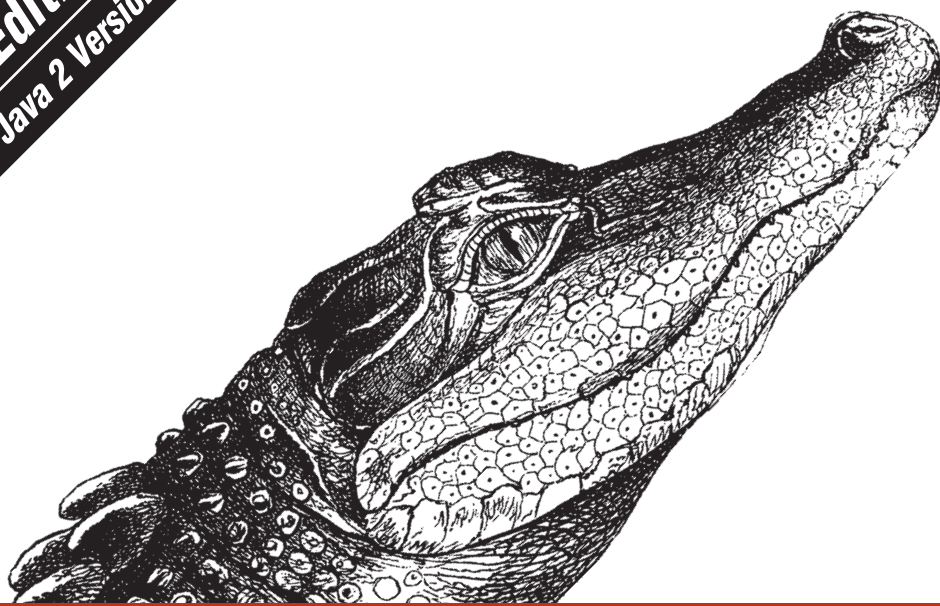


3rd Edition
Covers Java 2 Version 1.4

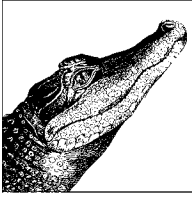


JAVA[™] EXAMPLES IN A NUTSHELL

A Tutorial Companion to Java in a Nutshell

O'REILLY[®]

David Flanagan



13

Printing

The `Graphics` and `Graphics2D` objects represent a “drawing surface”; in Chapter 12, we saw examples of using both the screen and an off-screen buffer as drawing surfaces. Printing in Java is simply a matter of obtaining a `Graphics` object that uses a printer as a drawing surface.* Once you have a `Graphics` object, you can print text and draw graphics to the printer, just as you do onscreen.

The tricky thing about printing in Java is obtaining the `Graphics` object that represents the printer. The API for doing this keeps changing:

- Java 1.1 added the first simple printing API using the `java.awt.PrintJob` class. The major weakness of this API is that it does not support Java 2D graphics.
- Java 1.2 defined a more advanced printing API in the new package `java.awt.print`. This new API supports Java 2D graphics, includes a `Printable` interface, and provides the ability to explicitly set printing attributes such as page margins, orientation, and use of color. It also allows print jobs to be initiated without displaying a **Print** dialog to the user.
- Java 1.3 enhanced the Java 1.1 API by adding the ability to define printing attributes with the `java.awt.JobAttributes` and `java.awt.PageAttributes` classes. Unfortunately, the API is still limited to drawing with the basic `Graphics` object, and it cannot use the Java 2D graphics methods defined in `Graphics2D`.
- Java 1.4 defined a new API in the `javax.print` package and subpackages. This new API is interoperable with the Java 1.2 API but is substantially new. It is the most complex API yet, introducing the ability to select a printer from a list of available printers based on printer name, capabilities, or location; print to files instead of printers; track the progress of print jobs with event listeners; and spool text and image files directly to a printer without actually drawing them.

* Although, as we'll see, Java 1.4 introduces facilities for spooling text and image files directly to a printer without having to draw the text or images to a `Graphics` object.

As you can tell from this list of API revisions, printing is a difficult topic, and it is hard to get it right. This applies to implementations as well as APIs, and you may find that your Java implementation does not support printing as well as you would like. While the most common cases typically work, you may run into difficulties if you push any of these APIs too hard.

This chapter includes examples of each of these printing APIs, including extended examples of the Java 1.1 and Java 1.4 APIs. It starts by developing a simple Swing component that displays an image. Three revisions of this component illustrate each of the three printing APIs.

Printing with the Java 1.1 API

Example 13-1 is a Swing component that displays a fractal image known as a “Julia set,” pictured in Figure 13-1. The image is fascinating, and the mathematics interesting, but the real point of the example is the `print()` method, which demonstrates how to print the Julia set using the Java 1.1 API and the Java 1.3 extensions to that API. The `print()` code is straightforward: it sets some default attribute values, then displays a dialog box to the user, to allow him to modify those attributes or cancel the print request. The dialog box returns a `PrintJob` object. The `print()` method then obtains a `Graphics` object from the `PrintJob`. Next, it draws the Julia set to this `Graphics` object and, finally, calls the `Graphics.dispose()` method to tell the printer that printing is done. These basic steps are repeated in the next two examples as well: when studying the examples of the Java 1.2 and Java 1.4 APIs, look for the code that sets attributes, displays a dialog, and obtains the `Graphics` object.

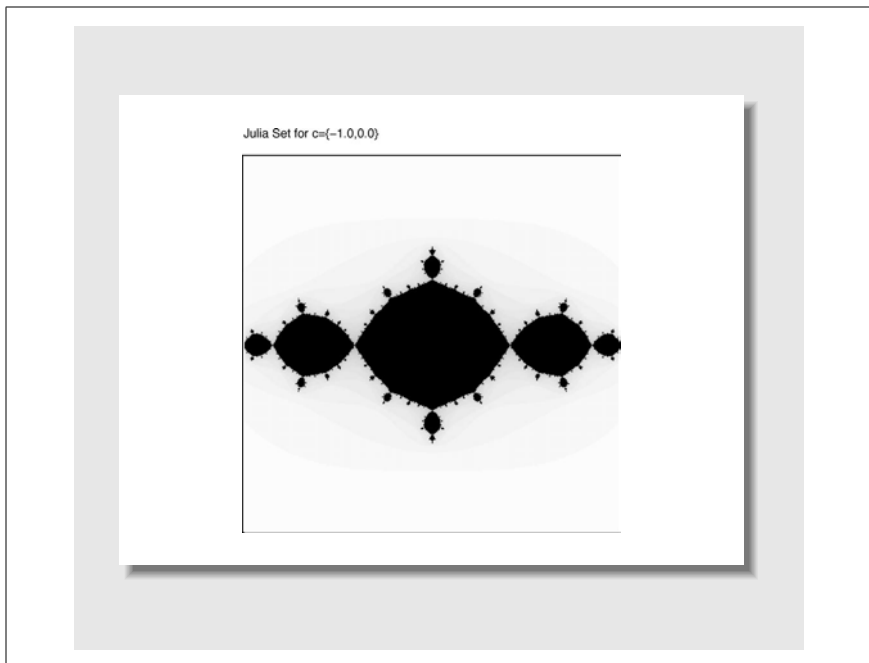


Figure 13-1. A Julia set printed with the Java 1.1 API

Example 13-1 does not include a `main()` method. To display the component, use the `ShowBean` program from Chapter 11, and then use the **Commands** menu to test the `print()` method:

```
java je3.gui.ShowBean je3.print.JuliaSet1
```

