Some Java API’s

- Java PrintService (JPS) API
- Java Logging API
- Java Preferences API

Printing in Java

- Java printing principally exists under the two top level packages:
  - java.awt.print - Java 2D printing, since JDK 1.2
  - javax.print - aka the Java Print Service (JPS) API, since JDK 1.4
- The Java™ Print Service is an API that allows printing on all Java platforms, including platforms requiring a small footprint, such as a J2ME profile, but still supports the current Java 2 Print API and offers following improvements:
  - Both client and server applications can discover and select printers (programmatic printer discovery)
  - Implementations of standard Internet Printing Protocol (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
- Printing with the Print Service API involves a three-part process of discovery, specification, and printing

Java Print Service packages

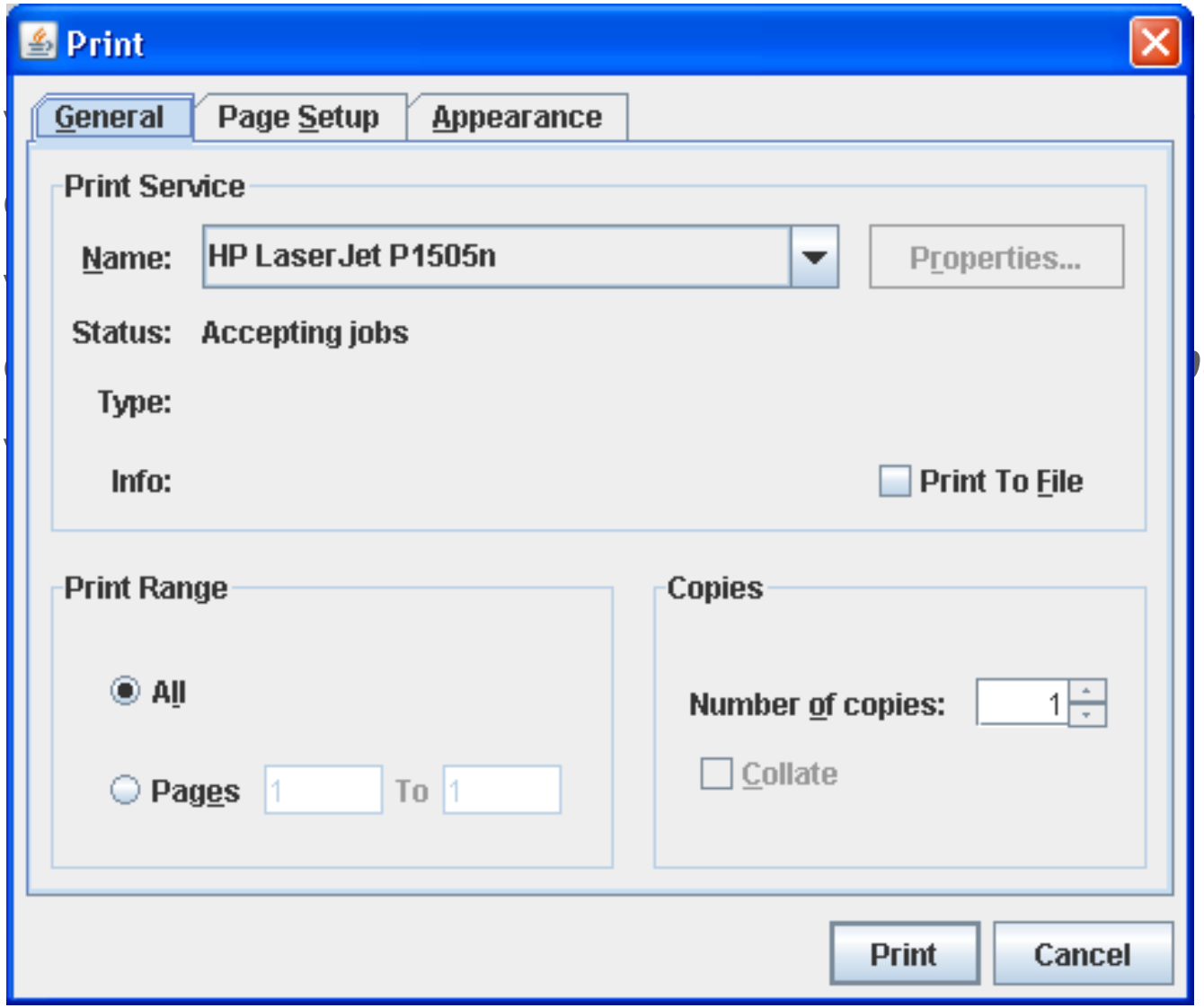
Package	Description
javax.print	The main package of the API. Holds interfaces and classes to: 1. Discover Print Services (Printers) 2. Specify the print data format (DocFlavor) 3. Create print jobs from a print service 4. Send the print data to a printer or a stream
javax.print.attribute	Describes the types of attributes and how they can be collected into sets. Includes classes and interfaces defining the five different kinds of attributes, each of which describes the capabilities of one piece of the printing process
javax.print.attribute.standart	Enumerates all of the standard attributes supported by the API, most of which are implementations of attributes specified in the IPP specification
javax.print.event	Contains classes that allow applications to register for events on print jobs and print services

