# Printing Programming Guide for Mac

Developer

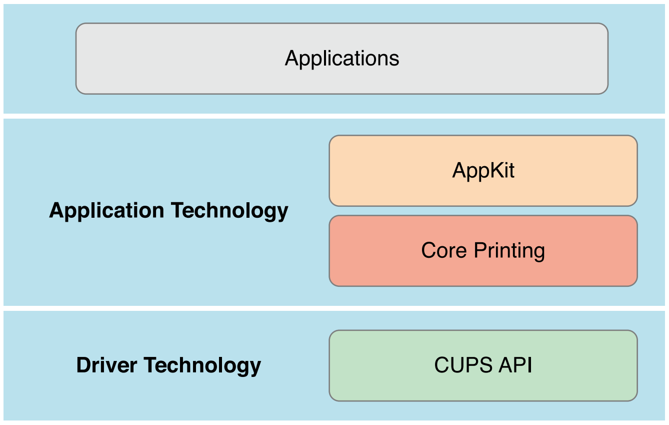
# Contents

# Figures and Listings

SwiftObjective-C

# About Printing on the Mac

Like many other technologies in OS X, printing technology is layered. The top layer is the custom application code that you write to generate the printed output you want. The AppKit layer provides the printing classes that Cocoa apps use to print. That layer is the focus of this book. The Core Printing layer is a C API that most Cocoa app developers will never use directly because it is ideal for writing command-line tools or performing printing tasks that don't require a user interface. The Common UNIX Printing System (CUPS) layer provides the low-level services, print queue management, and driver interfaces needed to communicate with printing devices. As an app developer, you don't need to know anything about CUPS.



**Note:** This document was previously titled *Printing Programming Topics for Cocoa* .

## At a Glance

Most Cocoa apps provide printing support in one form or another. When you create a Cocoa app, the Print command is automatically provided in the File menu. It's straightforward for apps to implement printing.

### Printing is Designed to be Easy-to-Use and Flexible

The printing system does as much as possible automatically for your app. If your app's printing needs are simple, you might need to write only a few lines of code. But if your app needs to print precisely formatted pages, you'll find that OS X printing provides all the flexibility you need.

**Relevant Chapter:** Printing System Workflow and User Interface (page 8)

## NSView and NSDocument Objects Each Support Printing

The printing system API works in conjunction with the `NSView` and `NSDocument` classes. Each class has API that responds to print messages. Your app will either be view-based or document-based. Printing is easy to use in either type of app. The basic concepts are the same with only minor differences in the API available to each.

**Relevant Chapter:** Printing From Your App (page 18)

## Layout Options Let You Format for Paper

Most of the time you'll want to print content other than what shows on the display, if only to add page numbers or margins. You can add borders, crop marks, and other features to the page as well as set up layout options that work best with paper.

**Relevant Chapter:** Laying Out Page Content (page 23)

## If it Needs to, Your App Can Manage the Printing Workflow

Most apps can let the printing system take care of managing Print and Page Setup panels, querying printers, and managing print information. If your app is not a typical app, you can manage various objects in the printing system. For example, your app can add an accessory view to the Print panel to allow users to set app-specific printing features.

**Relevant Chapters:** Managing and Extending the Print Panel (page 27), Managing Print Information Objects (page 31), and Managing Page Layout Objects (page 33)

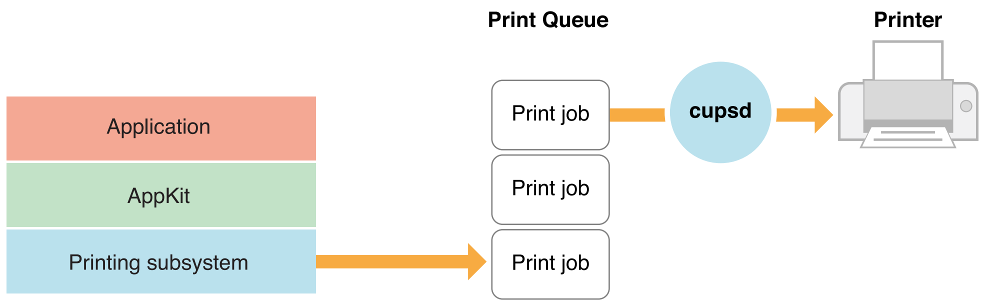
# Printing System Workflow and User Interface

Gone are the days when users had to download and install printer drivers and OS X had to package and distribute large packages of them. Printing on the Mac is not only easy for the user, but the modern API provided by OS X is easy for you to use in your app. Printing a single page from an app can take just a few lines of code. The code for controlling pagination, margins, headers, and footers for multipage documents with precise layout needs more code, but it is straightforward to write.

## The Printing System Workflow

Printing is generally initiated by the user choosing the Print menu command. An app uses the AppKit printing API to assemble the elements of a print job, including the content to print and information about the print job. The app then presents the Print panel as described in The Printing User Interface (page 9). The user can then set printing options before clicking Print. At that point, the AppKit framework asks the app to draw the content to print. AppKit records what the app draws as PDF data and then hands off that data to the printing subsystem.