Example 13-1. JuliaSet1.java

```
package je3.print;
import javax.swing.*;
import java.awt.*;
import java.awt.image.*;

/**
 * This class is a Swing component that computes and displays a fractal image
 * known as a "Julia set". The print() method demonstrates printing with the
 * Java 1.1 printing API, and is the main point of the example. The code
 * that computes the Julia set uses complex numbers, and you don't need to
 * understand it.
 */
public class JuliaSet1 extends JComponent {
    // These constants are hard-coded for simplicity
    double x1=-1.5, y1=-1.5, x2=1.5, y2=1.5; // Region of complex plane
    int width = 400, height = 400; // Mapped to these pixels
    double cx, cy; // This complex constant defines the set we display
    BufferedImage image; // The image we compute

    // We compute values between 0 and 63 for each point in the complex plane.
    // This array holds the color values for each of those values.
    static int[] colors;
    static { // Static initializer for the colors[] array.
        colors = new int[64];
        for(int i = 0; i < colors.length; i++) {
            colors[63-i] = (i*4 << 16) + (i*4 << 8) + i*4; // grayscale
            // (i*4) ^ ((i * 3)<<6) ^ ((i * 7)<<13); // crazy technicolor
        }
    }

    // No-arg constructor with default values for cx, cy.
    public JuliaSet1() { this(-1, 0); }

    // This constructor specifies the {cx,cy} constant.
    // For simplicity, the other constants remain hardcoded.
    public JuliaSet1(double cx, double cy) {
        this.cx = cx;
        this.cy = cy;
        setPreferredSize(new Dimension(width, height));
        computeImage();
    }

    // This method computes a color value for each pixel of the image
    void computeImage() {
        // Create the image
        image = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);
    }
}
```

Example 13-1. JuliaSet1.java (continued)

```
// Now loop through the pixels
int i,j;
double x, y;
double dx = (x2 - x1)/width;
double dy = (y2 - y1)/height;
for(j = 0, y = y1; j < height; j++, y += dy) {
    for(i = 0, x = x1; i < width; i++, x += dx) {
        // For each pixel, call testPoint() to determine a value.
        // Then map that value to a color and set it in the image.
        // If testPoint() returns 0, the point is part of the Julia set
        // and is displayed in black. If it returns 63, the point is
        // displayed in white. Values in-between are displayed in
        // varying shades of gray.
        image.setRGB(i, j, colors[testPoint(x,y)]);
    }
}

// This is the key method for computing Julia sets. For each point z
// in the complex plane, we repeatedly compute  $z = z*z + c$  using complex
// arithmetic. We stop iterating when the magnitude of z exceeds 2 or
// after 64 iterations. We return the number of iterations-1.
public int testPoint(double zx, double zy) {
    for(int i = 0; i < colors.length; i++) {
        // Compute  $z = z*z + c$ ;
        double newx = zx*zx - zy*zy + cx;
        double newy = 2*zx*zy + cy;
        zx = newx;
        zy = newy;
        // Check magnitude of z and return iteration number
        if (zx*zx + zy*zy > 4) return i;
    }
    return colors.length-1;
}

// This method overrides JComponent to display the Julia set.
// Just scale the image to fit and draw it.
public void paintComponent(Graphics g) {
    g.drawImage(image,0,0,getWidth(), getHeight(),this);
}

// This method demonstrates the Java 1.1 java.awt.PrintJob printing API.
// It also demonstrates the JobAttributes and PageAttributes classes
// added in Java 1.3. Display the Julia set with ShowBean and use
// the Command menu to invoke this print command.
public void print() {
    // Create some attributes objects. This is Java 1.3 stuff.
    // In Java 1.1, we'd use a java.util.Preferences object instead.
    JobAttributes jattrs = new JobAttributes();
    PageAttributes pattrs = new PageAttributes();
}
```

Example 13-1. JuliaSet1.java (continued)

```
// Set some example attributes: monochrome, landscape mode
pattrrs.setColor(PageAttributes.ColorType.MONOCHROME);
pattrrs.setOrientationRequested(
    PageAttributes.OrientationRequestedType.LANDSCAPE);
// Print to file by default
jattrrs.setDestination(JobAttributes.DestinationType.FILE);
jattrrs.setFileName("juliaset.ps");

// Look up the Frame that holds this component
Component frame = this;
while(!(frame instanceof Frame)) frame = frame.getParent();

// Get a PrintJob object to print the Julia set with.
// The getPrintJob() method displays a print dialog and allows the user
// to override and modify the default JobAttributes and PageAttributes
Toolkit toolkit = this.getToolkit();
PrintJob job = toolkit.getPrintJob((Frame)frame, "JuliaSet1",
    jattrrs, pattrrs);

// We get a null PrintJob if the user clicked cancel
if (job == null) return;

// Get a Graphics object from the PrintJob.
// We print simply by drawing to this Graphics object.
Graphics g = job.getGraphics();

// Center the image on the page
Dimension pagesize = job.getPageDimension(); // how big is page?
Dimension panesize = this.getSize(); // how big is image?
g.translate((pagesize.width-panesize.width)/2, // center it
    (pagesize.height-panesize.height)/2);

// Draw a box around the Julia set and label it
g.drawRect(-1, -1, panesize.width+2, panesize.height+2);
g.drawString("Julia Set for c={" + cx + "," + cy + "}",
    0, -15);

// Set a clipping region
g.setClip(0, 0, panesize.width, panesize.height);

// Now print the component by calling its paint method
this.paint(g);

// Finally tell the printer we're done with the page.
// No output will be generated if we don't call dispose() here.
g.dispose();
}
}
```

Printing with the Java 1.2 API

Example 13-2 is a listing of `JuliaSet2`. This subclass of Example 13-1 displays a different Julia set (pictured in Figure 13-2) and overrides the `print()` method, to demonstrate printing out the set using the Java 1.2 API. Like its superclass, `JuliaSet2` should be tested using the `ShowBean` program.