Locating a printer

- Printer objects are called *print services*, and the identification process is called a *lookup*
- The javax.print.PrintServiceLookup provides three static methods that applications use to locate printers:
  - lookupDefaultPrintService();
    - returns the default print service.
  - lookupPrintServices(DocFlavor flavor, AttributeSet attributes)
    - lookupPrintServices() returns the set of printers that support printing a specific document type (such as GIF) with a specific set of attributes (such as two sided).
  - lookupMultiDocPrintServices(DocFlavor[] flavors, AttributeSet attributes)
    - provides support for printing multiple documents at once.
- The ServiceUI class provides a single method to display the printer selection dialog:
  - printDialog(GraphicsConfiguration gc, int x, int y, PrintService[] services, PrintService defaultService, DocFlavor flavor, PrintRequestAttributeSet attributes)

Example – lookup print service

```
PrintService p
      PrintServ                                                    ;
PrintService d
      PrintServ
PrintService s                                                     00,
      printServ
```



Specifying the output format

- The DocFlavor class is used to identify the MIME (Multipurpose Internet Mail Extensions) type of the object you want to print
- JPS provides seven subclasses of DocFlavor as inner classes of itself to define formats
- These classes can be broken up into three subsets of MIME types:
  - byte-oriented:
    - BYTE_ARRAY
    - INPUT_STREAM
    - URL
  - character-oriented :
    - CHAR_ARRAY
    - READER
    - STRING
  - service-oriented inner class is SERVICE_FORMATTED

Specifying the output format: byte MIMES

- There are 19 byte-oriented MIME type flavors:
  1. AUTOSENSE
  2. GIF
  3. JPEG
  4. PCL
  5. PDF
  6. PNG
  7. POSTSCRIPT
  8. TEXT_HTML_HOST
  9. TEXT_HTML_US_ASCII
  10. TEXT_HTML_UTF_16
  11. TEXT_HTML_UTF_16BE
  12. TEXT_HTML_UTF_16LE
  13. TEXT_HTML_UTF_8
  14. TEXT_PLAIN_HOST
  15. TEXT_PLAIN_US_ASCII
  16. TEXT_PLAIN_UTF_16
  17. TEXT_PLAIN_UTF_16BE
  18. TEXT_PLAIN_UTF_16LE
  19. TEXT_PLAIN_UTF_8

Specifying the output format: char and service MIMES

BDA

- The character-oriented streams provides just two formats:
  1. TEXT_HTML
  2. TEXT_PLAIN
- The service-oriented streams includes three formats:
  - PAGEABLE
  - PRINTABLE
  - RENDERABLE_IMAGE

- Configuration of the flavor is done as shown here:

```
DocFlavor flavor = DocFlavor.INPUT_STREAM.PNG;
```

Specifying the print attributes

BDA

- To specify attributes, you need to work with one of two classes:
  - DocAttributeSet specifies the characteristics for a single document.
  - PrintRequestAttributeSet specifies the characteristics of a single print job.
- The javax.print.attribute package holds about 70 different attributes, each of which is defined as a separate class
- Standart Attributes:
  - OrientationRequested
  - Copies
  - Media
  - Destination (printing to file)
  - SheetCollate
  - Sides (ONE_SIDED, DUPLEX etc.)

```
PrintRequestAttributeSet pras = new
  HashPrintRequestAttributeSet();

pras.add(new Copies(5));
```

