

In this chapter:

- *PrintGraphics Interface*
- *PrintJob Class*
- *Component Methods*
- *Printing Example*
- *Printing Arbitrary Content*

Printing

Java 1.1 introduces the ability to print, a capability that was sadly missing in Java 1.0, even though the `Component` class had `print()` and `printAll()` methods. However, it is possible to print arbitrary content, including multipage documents. The printing facility in Java 1.1 is designed primarily to let a program print its display area or any of the components within its display.

Printing is implemented with the help of one public interface, `PrintGraphics`, and one public class, `PrintJob`, of AWT. The real work is hidden behind classes provided with the toolkit for your platform. On Windows NT/95 platforms, these classes are `sun.awt.windows.WPrintGraphics` and `sun.awt.windows.WPrintJob`. Other platforms have similarly named classes.

Printing from an applet has security implications and is restricted by the `SecurityManager`. It is reasonable to suppose that a browser will make it possible to print a page containing an applet; in fact, Netscape has done so ever since Navigator 3.0. However, this ability might not take advantage of Java's printing facility. It isn't reasonable to suppose that an applet will be able to initiate a print job on its own. You might allow a signed applet coming from a trusted source to do so, but you wouldn't want to give any random applet access to your printer. (If you don't understand why, imagine the potential for abuse.)

17.1 PrintGraphics Interface

Printing is similar to drawing an object on the screen. Just as you draw onto a graphics context to display something on the screen, you draw onto a "printing context" to create an image for printing. Furthermore, the printing context and

graphics context are very closely related. The graphics context is an instance of the class `Graphics`. The printing context is also an instance of `Graphics`, with the additional requirement that it implement the `PrintGraphics` interface. Therefore, any methods that you use to draw graphics can also be used for printing. Furthermore, the `paint()` method (which a component uses to draw itself on the screen) is also called when a component must draw itself for printing.

In short, to print, you get a special `Graphics` object that implements the `PrintGraphics` interface by calling the `getGraphics()` method of `PrintJob` (discussed later in this chapter) through `Toolkit`. You then call a component's `print()` or `printAll()` method or a container's `printComponents()` method, with this object as the argument. These methods arrange for a call to `paint()`, which can draw on the printing context to its heart's content. In the simple case where you're just rendering the component on paper, you shouldn't have to change `paint()` at all. Of course, if you are doing something more complex (that is, printing something that doesn't look exactly like your component), you'll have to modify `paint()` to determine whether it's painting on screen or on paper, and act accordingly. The code would look something like this:

```
public void paint(Graphics g) {
    if (g instanceof PrintGraphics) {
        // Printing
    }else {
        // Painting
    }
}
```

If the graphics object you receive is an instance of `PrintGraphics`, you know that `paint()` has been called for a print request and can do anything specific to printing. As I said earlier, you can use all the methods of `Graphics` to draw on `g`. If you're printing, though, you might do anything from making sure that you print in black and white to drawing something completely different. (This might be the trick you use to print the contents of a component rather than the component itself. However, as of Java 1.1, it's impossible to prevent the component from drawing itself. Remember that your `paint()` method was never responsible for drawing the component; it only drew additions to the basic component. For the time being, it's the same with printing.)

When you call `printComponents()` on a `Container`, all the components within the container will be printed. Early beta versions of 1.1 only painted the outline of components within the container. The component should print as it appears on the screen.

17.1.1 Methods

public abstract PrintJob getPrintJob () ★

The `getPrintJob()` method returns the `PrintJob` instance that created this `PrintGraphics` instance.

This seems like circular logic: you need a `PrintJob` to create a `PrintGraphics` object, but you can get a `PrintJob` only from a `PrintGraphics` object. To break the circle, you can get an initial `PrintJob` by calling the `getPrintJob()` method of `Toolkit`. `getPrintJob()` looks like it will be useful primarily within `paint()`, where you don't have access to the original `PrintJob` object and need to get it from the graphics context.