**Figure 1-1** The printing system architecture



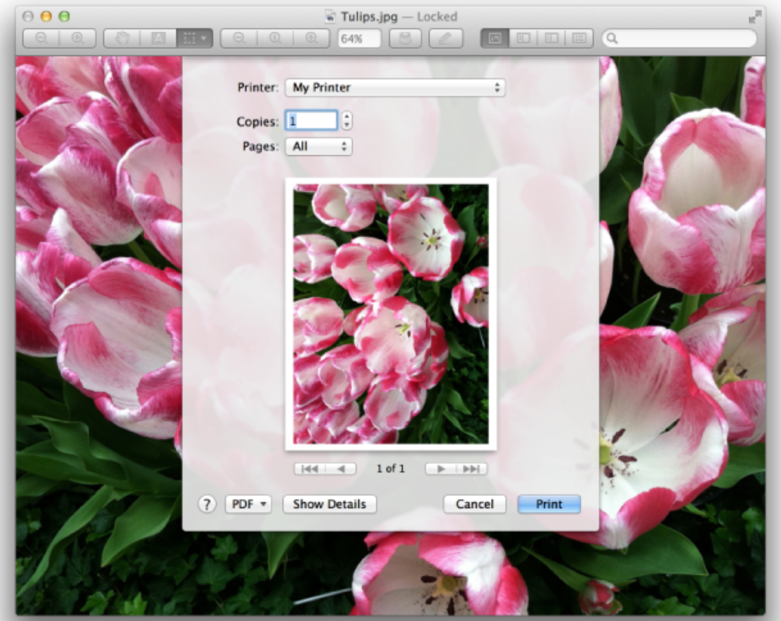
The printing subsystem writes the PDF data it receives to storage (that is, spools the data). It also captures information about the print job. The printing subsystem manages the combined print data and metadata for each print job in a first-in-first-out print queue. Multiple apps on a computer can submit multiple print jobs to the printing subsystem, and all of these are placed in the print queue. Each computer has one queue for all print jobs regardless of the originating app or destination printer.

When a print job rises to the top of the queue, the system printing daemon (`cupsd`) considers the destination printer's requirements and, if necessary, converts the print data to a form that is usable by the printer. The printing subsystem reports error conditions such as "Out of Paper" to the user as alerts. It also reports the progress of print jobs programmatically to the PrintProxy app, which displays information such as "page 2 of 5" for a print job. (The PrintProxy app takes on the name of the printer. It's the app the displays the queued print jobs for that printer and allows the user to pause, resume, and cancel jobs that are in the queue.)

## The Printing User Interface

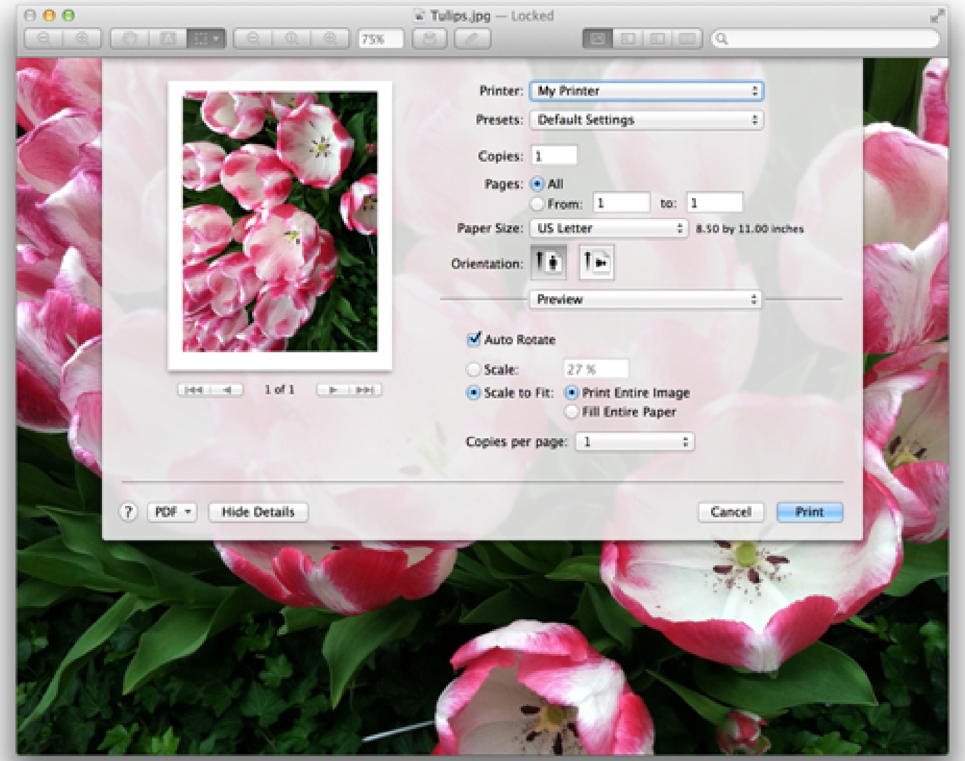
When the user chooses Print from the File menu, they are presented with a Print panel as shown in Figure 1-2. The Printer pop-up menu is populated with the last printer used. The user can page through a preview of a document that has multiple pages and can choose to print all pages or set a range. For most users, the simple Print panel is all they ever need. They might never see any of the other user interface elements described in this section. But for those users who require more control over the printing process and the printers they use, OS X provides it.

**Figure 1-2** The Print panel as a sheet in Preview



The user can click the Show Details button to control a variety of printing options including layout, color matching, and paper handling. Many options are dependent on the features available for the specified printer, such as the availability of duplex printing. An app can supply its own options, as shown in Figure 1-3. In this example, Preview supports rotation, scaling, copies per page, and how to fill the page with the image.