Setting the content

BDA

- The Doc interface provides the data to the print job:
  - `public DocFlavor getDocFlavor();`
  - `public Object getPrintData() throws IOException;`
  - `public DocAttributeSet getAttributes();`
  - `public Reader getReaderForText() throws IOException;`
  - `public InputStream getStreamForBytes() throws IOException;`
- The implementation of the interface is the SimpleDoc class
  - With a single constructor, you provide the content, the flavor and the attributes

  public SimpleDoc(Object printData, DocFlavor flavor, DocAttributeSet attributes)

Setting content example

BDA

```
DocFlavor flavor = DocFlavor.INPUT_STREAM.PNG;
String filename = ...;
FileInputStream fis = new FileInputSteam(filename);
DocAttributeSet das = new HashDocAttributeSet();
Doc doc = new SimpleDoc(fis, flavor, das);
```

Printing

- The actual print job is executed through the print() method of the DocPrintJob, retrieved from the PrintService

```
DocPrintJob job = ...;
PrintRequestAttributeSet pras = ...;
Doc doc = ...;
job.print(doc, pras);
```

Drawing graphics

- The DocFlavor.SERVICE_FORMATTED works with three interfaces:
  - java.awt.print.Printable
    - This model only uses one PagePainter for the entire document
    - Pages are rendered in sequence, starting with page zero
    - When the last page prints, PagePainter must return the NO_SUCH_PAGE value
    - The total number of pages to be printed cannot be calculated in advance using this model
  - java.awt.print.Pageable
    - Each page can have its own painter. For example, you could have a painter implemented to print the cover page, another painter to print the table of contents, and a third to print the entire document.
    - You can set a different page format for each page in the book. In aPageable job, you can mix portrait and landscape pages.
    - The print subsystem might ask your application to print pages out of sequence, and some pages may be skipped if necessary. Again, you don't have to worry about this as long as you can supply any page in your document on demand.
    - The Pageable job doesn't need to know how many pages are in the document.
  - java.awt.image.renderable.RenderableImage

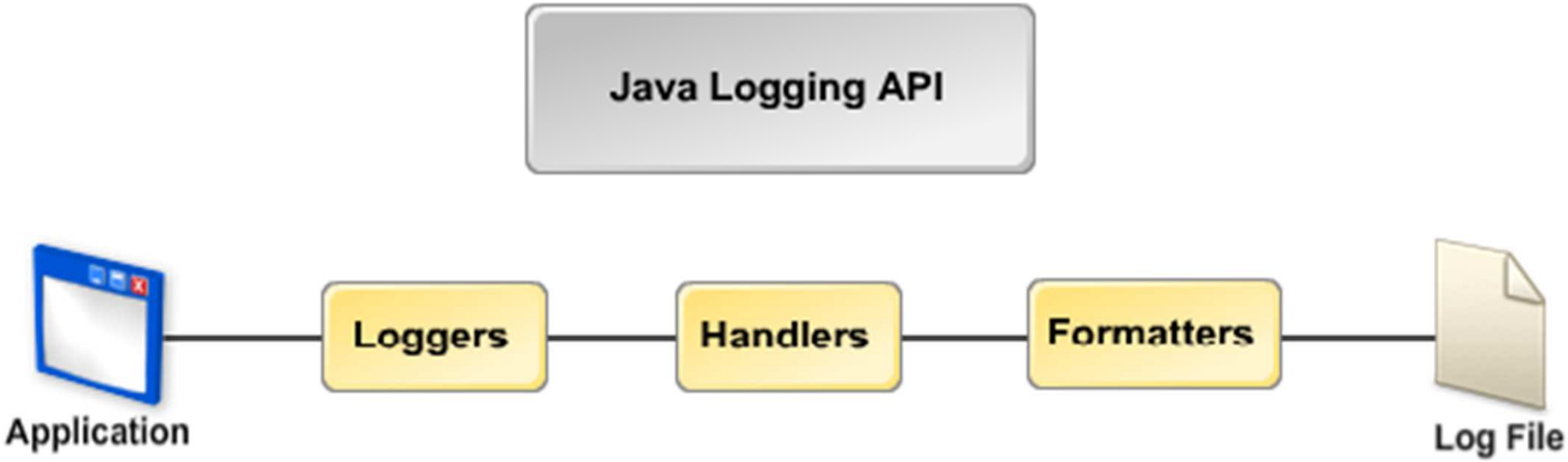
API definition:Printable and Pageable

Name	Type	Description
int print(Graphics graphics, PageFormat pageFormat, int pageIndex)	Method	Requests that the graphics handle using the given page format render the specified page.
NO_SUCH_PAGE	Value	This is a constant. Return this value to indicate that there are no more pages to print.
PAGE_EXISTS	Value	The print() method returns PAGE_EXISTS. It indicates that the page passed as a parameter to print() has been rendered and does exists.

