Printing Programming Topics for Cocoa

Graphics & Animation: Printing

2006-06-28

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Introduction to Printing 5

Organization of This Document 5

Cocoa Printing Architecture 7

Print Operations 9

How Print Information Is Stored 11

Pagination 13

Selecting the Page Bounds 13
- Custom Pagination 13
- Tiling Across Pages 13
- Clipping to the Page 14
- Fitting to the Page 14

Positioning the Page 14
Adorning the Page 14

Customizing a View's Drawing for Printing 17

Providing a Custom Pagination Scheme 19

Creating a Print Job 21

Customizing the Print Operation 22
- Suppressing the Print Panel 22
- Modifying the Print Panel 23

Generating EPS and PDF Data 23

Using a Print Panel 25

Using a Page Setup Panel 27

Printing Documents 29

Using a Page Setup Panel with NSDocument 29
Printing a Document 29

Accessing Printer Information 31

Creating a Printer 31
Retrieving Printer Attributes 31
Retrieving Values from the PPD Table 32
Retrieving Values from the Option and Argument Translation Tables 33
Retrieving Values from the Order Dependency Table 34
Retrieving Values from the UIConstraints Table 34

Document Revision History 37

Introduction to Printing

This programming topic covers Cocoa's printing architecture and explains how to support printing in a Cocoa application. Cocoa provides a set of classes that work together to make basic printing support easy. These behaviors can also be customized to provide the level of control your application needs in its printing behavior.

To understand how Cocoa printing works in the context of the Mac OS X printing system as a whole, you should first read *Mac OS X Printing System Overview*

Organization of This Document

This programming topic contains these articles:

- "Cocoa Printing Architecture" (page 7) provides an overview of the classes involved in printing and how they work together.
- "Print Operations" (page 9) describes NSPrintOperation, the class that controls a print job.
- "How Print Information Is Stored" (page 11) describes NSPrintInfo, the class that stores information for print jobs.
- "Pagination" (page 13) describes the options for placing an NSView object onto the printed page.
- "Creating a Print Job" (page 21) describes how to create a print job and customize it to fit your needs.
- "Customizing a View's Drawing for Printing" (page 17) describes how a view can detect that it is drawing to a printer and modify its behavior accordingly.
- "Providing a Custom Pagination Scheme" (page 19) shows an example of how a view can implement its own pagination scheme.
- "Using a Page Setup Panel" (page 27) describes how to use and extend the Page Setup panel.
- "Using a Print Panel" (page 25) describes how to use and extend the Print panel.
- "Printing Documents" (page 29) describes how to make use of NSDocument's printing features.
- "Accessing Printer Information" (page 31) describes how to create printer objects and retrieve printer attributes.

Cocoa Printing Architecture

Cocoa's printing architecture is composed of a half-dozen interconnected classes. Separate classes represent the Page Setup panel, the Print panel, the print settings, the print operation, the printer, and the object to be printed. Each of these classes is described here.

- NSView generates the data to be printed. Because Cocoa drawing is device independent, a view generates print commands simply by drawing itself with regular Application Kit and Core Graphics drawing commands after the display device has been changed to a printer. A view can be told to print itself by invoking its `print:` method.
- NSPrintInfo stores the options that control how a print job is performed. This information includes the paper size, number of copies, print margins, and so on. A shared instance holding default settings is used by the other objects of the print system if you do not provide your own instance for a particular print job.
- NSPageLayout displays the Page Setup panel where the user selects the paper type and orientation. The settings are stored in an NSPrintInfo object. You can add a custom accessory view to the Page Setup panel to display application-specific options.
- NSPrintPanel displays the Print panel where the user selects the output options for a print job, such as the number of copies and the pages to print. The settings are stored in an NSPrintInfo object. You can add a custom accessory view to the Print panel to display application-specific options.
- NSPrintOperation manages a print job. It displays the Print panel, optionally spawns a new thread to process the print job, sets up the print environment, and tells the NSView to print itself. It can also generate Encapsulated PostScript (EPS) or Portable Document Format (PDF) data instead of sending the results to a printer.
- NSPrinter describes a printer's capabilities as defined in its PostScript Printer Description (PPD) file. (Despite its name, non-PostScript printers can have PPDs, too.) Printers can be specified either by name or by type (make and model). You can also use several class methods to obtain a list of all the printers, organized either by name or by type, that the user has added to his or her printer list.

Printing is generally initiated by the user choosing the Print menu command, which usually sends either a `print:` or `printDocument:` message, depending on whether the application is NSDocument-based or not, up the responder chain. You receive the message either in a custom NSView object (if it has the keyboard focus), a window delegate, or an NSDocument object. Upon receipt, you create an NSPrintOperation object to manage the print job, tell it which NSView to print, add an accessory view to its print panel, if desired, and then run the operation. The NSView class defines several methods that you can override to control how the view is divided between multiple pages. From there, the view's `drawRect:` method draws the view's contents.

Subsequent sections of this document describe in greater detail how each part of Cocoa's printing architecture works.

Print Operations

An NSPrintOperation object controls the process that creates a print job. Print jobs are normally sent to a printer, but they can also be used to generate Portable Document Format (PDF) and Encapsulated PostScript (EPS) data for your application. An NSPrintOperation object does not generate the print code itself; it just controls the overall process, relying on an NSView object to generate the actual code.