**Figure 1-3** The Print panel with Preview's custom accessory view

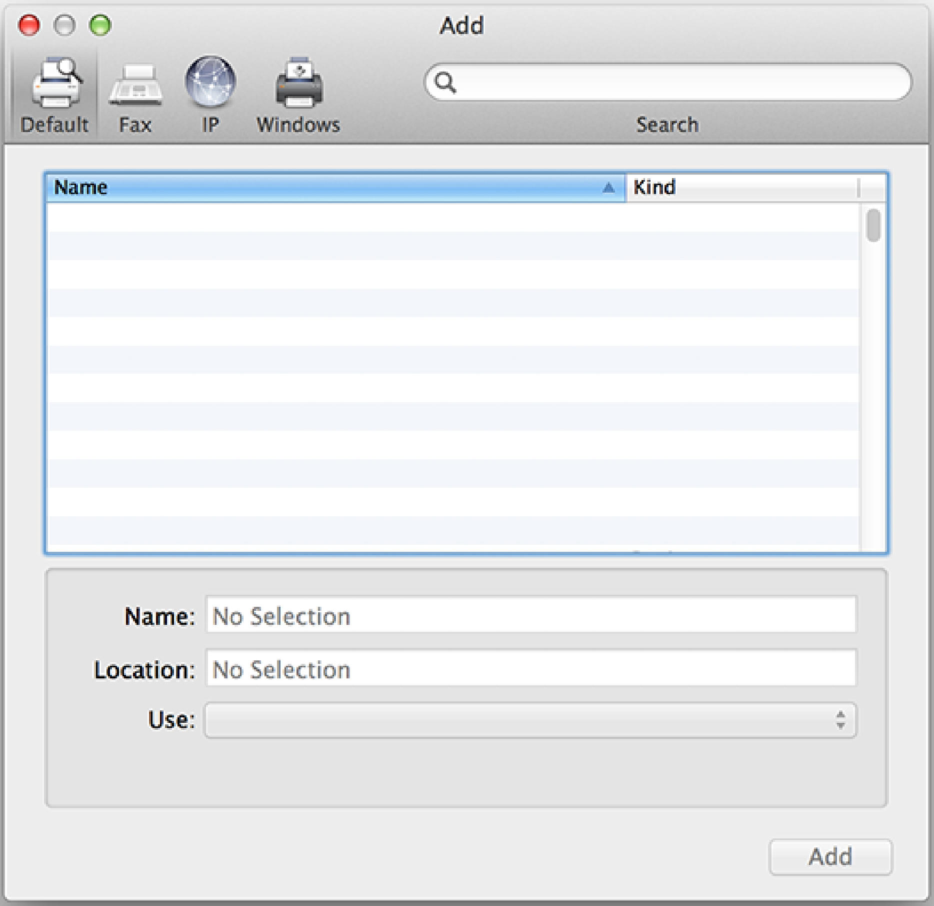


Printer: My Printer
Presets: Default Settings
Copies: 1
Pages: [✓ All]
From: 1 to: 1
Paper Size: US Letter 8.50 by 11.00 inches
Orientation:
Preview
[✓ Auto Rotate]
Scale: 27 %
[✓ Scale to Fit:] [✓ Print Entire Image] Fill Entire Paper
Copies per page: 1

Apps can choose whether to display the Print panel as a sheet or a a separate window, although it's most common to show a sheet so the user can easily see the window to which the panel applies.

Choosing a printer for printing is easy for the user, which they do through the Printer pop-up menu in the Print panel. The user can choose from a list of nearby printers and OS X sets up the printer automatically. A user who needs more control on adding printing devices can choose Add Printer to access the Add window as shown in Figure 1-4. This window is automatically populated by the devices that OS X can detect, including AirPrint printers. Users have the option to add specific host name or IP addresses as well as to add Fax and Windows workgroup printers.

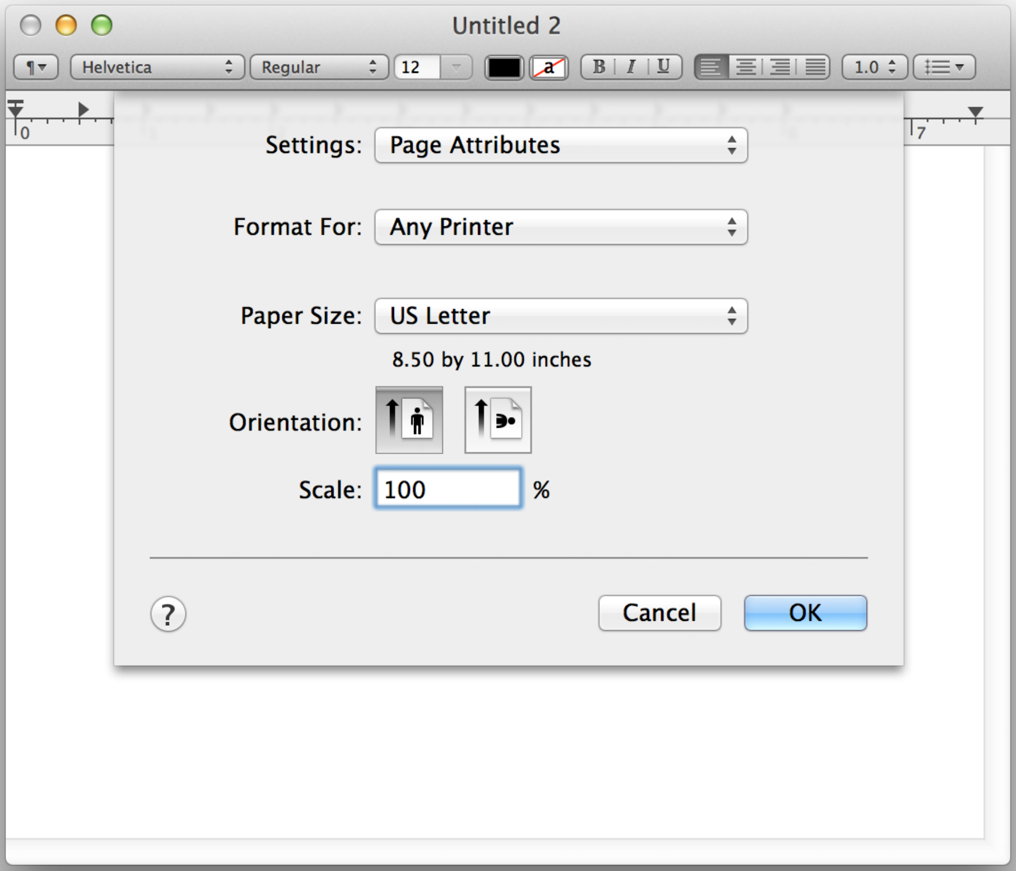
**Figure 1-4** The Add Printer window



Name: No Selection
Location: No Selection
Use:

Some apps also provide users with the option to setup the page through the Page Setup pane as shown in Figure 1-5. Most of the time users shouldn't need to perform any setup through this panel. So if your app doesn't need it, make the user experience simpler by not providing a Page Setup command.

