# PDF Kit Programming Guide

2005-11-09

EFTA00612172

# Contents

# Tables, Figures, and Listings

# Introduction to PDF Kit Programming Guide

PDF Kit is a technology that allows you to display and manipulate PDF documents in your applications. By implementing aspects of Adobe's PDF specification for you, PDF Kit minimizes development time on your part. Even Apple's own applications such as Safari and Preview use PDF Kit to display PDF content.

PDF Kit is available in Mac OS X v10.4 and later.

## Who Should Read This Document

The audience for this document is any Cocoa developer who wants to display, edit, or otherwise manipulate PDF documents in applications. This document assumes you have some familiarity with Cocoa and Objective-C programming.

PDF Kit is extremely flexible, allowing you as much or as little control over PDF documents as you like. Whether you simply want to display a PDF document in a help window, or if you want to create a full-featured PDF editor, PDF Kit can simpify your task.

## Organization of This Document

This document is organized into the following chapters:

- "PDF Kit Concepts" (page 9) gives a brief overview of PDF features and concepts and outlines the various PDF Kit classes.
- "PDF Kit Tasks" (page 17) shows how to implement common features using PDF Kit.

## See Also

Adobe Corporation's PDF specification is the main resource for describing PDF documents. You can download the latest version (in PDF form, naturally) from the following website:

http://partners.adobe.com/public/developer/pdf/index_reference.html

# PDF Kit Concepts

This chapter gives an overview of PDF concepts and PDF Kit classes. If you are already familiar with the elements of a PDF document, you can skip "PDF Basics" (page 9) and go directly to "PDF Kit Classes" (page 14).

## PDF Basics

A PDF is a document stored using Adobe Corporation's Portable Document Format. The PDF specification, based on the PostScript drawing language, can describe almost any combination of text and images as well as interactive elements.

The basic building block of a PDF is the document itself. Within the document, you can have various pages and an outline. Within a page you can have text, annotations, and so on.

For detailed information about the PDF format, see the PDF specification, which you can download from:

http://partners.adobe.com/public/developer/pdf/index_reference.html

Note that if you simply want to display PDF documents in your application, you generally don't need the level of detail that the PDF specification provides.

### Documents

The fundamental building block for a PDF is the document itself. The document is typically stored on disk as a file.

Documents support versioning and can be tagged with metadata such as the author, creation date, and so on.

A document can be encrypted, requiring a password to view it. Two levels of encryption exist:

- User level encryption: If the user successfully obtains user-level permissions, he or she can view the document but may be restricted from printing or copying the document.
- Owner-level encryption: A user who obtains owner-level permissions can view the document and has full usage permissions.

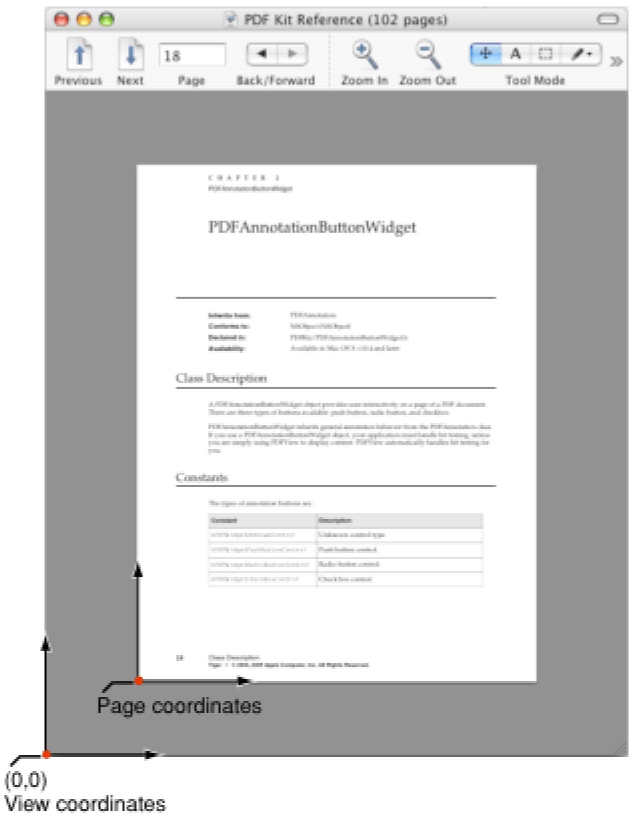
Many encrypted PDF documents have a "dummy" user password which is the empty string. Most PDF document parsers (including PDF Kit) automatically try the empty string password on encrypted documents, and if successful, simply display the document. Therefore, a document that is technically encrypted may not necessarily prompt the user for a password.

## Pages

A PDF document consists of a number of pages. These are the metaphorical equivalent of pages in a physical book, and they are what the user sees onscreen. However, unlike a physical page, PDF pages can contain hyperlinks and annotations. Pages can support cropping as well, which can be useful if you want to hide extraneous portions (such as registration marks) during display.