NSPrintOperation works in conjunction with two other objects: an NSPrintInfo object, which specifies how the code should be generated, and an NSView object, which performs the actual code generation. You specify these two objects when you create an NSPrintOperation object. If no NSPrintInfo is specified, NSPrintOperation uses the application's shared NSPrintInfo, which contains default values. (A shared NSPrintInfo object is automatically created for an application.)

The shared NSPrintInfo works well for applications that are not document-based. However, document-based applications should create an NSPrintInfo for each document that might be printed and use that object instead. This allows users to set printing attributes on a per-document basis.

Once created, a print operation can be configured in several ways. By default a print operation displays the Print panel to allow the user to modify the current print settings. You can prevent this with the `setShowPanels:` method. You can also tell the print operation to include a custom accessory view in the standard Print panel. The accessory view can present application-specific print settings to the user. (Print operations that create PDF or EPS data never display the Print panel.)

By default, the print operation performs the data generation on the current thread. This thread is normally the application's main thread, or the thread that processes user events. You can tell the print operation to instead spawn a new thread and generate the print job on it, using the `setCanSpawnSeparateThread:` method. This allows your application to continue processing events, instead of blocking. (Print operations that create PDF or EPS data always run on the current thread.)

After configuring the print operation, you tell it to run by sending it the `runOperation` or `runOperationModalForWindow:delegate:didRunSelector:contextInfo:` message. The first runs the Print panel as an application modal dialog and blocks until the print operation completes. The second displays the Print panel as a sheet on a specified window and returns immediately, sending a message to the delegate when the print operation completes.

How Print Information Is Stored

An NSPrintInfo object stores the attributes that describe a print job. A print info object is passed to an NSPrintOperation object, which makes a copy of it to use during an operation. Normally you don't set NSPrintInfo attributes directly—this is done by instances of NSPageLayout and NSPrintPanel. The NSView that's being printed may also supersede some NSPrintInfo settings, such as the pagination and orientation attributes.

A shared print info object is automatically created for an application and is used by default for all print jobs for that application if you do not provide your own instance to the NSPrintOperation, NSPageLayout, and NSPrintPanel objects. You can create your own print info object with different default settings and make it the shared instance with the NSPrintInfo class method `setSharedPrintInfo:`. You get the shared NSPrintInfo object using the `sharedPrintInfo` class method.

In an NSDocument-based application, each NSDocument has its own print info object, which you can obtain with NSDocument's `printInfo` method. Unless you set one yourself, using NSDocument's `setPrintInfo:` method, the document initially uses a copy of the application's shared object. When the user makes changes in the Page Setup panel, the document's print info object is automatically updated with the new print settings.

An NSPrintInfo object contains a dictionary that stores its attribute settings. The dictionary keys are described in the "Constants" section of NSPrintInfo.

There are a couple cases where you may want to record the settings of a print info object. First, you may want to remember the print settings used the last time your application was run. In this case you could record the print info object as an application preference each time the user prints something and then restore the settings when the application launches. However, because the dictionary that stores an NSPrintInfo object's print settings includes non-property list values, it is not a proper property list object. Therefore, it cannot be converted to a plist format and saved directly as a preference value. Instead, you need to use the NSKeyedArchiver (or NSArchiver) class method `archivedDataWithRootObject:` to encode the NSPrintInfo object as an NSData object, which *can* be stored in a property list or saved to a file. To restore the NSPrintInfo object, reload the NSData object and then use the NSKeyedUnarchiver (or NSUnarchiver) class method `unarchiveObjectWithData:` to decode the NSPrintInfo information.

The second case of saving print info objects applies to document-based applications. Print settings are often document specific. For example, a user may print a wide spreadsheet in landscape mode. That setting should be remembered each time the document is printed but should not be used for any other documents, which the user may prefer to print in portrait mode.Therefore, each document should have its own print info object that is saved with the document and used each time that particular document is printed. As before, you should encode the NSPrintInfo object into an NSData object. Then, you should write the data to the document's file.

Pagination

When a view is printed, there are several options for how it is placed on the page. If the view is larger than a single page, the view can be clipped, resized, or tiled across multiple pages. The view's location on each page can be adjusted. Finally, the view can add adornments to each page. The following sections describe the options available for placing the view onto a page.

Selecting the Page Bounds

When a view is too large to fit onto a single page, the view can be printed in one of several ways. The view can tile itself out onto separate logical pages so that its entire visible region is printed. Alternatively, the view can clip itself and print only the area that fits on the first page. Finally, the view can resize itself to fit onto a single page. These options can be set using the NSPrintInfo object's `setHorizontalPagination:` and `setVerticalPagination:` methods with the constants `NSClipPagination`, `NSFitPagination`, and `NSAutoPagination`. The separate methods for horizontal and vertical pagination allow you to mix these behaviors. For example, you can clip the image in one dimension, but tile it in the other. If these options are not sufficient, the view can also implement its own pagination scheme. The following sections describe each option.