Method name	Description
int getNumberOfPages()	Returns the number of pages in the document.
PageFormat getPageFormat(int pageIndex)	Returns the page's PageFormat as specified by pageIndex.
Printable getPrintable(int pageIndex)	Returns the Printable instance responsible for rendering the page specified bypageIndex.

Java Logging API

- Applications make logging calls on *Logger* objects
- Logger objects allocate *LogRecord* objects which are passed to *Handler* objects for publication
- Both Loggers and Handlers may use logging *Levels* and *Filters*
- Handler can use a *Formatter* to localize and format the message



Logger

- A logger enables an application to log events
- Loggers are normally named entities, using dot-separated names such as “org.bda.jap“
  - The namespace is hierarchical and is managed by the LogManager
- Loggers inherit attributes from their parent:
  - **Logging level**. If a Logger's level is set to be null then the Logger will use an effective Level that will be obtained by walking up the parent tree and using the first non-null Level.
  - **Handlers**. By default a Logger will log any output messages to its parent's handlers, and so on recursively up the tree.
  - **Resource bundle names**. If a logger has a null resource bundle name, then it will inherit any resource bundle name defined for its parent, and so on recursively up the tree.

Logging Levels

- The logging system defines a class called Level to specify the importance of a logging record

Level	Description
OFF	Used to turn off logging
SEVERE	The highest value; intended for extremely important messages (e.g. fatal program errors).
WARNING	Intended for warning messages.
INFO	Informational runtime messages.
CONFIG	Informational messages about configuration settings/setup.
FINE	Used for greater detail, when debugging/diagnosing problems.
FINER	Even greater detail.
FINEST	The lowest value; greatest detail.
ALL	Enables logging of all records