**Figure 1-5** The Page Setup panel



Settings: Page Attributes

Format For: Any Printer

Paper Size: US Letter

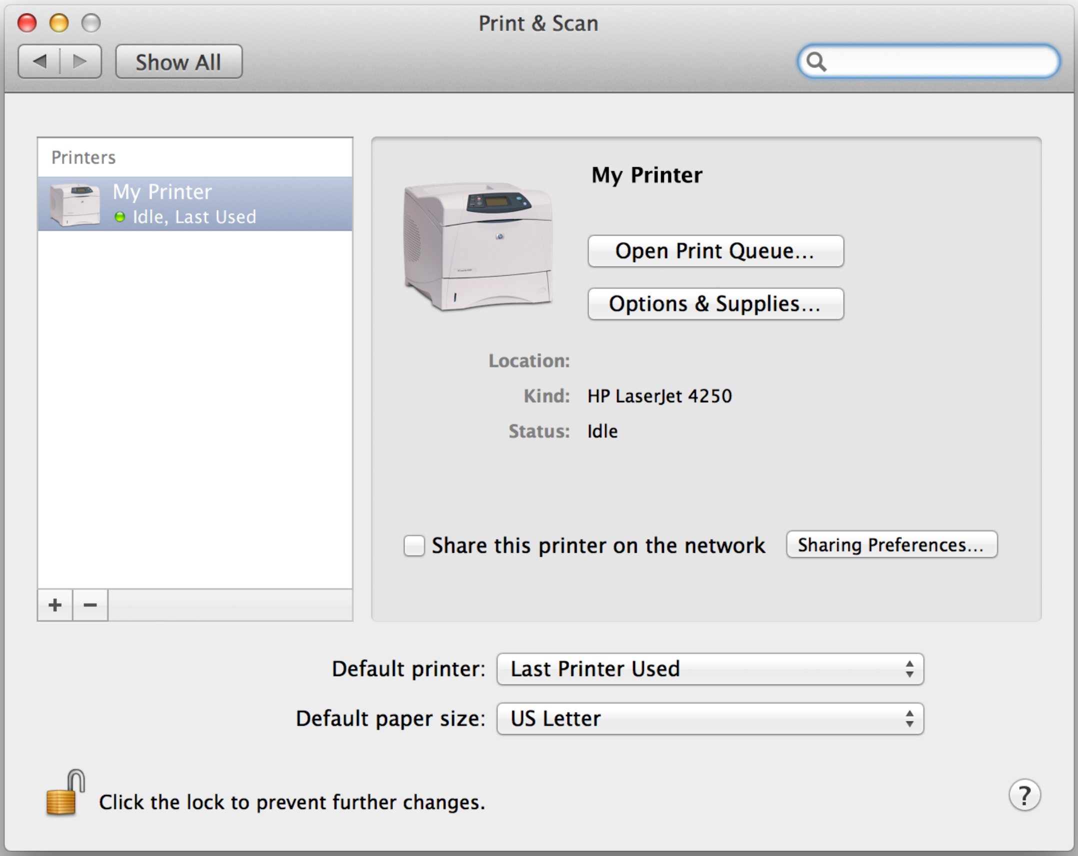
8.50 by 11.00 inches

Orientation:

Scale: 100 %

Users can manage printers through the Print & Scan preference pane in System Preferences, as shown in Figure 1-6.

**Figure 1-6** The Print & Scan preferences pane



Location:
Kind: HP LaserJet 4250
Status: Idle

☐ Share this printer on the network

Default printer: Last Printer Used

Default paper size: US Letter

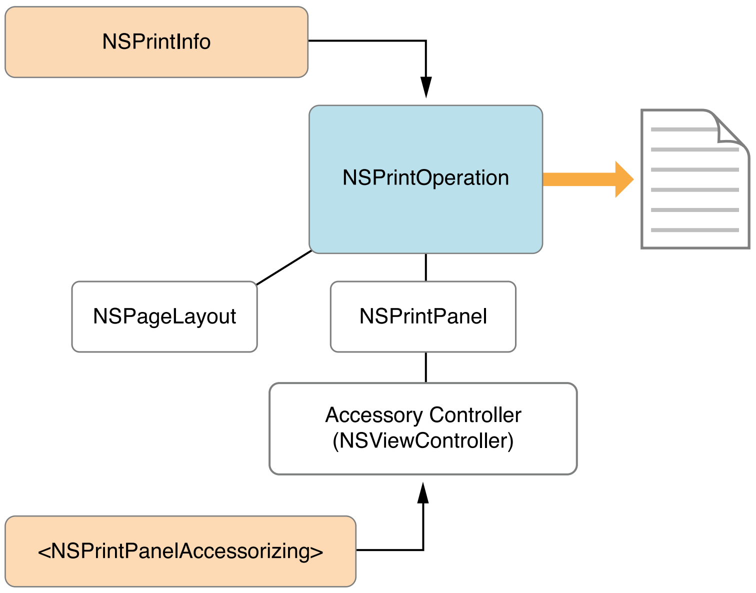
Click the lock to prevent further changes.

OS X automatically adds recently used printers to the list but a user can add others as well as delete any in the list. Opening a print queue provides the user with the option to pause a print job and to access printer settings. By clicking the Options & Supplies button, the user can set the printer name, manage driver options, check supply levels, and open a printer's utility app.

# The AppKit Printing API

The AppKit framework publishes the programmatic interface that supports printing in your app. The API includes five classes and one formal protocol. Objects of these classes and the delegate implementing the protocol have the runtime relationships shown in Figure 2-1.