Custom Pagination

To provide a completely custom pagination scheme that does not use NSView's built-in pagination support, a view needs to implement only two simple methods. A custom view signals that it will be calculating each page's dimensions by overriding the `knowsPageRange:` method to return `YES`. This method also returns by reference the page number range available for printing. Within the Print panel, the user can select a subset of this range for printing. The pagination machinery then sends a `rectForPage:` message to the view for each page in the user-selected range. This method is sent before each page is printed. Your implementation of `rectForPage:` should use the page number and the current printing information to calculate an appropriate rectangle in the view's coordinate system. The vertical and horizontal pagination settings in the NSPrintInfo object are ignored (unless your implementation takes them into account).

See "Providing a Custom Pagination Scheme" (page 19) for an example of a view implementing its own pagination scheme.

Tiling Across Pages

If the view does not supply its own pagination information and one of the print info object's pagination settings is `NSAutoPagination`, NSView tries to fit as much of the view being printed onto a logical page, slicing the view into the largest possible chunks along the given direction (horizontal or vertical). This is sufficient for many views, but if a view's image must be divided only at certain places—between lines of text or cells in a table, for example—the view can adjust the automatic mechanism to accommodate this by reducing the height or width of each page.

Before printing begins, the view calculates the positions of all the row and column page breaks and gives you an opportunity to adjust them. `adjustPageHeightNew:top:bottom:limit:` (in Objective-C only) provides an out parameter for the new bottom coordinate of the page, followed by the proposed top and bottom. An additional parameter limits the height of the page; the bottom can't be moved above it. `adjustPageWidthNew:left:right:limit:` (in Objective-C only) works in the same way to allow the view to adjust the width of a page. The limits are calculated as a percentage of the proposed page's height or width. Your view subclass can also customize this percentage by overriding the methods `heightAdjustLimit` and `widthAdjustLimit` (in both Objective-C and Java) to return the fraction of the page that can be adjusted; a value of zero indicates that no adjustments are allowed whereas a value of one indicates that the right or bottom edge of the page bounds can be adjusted all the way to the left or top edge.

Clipping to the Page

If one of the print info object's pagination values is `NSClipPagination`, the view is clipped to a single page along that dimension. If the horizontal pagination is set to clipped, the left most section of the view is printed, clipped to the width of a single page. If the vertical pagination is set to clipped, the top most section of the view is printed, clipped to the height of a single page.

Fitting to the Page

If the print info object's pagination setting is `NSFitPagination`, the image is resized to fit onto the page. Although vertical and horizontal pagination need not be the same, if either dimension is scaled, the other dimension is scaled by the same amount to avoid distorting the image. If both dimensions are scaled, the scaling factor that produces the smaller image is used, thereby avoiding both distortion and clipping. Note that print info object's scaling factor (`NSPrintScalingFactor`), which the user sets in the Page Layout panel, is independent of the scaling that's imposed by pagination and is applied after the pagination scaling.

Positioning the Page

The next stage of pagination involves placing the image to be printed on the logical page. NSView's instance method `locationOfPrintRect:` places it according to several print info attributes. By default it places the image in the upper left corner of the page, but if the print info object's `isHorizontallyCentered` or `isVerticallyCentered` methods return `YES`, it centers a single-page image along the appropriate axis. A multiple-page document, however, is always placed at the top left corner of the page so that the divided pieces can be assembled at their edges.

Override this method to position the image yourself. The returned point is relative to the bottom-left corner of the paper in page coordinates. You need to include the page margins in calculating the position.

Adorning the Page

After the NSView has sliced out a rectangle and positioned it on a page, it's given a chance to add extra marks to the page, such as crop marks or page numbers. `drawPageBorderWithSize:` is used for logical pages, and is invoked once for each page.

See the "Providing a Custom Pagination Scheme" (page 19) task for more information on using `drawPageBorderWithSize:`.

Customizing a View's Drawing for Printing

A view gets drawn for the printer the same way as for drawing to the screen. `drawRect:` messages are sent to the view identifying what region should be drawn. Core Graphics then translates the drawing operations performed by the view into the appropriate representations for displaying on the screen or rendering on a printer.

In some cases you need to behave differently when drawing to the screen versus to a printer. For example, a selected object on screen may have handles or a highlight that should not be drawn when printing. To test whether you are drawing to the screen, send the message `currentContextDrawingToScreen` to NSGraphicsContext, as shown here:

```
- (void)drawRect:(NSRect)r {
    if ( [NSGraphicsContext currentContextDrawingToScreen] ) {
        // Draw screen-only elements here
    } else {
        // Draw printer-only elements here
    }
    // Draw common elements here
}
```

You may also need to adjust your drawing based on an attribute in the print operation's NSPrintInfo object. You can get the current print operation with the NSPrintOperation class method `currentOperation` and then get its NSPrintInfo object from the `printInfo` method.

```
NSPrintOperation *op = [NSPrintOperation currentOperation];
NSPrintInfo *pInfo = [op printInfo];
```

Providing a Custom Pagination Scheme

To provide a completely custom pagination scheme that does not use NSView's built-in pagination support, a view must override the `knowsPageRange:` method to return `YES`. It should also return by reference the page range for the document. Before printing each page, the pagination machinery sends the view a `rectForPage:` message. Your implementation of `rectForPage:` should use the supplied page number and the current printing information to calculate an appropriate drawing rectangle in the view's coordinate system.