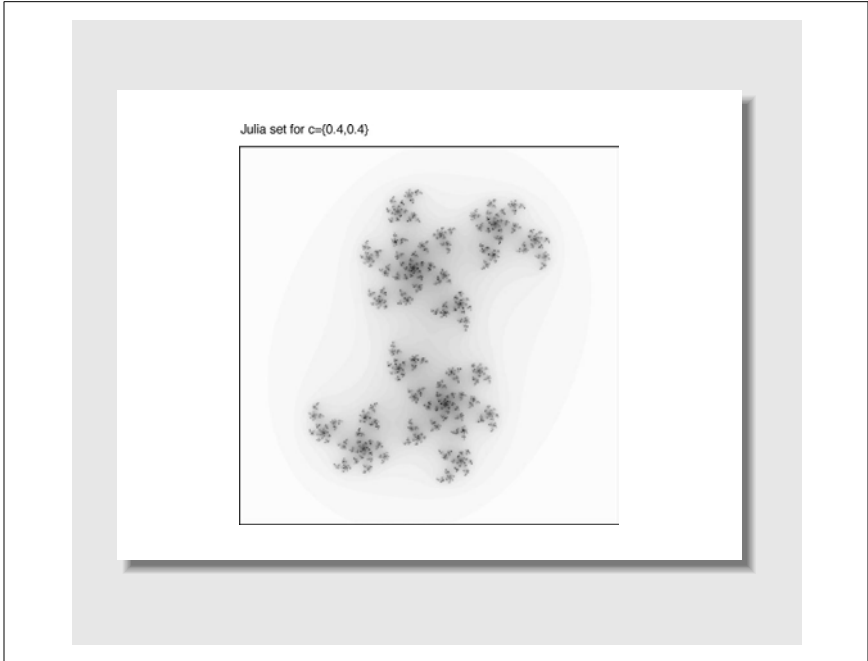


Figure 13-2. A Julia set printed with the Java 1.2 API

The key to this example is the inner class `PrintableComponent`, which implements the `java.awt.print.Printable` interface to print a component and a title for that component. Pay attention to the return value of the `Printable.print()` method: the `Printable` interface can represent multipage documents, and uses its return value to tell the printing system when the last page of the document has been reached.

The `java.awt.print` package also defines an interface named `Pageable`, which is useful for defining multipage documents for which the number of pages is known in advance. It allows different pages of the document to be printed using different `PageFormat` objects, which allows for different margins, orientations, and so forth. `Pageable` documents are not demonstrated in this chapter.

Example 13-2. `JuliaSet2.java`

```
package je3.print;
import javax.swing.*;
import java.awt.*;
import java.awt.print.*; // This package is the Java 1.2 printing API
```

Example 13-2. *JuliaSet2.java* (continued)

```
/**
 * This class extends JuliaSet1 and overrides the print() method to demonstrate
 * the Java 1.2 printing API.
 */
public class JuliaSet2 extends JuliaSet1 {
    public JuliaSet2() { this(.4, .4); } // Display a different set by default
    public JuliaSet2(double cx, double cy) { super(cx,cy); }

    // This method demonstrates the Java 1.2 printing API.
    // Test it using the ShowBean program.
    public void print() {
        // Java 1.1 used java.awt.PrintJob.
        // In Java 1.2 we use java.awt.print.PrinterJob
        PrinterJob job = PrinterJob.getPrinterJob();

        // Alter the default page settings to request landscape mode
        PageFormat page = job.defaultPage();
        page.setOrientation(PageFormat.LANDSCAPE); // landscape by default

        // Tell the PrinterJob what Printable object we want to print.
        // PrintableComponent is defined as an inner class below
        String title = "Julia set for c={" + cx + "," + cy + "}";
        Printable printable = new PrintableComponent(this, title);
        job.setPrintable(printable, page);

        // Call the printDialog() method to give the user a chance to alter
        // the printing attributes or to cancel the printing request.
        if (job.printDialog()) {
            // If we get here, then the user did not cancel the print job
            // So start printing, displaying a dialog for errors.
            try { job.print(); }
            catch(PrinterException e) {
                JOptionPane.showMessageDialog(this, e.toString(),
                    "PrinterException",
                    JOptionPane.ERROR_MESSAGE);
            }
        }
    }
}

// This inner class implements the Printable interface for an AWT component
public static class PrintableComponent implements Printable {
    Component c;
    String title;
    public PrintableComponent(Component c, String title) {
        this.c = c;
        this.title = title;
    }

    // This method should print the specified page number to the specified
    // Graphics object, abiding by the specified page format.
    // The printing system will call this method repeatedly to print all
    // pages of the print job. If pagenum is greater than the last page,
```

Example 13-2. JuliaSet2.java (continued)

```
// it should return NO_SUCH_PAGE to indicate that it is done. The
// printing system may call this method multiple times per page.
public int print(Graphics g, PageFormat format, int pageNum) {
    // This implementation is always a single page
    if (pageNum > 0) return Printable.NO_SUCH_PAGE;

    // The Java 1.2 printing API passes us a Graphics object, but we
    // can always cast it to a Graphics2D object
    Graphics2D g2 = (Graphics2D) g;

    // Translate to accomodate the requested top and left margins.
    g2.translate(format.getImageableX(), format.getImageableY());

    // Figure out how big the drawing is, and how big the page
    // (excluding margins) is
    Dimension size = c.getSize(); // component size
    double pageWidth = format.getImageableWidth(); // Page width
    double pageHeight = format.getImageableHeight(); // Page height

    // If the component is too wide or tall for the page, scale it down
    if (size.width > pageWidth) {
        double factor = pageWidth/size.width; // How much to scale
        g2.scale(factor, factor); // Adjust coordinate system
        pageWidth /= factor; // Adjust page size up
        pageHeight /= factor;
    }
    if (size.height > pageHeight) { // Do the same thing for height
        double factor = pageHeight/size.height;
        g2.scale(factor, factor);
        pageWidth /= factor;
        pageHeight /= factor;
    }

    // Now we know the component will fit on the page. Center it by
    // translating as necessary.
    g2.translate((pageWidth-size.width)/2,(pageHeight-size.height)/2);

    // Draw a line around the outside of the drawing area and label it
    g2.drawRect(-1, -1, size.width+2, size.height+2);
    g2.drawString(title, 0, -15);

    // Set a clipping region so the component can't draw outside of
    // its own bounds.
    g2.setClip(0, 0, size.width, size.height);

    // Finally, print the component by calling its paint() method.
    // This prints the background, border, and children as well.
    // For Swing components, if you don't want the background, border,
    // and children, then call printComponent() instead.
    c.paint(g);
}
```