Note that most objects on a page are specified in page space, rather than view space. That is, the coordinate system is in points (72 points per inch), with the origin at the bottom left of the page, not the view. Page space doesn't care about zooming, display mode, and so on. An item that has bounds of, say 32 points square, retains those bounds regardless of display size. Figure 1-1 (page 10) compares the two coordinate systems:

**Figure 1-1** View space versus page space



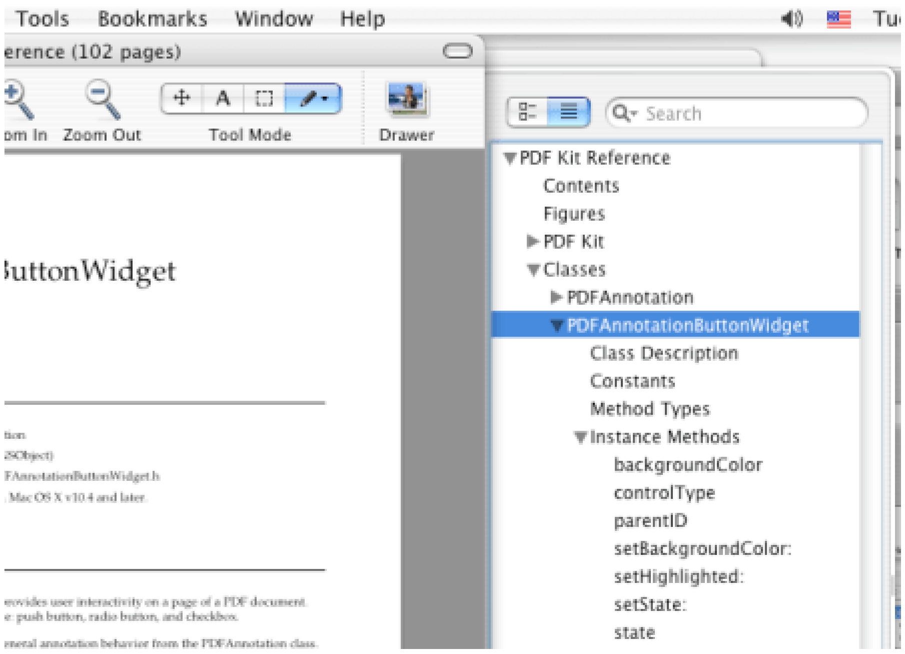
The PDFView class contains a number of conversion methods to translate coordinates from view space to page space and vice versa.

EFTA00612181

## Outlines

An outline is like an interactive table of contents, showing the chapter or structure hierarchy of the document. Outlines make it easy for users to see the structure of a document and to jump to a particular location.

**Figure 1-2** An outline for a PDF document



Not all PDF documents contain an outline.

## Annotations

Annotations are "extra" elements that can appear on a PDF page in addition to the standard text and images. Some annotations merely add visual features, such as lines, circles, and such, while others can have some interactive behavior.

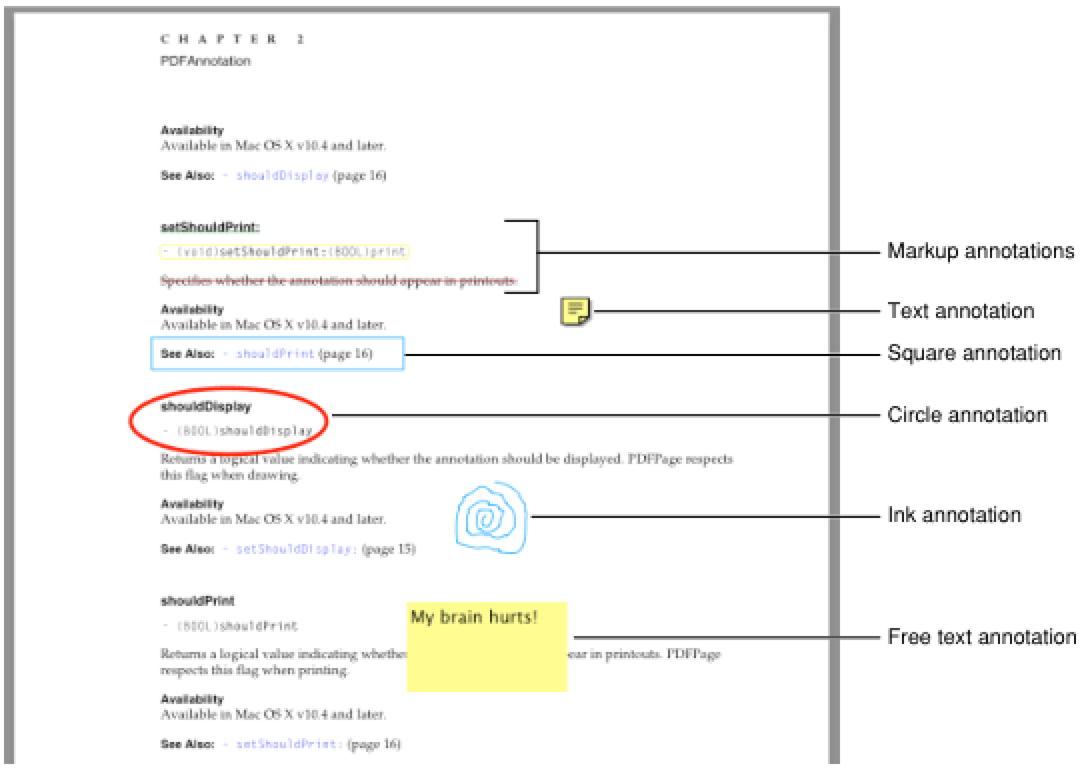
Some examples of annotations include:

- "Sticky notes" displaying text.
- Note icons that can display text when clicked upon.
- Editable text fields that can accept user text.
- Buttons, such as checkboxes. Such annotations, along with editable text fields, may be useful in forms to be filled out by the user.
- Circles, arbitrary lines, and boxes.
- Links to other documents, or to other sections within a document.
- Highlighting, strike-throughs, and other text markups.