- It is also possible to define custom levels

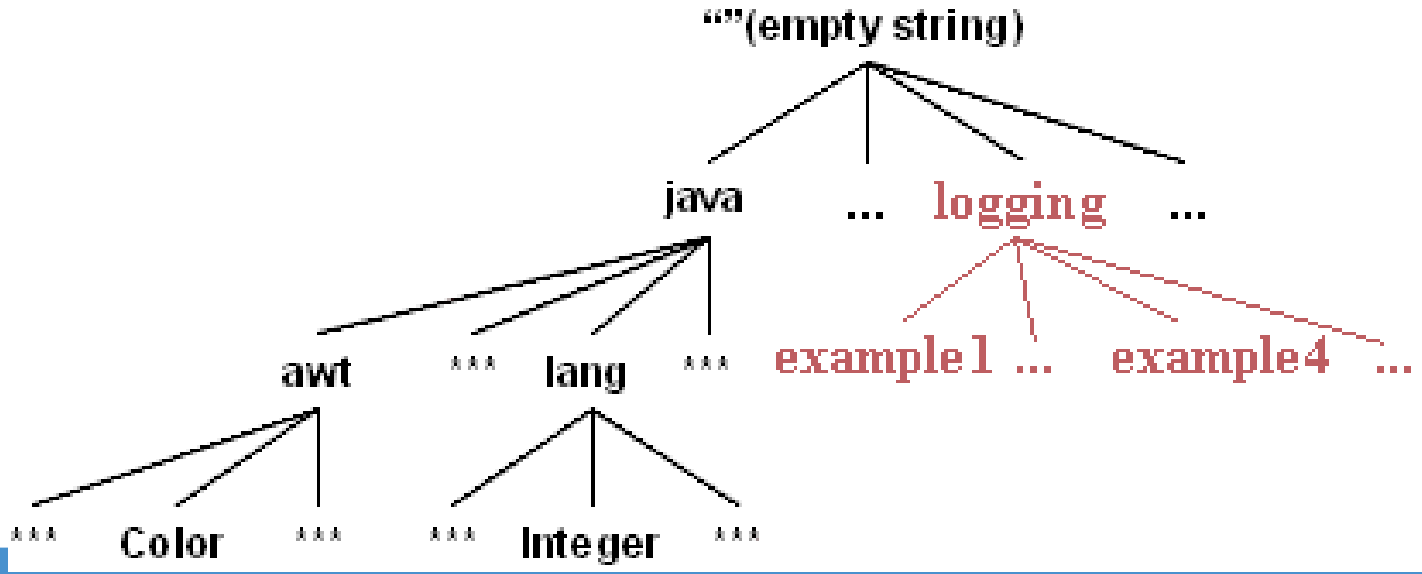
Handlers

- Handlers listen to messages with specified log levels. Provided handlers:
  - *StreamHandler:* A simple handler for writing formatted records to an OutputStream.
  - *ConsoleHandler:* A simple handler for writing formatted records to System.err
  - *FileHandler:* A handler that writes formatted log records either to a single file, or to a set of rotating log files.
  - *SocketHandler:* A handler that writes formatted log records to remote TCP ports.
  - *MemoryHandler:* A handler that buffers log records in memory.

The LogManager

- It provides much of the control over what gets logged and where it gets logged
- Manages a set of logging control properties:
  - JVM param: java.util.logging.config.file
- Logging can be managed at class level, package level or set of packages

```
LogManager.getLogManager().setLevel("logging", Level.FINE);
```



Formatter

- A Formatter provides support for formatting LogRecords. The Formatter takes a LogRecord and converts it to a string:
  - *SimpleFormatter*: Writes brief "human-readable" summaries of log records.
  - *XMLFormatter*: Writes detailed XML-structured information.

Configuration file example

```
# Specify the handlers to create in the root logger
handlers = java.util.logging.ConsoleHandler, java.util.logging.FileHandler

# Set the default logging level for the root logger
.level = ALL

# Set the default logging level for new ConsoleHandler instances
java.util.logging.ConsoleHandler.level = INFO

# Set the default logging level for new FileHandler instances
java.util.logging.FileHandler.level = ALL

# Set the default formatter for new ConsoleHandler instances
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

# Set the default logging level for the logger named com.mycompany
com.mycompany.level = ALL
```

Logging example

```
public class HelloWorld {
  private static Logger theLogger =
   Logger.getLogger(HelloWorld.class.getName());


  public static void main(String[] args) {
      HelloWorld hello = new HelloWorld("Hello world!");
      hello.sayHello();
  }
  private String theMessage;
  public HelloWorld(String message) {
      theMessage = message;
  }
  public String sayHello() {
      theLogger.info("Hello logging!");
      return theMessage;
  }
}
```

Java Preferences API

BDA

- The Preferences API allows preference data to be divided into two categories, "System" preferences and "User" preferences and are structured as a hierarchical tree
  - java.util.prefs
- An instance of java.util.prefs.Preferences represents a particular node
  - Preferences.userRoot();
  - Preferences.systemRoot();
  - Preferences.userNodeForPackage(Object)
  - Preferences.systemNodeForPackage(Object)
- Preferences are stored in an operating system dependent manner, eg in the Windows registry or a Mac preferences file

Writing data

- The Preferences class provides a series of put() methods
- You can store Strings, booleans, doubles, floats, integers, longs, and byte arrays (think serialization):
  - put(String key, String value)
  - putBoolean(String key, boolean value)
  - putByteArray(String key, byte value[])
  - putDouble(String key, double value)
  - putFloat(String key, float value)
  - putInt(String key, int value)
  - putLong(String key, long value)
- The key for a specific preference is limited to a length of Preferences.MAX_KEY_LENGTH (80) characters
- values are limited to Preferences.MAX_VALUE_LENGTH (8192) characters

Reading data

- Getting specific preferences is done through a series of get() methods
  - get(String key, String default)
  - getBoolean(String key, boolean default)
  - getByteArray(String key, byte default[])
  - getDouble(String key, double default)
  - getFloat(String key, float default)
  - getInt(String key, int default)
  - getLong(String key, long default)
- You must provide a default value in the event that the backing store is unavailable, or for when something hasn't been saved yet
- You can find a list of the keys associated with a node with the keys() method

Exporting and Importing Preferences

- The Preferences API also provides export and import facilities for moving the settings on one machine to another:
  - exportNode(OutputStream os)
  - exportSubtree(OutputStream os)
  - importPreferences(InputStream is)
- The information is stored in UTF-8 encoded XML format