Example 13-2. *JuliaSet2.java* (continued)

```
        // Tell the PrinterJob that the page number was valid
        return Printable.PAGE_EXISTS;
    }
}
```

Printing with the Java 1.4 API

Example 13-3 is a further subclass of our Julia set component, which once again overrides the `print()` method and this time uses the Java 1.4 Printing API to produce the printed output shown in Figure 13-3. In addition, it defines a `save()` method that uses the Java 1.4 Printing API to print the image of the Julia set directly to a PostScript file. The actual printing of the image is done by the `PrintableComponent` class inherited from the previous example: what is different in this example is the way the `Printable` object is sent to a printer.

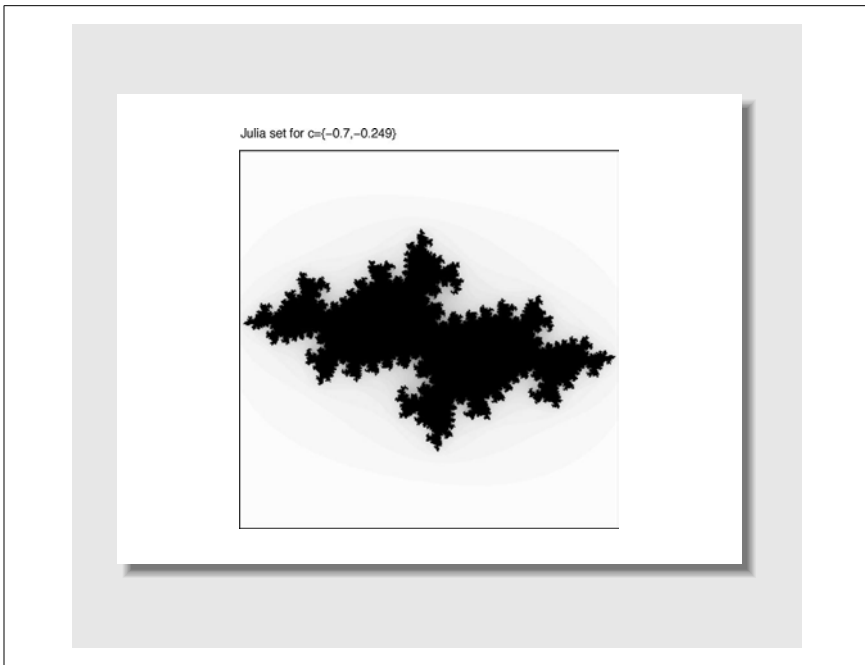


Figure 13-3. A Julia set printed with the Java 1.4 API

The `print()` and `save()` methods start by obtaining an appropriate `PrintService` object that represents a printer or print engine that can print to a file. Both methods then call the `printToService()` method, which uses the inherited `PrintableComponent` class to print the Julia set to the `PrintService`. One interesting feature of this method is its use of a `PrintJobListener` to monitor the progress of the print job and display its current status in a dialog box.

Example 13-3. JuliaSet3.java

```
package je3.print;
import javax.swing.*;
import java.awt.print.*; // Java 1.4 can use the 1.2 Printable API
import javax.print.*; // Java 1.4 API
import javax.print.event.*;
import javax.print.attribute.*;
import javax.print.attribute.standard.*;
import java.io.*;

/**
 * This class extends JuliaSet2, and its print() and save() methods demonstrate
 * the Java 1.4 printing API.
 */
public class JuliaSet3 extends JuliaSet2 {
    public JuliaSet3() { super(-.7, -.25); }

    // This method overrides JuliaSet2.print() and demonstrates the javax.print
    // printing API.
    public void print() {
        // Get a list of all printers that can handle Printable objects.
        DocFlavor flavor = DocFlavor.SERVICE_FORMATTED.PRINTABLE;
        PrintService[] services =
            PrintServiceLookup.lookupPrintServices(flavor, null);

        // Set some printing attributes
        PrintRequestAttributeSet printAttributes =
            new HashPrintRequestAttributeSet();
        printAttributes.add(OrientationRequested.LANDSCAPE); // landscape mode
        printAttributes.add(Chromaticity.MONOCHROME); // print in mono

        // Display a dialog that allows the user to select one of the
        // available printers and to edit the default attributes
        PrintService service = ServiceUI.printDialog(null, 100, 100,
            services, null, null,
            printAttributes);

        // If the user canceled, don't do anything
        if (service == null) return;

        // Now call a method defined below to finish the printing
        printToService(service, printAttributes);
    }

    // This method is like print() above but prints to a PostScript file
    // instead of printing to a printer.
    public void save() throws IOException {
        // Find a factory object for printing Printable objects to PostScript.
        DocFlavor flavor = DocFlavor.SERVICE_FORMATTED.PRINTABLE;
        String format = "application/postscript";
    }
}
```

Example 13-3. *JuliaSet3.java* (continued)

```
StreamPrintServiceFactory factory = StreamPrintServiceFactory.  
    lookupStreamPrintServiceFactories(flavor, format)[0];  
  
// Ask the user to select a file and open the selected file  
JFileChooser chooser = new JFileChooser();  
if (chooser.showSaveDialog(this)!=JFileChooser.APPROVE_OPTION) return;  
File f = chooser.getSelectedFile();  
FileOutputStream out = new FileOutputStream(f);  
  
// Obtain a PrintService that prints to that file  
StreamPrintService service = factory.getPrintService(out);  
  
// Do the printing with the method below  
printToService(service, null);  
  
// And close the output file.  
out.close();  
}  
  
// Print the Julia set to the sepecified PrintService using the specified  
// attributes.  
public void printToService(PrintService service,  
    PrintRequestAttributeSet printAttributes)  
{  
    // Wrap ourselves in the PrintableComponent class defined by JuliaSet2.  
    String title = "Julia set for c={" + cx + ", " + cy + "}";  
    Printable printable = new PrintableComponent(this, title);  
  
    // Now create a Doc that encapsulates the Printable object and its type  
    DocFlavor flavor = DocFlavor.SERVICE_FORMATTED.PRINTABLE;  
    Doc doc = new SimpleDoc(printable, flavor, null);  
  
    // Java 1.1 uses PrintJob.  
    // Java 1.2 uses PrinterJob.  
    // Java 1.4 uses DocPrintJob. Create one from the service  
    DocPrintJob job = service.createPrintJob();  
  
    // Set up a dialog box to monitor printing status  
    final JOptionPane pane = new JOptionPane("Printing...",  
        JOptionPane.PLAIN_MESSAGE);  
    JDialog dialog = pane.createDialog(this, "Print Status");  
    // This listener object updates the dialog as the status changes  
    job.addPrintJobListener(new PrintJobAdapter() {  
        public void printJobCompleted(PrintJobEvent e) {  
            pane.setMessage("Printing complete.");  
        }  
        public void printDataTransferCompleted(PrintJobEvent e) {  
            pane.setMessage("Document transfered to printer.");  
        }  
    })  
}
```