**Figure 2-1** The classes and protocol in the AppKit printing API


NSPrintInfo
NSPrintOperation
NSPageLayout
NSPrintPanel
Accessory Controller (NSViewController)
<NSPrintPanelAccessorizing>

These classes are in a layer above Core Printing, which is a C API used to create command-line tools or to perform printing tasks that don't display a user interface. The `NSPrintInfo` class provides direct access to Core Printing functionality. In Cocoa apps, Core Printing can be used to extend the functionality of the AppKit printing classes. However most apps shouldn't need to use the Core Printing API, so it is not discussed further in this document. If you want to find out more about Core Printing, see the sample code project *Cocoa Printing using Core Printing* and the technical note *Using Cocoa and Core Printing Together*.

## Overview of the Printing Classes and Protocol

Objects of the AppKit printing classes have specific roles and responsibilities.

## NSPrintOperation Manages a Print Job

An `NSPrintOperation` object is central to printing; without it, your app can't print. 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, and hands off the resulting content to the CUPS layer of the system. It can also generate Portable Document Format (PDF) data instead of sending the results to a printer.

`NSPrintOperation` works together 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 must specify a view when you create an `NSPrintOperation` object. You can optionally specify an `NSPrintInfo` object.

## NSPrintInfo Stores Options That Control How a Print Job is Performed

Print information includes the paper size, number of copies, print margins, whether to use a header and footer, and so on. The printing system automatically creates a shared `NSPrintInfo` object that holds defaults settings used by other objects of the printing system.

Normally you don't set `NSPrintInfo` attributes directly—this is done by instances of `NSPageLayout` and `NSPrintPanel`. The `NSView` that generates the printing content might also supersede some `NSPrintInfo` settings, such as the pagination and orientation attributes.

Your app should not create an `NSPrintInfo` object unless it needs to modify the default settings or save and restore custom settings. See Managing Print Information Objects (page 31).

## NSPrintPanel Creates and Displays the Print Panel

This class manages the standard system Print panel. Your app does not need to create an `NSPrintPanel` object unless you want to manage the printing workflow yourself or add custom print settings for your app (using an accessory view). If you create an instance of `NSPrintPanel` you need to display it and subsequently initiate the desired printing behavior.

If you add an accessory view to the Print panel to display app-specific options, you must adopt the `NSPrintPanelAccessorizing` protocol. See Managing and Extending the Print Panel (page 27).

## The NSPrintPanelAccessorizing Protocol Manages a Custom Accessory View

The `NSPrintPanelAccessorizing` protocol declares two methods that the `NSPrintPanel` class uses to get information from a printing accessory controller. You are required to implement the `localizedSummaryItems` method, which returns an array of dictionaries that contain the localized summary strings for the setting in your accessory view. It is optional for you to implement `keyPathsForValuesAffectingPreview`.

See Managing and Extending the Print Panel (page 27).

### NSPageLayout Displays the Page Setup Panel

This class manages the standard system Page Setup panel. Your app does not need to create an `NSPageLayout` object unless you want to manage the printing workflow yourself. If your app really needs to mange the Page Setup panel, it must display the Page Setup panel and subsequently initiate the desired printing behavior.

It is not typical for apps to create `NSPageLayout` objects. See Managing Page Layout Objects (page 33).

### NSView Draws the Content Your App Prints

As with screen-based drawing, the `NSView` class provides the underlying canvas for drawing printed content. If you have an app that already uses views to draw in your app's windows—which most Cocoa apps do—then you already have the basic code you need to draw printed content. By default, the printing workflows handle printing by taking the same views embedded in your windows and simply redirecting the output to a different destination.

In addition to drawing your custom content, the `NSView` class has methods for:

- Drawing header and footer content
- Paginating content
- Specifying alignment marks or virtual sheet borders on each logical page
- Specifying drawing crop marks or fold lines on each printed sheet

For more information on drawing content to `NSView` objects, see *Cocoa Drawing Guide* .

If your app supports printing text, you also need to be familiar with using the Cocoa Text System (see *Cocoa Text Architecture Guide* ). If you want to control text layout on the page, you need to use the `NSLayoutManager` class (see *Text Layout Programming Guide* ).

## Basic Printing Workflow

In a Cocoa app, printing is generally initiated by the user choosing the Print menu command, which usually sends either a `print:` or `printDocument:` message up the responder chain. Which message is sent depends on whether or not the app is document-based. The app receives the message either in a custom `NSView` object (if it has the keyboard focus), a window delegate, or an `NSDocument` object.

After receiving the message to print, the general workflow is as follows:

1. Create an `NSPrintOperation` object to manage the print job, providing the view that contains the content to print.
2. (Optional) Add an accessory view to the job's print panel.
3. Run the print operation.
4. (Optional) For a multipage job, override how the view is divided between multiple pages by using the methods of the `NSView` class.

The view's `drawRect:` method draws the view's contents.

Implementing printing in your app can be as easy as writing these few lines of code:

```
- (IBAction)print:(id)sender {
      NSPrintOperation *op;
      op = [NSPrintOperation printOperationWithView:self];
      if (op)
           [op runOperation];
      else
          // handle error here
}
```

OS X printing also provides support for custom formatting and layout. When you add those tasks, the workflow is a bit more complex, but straightforward. See Printing From Your App (page 18) and Laying Out Page Content (page 23).

# Printing From Your App

SwiftObjective-C

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) data for your app. An `NSPrintOperation` object controls the overall process, relying on an `NSView` object to generate the actual code. Once created, a print operation can be configured in several ways.

In a Cocoa App, there are two classes that provide infrastructure to handle printing:

- `NSView`
- `NSDocument`

The methods your app uses to handle printing depend on which class it uses for its content.

## Printing in an App That Uses NSView

Printing in apps that are not document-based works best for apps that have only one printable view (that is, an `NSView` object) in its main window that can be the first responder. For example, in a simple text editor, only the view holding the text document can have focus, so it is straightforward to implement printing in the text view. You can see an example of this architecture in the *TextEdit* sample code project.