EFTA00612182

Figure 1-3 (page 12) shows some annotation types available in PDF Kit.

**Figure 1-3** Some annotations available in PDF Kit



CHAPTER 2
PDFAnnotation

**Availability**
Available in Mac OS X v10.4 and later.

**See Also:** - shouldDisplay (page 16)

**setShouldPrint:**

- (void)setShouldPrint:(BOOL)print

~~Specifies whether the annotation should appear in printouts.~~

**Availability**
Available in Mac OS X v10.4 and later.

**See Also:** - shouldPrint (page 16)

**shouldDisplay**

- (BOOL)shouldDisplay

Returns a logical value indicating whether the annotation should be displayed. PDFPage respects this flag when drawing.

**Availability**
Available in Mac OS X v10.4 and later.

**See Also:** - setShouldDisplay: (page 15)

**shouldPrint**

- (BOOL)shouldPrint

Returns a logical value indicating whethe ear in printouts. PDFPage respects this flag when printing.

**Availability**
Available in Mac OS X v10.4 and later.

**See Also:** - setShouldPrint: (page 16)

These are the annotations that PDF Kit supports and can display in documents. However, PDF Kit can also support additional annotation types if they are specified using appearance streams. Appearance streams let you draw based on a drawing sequence rather than a specification based on a particular annotation type. For example, rather than specify "a circle annotation with a 20 point radius," an appearance stream would simply contain instructions for drawing a circle of that size.

Annotations often have content associated with them that your application can display. For example, text annotations typically appear as an icon in the PDF; when the user clicks on it, a window can open displaying its text.

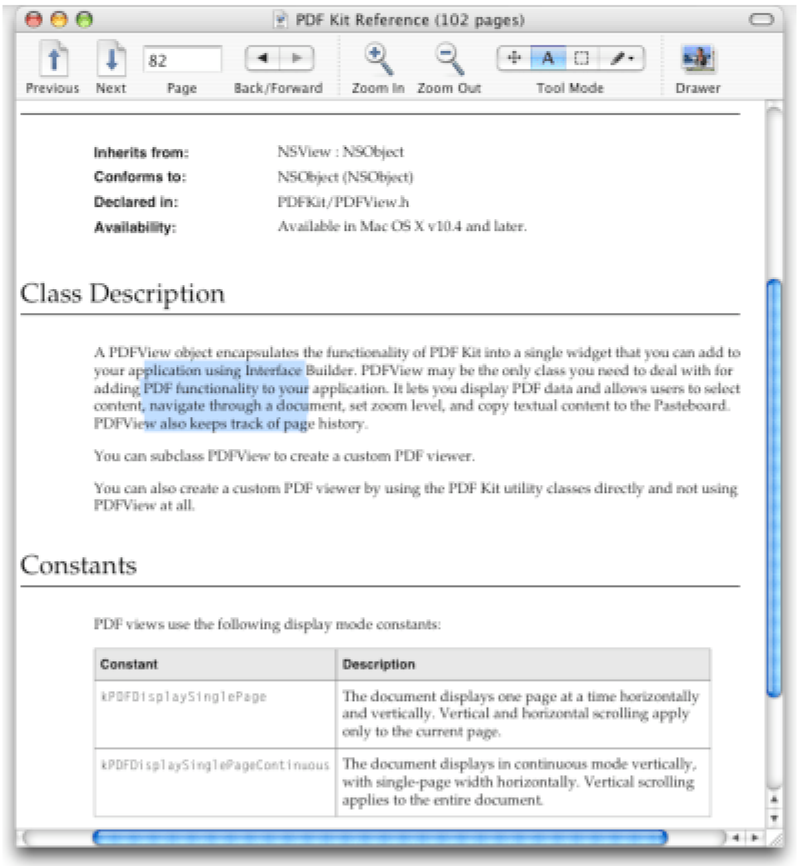
Note that PDF Kit does not supply a mechanism for displaying the annotation content; your application must create a window to display the content when the user clicks on an annotation.

**Note:** Currently, with the exception of link annotations, any annotations you create using PDF Kit cannot be modified after saving the document.

## Selections

PDF documents let the user select blocks of text, much like word processing applications. However, they offer greater flexibility in that text selections do not have to be linearly contiguous. For example, using PDF Kit, you could select a block of text within a page that doesn't have to be sequential, as shown in Figure 1-4 (page 13) Such selections can be useful if the document contains multicolumn pages, tables, or other unusual formatting.

**Figure 1-4** Arbitrary text selection in a PDF document



| | |
|---|---|
| **Inherits from:** | NSView : NSObject |
| **Conforms to:** | NSObject (NSObject) |
| **Declared in:** | PDFKit/PDFView.h |
| **Availability:** | Available in Mac OS X v10.4 and later. |

Class Description

A PDFView object encapsulates the functionality of PDF Kit into a single widget that you can add to your application using Interface Builder. PDFView may be the only class you need to deal with for adding PDF functionality to your application. It lets you display PDF data and allows users to select content, navigate through a document, set zoom level, and copy textual content to the Pasteboard. PDFView also keeps track of page history.