Example 13-3. *JuliaSet3.java* (continued)

```
        public void printJobRequiresAttention(PrintJobEvent e) {
            pane.setMessage("Check printer: out of paper?");
        }
        public void printJobFailed(PrintJobEvent e) {
            pane.setMessage("Print job failed");
        }
    });

    // Show the dialog, non-modal.
    dialog.setModal(false);
    dialog.show();

    // Now print the Doc to the DocPrintJob
    try {
        job.print(doc, printAttributes);
    }
    catch(PrintException e) {
        // Display any errors to the dialog box
        pane.setMessage(e.toString());
    }
}
```

Printing Multipage Text Documents

The printing examples we've seen so far print GUI components on a single page. Printing multipage documents is more interesting, but also trickier, because we have to decide where to place the page breaks. Example 13-4 shows how this can be done. This `HardcopyWriter` class is a custom `java.io.Writer` stream that uses the Java 1.1 Printing API to print the characters sent through it, inserting line breaks and page breaks as necessary.

The `HardcopyWriter` class includes two demonstration programs as inner classes. The first, `PrintFile`, reads a specified text file and prints it by sending its contents to a `HardcopyWriter` stream. The second, `Demo`, prints a demonstration page that shows off the font and tabbing capabilities of the class, as shown in Figure 13-4.

Example 13-4 is long but worth studying. In addition to demonstrating the Java 1.1 Printing API again, it shows an approach to paginating a text document. It is also a useful example of a custom `Writer` stream.



Figure 13-4. Demonstration page printed by HardcopyWriter

Example 13-4. HardcopyWriter.java

```

package je3.print;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.text.*;
import java.util.*;

/**
 * A character output stream that sends output to a printer.
 */
public class HardcopyWriter extends Writer {
    // These are the instance variables for the class
    protected PrintJob job;           // The PrintJob object in use
    protected Graphics page;          // Graphics object for current page
    protected String jobname;         // The name of the print job
    protected int fontsize;           // Point size of the font
    protected String time;             // Current time (appears in header)
    protected Dimension pagesize;     // Size of the page (in dots)
    protected int pagedpi;             // Page resolution in dots per inch

```

Example 13-4. HardcopyWriter.java (continued)

```
protected Font font, headerfont;           // Body font and header font
protected FontMetrics metrics;             // Metrics for the body font
protected FontMetrics headermetrics;      // Metrics for the header font
protected int x0, y0;                     // Upper-left corner inside margin
protected int width, height;              // Size (in dots) inside margins
protected int headery;                    // Baseline of the page header
protected int charwidth;                  // The width of each character
protected int lineheight;                 // The height of each line
protected int lineascent;                 // Offset of font baseline
protected int chars_per_line;            // Number of characters per line
protected int lines_per_page;            // Number of lines per page
protected int charnum = 0, linenum = 0;   // Current column and line position
protected int pagenum = 0;               // Current page number

// A field to save state between invocations of the write() method
private boolean last_char_was_return = false;

// A static variable that holds user preferences between print jobs
protected static Properties printprops = new Properties();

/**
 * The constructor for this class has a bunch of arguments:
 * The frame argument is required for all printing in Java.
 * The jobname appears left justified at the top of each printed page.
 * The font size is specified in points, as on-screen font sizes are.
 * The margins are specified in inches (or fractions of inches).
 */
public HardcopyWriter(Frame frame, String jobname, int fontsize,
                      double leftmargin, double rightmargin,
                      double topmargin, double bottommargin)
    throws HardcopyWriter.PrintCanceledException
{
    // Get the PrintJob object with which we'll do all the printing.
    // The call is synchronized on the static printprops object, which
    // means that only one print dialog can be popped up at a time.
    // If the user clicks Cancel in the print dialog, throw an exception.
    Toolkit toolkit = frame.getToolkit(); // get Toolkit from Frame
    synchronized(printprops) {
        job = toolkit.getPrintJob(frame, jobname, printprops);
    }
    if (job == null)
        throw new PrintCanceledException("User cancelled print request");

    pagesize = job.getPageDimension(); // query the page size
    pagedpi = job.getPageResolution(); // query the page resolution

    // Bug Workaround:
    // On Windows, getPageDimension() and getPageResolution don't work, so
    // we've got to fake them.
    if (System.getProperty("os.name").regionMatches(true,0,"windows",0,7)){
        // Use screen dpi, which is what the PrintJob tries to emulate
        pagedpi = toolkit.getScreenResolution();
    }
}
```

Example 13-4. *HardcopyWriter.java* (continued)

```
// Assume a 8.5" x 11" page size. A4 paper users must change this.
pagesize = new Dimension((int)(8.5 * pagedpi), 11*pagedpi);
// We also have to adjust the fontsize. It is specified in points,
// (1 point = 1/72 of an inch) but Windows measures it in pixels.
fontsize = fontsize * pagedpi / 72;
}

// Compute coordinates of the upper-left corner of the page.
// I.e. the coordinates of (leftmargin, topmargin). Also compute
// the width and height inside of the margins.
x0 = (int)(leftmargin * pagedpi);
y0 = (int)(topmargin * pagedpi);
width = pagesize.width - (int)((leftmargin + rightmargin) * pagedpi);
height = pagesize.height - (int)((topmargin + bottommargin) * pagedpi);

// Get body font and font size
font = new Font("Monospaced", Font.PLAIN, fontsize);
metrics = frame.getFontMetrics(font);
lineheight = metrics.getHeight();
lineascent = metrics.getAscent();
charwidth = metrics.charWidth('0'); // Assumes a monospaced font!

// Now compute the number of columns and lines
// that will fit inside the margins
chars_per_line = width / charwidth;
lines_per_page = height / lineheight;

// Get header font information
// And compute baseline of page header: 1/8" above the top margin
headerfont = new Font("SansSerif", Font.ITALIC, fontsize);
headermetrics = frame.getFontMetrics(headerfont);
headery = y0 - (int)(0.125 * pagedpi) -
    headermetrics.getHeight() + headermetrics.getAscent();

// Compute the date/time string to display in the page header
DateFormat df = DateFormat.getDateInstance(DateFormat.LONG,
                                           DateFormat.SHORT);
df.setTimeZone(TimeZone.getDefault());
time = df.format(new Date());

this.jobname = jobname;           // save name
this.fontsize = fontsize;        // save font size
}

/**
 * This is the write() method of the stream. All Writer subclasses
 * implement this. All other versions of write() are variants of this one
 */
public void write(char[] buffer, int index, int len) {
    synchronized(this.lock) { // For thread safety
        // Loop through all the characters passed to us
        for(int i = index; i < index + len; i++) {
```

