Document-Based Applications Overview

User Experience

2010-03-24

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

Simultaneously published in the United States and Canada.

Contents

Figures and Listings

Introduction to Document-Based Applications Overview

This document describes how to use the architecture supplied by the Application Kit to create applications that can create, open, load, and save multiple document files.

Who Should Read This Document

Every developer who wants to use the document architecture of the Application Kit should read this document.

To understand the information in this document you should have a general knowledge of Cocoa programming paradigms and, to understand the code examples, familiarity with the Objective-C language.

Organization of This Document

By using the document architecture provided by the Application Kit, you acquire many features of a well-crafted application "for free," but you must understand the architecture to implement it properly.

Some of the articles in this document describe window management using the `NSWindowController` class. However, window management with `NSWindowController` objects is a technique that is not limited to document-based applications. You can also use `NSWindowController` without an associated `NSWindow` object.

This document contains the following articles:

- "Document-Based Application Architecture" (page 11) provides an overview of the classes that make up a document-based application and defines a document in the context of the Application Kit framework.
- "The Roles of Key Objects in Document-Based Applications" (page 13) describes how the three major classes in the document-based application architecture play distinct but cooperating roles.
- "Implementing a Document-Based Application" (page 17) describes the tasks you must do, and those you might want to do, when implementing a document-based application.
- "Creating a Subclass of NSDocument" (page 23) describes how to subclass `NSDocument`.
- "Creating a Subclass of NSDocumentController" (page 27) describes how to subclass `NSDocumentController`.
- "Message Flow in the Document Architecture" (page 29) describes the default message flow among major objects of the document architecture.
- "Window Closing Behavior" (page 37) describes how the document architecture automates memory management for documents and their associated windows and window controllers.
- "Window Controllers and Nib Files" (page 39) describes the management of a nib file.

- "Storing Document Types Information in the Application's Property List" (page 41) describes how document-based applications use a property list to specify the document types the application can edit or view.
- "Saving HFS Type and Creator Codes" (page 45) describes how to save hierarchical file system type and creator codes in documents.
- "Creating Multiple-Document-Type Applications" (page 47) explains how to use multiple types of documents based on different `NSDocument` subclasses in a document-based application.
- "Autosaving in the Document Architecture" (page 49) describes the autosave feature built into the document architecture and explains how to turn it on in your application.
- "Error Handling in the Document Architecture" (page 51) explains how to take advantage of the Application Kit's robust error handling and provides some best practices advice for overriding methods that take `NSError` parameters.
- "Frequently Asked Questions" (page 53) answers common questions about the document-handling classes in the Application Kit.

Document-Based Application Architecture

A document-based application is one of the more common types of applications. It provides a framework for generating identically contained but uniquely composed sets of data that can be stored in files. Word processors and spreadsheet applications are two examples of document-based applications. Document-based applications do the following things:

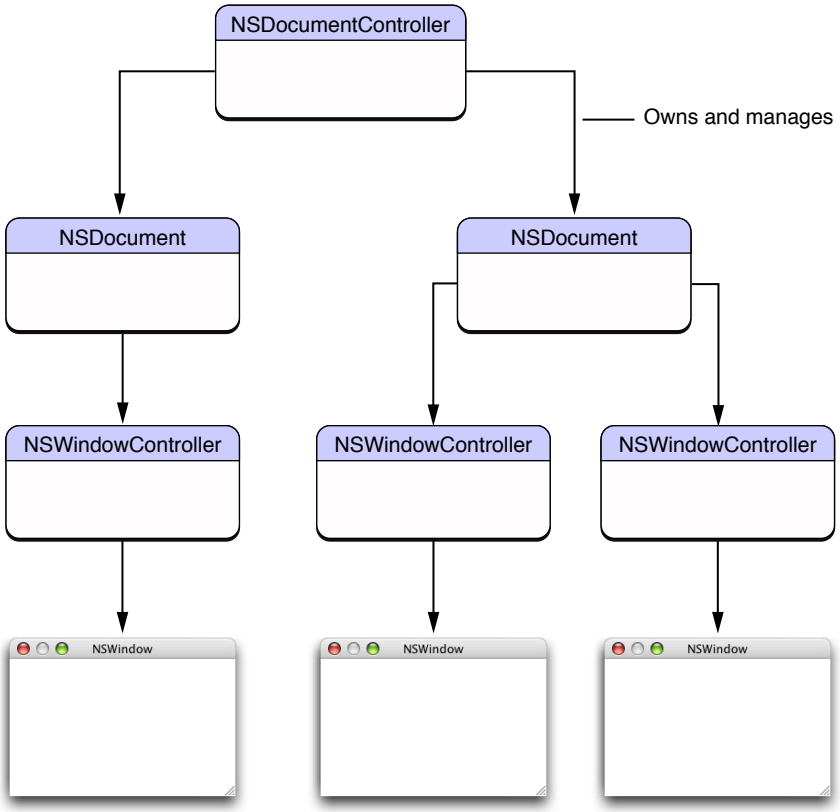
- Create new documents.
- Open existing documents that are stored in files.
- Save documents under user-designated names and locations.
- Revert to saved documents.
- Close documents (usually after prompting the user to save edited documents).
- Print documents and allow the page layout to be modified.
- Represent data of different types internally.
- Monitor and set the document's edited status and validate menu items.
- Manage document windows, including setting the window titles.
- Handle application and window delegation methods (such as when the application terminates).