You can subclass PDFView to create a custom PDF viewer.

You can also create a custom PDF viewer by using the PDF Kit utility classes directly and not using PDFView at all.

Constants

PDF views use the following display mode constants:

| Constant | Description |
|---|---|
| `kPDFDisplaySinglePage` | The document displays one page at a time horizontally and vertically. Vertical and horizontal scrolling apply only to the current page. |
| `kPDFDisplaySinglePageContinuous` | The document displays in continuous mode vertically, with single-page width horizontally. Vertical scrolling applies to the entire document. |

You can experiment with block selection by holding down the Option key while selecting text in Preview (in Mac OS X v10.4 and later).

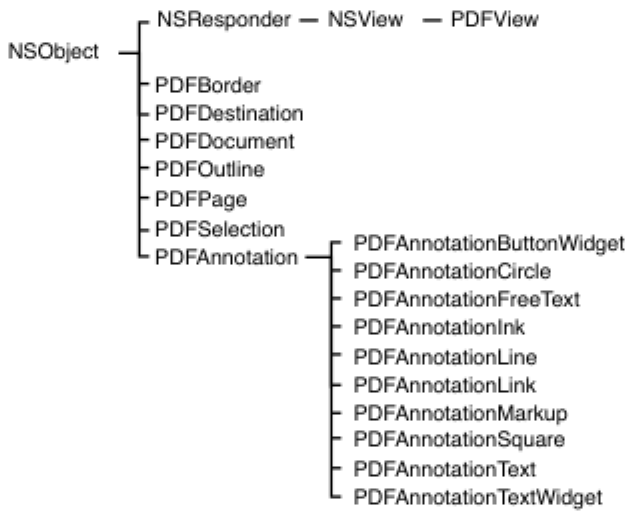
Selections are stored as selection objects, which also store additional data such as the page or pages containing the selection. This information is useful when presenting multiple selections to the user (for example, a list of search results).

EFTA00612184

## PDF Kit Classes

PDF Kit is divided into a number of different classes. With the exception of PDFView and PDFSelection, these classes correspond roughly to various objects in the PDF specification.

**Figure 1-5** The PDF Kit class hierarchy



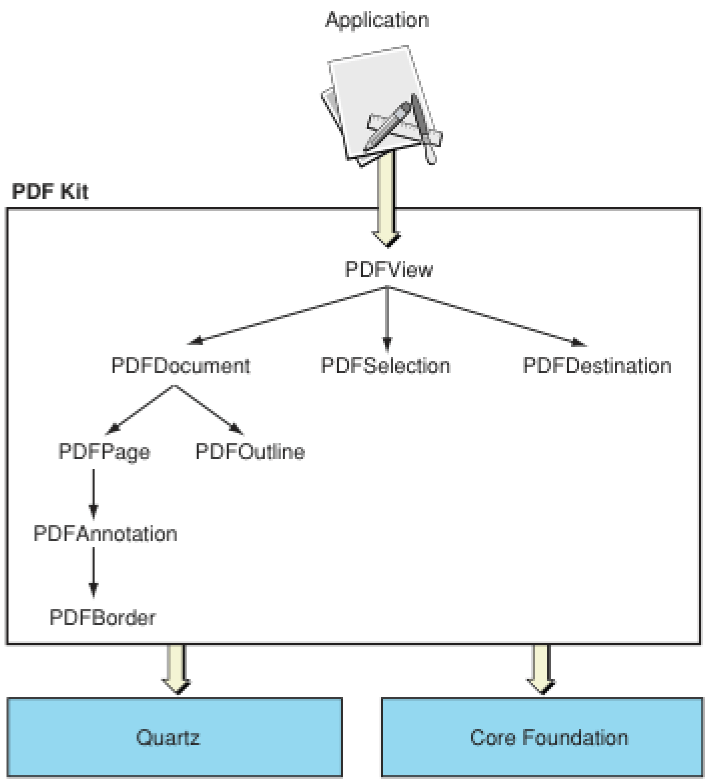
### The PDFView Class

The PDFView class, like the Web Kit WebView class, derives from the Application Kit NSView class. You can use a PDFView object directly in your application simply by placing it in a window using Interface Builder. Get the palette from `/Developer/Extras/Palettes/PDFKit.palette`.

PDFView may be the only PDF Kit class that you need to deal with. It lets you display PDF data in your application and allows users to select content and navigate through a document, set the zoom level, and copy textual content to the Pasteboard. Users can also drag-and-drop documents into PDFView.

PDFView calls upon the PDF utility classes to implement much of its functionality. If you want to add special features, you need to use or subclass from the utility classes.

**Figure 1-6** Utility classes as used by PDFView



## PDF Kit Utility Classes

The PDF Kit utility classes offer a mix of Foundation-like and Application Kit-like behavior. They are analogous to the NSString class, and its NSString Additions methods, in that many of them support drawing. These classes are subclasses of NSObject, as shown in Figure 1-5 (page 14).

### PDF Document

The primary PDF Kit utility class is PDFDocument, which represents PDF data or a PDF file. The other utility classes are either instantiated from methods in PDFDocument, as are PDFPage and PDFOutline; or support it, as do PDFSelection and PDFDestination.