Example 13-4. *HardcopyWriter.java* (continued)

```
// If we haven't begun a page (or a new page), do that now.
if (page == null) newpage();

// If the character is a line terminator, then begin new line,
// unless it is a \n immediately after a \r.
if (buffer[i] == '\n') {
    if (!last_char_was_return) newline();
    continue;
}
if (buffer[i] == '\r') {
    newline();
    last_char_was_return = true;
    continue;
}
else last_char_was_return = false;

// If it's some other non-printing character, ignore it.
if (Character.isWhitespace(buffer[i]) &&
    !Character.isSpaceChar(buffer[i]) && (buffer[i] != '\t'))
    continue;

// If no more characters will fit on the line, start new line.
if (charnum >= chars_per_line) {
    newline();
    // Also start a new page, if necessary
    if (page == null) newpage();
}

// Now print the character:
// If it is a space, skip one space, without output.
// If it is a tab, skip the necessary number of spaces.
// Otherwise, print the character.
// It is inefficient to draw only one character at a time, but
// because our FontMetrics don't match up exactly to what the
// printer uses, we need to position each character individually
if (Character.isSpaceChar(buffer[i])) charnum++;
else if (buffer[i] == '\t') charnum += 8 - (charnum % 8);
else {
    page.drawChars(buffer, i, 1,
                  x0 + charnum * charwidth,
                  y0 + (linenum*lineheight) + linescent);
    charnum++;
}
}
}

/**
 * This is the flush() method that all Writer subclasses must implement.
 * There is no way to flush a PrintJob without prematurely printing the
 * page, so we don't do anything.
 */
public void flush() { /* do nothing */ }
```

Example 13-4. *HardcopyWriter.java* (continued)

```
/**
 * This is the close() method that all Writer subclasses must implement.
 * Print the pending page (if any) and terminate the PrintJob.
 */
public void close() {
    synchronized(this.lock) {
        if (page != null) page.dispose(); // Send page to the printer
        job.end(); // Terminate the job
    }
}

/**
 * Set the font style. The argument should be one of the font style
 * constants defined by the java.awt.Font class. All subsequent output
 * will be in that style. This method relies on all styles of the
 * Monospaced font having the same metrics.
 */
public void setFontStyle(int style) {
    synchronized (this.lock) {
        // Try to set a new font, but restore current one if it fails
        Font current = font;
        try { font = new Font("Monospaced", style, fontsize); }
        catch (Exception e) { font = current; }
        // If a page is pending, set the new font. Otherwise newpage() will
        if (page != null) page.setFont(font);
    }
}

/** End the current page. Subsequent output will be on a new page. */
public void pageBreak() { synchronized(this.lock) { newpage(); } }

/** Return the number of columns of characters that fit on the page */
public int getCharactersPerLine() { return this.chars_per_line; }

/** Return the number of lines that fit on a page */
public int getLinesPerPage() { return this.lines_per_page; }

/** This internal method begins a new line */
protected void newline() {
    charnum = 0; // Reset character number to 0
    linenum++; // Increment line number
    if (linenum >= lines_per_page) { // If we've reached the end of page
        page.dispose(); // send page to printer
        page = null; // but don't start a new page yet.
    }
}

/** This internal method begins a new page and prints the header. */
protected void newpage() {
    page = job.getGraphics(); // Begin the new page
    linenum = 0; charnum = 0; // Reset line and char number
    pagenum++; // Increment page number
}
```

Example 13-4. HardcopyWriter.java (continued)

```
page.setFont(headerfont);           // Set the header font.
page.drawString(jobname, x0, headery); // Print job name left justified

String s = "- " + pagenum + " -";    // Print the page # centered.
int w = headermetrics.stringWidth(s);
page.drawString(s, x0 + (this.width - w)/2, headery);
w = headermetrics.stringWidth(time); // Print date right justified
page.drawString(time, x0 + width - w, headery);

// Draw a line beneath the header
int y = headery + headermetrics.getDescent() + 1;
page.drawLine(x0, y, x0+width, y);

// Set the basic monospaced font for the rest of the page.
page.setFont(font);
}

/**
 * This is the exception class that the HardcopyWriter constructor
 * throws when the user clicks "Cancel" in the print dialog box.
 */
public static class PrintCanceledException extends Exception {
    public PrintCanceledException(String msg) { super(msg); }
}

/**
 * A program that prints the specified file using HardcopyWriter
 */
public static class PrintFile {
    public static void main(String[] args) {
        try {
            if (args.length != 1)
                throw new IllegalArgumentException("Wrong # of arguments");
            FileReader in = new FileReader(args[0]);
            HardcopyWriter out = null;
            Frame f = new Frame("PrintFile: " + args[0]);
            f.setSize(200, 50);
            f.show();
            try {
                out = new HardcopyWriter(f, args[0], 10, .5, .5, .5, .5);
            }
            catch (HardcopyWriter.PrintCanceledException e) {
                System.exit(0);
            }
            f.setVisible(false);
            char[] buffer = new char[4096];
            int numchars;
            while((numchars = in.read(buffer)) != -1)
                out.write(buffer, 0, numchars);
            in.close();
            out.close();
        }
    }
}
```

