, Inc. All Rights Reserved.
 *
 * This software is the proprietary information of
 * Sun Microsystems, Inc.
 * Use is subject to license terms.
 */
import java.io.*;
import java.awt.*;
import java.net.*;
import java.awt.image.*;
import java.awt.print.*;
import javax.print.*;
import javax.print.attribute.*;
import javax.print.attribute.standard.*;

public class Print2DGraphicsDoc implements Printable {

    public Print2DGraphicsDoc() {

        /* Construct the print request specification.
         * The print data is a Printable object.
         * the request additionally specifies a job name, 2 copies,
         * and landscape orientation of the media.
         */
        DocFlavor flavor = DocFlavor.SERVICE_FORMATTED.PRINTABLE;
        PrintRequestAttributeSet aset = new
            HashPrintRequestAttributeSet();
        aset.add(OrientationRequested.LANDSCAPE);
        aset.add(new Copies(2));
        aset.add(new JobName("My job", null));

        /* locate a print service that can handle the request */
        PrintService[] services =
            PrintServiceLookup.lookupPrintServices(flavor, aset);
    }
}
```

```

if (services.length > 0) {
    System.out.println("selected printer " +
        services[0].getName());

    /* create a print job for the chosen service */
    DocPrintJob pj = services[0].createPrintJob();

    try {
        /*
         * Create a Doc object to hold the print data.
         */
        Doc doc = new PrintableDoc(this);

        /* print the doc as specified */
        pj.print(doc, aset);

        /*
         * Do not explicitly call System.exit() when print
         * returns.
         * Printing can be asynchronous so may be executing
         * in a separate thread.
         * If you want to explicitly exit the VM, use a
         * print job
         * listener to be notified when it is safe to do so.
         */

        } catch (PrintException e) {
            System.err.println(e);
        }
    }
}

public int print(Graphics g,PageFormat pf,int pageIndex) {

    if (pageIndex == 0) {
        Graphics2D g2d= (Graphics2D)g;
        g2d.translate(pf.getImageableX(), pf.getImageableY());
        g2d.setColor(Color.black);
        g2d.drawString("example string", 250, 250);
        g2d.fillRect(0, 0, 200, 200);
        return Printable.PAGE_EXISTS;
    } else {
        return Printable.NO_SUCH_PAGE;
    }
}

public static void main(String arg[]) {
    Print2DGraphicsDoc sp = new Print2DGraphicsDoc();
}

class PrintableDoc implements Doc {

```

```

private Printable printable;

public PrintableDoc(Printable printable) {
    this.printable = printable;
}

public DocFlavor getDocFlavor() {
    return DocFlavor.SERVICE_FORMATTED.PRINTABLE;
}

public DocAttributeSet getAttributes() {
    return null;
}

public Object getPrintData() throws IOException {
    return printable;
}

public Reader getReaderForText() throws IOException {
    return null;
}

public InputStream getStreamForBytes() throws IOException {
    return null;
}
}
}

```

## Example: Print2DtoStream.java

---

```
/*
 * Copyright 2001 Sun Microsystems, Inc. All Rights Reserved.
 *
 * This software is the proprietary information of
 * Sun Microsystems, Inc.
 * Use is subject to license terms.
 */

import java.io.*;
import java.awt.*;
import java.awt.print.*;
import javax.print.*;
import javax.print.attribute.*;
import javax.print.attribute.standard.*;

/*
 * Use the Java(TM) Print Service API to locate a service which can
 * export 2D graphics to a stream as Postscript. This may be spooled
 * to a Postscript printer, or used in a postscript viewer.
 */
public class Print2DtoStream implements Printable{

    public Print2DtoStream() {

        /* Use the pre-defined flavor for a Printable from an
         * InputStream */
        DocFlavor flavor = DocFlavor.SERVICE_FORMATTED.PRINTABLE;

        /* Specify the type of the output stream */
        String psMimeType =
            DocFlavor.BYTE_ARRAY.POSTSCRIPT.getMimeType();