The Major Classes

Three Application Kit classes provide an architecture for document-based applications, called the document architecture, that simplifies the work developers must do to implement the features listed above. These classes are `NSDocument`, `NSWindowController`, and `NSDocumentController`.

Objects of these classes divide and orchestrate the work of creating, saving, opening, and managing the documents of an application. They are arranged in a tiered one-to-many relationship, as depicted in Figure 1. An application can have only one `NSDocumentController`, which creates and manages one or more `NSDocument` objects (one for each New or Open operation). In turn, an `NSDocument` object creates and manages one or more `NSWindowController` objects, one for each of the windows displayed for a document. In addition, some of these objects have responsibilities analogous to `NSApplication` and `NSWindow` delegates, such as approving major events like closing and quitting.

Figure 1 Relationships among NSDocumentController, NSDocument, and NSWindowController



What Is a Document?

Conceptually, a document is a container for a body of information identified by a name under which it is stored in a disk file. In this sense, however, the document is not the same as the file but is an object in memory that owns and manages the document data. In the context of the Application Kit, a document is an instance of a custom `NSDocument` subclass that knows how to represent internally, in one or more formats, persistent data that is displayed in windows. A document can read that data from a file and write it to a file. It is also the first-responder target for many menu commands related to documents, such as Save, Revert, and Print. A document manages its window's edited status and is set up to perform undo and redo operations. When a window is closing, the Application Kit first asks the document, before it asks the window delegate, to approve the closing.

To create a useful `NSDocument` subclass, you must override some methods, and you might want to override others. The `NSDocument` class itself knows how to handle document data as undifferentiated lumps; although it understands that these lumps are typed, it knows nothing about particular types. In their overrides of the data-based reading and writing methods, subclasses must add the knowledge of particular types and how data of the document's native type is structured internally. Subclasses are also responsible for the creation of the window controllers that manage document windows and for the implementation of undo and redo. The `NSDocument` class takes care of much of the rest, including generally managing the state of the document.

The Roles of Key Objects in Document-Based Applications

Instances of the three major classes in the document architecture play distinct but cooperating roles, as described in this article.

The Role of NSDocumentController

The primary job of an application's `NSDocumentController` object is to create and open documents, and to track and manage those documents. When a user chooses New from the File menu, an `NSDocumentController` object gets the appropriate `NSDocument` subclass from the `CFBundleDocumentTypes` property in the application's information property list, allocates an instance of this class, and initializes this instance by invoking the `NSDocument` method `initWithType:error:`. When the user chooses Open from the File menu, the `NSDocumentController` object displays the Open panel, gets the user's selection, finds the `NSDocument` subclass for the file (based on its document type information), allocates an instance of this class, and initializes the object and loads document data by invoking the `NSDocument` method `initWithContentsOfURL:ofType:error:`. In both cases, the `NSDocumentController` object adds a reference to the document object to an internal list to facilitate the management of its documents. It has a notion of the current document as the document whose window is currently main. (The foremost application window that is the focus of the user's attention is called the main window.)

The `NSDocumentController` object manages the Open Recent menu for the application, maintaining a list of the URLs of documents the application has recently handled. It notes recent documents during the opening, saving, reverting, and closing of documents.