When your user interface contains multiple views that can have focus, such as multiple `NSTextField` objects, view-based printing doesn't work well. When the user chooses the Print command, the view that receives the `print:` message prints itself, but nothing else. If the focus is in a text field, for example, only the contents of that text field are printed. This probably is not the desired behavior. Instead, your app needs to take a more document-based approach. See Printing in an App That Uses NSDocument (page 19).

In a view-based app, your app receives a `print:` message when the user chooses Print from the File menu. In your implementation of the `print:` method, you create an `NSPrintOperation` object initialized with the view to print and, optionally, the `NSPrintInfo` object holding the print settings.

The print operation is not started, until you invoke one of the `runOperation` methods of `NSPrintOperation` as shown in Listing 3-1.

**Listing 3-1** A simple implementation of the print: method for a view-based app

```
- (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 Print panel (which is an `NSPrintPanel` object created automatically by the printing system) to obtain the print settings from the user. The app's shared `NSPrintInfo` object is used for the initial settings.

## Printing in an App That Uses NSDocument

An app that uses the `NSDocument` class 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.

In a document-based app, when the user chooses Print from the File menu, the printing system sends a `printDocument:` message, which only the `NSDocument` class implements. The `NSDocument` object associated with the app's main window receives the message and invokes the method `printDocumentWithSettings:showPrintPanel:delegate:didPrintSelector:contextInfo:` with `YES` as the argument for `showPrintPanel`.

If you specify to show the Print panel, the method presents it and prints only if the user approves the panel. The method adds the `NSPrintInfo` attributes, from the `printSettings` dictionary you pass, to a copy of the document's print info, and the resulting print info settings are used for the operation. When printing is complete or canceled, the method sends the message selected by `didPrintSelector` to the delegate, with the `contextInfo` as the last argument. The method selected by `didPrintSelector` must have the same signature as:

```
- (void)document:(NSDocument *)document didPrint:(BOOL)didPrintSuccessfully
contextInfo: (void *)contextInfo;
```

The default implementation of `printDocumentWithSettings:showPrintPanel:delegate:didPrintSelector:contextInfo:` invokes `printOperationWithSettings:error:`. You must override this method to enable printing in your app; the default implementation raises an exception.

If the `printDocumentWithSettings:showPrintPanel:delegate:didPrintSelector:contextInfo:` returns `nil`, it presents the error to the user before messaging the delegate. Otherwise it invokes this line of code:

```
[thePrintOperation setShowsPrintPanel:showPrintPanel];
```

followed by this code:

```
[self runModalPrintOperation:thePrintOperation
                    delegate:delegate
              didRunSelector:didPrintSelector
                 contextInfo:contextInfo];
```

## Customizing Content for the Printed Page, Not the Display

In some cases it might be preferable for your app to send content to the printer that is not identical to that drawn onscreen. For example, the main window of a database app might 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

You have two options for customizing content for the printed page:

1. In your app's drawRect: method, branch the code 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
    }
    ```

> **Note:** If you are using layer-backed views, the system still calls your view's `drawRect:` method, thus allowing your printing branch to execute.

2. Create a view that's used only for printing.

When printing to a view object that your app uses only for printing you need to:

- Create a view that will be used only for printing, making sure you size it appropriately.
- Create a print operation using the print view.

```
NSPrintOperation *printOp = [NSPrintOperation
printOperationWithView:myPrintingView

                          printInfo:[self printInfo]];
```

- Ideally, also specify that the print operation can run in a separate thread. This causes the print progress panel to appear as a sheet on the document window.

```
[printOp setCanSpawnSeparateThread:YES];
```

You might 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];
```

## Generating PDF Data

A print operation does not have to send its results to a printer. You can have the operation generate PDF 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 the `PDFOperation` class method to create the `NSPrintOperation` object instead of one of the `printOperation` methods. You can identify whether a print operation is generating PDF 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.

The `NSView` class provides several convenience methods for generating PDF data. The data can be returned in an `NSData` object or written to a pasteboard. The `NSView` class implements `dataWithPDFInsideRect:` and `writePDFInsideRect:toPasteboard:`.

These methods create and run an `NSPrintOperation` object, just as the `print:` method 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.

## Printing on Another Thread

By default, the print operation performs the data generation on the current thread. This thread is normally the app'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 app to continue processing events, instead of blocking. (Print operations that create PDF data always run on the current thread.)

# Laying Out Page Content

SwiftObjective-C

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 for Content That Exceed a Single Page

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 the built-in pagination support of the `NSView` class, a view needs to implement only two simple methods and set

1. Set up the pagination mode (`NSPrintingPaginationMode`) using the the appropriate method of `NSPrintInfo` (`setHorizontalPagination:` or `setVerticalPagination:`
2. Override the `knowsPageRange:` method so it returns `YES` to indicate the custom view will collocate the dimension of each page.
3. Implement the `rectForPage:` method so it uses the page page number and the current printing information to calculate an appropriate rectangle in the view's coordinate system. The printing system sends a `rectForPage:` message to your app before each page is printed, base on the range of pages the user selects in the Print panel. Note that the vertical and horizontal pagination settings in the `NSPrintInfo` object are ignored (unless your implementation takes them into account).

Listing 4-1 shows a simple implementation that splits a view vertically into pages that have the maximum size. The code does not show setting the pagination mode, which you must do.

**Listing 4-1** Code that splits the view vertically into pages

```
// 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;
}
```

## Adding Page Numbers, Crop Marks, and Date-Time Strings to the Page

When you perform custom pagination, you can override the `drawPageBorderWithSize:` method to add extra features to the page, such as crop marks, date/time strings, or page numbers. When you override `drawPageBorderWithSize:`:

1. Save the view's existing body frame—you will need to restore it at the end of the method.
2. 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.
3. Add your custom border elements to all four margin areas (top, bottom, left, and right).