Example 13-4. *HardcopyWriter.java* (continued)

```
        catch (Exception e) {
            System.err.println(e);
            System.err.println("Usage: " +
                "java HardcopyWriter$PrintFile <filename>");
            System.exit(1);
        }
        System.exit(0);
    }
}

/**
 * A program that prints a demo page using HardcopyWriter
 */
public static class Demo extends Frame implements ActionListener {
    /** The main method of the program. Create a test window */
    public static void main(String[] args) {
        Frame f = new Demo();
        f.show();
    }
    // Buttons used in this program
    protected Button print, quit;

    /** Constructor for the test program's window. */
    public Demo() {
        super("HardcopyWriter Test"); // Call frame constructor
        Panel p = new Panel(); // Add a panel to the frame
        this.add(p, "Center"); // Center it
        p.setFont(new Font("SansSerif", // Set a default font
            Font.BOLD, 18));
        print = new Button("Print Test Page"); // Create a Print button
        quit = new Button("Quit"); // Create a Quit button
        print.addActionListener(this); // Specify that we'll handle
        quit.addActionListener(this); // button presses
        p.add(print); // Add the buttons to panel
        p.add(quit);
        this.pack(); // Set the frame size
    }

    /** Handle the button presses */
    public void actionPerformed(ActionEvent e) {
        Object o = e.getSource();
        if (o == quit) System.exit(0);
        else if (o == print) printDemoPage();
    }

    /** Print the demo page */
    public void printDemoPage() {
        // Create a HardcopyWriter, using a 10 point font and 3/4" margins.
        HardcopyWriter hw;
        try { hw=new HardcopyWriter(this, "Demo Page",10,.75,.75,.75,.75);}
        catch (HardcopyWriter.PrintCanceledException e) { return; }
    }
}
```


Example 13-4. *HardcopyWriter.java* (continued)

```
        // And mark the lower-left and lower-right
        out.print("+"); // lower-left
        for(int i=0;i<cols-2;i++) out.print(" "); // space-over
        out.print("+"); // lower-right

        // Close the output stream, forcing the page to be printed
        out.close();
    }
}
```

Advanced Printing with Java 1.4

The Java 1.4 Printing API is substantially more complex than the Java 1.1 and Java 1.2 APIs. It is a command-line utility that can:

- List available printers capable of handling specific printing requests (color printing, collation, stapling, etc.)
- Query the status of a particular named printer
- Spool text or image files directly to a printer
- Convert GIF, JPEG, and PNG image files to PostScript files

Example 13-5 demonstrates some of these advanced features. The example does not create a GUI, and, unlike the other examples in this chapter, it prints without using a `Graphics` object. The code is well-commented and straightforward; the example is long only because it demonstrates several different features of the Java 1.4 API.

Example 13-5. *Print.java*

```
package je3.print;
import javax.print.*;
import javax.print.event.*;
import javax.print.attribute.*;
import javax.print.attribute.standard.*;
import java.io.*;

/**
 * This utility program demonstrates the javax.print API and allows you to
 * list available printers, query a named printer, print text and image files
 * to a printer, and print to PostScript files.
 *
 * Usage:
 * java Print -i inputfile [-q] [-p printer] [-ps outputfile] [attributes]
 */
public class Print {
    public static void main(String[] args) throws IOException {
        // These are values we'll set from the command-line arguments
        boolean query = false;
        String printerName = null;
        String inputFileName = null;
```

Example 13-5. Print.java (continued)

```
String outputFileName = null;
String outputFileType = null;
PrintRequestAttributeSet attributes =
    new HashPrintRequestAttributeSet();

// Loop through the arguments
for(int i = 0; i < args.length; i++) {
    if (args[i].equals("-q")) query = true; // Is this a query?
    else if (args[i].equals("-p"))        // Specific printer name
        printerName = args[++i];
    else if (args[i].equals("-i"))        // The file to print
        inputFileName = args[++i];
    else if (args[i].equals("-ps")) {     // Print it to this file
        // Sun's Java 1.4 implementation only supports PostScript
        // output. Other implementations might offer PDF, for example.
        outputFileName = args[++i];
        outputFileType = "application/postscript";
    }
    // The rest of the arguments represent common printing attributes
    else if (args[i].equals("-color"))    // Request a color printer
        attributes.add(Chromaticity.COLOR);
    else if (args[i].equals("-landscape")) // Request landscape mode
        attributes.add(OrientationRequested.LANDSCAPE);
    else if (args[i].equals("-letter"))   // US Letter-size paper
        attributes.add(MediaSizeName.NA_LETTER);
    else if (args[i].equals("-a4"))       // European A4 paper
        attributes.add(MediaSizeName.ISO_A4);
    else if (args[i].equals("-staple"))   // Request stapling
        attributes.add(Finishings.STAPLE);
    else if (args[i].equals("-collate"))  // Collate multiple copies
        attributes.add(SheetCollate.COLLATED);
    else if (args[i].equals("-duplex"))   // Request 2-sided
        attributes.add(Sides.DUPLEX);
    else if (args[i].equals("-2"))        // 2 pages to a sheet
        attributes.add(new NumberUp(2));
    else if (args[i].equals("-copies"))   // how many copies
        attributes.add(new Copies(Integer.parseInt(args[++i])));
    else {
        System.out.println("Unknown argument: " + args[i]);
        System.exit(1);
    }
}

if (query) {
    // If the -q argument was specified, but no printer was named,
    // then list all available printers that can support the attributes
    if (printerName == null) queryServices(attributes);
    // Otherwise, look for a named printer that can support the
    // attributes and print its status
    else queryPrinter(printerName, attributes);
}
else if (outputFileName != null)
```

Example 13-5. *Print.java* (continued)

```
        // If this is not a query and we have a filename, print to a file
        printToFile(outputFileName, outputFileType,
                    inputFileName, attributes);
    else
        // Otherwise, print to the named printer, or to the default
        // printer otherwise.
        print(printerName, inputFileName, attributes);

    // The main() method ends here, but there may be a printing thread
    // operating in the background. So the program may not terminate
    // until printing completes.
}

// List names of all PrintServices that can support the attributes
public static void queryServices(PrintRequestAttributeSet attributes) {
    // Find all services that can support the specified attributes
    PrintService[] services =
        PrintServiceLookup.lookupPrintServices(null, attributes);
    // Loop through available services
    for(int i = 0; i < services.length; i++) {
        // Print service name
        System.out.print(services[i].getName());

        // Then query and print the document types it can print
        DocFlavor[] flavors = services[i].getSupportedDocFlavors();
        for(int j = 0; j < flavors.length; j++) {
            // Filter out DocFlavors that have a representation class other
            // than java.io.InputStream.
            String repclass = flavors[j].getRepresentationClassName();
            if (!repclass.equals("java.io.InputStream"))continue;
            System.out.println("\t" + flavors[j].getMimeType());
        }
    }
}

// List details about the named printer
public static void queryPrinter(String printerName,
                                PrintRequestAttributeSet attributes)
{
    // Find the named printer
    PrintService service = getNamedPrinter(printerName, attributes);
    if (service == null) {
        System.out.println(printerName + ": no such printer capable of " +
                            "handling the specified attributes");
        return;
    }

    // Print status and other information about the printer
    System.out.println(printerName + " status:");
    Attribute[] attrs = service.getAttributes().toArray();
}
```