You initialize a PDFDocument object with PDF data or with a URL to a PDF file. You can then ask for the page count, add or delete pages, perform a find, or parse selected content into an NSString object.

### PDFPage

As you might expect, the PDFPage class represents pages in a PDF document. Your application instantiates a PDFPage object by asking for one from a PDFDocument object. PDF page objects are what the user sees onscreen, and a view may display more than one page at a time. You can use PDFPage to render PDF content onscreen, add annotations, count characters, define selections, and get the textual content of a page as an NSString or NSAttributedString object.

## PDFOutline

In addition to displaying the actual document content, PDF Kit can also present outline information if that is included in the PDF. A PDFOutline object represents a parent or child element in an outline hierarchy.

Outlines are composed of a hierarchy of PDFOutline objects. The top level is the root outline object, which acts only as a container for other outline objects. The root outline is invisible to the user.

## PDFSelection

A PDFSelection object encompasses a span of text in a PDF document. You don't create PDF selections directly. You get PDFSelection objects as return values from selection methods that you invoke on PDFPage or PDFDocument objects, and as the return values from successful searches.

Selections on a PDF view may span multiple pages, may be noncontiguous, or both. For example, you can select the text in a single column of consecutive two-column pages. You can get the text and pages covered from a selection, combine selections, or extend selections in either direction.

## PDFAnnotation

A PDFAnnotation object can represent a variety of content other than the primary textual content in a PDF file: links, form elements, highlighting circles, and so on. Each annotation is associated with a specific location on a page, and may offer interactivity with the user.

PDFAnnotation is an abstract superclass of the concrete classes shown in Figure 1-3 (page 12). The various concrete classes represent annotation types that PDF Kit supports.

## PDFBorder

PDFBorder objects encapsulate the drawing behavior for the border of a PDFAnnotation object. A PDF border lets you specify such attributes as line style (for example, solid, dashed, or beveled), line width, and corner radius.

# PDF Kit Tasks

This chapter shows how you can implement common tasks with PDF Kit.

## Implementing a PDFView

Most developers will simply want to display PDF information in their views, so PDFView will fit the bill nicely.

A PDFView user interface element is available in Interface Builder, so you should use that wherever you want your application to display PDF content. Note that you need to install the PDF Kit palette in `/Developer/Extras/Palettes/PDFKit.palette` to make PDFViews available.

To add the PDFKit palette in Interface Builder, select the Palettes tab in the Preferences panel. Click the Add… button, navigate to the `/Developer/Extras/Palettes` folder, and select the PDFKit palette. Next, select the Customize Toolbar menu item in the Tools/Palettes menu and drag the PDFKit palette to the toolbar to make it visible.

After adding the PDFView element in your nib file, you can add PDF content from your application by calling the PDFDocument method `initWithURL`. For example, you could use code like the following:

```
PDFDocument *pdfDoc;

pdfDoc = [[[PDFDocument alloc] initWithURL: [NSURL fileURLWithPath: [self
fileName]]] autorelease];
[_pdfView setDocument: pdfDoc];
```

Alternatively, if your PDF data is stored in a different form, you can use the PDFDocument method `initWithData:`. Users can also drag-and-drop documents into the view.

The resulting PDFView handles the basic functionality required to view and navigate through a PDF document. Simple scrolling and live links work automatically, and you also get a contextual menu to handle scaling and navigation. To experiment with the “free” features of a PDFView, you can simply drag a PDF document into your view while in interface test mode in Interface Builder.

## PDF Kit in Preview

Preview in Mac OS X v10.4 uses PDF Kit, so you can use this application as a guide to see what is possible in your own PDF views. Many method calls in PDF Kit have a comparable menu item in Preview. Table 2-1 (page 18) shows the correspondance between the various menu items and their API equivalents.

**Table 2-1** Preview Menu Items versus PDF Kit methods

| Menu | Submenu | Method |
|---|---|---|
| View | PDF Display | PDFView: - `setDisplayMode:` and - `setDisplayBox:` |
| | Zoom In | PDFView: - `zoomIn:` |
| | Zoom Out | PDFView: - `zoomOut:` |
| Go | Next | PDFView: - `goToNextPage:` |
| | Previous | PDFView: - `goToPreviousPage:` |
| | Go to Page | PDFView: - `goToPage:` |
| | First | PDFView: - `goToFirstPage:` |
| | Last | PDFView: - `goToLastPage:` |
| | Back | PDFView: - `goBack:` |
| | Forward | PDFView: - `goForward:` |
| Tools | Rotate Left | PDFPage: - `setRotation:` to - `rotation` -90 |
| | Rotate Right | PDFPage: - `setRotation:` to - `rotation` +90 |
| | Annotation | `initWithBounds:` for the appropriate annotation subclass (such as PDFAnnotationCircle) |

## Creating Outlines

Many PDF documents contain outlines, so your application will usually want to display this information as well.

Note that because some PDF documents do not contain outline information, you should probably make the outline display an optional element in your user interface. For example, you can display outlines in a drawer, which can be closed if not needed.

You can use the NSOutlineView class to display your PDF outlines. Instances of this class automatically display the outline hierarchy with disclosure triangles and live links to the appropriate PDF pages. Listing 2-1 (page 19) shows how you might do so.