`NSDocumentController` is hard-wired to respond appropriately to certain application events, such as when the application starts up, when it terminates, when the system is shutting down, and when documents are opened or printed from the Finder. If you wish, you can make a custom object the application delegate and implement the delegate methods invoked as a result of the same events, and these methods will be invoked instead. However, the default `NSDocumentController` object is an adequate application controller for most situations, and typically you should not need to subclass it. If you require additional behavior, such as displaying About panels and handling application preferences, you can have a custom controller object perform these duties rather than a subclass of `NSDocumentController`. Nonetheless, it is possible to subclass `NSDocumentController` if necessary, as described in "Creating a Subclass of NSDocumentController" (page 27).

The Role of NSDocument

The primary job of an `NSDocument` object is to represent, manipulate, store, and load the persistent data associated with a document. As such it is a model-controller. Based on the document types it claims to understand (as specified in the `CFBundleDocumentTypes` property of the application's information property list), a document must be prepared to:

- Provide the data displayed in windows (and represented internally) in a supported document type.
- Store document data in a file at a specified location in the file system.
- Read document data stored in a file.

With the assistance of its window controllers, an `NSDocument` object manages the display and capture of the data in its windows. By some special hard-wiring of the Application Kit, the `NSDocument` object associated with the key window is the recipient of first-responder action messages when users save, print, revert, and close documents. In response to the appropriate action, it knows how to run and manage the Save panel and the Page Layout panel.

A fully implemented `NSDocument` object knows how to track its edited status, print document data, and perform undo and redo operations. Although these behaviors aren't completely provided by default, the `NSDocument` object does assist the developer in implementing each. For edited-status tracking, the `NSDocument` object provides an API for updating a change counter. For undo/redo operations, the `NSDocument` object by default lazily creates an `NSUndoManager` instance when one is requested, responds appropriately to Undo and Redo menu commands, and updates the change counter when undo and redo operations are performed. For printing, the `NSDocument` object facilitates the display of the Page Layout panel and the subsequent modification of the `NSPrintInfo` object used in printing.

The Role of NSWindowController

An `NSWindowController` object manages one window associated with a document, which is usually stored in a nib file. As such it is a view-controller. If a document has multiple windows, each window has its own window controller. For example, a document might have a main data-entry window and a window listing records for selection; each window would have its own `NSWindowController` object. When requested by its owning `NSDocument` object, an `NSWindowController` object loads the nib file containing a window and displays it. It also assumes responsibility for closing windows properly (after ensuring that the data they display is saved).

The `NSWindowController` class offers additional behavior to document-based applications, such as cascading document windows in relation to each other, so they don't completely obstruct one another.

Subclasses of `NSWindowController` are optional. Applications can often use the default instance. Subclasses can augment `NSWindowController` to perform different nib-loading and setup tasks or to customize the titles of windows.

Typical Usage Patterns

This section describes three ways to use the document architecture, starting with the simplest and proceeding to the most complex.

The simplest way to use the document architecture is appropriate for documents that have only one window and are simple enough that there isn't much benefit in splitting the controller layer into a model-controller and a view-controller. In this case, you need only to create a subclass of `NSDocument`. The `NSDocument` subclass provides storage for the model and the ability to load and save document data. It also has any

outlets and actions required for the user interface. It overrides `windowNibName` to return the nib file name used for documents of this type. The `NSDocument` object automatically creates an `NSWindowController` object to manage that nib file, but the `NSDocument` object serves as the nib file's owner.

If your document has only one window, but it is complex enough that you'd like to split up some of the logic in the controller layer, you can subclass `NSWindowController` as well as `NSDocument`. In this case, any outlets and actions and any other behavior that is specific to the management of the user interface goes into the `NSWindowController` subclass. Your `NSDocument` subclass must override `makeWindowControllers` instead of `windowNibName`. The `makeWindowControllers` method should create an instance of your `NSWindowController` subclass and add it to the list of managed window controllers with `addWindowController:`. The `NSWindowController` object should be the file's owner for the nib file because this creates better separation between the view-related logic and the model-related logic. This approach is recommended for all but the simplest cases.