   You typically use the `drawAtPoint:` method for drawing. 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 want 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.
4. 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.

## Tiling Content 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. The `adjustPageHeightNew:top:bottom:limit:` method 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. The `adjustPageWidthNew:left:right:limit:` method 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` 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 Content 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 Content 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 Content on the Logical Page

The `NSView` method `locationOfPrintRect:` places content 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 point returned by `locationOfPrintRect:` is relative to the bottom-left corner of the paper in page coordinates. You need to include the page margins when calculating the position.

After the `NSView` position the rectangle on the page, it invokes `drawPageBorderWithSize:`. If you haven't implemented this method, nothing happens. If you have implemented `drawPageBorderWithSize:`, any extra marks—crop marks, page numbers, and so on—you draw in this method are added to the page. The `drawPageBorderWithSize:` method is invoked by the printing system once for each page.

See Custom Pagination (page 23) for more information on using `drawPageBorderWithSize:`.

# Managing and Extending the Print Panel

Objective-CSwift
An `NSPrintPanel` object creates and displays a Print panel that allows the user to modify the current print settings, such as a page range and the number copies to print. Running a print operation causes the Print panel to appear. You can prevent the panel from show by supplying `No` to the `setShowsPrintPanel:` method. (Print operations that create PDF or EPS data never display the Print panel.)

## Suppressing the Print Panel

By default, an `NSPrintOperation` object object displays a Print panel allowing the user to select printing options, such as number of copies to print and range of pages to print. After the user chooses options for the first time, you might want to offer the user the ability to bypass the Print panel and print immediately using the previous print settings.

You can suppress the display of the Print panel by sending `setShowsPrintPanel:` with a `NO` argument to the `NSPrintOperation` object before running the operation.

However, make sure that any non-default settings in the `NSPrintPanel` object that would normally be selected from an `NSPrintPanel` object are set to reasonable values—a copy of an `NSPrintPanel` object used in a previous print job will have the correct values, as shown in Listing 5-1.

**Listing 5-1** Ensuring the print panel object has reasonable values

```
// 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 setShowsPrintPanel:NO];
    [op runOperation];
}
```

## Modifying the Print Panel

By default, an `NSPrintOperation` object displays a standard Print panel. If you need to add some app-specific options, you can add an accessory view. You have the option of loading the accessory view from a nib file or creating it programmatically.

To add a custom accessory view to the standard Print panel, follow these steps:

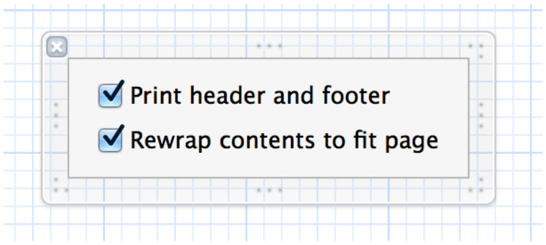
1. Use Interface Builder in Xcode to create an accessory view, placing controls that correspond to your app-specific print settings. (For example, see Figure 5-1 (page 29).)

   The accessory view displays when the user chooses your app's name in the pane-selection pop-up menu in the Print panel. The panel automatically resizes in the vertical direction to accommodate the view you add. If your view is too wide, the content will be clipped. If possible, you should make your accessory view the same size as the standard views in the panel

2. Subclass `NSViewController` to create a print panel accessory view controller.
3. Implement the required protocol method `localizedSummaryItems`.
4. (Optional) Implement `keyPathsForValuesAffectingPreview`.
5. Get the print panel associated with the print operation.
6. Call the `addAccessoryController:` method to add the accessory view to the print panel.

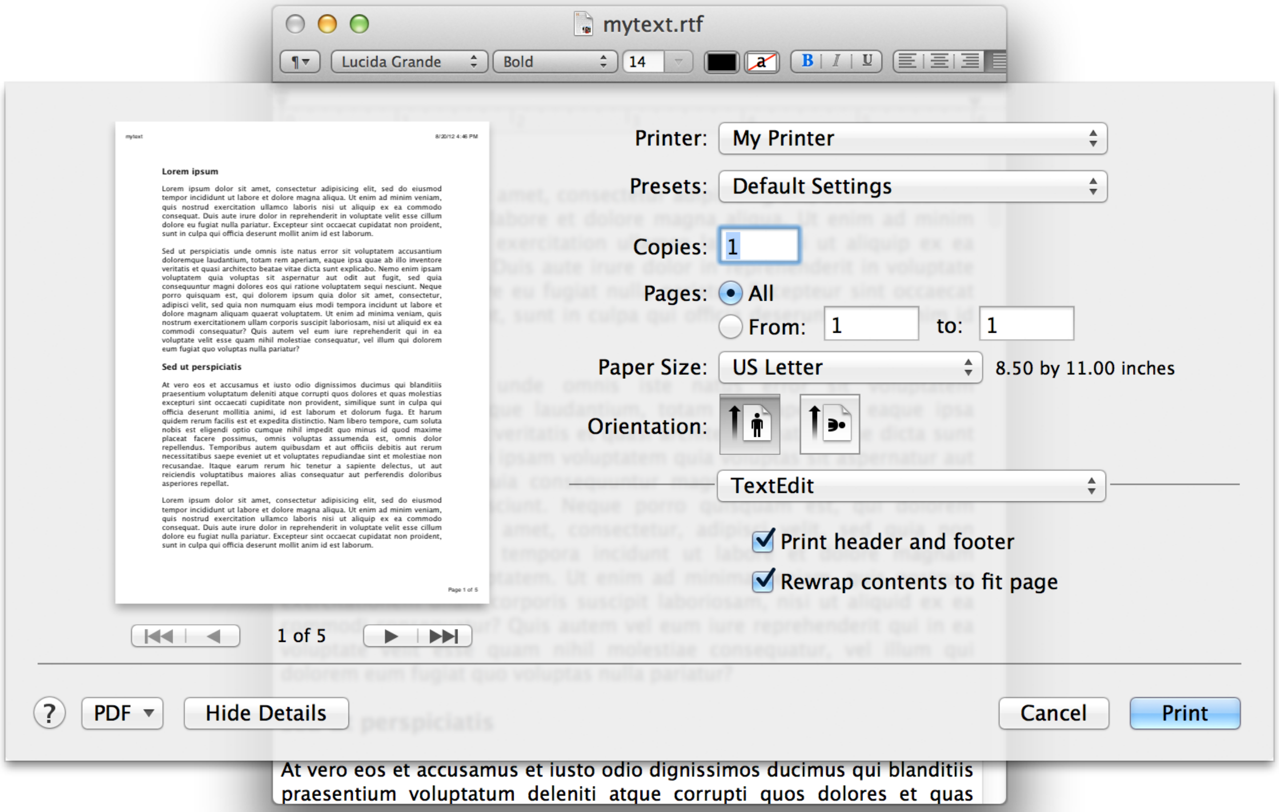
7. Run the print operation to trigger showing the Print panel