**Listing 2-1** Loading PDF outline information

```
_outline = [[[_pdfView document] outlineRoot] retain];                    // 1
    if (_outline)
    {
        [_noOutlineText removeFromSuperview];                             // 2
        _noOutlineText = NULL;

        [_outlineView reloadData];                                        // 3
    }
    else
    {
        [[_outlineView enclosingScrollView] removeFromSuperview];         // 4
        _outlineView = NULL;
    }
```

Here is how the code works:

1. Obtains the root (topmost) outline element. `_outline` is an instance of PDFOutline.

2. If a root outline exists, removes placeholder text indicating "No Outline."

3. Loads PDF outline information into the outline view. The outline view calls your delegate methods to determine the elements in the outline hierarchy.

4. If the root outline does not exist, removes the outline view, leaving behind placeholder text.

After you invoke the `reloadData` method, the outline view calls various data source delegate methods to populate the outline. These delegate methods are defined in the NSOutlineViewDataSource protocol. Your application must implement these methods so that the proper PDF data is added to the outline.

Listing 2-2 (page 19) shows the delegate method for obtaining the number of childen, which simply returns the value obtained by the PDFOutline method `numberOfChildren`. If the item parameter is `NULL`, this method returns the number of children for the root outline.

**Listing 2-2** Delegate method for determining the number of children

```
- (int) outlineView: (NSOutlineView *) outlineView numberOfChildrenOfItem: (id)
 item
{
    if (item == NULL)
    {
        if (_outline)
            return [_outline numberOfChildren];
        else
            return 0;
    }
    else
        return [(PDFOutline *)item numberOfChildren];
}
```

Listing 2-3 (page 20) shows the delegate method for obtaining a particular child outline by calling the PDFOutline method `childAtIndex`. If the item parameter is `NULL`, this method returns the appropriate child of the root outline.

**Listing 2-3** Delegate method for obtaining a child element

```
- (id) outlineView: (NSOutlineView *) outlineView child: (int) index ofItem:
(id) item
{
    if (item == NULL)
    {
        if (_outline)
            return [[_outline childAtIndex: index] retain];
        else
            return NULL;
    }
    else
        return [[(PDFOutline *)item childAtIndex: index] retain];
```

Listing 2-4 (page 20) shows a delegate method for determining if an outline element is expandable (that is, whether it has child outlines).

**Listing 2-4** Delegate method for determining if an element has children

```
- (BOOL) outlineView: (NSOutlineView *) outlineView isItemExpandable: (id) item
{
    if (item == NULL)
    {
        if (_outline)
            return ([_outline numberOfChildren] > 0);
        else
            return NO;
    }
    else
        return ([(PDFOutline *)item numberOfChildren] > 0);
}
```

Listing 2-5 (page 20) shows a delegate method for obtaining an outline element's label, which calls the PDFOutline method `label`. The label is simply the string that is displayed in the outline view (for example, a chapter title).

**Listing 2-5** Delegate method for obtaining an element's contents

```
- (id) outlineView: (NSOutlineView *) outlineView
        objectValueForTableColumn: (NSTableColumn *) tableColumn
        byItem: (id) item
{
    return [(PDFOutline *)item label];
}
```

When the user selects an outline element, your application should update the PDF display to show the page corresponding to that element. The simplest way to do so is to call the PDFView's `goToDestination` method, as shown in Listing 2-6 (page 20).

**Listing 2-6** Displaying the page associated with an outline element

```
- (IBAction) takeDestinationFromOutline: (id) sender
{
    [_pdfView goToDestination: [[sender itemAtRow:
                                [sender selectedRow]] destination]];
}
```

In addition, if the user scrolls or otherwise moves through the document, your application should update the outline to highlight the outline element that corresponds to the currently displayed page. You can do so by installing a notification handler to be called each time the page changes (that is, when PDFViewPageChangedNotification is posted). Listing 2-7 (page 21) hows how you might do so.

**Listing 2-7** Updating the outline when the page changes

```
- (void) pageChanged: (NSNotification *) notification
{
    unsigned int    newPageIndex;
    int             numRows;
    int             i;
    int             newlySelectedRow;

    if ([[_pdfView document] outlineRoot] == NULL)                          // 1
        return;

    newPageIndex = [[_pdfView document] indexForPage:                      // 2
                                        [_pdfView currentPage]];

    // Walk outline view looking for best firstpage number match.
    newlySelectedRow = -1;
    numRows = [_outlineView numberOfRows];
    for (i = 0; i < numRows; i++)                                           // 3
    {
        PDFOutline  *outlineItem;

        // Get the destination of the given row....
        outlineItem = (PDFOutline *)[_outlineView itemAtRow: i];

        if ([[_pdfView document] indexForPage:
                    [[outlineItem destination] page]] == newPageIndex)
        {
            newlySelectedRow = i;
            [_outlineView selectRow: newlySelectedRow
                                byExtendingSelection: NO];
            break;
        }
        else if ([[_pdfView document] indexForPage:
                    [[outlineItem destination] page]] > newPageIndex)
        {
            newlySelectedRow = i - 1;
            [_outlineView selectRow: newlySelectedRow
                                byExtendingSelection: NO];
            break;
        }
    }

    if (newlySelectedRow != -1)                                             // 4
        [_outlineView scrollRowToVisible: newlySelectedRow];
}
```

Here is how the code works:

1. Checks to see if a root outline exists. If not, then there is no outline to update, so simply return.

EFTA00612192

2. Obtains the index value for the current page. The PDFView method `currentPage` returns the PDFPage object, and the PDFDocument method `indexForPage` returns the actual index for that page. This index value is zero-based, so it doesn't necessarily correspond to a page number.