If your document requires or allows multiple windows for a single document, you should subclass `NSWindowController` as well as `NSDocument`. In your `NSDocument` subclass you override `makeWindowControllers` just as in the second procedure described above. However, in this case you might create more than one instance of `NSWindowController`, possibly from different subclasses of `NSWindowController`. Some applications need several different windows to represent one document. Therefore you probably need several different subclasses of `NSWindowController` and you must create one of each in `makeWindowControllers`. Some applications need only one window for a document but want to allow the user to create several copies of the window for a single document (sometimes this is called a multiple-view document) so that the user can have each window scrolled to a different position or displayed in different ways. In this case, your `makeWindowControllers` override may create only one `NSWindowController` object, but there will be a menu command or similar control that allows the user to create others.

Documents and Scripting

Scripting support is mostly automatic for applications based on the document architecture, for several reasons. First, `NSDocument` and the other classes in the document architecture directly implement the standard document scripting class (as expected by AppleScript) and automatically support many of the scripting commands that apply to documents. Second, because the document architecture is intended to work with application designs that use model-view-controller (MVC) separation, and because scripting support depends on many of the same design points, applications that use the document architecture are already in better shape to support scripting than other applications that are not designed that way. Finally, the document plays an important role in the scripting API of most applications; the `NSDocument` class fills that role and provides a good starting point for allowing scripted access to the model layer of your application.

If an application is not based on the document architecture, making it scriptable requires you to duplicate work you would otherwise get for free. The TextEdit example application shows how to make a document-based application that is not based on `NSDocument` scriptable. See the Sketch example project for an example of how to implement a scriptable `NSDocument`-based application.

Implementing a Document-Based Application

It is possible to put together a document-based application without having to write much code. If your requirements are minimal, you can use the default `NSWindowController` instance and default `NSDocumentController` instance provided by the Application Kit. You have only to create a document project, compose the human interface, implement a subclass of `NSDocument`, and add any other custom classes or behavior required by your application.

The following sections step you through the tasks you must do, and those you might want to do, when implementing a document-based application. Where something is described in detail elsewhere, such as overridden methods in `NSDocument` subclasses, you are referred there.

As for the three classes behind document-based applications, two likely concerns are the number of required objects and whether subclassing is necessary. The following table summarizes this information:

Class	How many objects?	Subclass?
`NSDocumentController`	1 per application	Optional (but unlikely)
`NSDocument`	1 per document	Required
`NSWindowController`	1 per window	Optional (but likely)

The Document-Based Application Project Template

Cocoa's development environment provides a Cocoa Document-based Application project template in Xcode to expedite the development of document-based applications. This project template provides the following things:

- A nib file for the application's document

  This nib file is named `MyDocument.nib`. A subclass of `NSDocument` named `MyDocument` is made file's owner of the nib file. It has an outlet named `window` connected to its window object. The window has only a text field on it with the words "Your document contents here".

- The application's main nib file

  This nib file is named `MainMenu.nib`. It contains an application menu, a File menu (with all of its associated document commands), and an Edit menu with text editing commands and Undo and Redo menu items. These menu items, as well as all of the menu items of the File menu, are connected to the appropriate first-responder action methods. The About NewApplication menu item is connected to the `orderFrontStandardAboutPanel:` action method that displays a standard About window.

- A skeletal `NSDocument` subclass implementation

The project includes `MyDocument.h` and `MyDocument.m`. The latter file includes empty but commented blocks for the `dataRepresentationOfType:` and `loadDataRepresentation:ofType:` methods. These skeletal implementations are for applications targeted for systems that must be able to run on Mac OS X v10.3 and earlier. For applications that can require Mac OS X v10.4 and later, you should instead override the `dataOfType:error:` and `readFromData:ofType:error:` methods. It also includes a fully implemented `windowNibName` method that returns the name of the document window nib file, `MyDocument`, and an override of `windowControllerDidLoadNib:`.

- The application's information property list

  In the Xcode Target inspector, you can edit the file `Info.plist`, which contains placeholder values for global application keys, as well as for the `CFBundleDocumentTypes` key (which specifies information about the document types the application works with).

The steps described in the following sections show how to create a document-based application using Xcode's Cocoa Document-based Application project template. The following table lists the File menu first-responder action connections that should already exist in the template application.

File menu command	First-responder action
New	`newDocument:`
Open...	`openDocument:`
Open Recent > Clear Menu	`clearRecentDocuments:`
Close	`performClose:`
Save	`saveDocument:`
Save As...	`saveDocumentAs:`
Revert	`revertDocumentToSaved:`
Page Setup...	`runPageLayout:`
Print...	`printDocument:`

The template has similar ready-made connections for the Edit, Window, and Help menus. If your application does not support any of the supplied actions, such as printing, for example, you should remove the associated menu items from the nib.