The following example shows a very simple implementation that merely splits the view vertically into the maximum page-sized pieces:

```
// Return the number of pages available for printing
- (BOOL)knowsPageRange:(NSRangePointer)range {
    NSRect bounds = [self bounds];
    float printHeight = [self calculatePrintHeight];

    range->location = 1;
    range->length = NSHeight(bounds) / printHeight + 1;
    return YES;
}

// Return the drawing rectangle for a particular page number
- (NSRect)rectForPage:(int)page {
    NSRect bounds = [self bounds];
    float pageHeight = [self calculatePrintHeight];
    return NSMakeRect( NSMinX(bounds), NSMaxY(bounds) - page * pageHeight,
                       NSWidth(bounds), pageHeight );
}

// Calculate the vertical size of the view that fits on a single page
- (float)calculatePrintHeight {
    // Obtain the print info object for the current operation
    NSPrintInfo *pi = [[NSPrintOperation currentOperation] printInfo];

    // Calculate the page height in points
    NSSize paperSize = [pi paperSize];
    float pageHeight = paperSize.height - [pi topMargin] - [pi bottomMargin];

    // Convert height to the scaled view
    float scale = [[[pi dictionary] objectForKey:NSPrintScalingFactor]
                    floatValue];
    return pageHeight / scale;
}
```

As part of the custom pagination, you can also add extra features to the page, such as crop marks, date/time strings, or page numbers. This is done, for each page, with `drawPageBorderWithSize:`.

You must override `drawPageBorderWithSize:` to make it functional, as its default implementation prints nothing to the page. In this method, first save the view's existing body frame—it will need to be restored at the end of the method. Once the old frame is saved, resize the body frame to a rect with origin `(0, 0)` and a size equal to the incoming *`borderSize`* parameter. This new frame now encompasses the margins instead of hiding them.

Once the frame is expanded, you can add your custom border elements to all four margin areas (top, bottom, left, and right). Drawing is typically done with `drawAtPoint:`. Any set of drawing calls must be preceded by `lockFocus:` and followed by `unlockFocus:`, otherwise `drawPageBorderWithSize:` will not draw anything to the page for those calls. Use the paper and margin dimensions from the print info object to constrain the printable area and prevent `drawPageBorderWithSize:` from printing within the body text frame. If you wish to print within the body text frame—to print a watermark, for example—do so by printing directly in the newly enlarged frame and ignoring the margin constraints.

Reset the frame to the body text area before exiting the method; this assures the next page of content will print only within the paginated portion of the view.

Creating a Print Job

You create NSPrintOperation objects in response to the user choosing the Print menu command. You initialize the print operation object with the view to be printed and, optionally, the NSPrintInfo object holding the print settings. The operation is not started, however, until you invoke one of NSPrintOperation's `runOperation` methods. For example, a view can have a simple `print:` method as in the following example, which merely starts a print operation for the view that receives the `print:` message:

```
- (void)print:(id)sender {
    [[NSPrintOperation printOperationWithView:self] runOperation];
}
```

This implementation of `print:` starts by creating an NSPrintOperation object, which manages the process of generating proper code for a printer device. When run, the NSPrintOperation object creates and displays a modal print panel, an NSPrintPanel object, to obtain the print settings from the user. The application's shared NSPrintInfo object is used for the initial settings. The method blocks until the print operation is complete.

This NSView-based implementation works best when you have only one printable view in your window that can ever be the first responder. For example, in a simple text editor, only the view holding the text document can have the focus, so it is safe to implement printing in the text view. You can see an example of this architecture in the TextEdit example project.

When your user interface contains multiple views that can have focus, such as multiple NSTextFields, this implementation breaks down. When you choose the Print menu command, the view that receives the `print:` message prints itself, but nothing else. If the focus is in an NSTextField, for example, only the contents of that text field is printed. This probably is not the desired behavior. Instead, your application needs to take a more document-based approach.

In a document-based model, you need to customize the Print menu command (as usually defined in the main nib file) to send a different message that one of your objects higher in the responder chain, such as a window or application delegate, implements. Your class can then identify, or perhaps construct, the view that you want printed, regardless of which particular view in the window has the current focus. You can also assign separate NSPrintInfo objects to each open document or window.

NSDocument-based applications, such as the Sketch example project, behave as just described. If you create an NSDocument-based application in Xcode, the Print menu command in the default main nib file sends a `printDocument:` message instead of `print:`. If you provide your own document architecture, you must edit the nib file yourself.