3. Iterate through each visible element in the outline, checking to see if one of the following occurs:

   - The index of an outline element matches the index of the new page. If so, highlight this element (using the NSTableView method `selectRow:byExtendingSelection`).
   - The index of the outline element is larger than the index of the page. If so, a match was not possible as the index corresponds to a hidden child of a visible element. In this case, use `selectRow` to highlight the parent outline element (the current row -1 ).

4. Call the NSTableView method `scrollRowToVisible` to adjust the outline view (if necessary) to make the highlighted element visible.

## Searching a PDF Document

Users often need to search through a PDF document. PDF Kit offers two methods for doing so:

- Searching string-by-string through the document. That is, when the user searches for *string*, PDF Kit returns the first occurrence of the string. Additional searches ("Find Again") return successive instances of *string*. This search method is synchronous.
- Obtaining a listing of all occurrences of *string* in a document. This search method may be synchronous or asynchronous.

For simple string-by-string searching, your application can simply call the PDFDocument method `findString:fromSelection:withOptions:`.

```
- (PDFSelection *) findString:(NSString *)string
         fromSelection:PDFSelection *selection:withOptions:(int)options
```

To display the selection returned, you can call the PDFView method `setCurrentSelection`, which highlights the selection, followed by `scrollSelectionToVisible`.

You can specify the following options:

- `NSCaseInsensitiveSearch`: Ignore case when making a match.
- `NSLiteralSearch`: Search for contiguous words, separated by spaces.
- `NSBackwardsSearch`: Search backwards from the current selection.

By passing `NULL` for the selection, you can begin the search from the beginning (or end) of the document. By passing the most recent match for the selection, you can implement "Find Again" behavior. If the `findString` call returns `NULL`, that means that either the string was not found, or the search reached the end (or beginning) of the document.

To obtain all the occurrences of a given string, you can use either of the following PDFDocument methods:

- `findString:withOptions:` to synchronously obtain an NSArray object holding all the matches
- `beginFindString:withOptions:` to asynchronously begin a search for all occurrences. PDF Kit calls your delegate method each time a match is found.

Unless you are sure that the search will be brief, you should choose to use `beginFindString:withOptions:`

As this is an asynchronous search, PDFDocument includes two other useful find-related methods:

- `isFinding` to determine if a search in currently in progress
- `cancelFindString` to terminate a current search.

Listing 2-8 (page 23) shows how you might initiate a search:

**Listing 2-8** Beginning an asynchronous search

```
- (void) doFind: (id) sender
{
    if ([[_pdfView document] isFinding])                                    // 1
        [[_pdfView document] cancelFindString];

    if (_searchResults == NULL)                                             // 2
        _searchResults = [[NSMutableArray arrayWithCapacity: 10] retain];

    [[_pdfView document] beginFindString: [sender stringValue]              // 3
                withOptions: NSCaseInsensitiveSearch];
}
```

Here is how the code works:

1. Cancels any current searches.
2. Allocates a mutable array to hold the search results if one does not already exist.
3. Calls the PDFDocument method `beginFindString:withOptions:` with the desired search string.

During the search, PDF Kit sends out notifications that your application can react to:

- PDFDocumentDidBeginFindNotification
- PDFDocumentDidEndFindNotification
- PDFDocumentDidBeginPageFindNotification
- PDFDocumentDidEndPageFindNotification
- PDFDocumentDidFindMatchNotification

The first two notifications are sent when the search starts, or finishes a search. You can use these notifications to set up and remove progress bars or any other initializations.

The begin and end page notifications are sent when the search begins or ends searching a page in the document. You can use these notifications to update a progress bar or page counter.

EFTA00612194

The find match notification is sent whenever a match is found for the search string. Typically you will want to obtain the string selection and store it in an array for later display. However, in most cases it may be easier to use the PDFDocument delegate method `didMatchString`, which automatically passes you the matching selection. Listing 2-9 (page 24) shows how you might implement a delegate to take each matching string and add it to an array of search results in an NSTableView.

**Listing 2-9** Adding search results to a table view

```
- (void) didMatchString: (PDFSelection *) instance
{
    // Add page label to our array.
    [_searchResults addObject: [instance copy]];

    // Force a reload.
    [_searchTable reloadData];
}
```

Here `_searchResults` is an instance of NSMutableArray, and `_searchTable` is an instance of NSTableView.

To make sure that the NSTableView displays the search results correctly, you need to implement delegate data source methods (similar to those required for NSOutlineView). These delegate methods are defined in the NSTableDataSource protocol.

Listing 2-10 (page 24) shows a delegate method for determining the number of rows in the table view. This method simply obtains the number of items in the search results by calling the NSMutableArray method `count`.

**Listing 2-10** Determining the number of rows in the table

```
- (int) numberOfRowsInTableView: (NSTableView *) aTableView
{
    return ([_searchResults count]);
}
```

Listing 2-11 (page 24) shows a delegate data source method for obtaining the value of a particular column.