Create the Project and Compose the Interface

1. Launch Xcode and choose New Project from the File menu.

   In the New Project dialog, choose "Cocoa Document-based Application."

2. Provide a name and location on disk for the project.

3. Double-click the `MyDocument.nib` file in the Resources group in Xcode's Groups & Files pane. This opens the file in Interface Builder.

If you want to change the name of the nib file, you can save it under another name in Interface Builder and add it to the project. If you do this, you must also modify the string returned by the `windowNibName` method in the `NSDocument` subclass implementation.

4. Create the visible interface for the document window in Interface Builder.

5. If the objects on the document window require further outlets or actions, add these to the MyDocument subclass of `NSDocument`. Connect these actions and outlets via the File's Owner icon on the Instances display of the nib file window.

   Important: Do not generate an instance of MyDocument to make these connections.

   If you want to name your `NSDocument` subclass something other than `MyDocument` (the default name), change the name in Interface Builder and wherever it occurs in the Document header (`.h`) and implementation (`.m`) files. You must also change the name under the `NSDocumentClass` key in the `Info.plist` file.

6. If your document objects interact with other custom objects, such as model objects that perform specialized computations, define those objects in Interface Builder and make any necessary connections to them.

Complete the Information Property List

1. In Xcode, click the Targets group disclosure triangle, select the application target, and click the Info button in the Xcode toolbar (or choose Get Info from the File menu). Click the Properties tab in the Target Info window.

2. Replace the placeholder or default values in the information property list with those specific to your application. You can also edit the `Info.plist` file directly.

   See "Storing Document Types Information in the Application's Property List" (page 41) and "Creating Multiple-Document-Type Applications" (page 47) for information on the properties specific to documents.

   See also "Inspecting Targets" in Targets for information about the fields available in the Xcode Target inspector.

Implement the NSDocument Subclass

The following procedure just gives general guidelines. For more details see the `NSDocument` reference and "Creating a Subclass of NSDocument" (page 23). You might also read the Cocoa documentation covering undo, copy/paste, and printing.

1. In Xcode, open the header file of your `NSDocument` subclass (in the Classes group in Xcode's Groups & Files pane).

2. If you added outlets or actions to your `NSDocument` subclass in Interface Builder, add them to the subclass's header file. Also add any other required instance variables and include the declarations of new methods that you wish to be public, such as accessor methods.

   You can specify additional outlets in actions in the existing header file and then import them into the nib file by using Interface Builder's Read Files command in the Classes menu.

3. Open the subclass implementation file in the project's Classes group.

4. Although it's not usually necessary, you can override the designated initializer (`init`) and perhaps the document-opening initializer `initWithContentsOfFile:ofType:` to perform initializations specific to your subclass; be sure to invoke the superclass implementations. You can also implement `awakeFromNib` to initialize objects unarchived from the document's window nib files (but not the document itself).

5. Override the data-based reading and writing methods. For applications targeted for Mac OS X v10.4 and later, override `readFromData:ofType:error:` (to load document data of a certain type) and `dataOfType:error:` (to provide document data of a certain type).

   The following example implementations assume that the application has an `NSTextView` object configured typically with an `NSTextStorage` object to hold the document's data. The `NSDocument` object has `text` and `setText:` accessors for the document's `NSAttributedString` data model.

```
- (BOOL)readFromData:(NSData *)data ofType:(NSString *)typeName error:(NSError
 **)outError {
    BOOL readSuccess = NO;
    NSAttributedString *fileContents = [[NSAttributedString alloc]
            initWithData:data options:nil documentAttributes:nil
            error:outError];
    if (fileContents) {
        readSuccess = YES;
        [self setText:fileContents];
        [fileContents release];
    }
    return readSuccess;
}

- (NSData *)dataOfType:(NSString *)typeName error:(NSError **)outError {
    [textView breakUndoCoalescing];
   NSData *data = [textView dataFromRange:NSMakeRange(0, [[textView textStorage]
 length])
                               documentAttributes:nil
                               error:outError];
    if (!data && outError) {
        *outError = [NSError errorWithDomain:NSCocoaErrorDomain
                                code:NSFileWriteUnknownError userInfo:nil];
    }
    return data;
}
```

   For applications that must be able to run on Mac OS X v10.3 and earlier, you should implement `loadDataRepresentation:ofType:` and `dataRepresentationOfType:` instead.