System-provided `PrintGraphics` objects inherit their other methods from the `Graphics` class, which is discussed in Chapter 2, *Simple Graphics*.^{*} The one method that's worth noting here is `dispose()`. In a regular `Graphics` object, calling `dispose()` frees any system resources the object requires. For a `PrintGraphics` object, `dispose()` sends the current object to the printer prior to deallocating its resources. Calling `dispose()` is therefore equivalent to sending a form feed to eject the current page.

17.2 PrintJob Class

The abstract `PrintJob` class provides the basis for the platform-specific printing subclasses. Through `PrintJob`, you have access to properties like page size and resolution.

17.2.1 Constructor and Pseudo-Constructor

public PrintJob () ★

The `PrintJob()` constructor is public; however, the class is abstract, so you would never create a `PrintJob` instance directly.

Since you can't call the `PrintJob` constructor directly, you need some other way of getting a print job to work with. The proper way to get an instance of `PrintJob` is to ask the `Toolkit`, which is described in Chapter 15, *Toolkit and Peers*. The `getPrintJob()` method requires a `Frame` as the first parameter, a `String` as the second parameter, and a `Properties` set as the third parameter. Here's how you might call it:

^{*} Anything can implement the `PrintGraphics` interface, not just subclasses of `Graphics`. However, in order for `paint()` and `print()` to work, it must be a subclass of `Graphics`.

```
PrintJob pjob = getToolkit().getPrintJob(aFrame, "Job Title",
                                         (Properties)null);
```

The `Frame` is used to hold a print dialog box, asking the user to confirm or cancel the print job. (Whether or not you get the print dialog may be platform specific, but your programs should always assume that the dialog may appear.) The `String` is the job's title; it will be used to identify the job in the print queue and on the job's header page, if there is one.

The `Properties` parameter is used to request printing options, like page reversal. The property names, and whether the requested properties are honored at all, are platform specific. UNIX systems use the following properties:

```
awt.print.printer
awt.print.paperSize
awt.print.destination
awt.print.orientation
awt.print.options
awt.print.fileName
awt.print.numCopies
```

Windows NT/95 ignores the properties sheet. If the properties sheet is `null`, as in the previous example, you get the system's default printing options. If the properties sheet is non-`null`, `getPrintJob()` modifies it to show the actual options used to print the job. You can use the modified properties sheet to find out what properties are recognized on your system and to save a set of printing options for use on a later print job.

If you are printing multiple pages, each page should originate from the same print job.

According to Sun's documentation, `getPrintJob()` ought to return `null` if the user cancels the print job. However, this is a problem. On some platforms (notably Windows NT/95), the print dialog box doesn't even appear until you call the `getGraphics()` method. In this case, `getPrintJob()` still returns a print job and never returns `null`. If the user cancels the job, `getGraphics()` returns `null`.

17.2.2 Methods

public abstract Graphics getGraphics () ★

The `getGraphics()` method returns an instance of `Graphics` that also implements `PrintGraphics`. This graphics context can then be used as the parameter to methods like `paint()`, `print()`, `update()`, or `printAll()` to print a single page. (All of these methods result in calls to `paint()`; in `paint()`, you draw whatever you want to print on the `Graphics` object.)

On Windows NT/95 platforms, `getGraphics()` returns `null` if the user cancels the print job.

public abstract Dimension getPageDimension () ★

The `getPageDimension()` method returns the dimensions of the page in pixels, as a `Dimension` object. Since `getGraphics()` returns a graphics context only for a single page, it is the programmer's responsibility to decide when the current page is full, print the current page, and start a new page with a new `Graphics` object. The page size is chosen to roughly represent a screen but has no relationship to the page size or orientation.

public abstract int getPageResolution () ★

The `getPageResolution()` method returns the number of pixels per inch for drawing on the page. It is completely unclear what this means, since the number returned has no relationship to the printer resolution. It appears to be similar to the screen resolution.

public abstract boolean lastPageFirst () ★

The `lastPageFirst()` method lets you know if the user configured the printer to print pages in reverse order. If this returns `true`, you need to generate the last page first. If `false`, you should print the first page first. This is relevant only if you are trying to print a multipage document.

public abstract void end () ★

The `end()` method terminates the print job. This is the last method you should call when printing; it does any cleaning up that's necessary.

public void finalize () ★

The `finalize()` method is called by the garbage collector. In the event you forget to call `end()`, `finalize()` calls it for you. However, it is best to call `end()` as soon as you know you have finished printing; don't rely on `finalize()`.