Assuming the Print menu command now sends a `printDocument:` message, your document-based application can run a print operation as follows:

```
- (void)printDocument:(id)sender {
    // Assume documentView returns the custom view to be printed
    NSPrintOperation *op = [NSPrintOperation
                printOperationWithView:[self documentView]
                printInfo:[self printInfo]];
    [op runOperationModalForWindow:[self documentWindow]
                delegate:self
```

```
                didRunSelector:
                    @selector(printOperationDidRun:success:contextInfo:)
                contextInfo:NULL];
}

- (void)printOperationDidRun:(NSPrintOperation *)printOperation
                success:(BOOL)success
                contextInfo:(void *)info {
    if (success) {
        // Can save updated NSPrintInfo, but only if you have
        // a specific reason for doing so
        // [self setPrintInfo: [printOperation printInfo]];
    }
}
```

`printDocument:` now creates an NSPrintOperation with the document's NSPrintInfo object—not the application's shared NSPrintInfo object. It also uses the `runOperationModalForWindow:delegate:didRunSelector:contextInfo:` method to run the NSPrintPanel as a sheet on the document's window. This way, the application is not blocked and can continue processing events. The print operation delegate object, `self` in this case, is sent a message when the print operation completes. The callback method can check whether the operation was successful and perform additional actions if necessary. For example, on success the method can save the print operation's NSPrintInfo object for use by the next print operation. It is recommended that most applications *not* save the print info object between print operations. This ensures that the Print panel displays with the appropriate default settings each time.

Customizing the Print Operation

There are a number of ways you can customize the behavior of a print operation. These include running a print operation without a print panel and customizing the print panel.

Suppressing the Print Panel

By default, an NSPrintOperation object displays an NSPrintPanel object allowing the user to select printing options, such as number of copies to print and range of pages to print. After the user fills this out the first time, you may want to offer the user the ability to by-pass the Print panel and just print immediately using the previous print settings.

You can suppress the display of the NSPrintPanel object by sending `setShowPanels:` with a `NO` argument to the NSPrintOperation object before running the operation. However, make sure that any non-default settings in the NSPrintInfo object that would normally be selected from an NSPrintPanel object are set to reasonable values—a copy of an NSPrintInfo object used in a previous print job will have the correct values. This is illustrated in the following example:

```
// Invoked in response to the standard "Print..." menu command
- (void)print:(id)sender {
    NSPrintOperation *op = [NSPrintOperation printOperationWithView:self
                                    printInfo:[self printInfo]];
    if ( [op runOperation] )
        [self setPrintInfo:[op printInfo]];
}
```

```
// Invoked in response to a custom "Print Now" menu command
- (void)printWithNoPanel:(id)sender {
    NSPrintOperation *op;

    op = [NSPrintOperation printOperationWithView:self
                printInfo:[self printInfo]];
    [op setShowPanels:NO];
    [op runOperation];
}
```

Modifying the Print Panel

By default, an NSPrintOperation displays a standard NSPrintPanel. If you need to add some application-specific options, you can add an accessory view. You can load the accessory view from a nib file or create it programmatically. After creating the NSPrintOperation, send it your view with the `setAccessoryView:` method:

```
- (void)print:(id)sender {
    NSPrintOperation *op;

    op = [NSPrintOperation printOperationWithView:self];
    // Assume printAccessoryView loads or creates your custom view
    [op setAccessoryView:[self printAccessoryView]];
    [op runOperation];
}
```

If you need to make more extensive changes to the Print panel, you can subclass NSPrintPanel. You tell NSPrintOperation to use your custom subclass instead of the default panel using its `setPrintPanel:` method.

```
- (void)print:(id)sender {
    NSPrintOperation *op;
    MyPrintPanel *myPanel = [[MyPrintPanel alloc] init];

    op = [NSPrintOperation printOperationWithView:self];
    [op setPrintPanel:myPanel];
    [op runOperation];
    [myPanel release];
}
```

Generating EPS and PDF Data

A print operation does not have to send its results to a printer. You can have the operation generate raw PDF or EPS data and write the data either to an NSMutableData object you provide or to a file at a path you specify. To do so, use one of the `EPSOperation` or `PDFOperation` class methods to create the NSPrintOperation instead of one of the `printOperation` methods. You can identify whether a print operation is generating PDF or EPS data by sending it an `isCopyingOperation` message, which returns `YES` in this case; it returns `NO` if the data are being sent to a printer.

NSView provides several convenience methods for generating PDF and EPS data. The data can be returned in an NSData or written to a pasteboard. For PDF data, NSView implements `dataWithPDFInsideRect:` and `writePDFInsideRect:toPasteboard:`. For EPS the methods are `dataWithEPSInsideRect:` and `writeEPSInsideRect:toPasteboard:`.

These methods create and run an NSPrintOperation, like `print:` does, but the print panel is not displayed. They still use the shared NSPrintInfo object if one is provided, but do not allow the user to modify the defaults.

Using a Print Panel

NSPrintPanel creates a Print panel, which is used to query the user for information about a print job, such as which pages to print and how many copies.

When a `print:` message is sent to an NSView or NSWindow, an NSPrintOperation object is created to control the print operation (see "Creating a Print Job" (page 21) for details). An NSPrintOperation object uses an NSPrintPanel unless it is sent the `setShowPanels:` message, passing `NO` as the argument. If you want to use a custom subclass for a particular print operation, pass an instance of your subclass to the print operation using the `setPrintPanel:` message.

However, you rarely need to subclass NSPrintPanel because you can augment its display by adding your own accessory view using the `setAccessoryView:` method. Place controls for setting application-specific print settings on your accessory view. The accessory view is displayed when the user chooses the appropriate entry in the pane-selection pop-up menu in the Print panel. The application's name is used for the accessory view's entry in the menu. The panel automatically resizes to accommodate the view you add. If possible, you should make your accessory view the same size as the standard views in the panel.