**Figure 5-1** An accessory view created in Interface Builder



When you run a print operation, your accessory view is available for viewing in the Print panel. Figure 5-2 (page 29) shows the accessory view for the TextEdit sample code project. Figure 5-1 (page 29) shows the view as it appears in Interface Builder.

**Figure 5-2** The TextEdit accessory view



Printer: My Printer
Presets: Default Settings
Copies: 1
Pages: ● All
From: 1 to: 1
Paper Size: US Letter 8.50 by 11.00 inches
Orientation:
TextEdit
✓ Print header and footer
✓ Rewrap contents to fit page

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];
}
```

# Managing Print Information Objects

An `NSPrintInfo` object contains a dictionary that stores the attributes that describe a print job. The dictionary keys are described in the "Constants" section of `NSPrintInfo`.

## Setting a Shared Print Info Object

You can set your instance of `NSPrintInfo` as the shared instance using the method `setSharedPrintInfo:`. You get the shared `NSPrintInfo` object using the `sharedPrintInfo` class method.

## Saving Print Settings for an App

To reuse the print settings used the last time your app ran, record the print info object as an app preference each time the user prints something and then restore those settings when the app 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.

## Saving Print Settings for a Document

In a document-based app, each `NSDocument` instance has its own print info object, which you can obtain by calling the `printInfo` method of `NSDocument`. The document initially uses a copy of the app's shared print info object (unless you set one yourself). When the user makes changes in the Page Setup panel, the document's print info object is automatically updated with the new print settings.

Because print settings are often document specific, you might want to save them. 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.

# Managing Page Layout Objects

Objective-CSwift

An `NSPageLayout` object creates a panel that queries the user for information such as paper size and orientation. This information is stored in an `NSPrintInfo` object which is used when printing.

When the user chooses the Page Setup menu command from the File menu, the `runPageLayout:` the system sends the message up the responder chain and creates an `NSPageLayout` object. Either an `NSApplication` or an `NSDocument` object receives the message. Upon receipt, the Page Layout user interface is displayed to the user. An `NSApplication` object displays the page layout user interface as a modal panel, whereas an `NSDocument` object displays the page layout user interface as a sheet. The message handling and display is done automatically by the system with no code needed by your app.

## Handling the the Page Setup Panel

If the default implementation is not sufficient you can handle the Page Setup panel yourself. You create an `NSPageLayout` object by invoking the `pageLayout` method. To display the Page Setup user interface app-modally, use the `runModal` or `runModalWithPrintInfo:` methods. To display the Page Setup user interface document-modally as a sheet, use the `beginSheetWithPrintInfo:modalForWindow:delegate:didEndSelector:contextInfo:` method.

You rarely need to subclass `NSPageLayout` because you can augment its display by adding your own accessory view using the `addAccessoryController:` method. Place controls for setting app-specific print settings in your accessory view.

The accessory view is displayed when the user chooses the appropriate entry—which is the name of your app—in the Settings pop-up menu in the Page Setup panel. The panel automatically resizes in the vertical direction to accommodate the height of the view you add. You must make sure the width of your accessory view fits with the maximum width of the Page Setup panel. If possible, you should make your accessory view the same size as the standard views in the panel.

When running an app that is not document based, you must override the `runPageLayout:` method of the `NSApplication` class. You can also implement the method earlier in the responder chain. If you want to add an accessory view, your `runPageLayout:` method needs to call the `addAccessoryController:` method.

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

## Customizing a Page Setup Panel in a Document-based App

The `NSDocument` class implements the `runPageLayout:` method to handle the Page Setup menu command instead of letting `NSApplication` handle it. When the document receives this message, it gets the document's `NSPrintInfo` object and invokes the method `runModalPageLayoutWithPrintInfo:delegate:didRunSelector:contextInfo:` to display the Page Setup panel. To give your `NSDocument` subclass an opportunity to customize the `NSPageLayout` object, the printing system passes the `NSPageSetup` 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 printing system updates the `NSPrintInfo` object 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.

# Document Revision History

This table describes the changes to *Printing Programming Guide for Mac* .

| Date | Notes |
|---|---|
| 2012-12-20 | Added a note about layer-backed views, made minor editorial corrections. Changed the title (OS X to Mac), fixed typos, updated Managing and Extending the Print Panel (page 27), and added a note to Customizing Content for the Printed Page, Not the Display (page 20). |
| 2012-09-19 | Updated for OS X v10.8. Formerly titled Printing Programming Topics for Cocoa. Reorganized the content and revised to incorporate current best practices. Replaced references to deprecated methods with current methods. |
| 2004-01-17 | Added section on the proper usage of `drawPageBorderWithSize:`. |
| 2003-04-21 | Corrected error in code sample. |
| 2002-11-12 | Revision history was added to existing topic. It will be used to record changes to the content of the topic. |