17.3 Component Methods

The methods that start the printing process come from either the `Component` or `Container` class and are inherited by all their children. All components inherit the `printAll()` and `print()` methods. Containers also inherit the `printComponents()` method, in addition to `printAll()` and `print()`. A container should call `printComponents()` to print itself if it contains any components. Otherwise, it is sufficient to call `printAll()`.

These methods end up calling `paint()`, which does the actual drawing.

17.4 Printing Example

Now that you know about the different classes necessary to print, let's put it all together. Printing takes four steps:

1. Get the `PrintJob`:

```
PrintJob pjob = getToolkit().getPrintJob(this, "Job Title", (Properties)null);
```

2. Get the graphics context from the `PrintJob`:

```
Graphics pg = pjob.getGraphics();
```

3. Print by calling `printAll()` or `print()`. When this method returns, you can call `dispose()` to send the page to the printer:

```
printAll(pg);
pg.dispose(); // This is like sending a form feed
```

4. Clean up after yourself:

```
pjob.end();
```

The following code summarizes how to print:

```
// Java 1.1 only
PrintJob pjob = getToolkit().getPrintJob(this, "Print?", (Properties)null);
if (pjob != null) {
    Graphics pg = pjob.getGraphics();
    if (pg != null) {
        printAll(pg);
        pg.dispose();
    }
    pjob.end();
}
```

This code prints the current component: what you get from the printer should be a reasonable rendition of what you see on the screen. Note that we didn't need to modify `paint()` at all. That should always be the case if you want your printer output to look like your onscreen component.

17.5 Printing Arbitrary Content

Of course, in many situations, you want to do more than print the appearance of a component. You often want to print the contents of some component, rather than the component itself. For example, you may want to print the text the user has typed into a text area, rather than the text area itself. Or you may want to print the contents of a spreadsheet, rather than the collection of components that compose the spreadsheet.

Java 1.1 lets you print arbitrary content, which may include multipage documents. You aren't restricted to printing your components' appearance. In many ways, the steps required to print arbitrary content are similar to those we outlined previously. However, a few tricks are involved:

1. Get the `PrintJob`:

```
PrintJob pjob = getToolkit().getPrintJob(this, "Job Title", (Properties)null);
```

2. Get the graphics context from the `PrintJob`:

```
Graphics pg = pjob.getGraphics();
```

3. Don't call `printAll()` or `print()`. These methods will try to draw your component on the page, which you don't want. Instead, get the dimensions of the page by calling `getPageDimension()`:

```
pjob.getPageDimension();
```

4. Set the font for your graphics context; then get the font metrics from your graphics context.

```
Font times = new Font ("SansSerif", Font.PLAIN, 12);  
pg.setFont(times);  
FontMetrics tm = pg.getFontMetrics(times);
```

5. Draw whatever you want into the graphics context, using the methods of the `Graphics` class. If you are drawing text, it's your responsibility to do all the positioning, making sure that your text falls within the page boundaries. By the time you're through with this, you'll have the `FontMetrics` class memorized.
6. When you've finished drawing the current page, call `dispose()`; this sends the page to the printer and releases the resources tied up by the `PrintGraphics` object.

```
pg.dispose(); // This is like sending a form feed
```

7. If you want to print more pages, return to step 2.
8. Clean up after yourself:

```
pjob.end();
```

Remember to set a font for the `PrintGraphics` object explicitly! It doesn't have a default font.

An example that loads and prints a text file is available from this book's Web page.