        /* Locate factory which can export a GIF image stream as
         * Postscript */
        StreamPrintServiceFactory[] factories =
            StreamPrintServiceFactory.lookupStreamPrintServiceFactories(
```

```

        flavor, psMimeType);
    if (factories.length == 0) {
        System.err.println("No suitable factories");
        System.exit(0);
    }

    try {
        /* Create a file for the exported postscript */
        FileOutputStream fos = new FileOutputStream("out.ps");

        /* Create a Stream printer for Postscript */
        StreamPrintService sps = factories[0].getPrintService(fos);

        /* Create and call a Print Job */
        DocPrintJob pj = sps.createPrintJob();
        PrintRequestAttributeSet
            aset = new HashPrintRequestAttributeSet();

        Doc doc = new SimpleDoc(this, flavor, null);

        pj.print(doc, aset);
        fos.close();

    } catch (PrintException pe) {
        System.err.println(pe);
    } catch (IOException ie) {
        System.err.println(ie);
    }
}

public int print(Graphics g,PageFormat pf,int pageIndex) {

    if (pageIndex == 0) {
        Graphics2D g2d= (Graphics2D)g;
        g2d.translate(pf.getImageableX(), pf.getImageableY());
        g2d.setColor(Color.black);
        g2d.drawString("example string", 250, 250);
        g2d.fillRect(0, 0, 200, 200);
        return Printable.PAGE_EXISTS;
    } else {
        return Printable.NO_SUCH_PAGE;
    }
}

public static void main(String args[]) {
    Print2DtoStream sp = new Print2DtoStream();
}
}

```

## Example: PrintGIF.java

---

```
/*
 * Copyright 2001 Sun Microsystems, Inc. All Rights Reserved.
 *
 * This software is the proprietary information of Sun Microsystems,
 * Inc.
 * Use is subject to license terms.
 */

import java.io.*;
import javax.print.*;
import javax.print.attribute.*;
import javax.print.attribute.standard.*;

/*
 * Use the Java(TM) Print Service API to locate a print service which
 * can print a GIF-encoded image. A GIF image is printed according to
 * a job template specified as a set of printing attributes.
 */
public class PrintGIF {

    public static void main(String args[]) {

        /* Use the pre-defined flavor for a GIF from an InputStream */
        DocFlavor flavor = DocFlavor.INPUT_STREAM.GIF;

        /* Create a set which specifies how the job is to be printed
        */
        PrintRequestAttributeSet aset = new
            HashPrintRequestAttributeSet();
        aset.add(MediaSizeName.NA_LETTER);
        aset.add(new Copies(1));

        /* Locate print services which can print a GIF in the manner
        specified */
        PrintService[] pservices =
            PrintServiceLookup.lookupPrintServices(flavor, aset);
    }
}
```

```

    if (pservices.length > 0) {
        /* Create a Print Job */
        DocPrintJob printJob = pservices[0].createPrintJob();

        /* Create a Doc implementation to pass the print data */
        Doc doc = new InputStreamDoc("java2dlogo.gif", flavor);

        /* Print the doc as specified */
        try {
            printJob.print(doc, aset);
        } catch (PrintException e) {
            System.err.println(e);
        }
    } else {
        System.err.println("No suitable printers");
    }
}

class InputStreamDoc implements Doc {
    private String filename;
    private DocFlavor docFlavor;
    private InputStream stream;

    public InputStreamDoc(String name, DocFlavor flavor) {
        filename = name;
        docFlavor = flavor;
    }

    public DocFlavor getDocFlavor() {
        return docFlavor;
    }

    /* No attributes attached to this Doc - mainly useful for MultiDoc
    */
    public DocAttributeSet getAttributes() {
        return null;
    }

    /* Since the data is to be supplied as an InputStream delegate to
    * getStreamForBytes().
    */
    public Object getPrintData() throws IOException {
        return getStreamForBytes();
    }

    /* Not possible to return a GIF as text */
    public Reader getReaderForText()
        throws UnsupportedEncodingException, IOException {

```