Typically, you do not need to create an NSPrintPanel yourself. NSPrintOperation creates a standard panel for you if you do not set a different one with `setPrintPanel:`. Even if you need to add an accessory view, you can have NSPrintOperation add your view to its default panel using its `setAccessoryView:` method as shown here:

```
- (void)print:(id)sender {
    NSPrintOperation *op = [NSPrintOperation printOperationWithView:self];
    // Assume printAccessoryView exists and returns your custom view
    NSView *accView = [self printAccessoryView];
    [op setAccessoryView:accView];
    [op runOperation];
}
```

If you do need to create an NSPrintPanel, use the `printPanel` class method.

Using a Page Setup Panel

NSPageLayout creates a panel that queries the user for information such as paper size and orientation. This information is stored in an NSPrintInfo object, and is later used when printing. The NSPageLayout panel is created, displayed, and run when a `runPageLayout:` message is sent to the NSApplication or an NSDocument object. NSApplication runs it as a modal panel, whereas NSDocument runs it as a sheet. By default, this message is sent up the responder chain when the user chooses the Page Setup menu command.

If the NSApplication or NSDocument implementation is not sufficient you can handle the Page Setup panel yourself. Typically, you create an NSPageLayout object by invoking the `pageLayout` method. It can be run application modally using `runModal` or `runModalWithPrintInfo:`, or document modally as a sheet using `beginSheetWithPrintInfo:modalForWindow:delegate:didEndSelector:contextInfo:`.

You rarely need to subclass NSPageLayout because you can augment its display by adding your own accessory view using the `setAccessoryView:` method. Place controls for setting application-specific print settings on your accessory view. The accessory view is displayed when the user chooses the appropriate entry in the Settings pop-up menu in the Page Setup panel. The application's name is used for the accessory view's entry in the menu. The panel automatically resizes to accommodate the view you add. If possible, you should make your accessory view the same size as the standard views in the panel.

When running a non-NSDocument-based application, you need to override NSApplication's `runPageLayout:` method (or implement the method earlier in the responder chain), if you need to add an accessory view. Your method might look like this:

```
- (void)runPageLayout:(id)sender {
    NSPageLayout *pageLayout = [NSPageLayout pageLayout];
    // Assume layoutAccessoryView exists and returns your custom view
    NSView *accView = [self layoutAccessoryView];
    [pageLayout setAccessoryView:accView];
    [pageLayout runModal];
}
```

When running an NSDocument-based application, NSDocument's default implementation of `runPageLayout:` creates the page layout panel and then passes the object to its `preparePageLayout:` method. Override this second method to add an accessory view. NSDocument then runs the panel as a sheet attached to its window.

Printing Documents

An application that uses NSDocument to manage its documents gains additional infrastructure to handle document printing.

Because print settings may be different for different documents, each instance of NSDocument has its own NSPrintInfo object, which is accessed with the `printInfo` and `setPrintInfo:` methods.

Using a Page Setup Panel with NSDocument

NSDocument implements the `runPageLayout:` method to handle the Page Setup menu command instead of letting NSApplication handle it. When it receives this message, it gets the document's NSPrintInfo object and invokes `runModalPageLayoutWithPrintInfo:delegate:didRunSelector:contextInfo:` to display the Page Setup panel. To give your NSDocument subclass an opportunity to customize the NSPageLayout object, it passes the object to `preparePageLayout:` before displaying the panel. Override this method if you want to add an accessory view to the panel.

When the panel is dismissed with the OK button, NSDocument checks the return value of its `shouldChangePrintInfo:` method. If it returns `YES`, the default, the document's NSPrintInfo object is updated to reflect the new print settings. You can override this method to validate the new settings and return `NO` if the new settings should be discarded.

Printing a Document

In an NSDocument-based application, the Print menu command normally sends a `printDocument:` message, which only NSDocument implements. The NSDocument object associated with the application's main window receives the message and invokes `printShowingPrintPanel:` with `YES` as its argument. Because NSDocument does not manage the view or views displaying your document data, it cannot provide a default implementation for this printing method. Therefore, your NSDocument subclass *must* override `printShowingPrintPanel:` to create and run the print operation for the document.

When printing a document, you can use either a view object already displaying your data in a window or a special view object that you create just for printing. For example, a simple text editor may display text in a window the same way it prints it on the page, so it can use the same view for both cases. Alternatively, a database program's main window may contain an interface for browsing and editing the database, while the printed data needs to be formatted as a table. In this case, the document needs separate views for drawing in a window and for printing to a printer. If you have a good Model-View-Controller design, you can easily create a custom view that can draw the printer-specific version of your data model and use it when creating the print operation.