6. For applications targeted for Mac OS X v10.4 and later, If your application needs access to document data files, you can override `readFromURL:ofType:error:` and `writeToURL:ofType:error:`, respectively, instead. You can override `writeToURL:ofType:forSaveOperation:originalContentsURL:error:` instead of

`writeToURL:ofType:error:` if your document writing machinery needs access to the on-disk representation of the document revision that is about to be overwritten. If your document data is stored in file packages, you can override `readFromFileWrapper:ofType:error:` and `fileWrapperOfType:error:` instead.

Here are examples of URL-based reading and writing implementations which have the same assumptions as the previous examples.

```
- (BOOL)readFromURL:(NSURL *)inAbsoluteURL ofType:(NSString *)inTypeName
error:(NSError **)outError {
    BOOL readSuccess = NO;
    NSAttributedString *fileContents = [[NSAttributedString alloc]
                                initWithURL:inAbsoluteURL options:nil
                                documentAttributes:NULL error:outError];
    if (fileContents) {
        readSuccess = YES;
        [self setText:fileContents];
        [fileContents release];
    }
    return readSuccess;
}

- (BOOL)writeToURL:(NSURL *)inAbsoluteURL ofType:(NSString *)inTypeName
error:(NSError **)outError {
    NSData *data = [[self text] RTFFromRange:NSMakeRange(0,
                    [[self text] length]) documentAttributes:nil];
    BOOL writeSuccess = [data writeToURL:inAbsoluteURL
                            options:NSAtomicWrite error:outError];
    return writeSuccess;
}
```

For applications that must be able to run on Mac OS X v10.3 and earlier, you should override `readFromFile:ofType:` and `writeToFile:ofType:` instead.

7. Implement the method to create the window controller or controllers for the `NSDocument` object.

   If your document has only one window, the project template provides a default implementation:

   ```
   - (NSString *)windowNibName {
       return @"MyDocument";
   }
   ```

   If your document has more than one window, or if you have a custom subclass of `NSWindowController`, override `makeWindowControllers` instead. Make sure you add each created window controller to the list of such objects managed by the document using `addWindowController:`. This method causes the document to retain the window controller object, so be sure to release it after you add it.

8. You can implement`windowControllerWillLoadNib:` and `windowControllerDidLoadNib:` to perform any necessary tasks related to the window before and after it is loaded from the nib file.

   Here is an example:

   ```
   - (void)windowControllerDidLoadNib:(NSWindowController *)windowController {
       [super windowControllerDidLoadNib:windowController];
       [textView setAllowsUndo:YES];
       if (fileContents != nil) {
           [textView setString:fileContents];
   ```

```
            fileContents = nil;
        }
    }
```

9. Mark the document's dirty flag when it is edited.

   The flag returned by `isDocumentEdited` indicates whether the document has unsaved changes. Although the `NSDocument` object clears this flag when it saves or reverts a document, you must set this flag in your code, unless you are using the `NSDocument` object's default undo/redo mechanism. Normally, you respond to the appropriate delegation or notification messages sent when users edit a document, then invoke `updateChangeCount:` with an argument of `NSChangeDone` to set the dirty flag.

10. Write the code that prints the document's data.

    If you want users to be able to print a document, you must override `printOperationWithSettings:error:`, possibly providing a modified `NSPrintInfo` object.

11. Register undo and redo groups in your code. See the class description of `NSUndoManager` for details.

And, of course, you must implement any methods that are special to your `NSDocument` subclass.

Implement Additional Controller Classes

If the default `NSWindowController` instance provided by the Application Kit does not meet the needs of your document-based application, you can create a custom subclass of it. If you do so, you must override the `NSDocument makeWindowControllers` method to instantiate this custom class and add the created object to the document's list of window controllers. You should also ensure that your NSWindowController subclass is the nib file's owner.

If the default `NSDocumentController` object somehow does not meet all of your requirements for an application controller, such as handling user preferences or responding to uncommon application delegate messages, usually you should create a separate controller object (instead of subclassing `NSDocumentController`). For information on implementing `NSDocumentController` and `NSWindowController` subclasses, refer to the appropriate class documentation. See also "Creating a Subclass of NSDocumentController" (page 27) and "Frequently Asked Questions" (page 53) in this document.