```
        return null;
    }

    /* Return the print data as an InputStream.
    * Always return the same instance.
    */
    public InputStream getStreamForBytes() throws IOException {
        synchronized(this) {
            if (stream == null) {
                stream = new FileInputStream(filename);
            }
            return stream;
        }
    }
}
```

## Java™ Print Service Glossary

---

<b>attribute</b>	a description of: a capability of a <i>print service</i> , a characteristic of a <i>doc</i> , an instruction for processing a <i>doc</i> or an entire <i>print job</i> , or the state of a print job or printer.
<b>attribute category</b>	the name of the <i>attribute</i> without its range of values. The Java Print Service represents the attribute category with a class, such as the <code>OrientationRequested</code> class. An instance of an attribute class, such as an <code>OrientationRequested</code> object, is an <i>attribute value</i> .
<b>attribute set</b>	an <i>attribute</i> collection that usually is restricted to contain only a certain kind of attribute. The Java Print Service defines a superinterface for all attribute sets called <code>AttributeSet</code> and an interface for each kind of attribute set. The kinds of attribute sets are: <i>doc attribute set</i> , <i>print request attribute set</i> , <i>print job attribute set</i> , and <i>print service attribute set</i> .
<b>attribute value</b>	one of the range of values that an <i>attribute</i> supports. The Java Print Service represents each supported attribute with one of the attribute classes, such as the <code>OrientationRequested</code> class. The class itself is the <i>attribute category</i> . An instance of the class, such as an <code>OrientationRequested</code> object, is the attribute value. The term attribute often refers to the attribute value because the value of an attribute implies the attribute itself.
<b>doc</b>	a piece of print data. the Java Print Service uses the <code>Doc</code> interface to represent a piece of print data.
<b>doc attribute</b>	an <i>attribute</i> that specifies a characteristic of an individual document and the <i>print job</i> settings to be applied to an individual document.
<b>doc attribute set</b>	a collection of attributes that is restricted to only contain <i>doc attributes</i> .
<b>doc flavor</b>	a document type. The Java Print Service uses the <code>DocFlavor</code> class to represent a document type. A <code>DocFlavor</code> object encapsulates a <code>String</code> representing a MIME type, which indicates the format of the document, and a <code>String</code> representing a Java class, such as “ <code>java.io.InputStream</code> ”, that indicates how the document is sent to the printer.
<b>hash attribute set</b>	an <i>attribute set</i> that can contain different kinds of <i>attributes</i> . The Java Print Service represents this attribute set with the <code>HashAttributeSet</code> class, which implements <code>AttributeSet</code> .

<b>print job</b>	a submitted <i>print request</i> , which includes one or more pieces of print data and a set of processing instructions. The Java Print Service represents a print job with the <code>DocPrintJob</code> object, which is obtained from a <i>print service</i> .
<b>print job attribute</b>	an <i>attribute</i> that reports the status of a <i>print job</i> . A client usually does not specify these attributes since these attributes describe the print job after the client has submitted it. To specify settings for a whole print job, the client instead uses the <i>print request attributes</i> .
<b>print job attribute set</b>	a collection of <i>attributes</i> that is restricted to only contain <i>print job attributes</i> .
<b>print request</b>	an unsubmitted <i>print job</i> .
<b>print request attribute</b>	an <i>attribute</i> that specifies a setting applied to a whole <i>print job</i> and to all the documents in the print job.
<b>print request attribute set</b>	a collection of <i>attributes</i> that is restricted to only contain <i>print request attributes</i> .
<b>print service</b>	an interface that represents a particular printer and is a factory for <i>print jobs</i> .
<b>print service attribute</b>	an <i>attribute</i> that describes a printer. An example of such an attribute is one that describes the printer's location.
<b>print service attribute set</b>	a collection of <i>attributes</i> that is restricted to only contain <i>print service attributes</i> .
<b>print-service lookup</b>	the action of attempting to locate a set of <i>print services</i> representing printers that can print a given set of print data according to a given set of processing instructions. A client uses the <code>PrintServiceLookup</code> class to perform this action.
<b>stream print service</b>	a <i>print service</i> that prints data to a client-provided output stream.