After setting up the print operation, invoke NSDocument's `runModalPrintOperation:delegate:didRunSelector:contextInfo:` method to run the print operation and display the NSPrintPanel as a sheet on the document's window.

```
- (void)printShowingPrintPanel:(BOOL)showPanels {
    // Obtain a custom view that will be printed
    NSView *printView = [self printableView];

    // Construct the print operation and setup Print panel
    NSPrintOperation *op = [NSPrintOperation
                printOperationWithView:printView
                printInfo:[self printInfo]];
    [op setShowPanels:showPanels];
    if (showPanels) {
        // Add accessory view, if needed
    }

    // Run operation, which shows the Print panel if showPanels was YES
    [self runModalPrintOperation:op
                delegate:nil
                didRunSelector:NULL
                contextInfo:NULL];
}
```

Accessing Printer Information

An NSPrinter object gives you access to the capabilities of a particular printer as described in the printer's PostScript Printer Description (PPD) file. NSPrinter provides read-only access to the printer's features; you cannot configure a printer's settings.

Creating a Printer

You can create an NSPrinter object by specifying either the printer's name or its type. The user creates the list of printers recognized by NSPrinter using the Print Center application. The NSPrinter class methods `printerNames` and `printerTypes` return the names and types of these printers. You then create an NSPrinter object with `printerWithName:` or `printerWithType:`. If multiple printers have the same type, `printerWithType:` returns the NSPrinter object for the first printer that matches.

The following example creates an NSPrinter object for a printer named "My LaserWriter".

```
/* Create an NSPrinter object from a printer name. */
NSPrinter *printer = [NSPrinter printerWithName:@"My LaserWriter"];
```

Retrieving Printer Attributes

A printer normally has associated with it a PPD file that describes the capabilities of the printer. An NSPrinter object reads the PPD file that corresponds to the printer's type and stores the data it finds there in named tables. NSPrinter defines five key-value tables to store PPD information. The tables are identified by the names given below:

Name	Contents
`PPD`	General information about a printer type. This table contains the values for all entries in a PPD file except those with the `*OrderDependency` and `*UIConstraint` main keywords. The values in this table don't include the translation strings.
`PPDOptionTranslation`	Option keyword translation strings.
`PPDArgumentTranslation`	Value translation strings.
`PPDOrderDependency`	`*OrderDependency` values.
`PPDUIConstraints`	`*UIConstraint` values.

There are two principle methods for retrieving data from the NSPrinter tables:

- `stringForKey:inTable:` returns the value for the first occurrence of a given key in the given table.
- `stringListForKey:inTable:` returns an array of values, one for each occurrence of the key.

For both methods, the first argument is an NSString that names a key—which part of a PPD file entry the key corresponds to depends on the table (as explained in the following sections). The second argument names the table that you want to search. The values that are returned by these methods, whether singular or in an array, are always NSStrings, even if the value wasn't a quoted string in the PPD file.

The two methods described above are sufficient for retrieving any value from any table. NSPrinter provides a number of other methods, such as `booleanForKey:inTable:` and `intForKey:inTable:`, which retrieve single values and coerce them, if possible, into particular data types.

To check the integrity of a table, use the `isKey:inTable:` and `statusForTable:` methods. The former returns a boolean that indicates whether the given key is valid for the given table; the latter returns an error code that describes the general state of a table (in particular, whether it actually exists).

The following sections describe the contents of each table. To understand how NSPrinter parses the PPD file and constructs its tables, you need to be acquainted with the PPD file format. This is described in *PostScript Printer Description File Format Specification, version 4.0*, available from Adobe Systems Incorporated.

You can also see *Using PostScript Printer Description Files*, found on the Printing page of the Core Technologies section of the Mac OS X developer documentation. It contains important information if you want to install and use PPD files in Mac OS X. It details the differences between PPD support in Mac OS X and earlier versions of the Mac OS, provides information about localizing PPD files, shows how to specify that certain features should be grouped in the interface, describes where PPD files need to be installed, and tells what you need to do if you provide a printing dialog extension for your PostScript printer.

Retrieving Values from the PPD Table

A PPD file statement, or entry, associates a value with a main keyword:

```
*mainKeyword: value
```

The asterisk is literal; it indicates the beginning of a new entry. For example:

```
*ModelName: "Apple Laserwriter Pro 810"
*3dDevice: False
```

A main keyword can be qualified by an option keyword:

```
*mainKeyword optionKeyword: value
```

For example:

```
*PaperDensity Letter: "0.1"
*PaperDensity Legal: "0.2"
*PaperDensity A4: "0.3"
*PaperDensity B5: "0.4"
```

In addition, any number of entries may have the same main keyword with no option keyword yet give different values:

```
*InkName: ProcessBlack/Process Black
*InkName: CustomColor/Custom Color
```

```
*InkName: ProcessCyan/Process Cyan
*InkName: ProcessMagenta/Process Magenta
*InkName: ProcessYellow/Process Yellow
```

Keys for the PPD table are strings that name a main keyword or main keyword/option keyword pairing. In both cases, you exclude the main keyword asterisk. The following example invokes `stringForKey:inTable:` to retrieve the value for an un-optioned main keyword:

```
/* Sets sValue to FALSE. */
NSString *sValue = [printer stringForKey:@"3dDevice" inTable:@"PPD"];
```

To retrieve the value for a main keyword/option keyword pair, pass the keyword formatted as "mainKeyword/optionKeyword":

```
/* Sets sValue to "0.3". */
NSString *sValue = [printer stringForKey:@"PaperDensity/A4" inTable:@"PPD"];
```