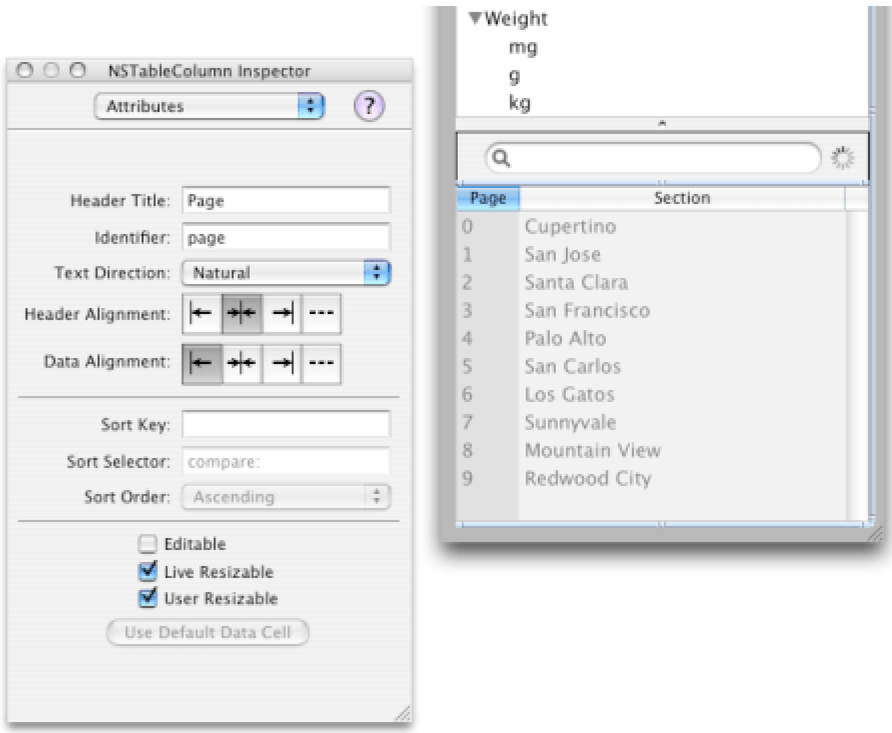
**Listing 2-11** Obtaining the value for a column

```
- (id) tableView: (NSTableView *) aTableView objectValueForTableColumn:
            (NSTableColumn *) theColumn
            row: (int) rowIndex
{
    if ([[theColumn identifier] isEqualToString: @"page"])
        return ([[[[_searchResults objectAtIndex: rowIndex] pages]
                     objectAtIndex: 0] label]);
    else if ([[theColumn identifier] isEqualToString: @"section"])
    {
        NSString    *label = [[[_pdfView document] outlineItemForSelection:
                         [_searchResults objectAtIndex: rowIndex]] label];
        return label;
    }
    else
        return NULL;
}
```

The table view calls this method whenever it needs to determine the value of a particular table element (such as when preparing the table for display).

The search results table in this example contains only two columns: the page containing the hit and the outline section that contains the hit. (In a more sophisticated example, you may want to display a portion of the text containing the matching string, as Preview does.) You reference these columns using the identifier tags you specified for them in the NSTableColumn inspector window in Interface Builder (as shown in Figure 2-1 (page 25)).

**Figure 2-1** Assigning column identifiers in Interface Builder



Header Title: Page
Identifier: page
Text Direction: Natural
Sort Key:
Sort Selector: compare:
Sort Order: Ascending
☐ Editable
☑ Live Resizable
☑ User Resizable

| Page | Section |
|---|---|
| 0 | Cupertino |
| 1 | San Jose |
| 2 | Santa Clara |
| 3 | San Francisco |
| 4 | Palo Alto |
| 5 | San Carlos |
| 6 | Los Gatos |
| 7 | Sunnyvale |
| 8 | Mountain View |
| 9 | Redwood City |

For this example:

- If the column identifier is "page," return the page number of the hit by calling the PDFSelection method `pages`.
- If the column identifier is "section," return the outline label for the selection by calling the PDFDocument method `outlineItemForSelection`.

When the user selects an item in the search results, your application should display the corresponding page. Listing 2-12 (page 25) shows how you might do so using an NSTableView notification.

**Listing 2-12** Handling a selection in the table

```
- (void) tableViewSelectionDidChange: (NSNotification *) notification
{
    int rowIndex;

    // What was selected.  Skip out if the row has not changed.
```

EFTA00612196

```
    rowIndex = [(NSTableView *)[notification object] selectedRow];              // 1
    if (rowIndex >= 0)
    {
        [_pdfView setCurrentSelection:                                            // 2
                    [_searchResults objectAtIndex: rowIndex]];
        [_pdfView scrollSelectionToVisible: self];                                // 3
    }
}
```

A table view sends the NSTableViewSelectionDidChangeNotification when an item is selected, and calls your delegate method `tableViewSelectionDidChange`. Here is how the code works:

1. Checks to see if the selection is valid.
2. Sets the current selection to the one that the user clicked.
3. Updates the PDF view to show the page containing the selection.

For more information about implementing these delegate methods, see *Table View Programming Guide* in Cocoa User Experience Documentation.

# Document Revision History

This table describes the changes to *PDF Kit Programming Guide*.

| Date | Notes |
|---|---|
| 2005-11-09 | Added information about how to add the PDFKit palette in Interface Builder. |
| 2005-04-29 | First public version. |