Example 13-5. Print.java (continued)

```
        for(int i = 0; i < attrs.length; i++)
            System.out.println("\t" + attrs[i].getName() + ": " + attrs[i]);
    }

    // Print the contents of the named file to the named printer (or to a
    // default printer if printerName is null) requesting the specified
    // attributes.
    public static void print(String printerName, String filename,
                            PrintRequestAttributeSet attributes)
        throws IOException
    {
        // Look for a printer that can support the attributes
        PrintService service = getNamedPrinter(printerName, attributes);
        if (service == null) {
            System.out.println("Can't find a printer " +
                               "with specified attributes");
            return;
        }
        // Print the file to that printer. See method definition below
        printToService(service, filename, attributes);
        // Let the user know where to pick up their printout
        System.out.println("Printed " + filename + " to " + service.getName());
    }

    // Print to an output file instead of a printer
    public static void printToFile(String outputFileName,
                                   String outputFileType,
                                   String inputFileName,
                                   PrintRequestAttributeSet attributes)
        throws IOException
    {
        // Determine whether the system can print to the specified type, and
        // get a factory object if so.
        // The name of this static method is way too long!
        StreamPrintServiceFactory[] factories = StreamPrintServiceFactory.
            lookupStreamPrintServiceFactories(null, outputFileType);

        // Error message if we can't print to the specified output type
        if (factories.length == 0) {
            System.out.println("Unable to print files of type: " +
                               outputFileType);
            return;
        }

        // Open the output file
        FileOutputStream out = new FileOutputStream(outputFileName);
        // Get a PrintService object to print to that file
        StreamPrintService service = factories[0].getPrintService(out);
```

Example 13-5. *Print.java* (continued)

```
// Print using the method below
printToService(service, inputFileName, attributes);
// And remember to close the output file
out.close();
}

// Print the contents of the named file to the specified PrintService,
// requesting the specified attributes.
// This is shared code used by print() and printToFile() above.
public static void printToService(PrintService service, String filename,
                                PrintRequestAttributeSet attributes)
    throws IOException
{
    // Figure out what type of file we're printing
    DocFlavor flavor = getFlavorFromFilename(filename);
    // Open the file
    InputStream in = new FileInputStream(filename);
    // Create a Doc object to print from the file and flavor.
    Doc doc = new SimpleDoc(in, flavor, null);
    // Create a print job from the service
    DocPrintJob job = service.createPrintJob();

    // Monitor the print job with a listener
    job.addPrintJobListener(new PrintJobAdapter() {
        public void printJobCompleted(PrintJobEvent e) {
            System.out.println("Print job complete");
            System.exit(0);
        }
        public void printDataTransferCompleted(PrintJobEvent e) {
            System.out.println("Document transferred to printer");
        }
        public void printJobRequiresAttention(PrintJobEvent e) {
            System.out.println("Print job requires attention");
            System.out.println("Check printer: out of paper?");
        }
        public void printJobFailed(PrintJobEvent e) {
            System.out.println("Print job failed");
            System.exit(1);
        }
    });

    // Now print the document, catching errors
    try {
        job.print(doc, attributes);
    }
    catch(PrintException e) {
        System.out.println(e);
        System.exit(1);
    }
}
```

Example 13-5. *Print.java* (continued)

```
// A utility method to look up printers that can support the specified
// attributes and return the one that matches the specified name.
public static PrintService getNamedPrinter(String name,
                                           PrintRequestAttributeSet attrs)
{
    PrintService[] services =
        PrintServiceLookup.lookupPrintServices(null, attrs);
    if (services.length > 0) {
        if (name == null) return services[0];
        else {
            for(int i = 0; i < services.length; i++) {
                if (services[i].getName().equals(name)) return services[i];
            }
        }
    }
    return null;
}

// A utility method to return a DocFlavor object matching the
// extension of the filename.
public static DocFlavor getFlavorFromFilename(String filename) {
    String extension = filename.substring(filename.lastIndexOf('.')+1);
    extension = extension.toLowerCase();
    if (extension.equals("gif"))
        return DocFlavor.INPUT_STREAM.GIF;
    else if (extension.equals("jpeg"))
        return DocFlavor.INPUT_STREAM.JPEG;
    else if (extension.equals("jpg"))
        return DocFlavor.INPUT_STREAM.JPEG;
    else if (extension.equals("png"))
        return DocFlavor.INPUT_STREAM.PNG;
    else if (extension.equals("ps"))
        return DocFlavor.INPUT_STREAM.POSTSCRIPT;
    else if (extension.equals("txt"))
        return DocFlavor.INPUT_STREAM.TEXT_PLAIN_HOST;
    // Fallback: try to determine flavor from file content
    else return DocFlavor.INPUT_STREAM.AUTONSENSE;
}
}
```

Exercises

Exercise 13-1. In Java 1.3, `java.awt.JobAttributes` and `java.awt.PageAttributes` are new classes that enhance the Java 1.1 Printing API. Read the documentation for these classes, then modify the `HardcopyWriter` class so that you can pass `PageAttributes` and `JobAttributes` objects to the `HardcopyWriter` constructor. Test your modification by writing a program that prints in landscape mode and uses duplex (two-sided) printing (if you have a printer that supports this feature).

- Exercise 13-2.** With the Java 1.2 Printing API, the `print()` method of a `Printable` object may be called multiple times per page to facilitate the printing of high-resolution images and drawings. For this reason, it is not possible to print a character stream with the Java 1.2 API, as is done with the Java 1.1 API in the `HardcopyWriter` class. Nevertheless, it is still possible to print multipage text documents using the Java 1.2 API. Write a program that reads a plain text file, paginates it, and prints its contents using the Java 1.2 API. If you paginate the entire document in an initial pass, then you can implement the `Pageable` interface. Otherwise, if you paginate as you print, you'll have to implement the `Printable` interface. In either case, you should be able to reuse code from `HardcopyWriter`.
- Exercise 13-3.** Modify your program from the previous example so that the `Printable` or `Pageable` object you implemented is printed with a Java 1.4 `javax.print.PrintService` object instead of a Java 1.2 `java.print.PrinterJob`.
- Exercise 13-4.** Modify your program from the previous exercise again to support “2-up” printing—printing two reduced-size pages side by side in landscape mode. You'll need to use Java 2D scaling and rotation features to accomplish this.