If you pass a main keyword (only) as the first argument to the method, and if that keyword has options in the PPD file, the method returns an empty string. If it doesn't have options, it returns the value of the first occurrence of the main keyword:

```
/* Sets sValue to an empty string. */
NSString *sValue = [printer stringForKey:@"PaperDensity" inTable:@"PPD"];

/* Sets sValue to "ProcessBlack". */
NSString *sValue = [printer stringForKey:@"InkName" inTable:@"PPD"];
```

To retrieve the values for all occurrences of a main keyword, use the `stringListForKey:inTable:` method giving the main keyword only:

```
/* Sets sList to an array containing "ProcessBlack", "CustomColor", etc. */
NSArray *sList = [printer stringListForKey:@"InkName" inTable:@"PPD"];
```

In addition, `stringListForKey:inTable:` can be used to retrieve all the options for a main keyword (given that the main keyword has options):

```
/* Sets sList to an array containing "Letter", "Legal", "A4", etc. */
NSArray *sList = [printer stringListForKey:@"PaperDensity" inTable:@"PPD"];
```

Retrieving Values from the Option and Argument Translation Tables

Option keywords and values can sport translation strings. A translation string is a textual description, appropriate for display in a user interface, of the option or value. An option or value is separated from its translation string by a slash:

```
*Resolution 300dpi/300 dpi: " ... "
*InkName: ProcessBlack/Process Black
```

In the first example, the `300dpi` option would be presented in a user interface as "300 dpi." In the second example, the translation string for the `ProcessBlack` value is set to "Process Black".

A key for a translation table is similar to a key for the PPD table: It's a main keyword or main/option keyword pair (again excluding the asterisk). However, the values that are returned from the translation tables are the translation strings for the option or argument (value) portions of the PPD file entry. For example:

```
/* Sets sValue to "300 dpi". */
```

```
NSString *sValue = [printer stringForKey:@"Resolution/300dpi"
    inTable:@"PPDOptionTranslation"];
```

As with the PPD table, use `stringListForKey:inTable:` to request an array of all occurrences of a main keyword:

```
/* Sets sList to an array containing "Process Black", "Custom Color", etc. */
NSArray *sList = [printer stringListForKey:@"InkName"
    inTable:@"PPDArgumentTranslation"];
```

Retrieving Values from the Order Dependency Table

NSPrinter treats entries that have an `*OrderDependency` main keyword specially. Such entries take the following forms (the bracketed element is optional):

```
*OrderDependency: real section *mainKeyword [optionKeyword]
```

Below is an example of an `*OrderDependency` entry:

```
*OrderDependency: 10 AnySetup *Resolution
```

These entries are stored in the `PPDOrderDependency` table. To retrieve a value from this table, always use `stringListForKey:inTable:`. The value passed as the key is, again, a main keyword or main keyword/option keyword pair; however, these values correspond to the mainKeyword and optionKeyword parts of an order dependency entry's value. The method returns an NSArray of two NSStrings that correspond to the real and section values for the entry. For example:

```
/* Sets sList to an array containing "10" and "AnySetup". */
NSArray *sList = [printer stringListForKey:@"Resolution"
    inTable:@"PPDOrderDependency"]
```

Retrieving Values from the UIConstraints Table

NSPrinter treats entries that have a `*UIConstraint` main keyword specially. Such entries take the following form (the bracketed elements are optional):

```
*UIConstraint: *mainKeyword1 [optionKeyword1] *mainKeyword2 [optionKeyword2]
```

There may be more than one UIConstraint entry with the same `mainKeyword1` or `mainKeyword1`/`optionKeyword1` value. Below are some examples of `*UIConstraint` entries:

```
*UIConstraint: *Option3 None *PageSize Legal
*UIConstraint: *Option3 None *PageRegion Legal
```

These entries are stored in the `PPDUIConstraints` table. Retrieving a value from the `PPDUIConstraints` table is similar to retrieving a value from the `PPDOrderDependency` table: always use `stringListForKey:inTable:`. The key corresponds to mainKeyword1/optionKeyword1.

The NSArray returned by `stringListForKey:inTable:` contains the mainKeyword2 and optionKeyword2 values (with the keywords stored as separate elements in the NSArray) for every `*UIConstraints` entry that has the given mainKeyword1/optionKeyword1 value. For example:

```
/* Sets sList to an array containing:
```

```
    "PageSize", "Legal", "PageRegion", and "Legal" */
NSArray *sList = [printer stringListForKey:@"Option3/None"
    inTable:@"PPDUIConstraints"]
```

Note that the main keywords that are returned in the NSArray don't have asterisks. Also, the NSArray that's returned always alternates main and option keywords. If a particular main keyword doesn't have an option associated with it, the string for the option will be empty (but the entry in the NSArray for the option will exist).

Document Revision History

This table describes the changes to *Printing Programming Topics for Cocoa*.

Date	Notes
2006-06-28	Fixed code example in "Providing a Custom Pagination Scheme."
2004-01-17	Added section on the proper usage of `drawPageBorderWithSize:` to "Providing a Custom Pagination Scheme" (page 19).
2003-04-21	Corrected error in code sample in "Printing Documents" (page 29).
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.