Creating a Subclass of NSDocument

Every application that takes advantage of the document architecture must create at least one subclass of `NSDocument`. This architecture requires that you override some `NSDocument` methods (among several choices), and recommends overriding several others in certain situations. This article also includes advice to consider when overriding these and other `NSDocument` methods, including `init` and `displayName`.

Required Method Overrides

The functional areas described by items in the following subsections require you to override `NSDocument` methods. You must override one reading and one writing method. In the simplest case, you can override the two data-based reading and writing methods described in the following subsection. If you need to deal with the location of the file, then you can override the URL reading and writing methods instead. If your application supports document files that are file packages, then you can override the file-wrapper reading and writing methods instead.

Note that these methods read and write entire files at once. If your application has a large data set, you may want instead to read and write pieces of your files at different times.

Data-based reading and writing methods

The `dataOfType:error:` method may be overridden to create and return document data (packaged as an `NSData` object) of a supported type, usually in preparation for writing that data to a file. The `readFromData:ofType:error:` method may be overridden to convert an `NSData` object containing document data of a certain type into the document's internal data structures and display that data in a document window. The `NSData` object usually results from the document reading a document file.

For applications that must be able to run on Mac OS X v10.3 and earlier, implement `dataRepresentationOfType:` and `loadDataRepresentation:ofType:` instead.

Location-based reading and writing methods

By default, the `writeToURL:ofType:error:` method writes data to a file after obtaining the data from the `fileWrapperOfType:error:` method, which gets it from the `dataOfType:error:` method. The `readFromURL:ofType:error:` method reads data from a file, creates an `NSFileWrapper` object from it, and gives this object to the `readFromFileWrapper:ofType:error:` method; if this object represents a simple file, it is passed to the `readFromData:ofType:error:` method for processing; otherwise (if the object represents a directory), the `readFromFileWrapper:ofType:error:` method can be overridden to handle the situation. Subclasses can override any of these methods instead of the data-based reading and writing methods if the way `NSDocument` reads and writes document data is not sufficient; their override implementations, however, must also assume the loading duties of the data-based reading and writing methods.

For applications that must be able to run on Mac OS X v10.3 and earlier, implement `writeToFile:ofType:`, `fileWrapperRepresentationOfType:`, `readFromFile:ofType:`, and `loadFileWrapperRepresentation:ofType:` instead.

Advice for overrides of reading and writing methods

Don't invoke `fileURL` (or `fileName`, the method was that deprecated in favor if it, in Mac OS X v10.4), `fileType`, or `fileModificationDate` from within your overrides. During reading, which typically happens during object initialization, there is no guarantee that `NSDocument` properties like the file's location or type have been set yet. Your overridden method should be able to determine everything it needs to do the reading from the passed-in parameters. During writing, your document may be asked to write its contents to a different location or using a different file type. Again, your overridden method should be able to determine everything it needs to do the writing from the passed-in parameters.

If your override cannot determine all of the information it needs from the passed-in parameters, consider overriding another method. For example, if you see the need to invoke `fileURL` from within an override of `readFromData:ofType:error:`, perhaps you should instead override `readFromURL:ofType:error:`. For another example, if you see the need to invoke `fileURL` from within an override of `writeToURL:ofType:error:`, perhaps you should instead override `writeToURL:ofType:forSaveOperation:originalContentsURL:error:`.

If your `NSDocument` subclass uses an `NSTextView` object to display its data, and it has an active `NSUndoManager` object, then your document-writing method override (or another override called during document saving) should send the `NSTextView` object a `breakUndoCoalescing` message. Sending this message when saving the text view's contents preserves proper tracking of unsaved changes and the document's dirty state. For more information about supporting undo in a document-based application, see "How do I implement undo?" (page 59). Message flow during document saving is described in "Saving a Document" (page 33).

Concurrent Document Opening

In Mac OS X v10.6 and later, `NSDocument` has the ability to read documents concurrently, using background threads. A class method of `NSDocument`, `canConcurrentlyReadDocumentsOfType:`, gives you control over this capability. Your `NSDocument` subclass can override this method to return `YES` to enable loading of documents concurrently, using background threads. When this facility is enabled in this way, `initWithContentsOfURL:ofType:error:` executes on a background thread when opening files via the Open panel or from the Finder. This allows concurrent reading of multiple documents and also allows the application to be responsive while reading a large document.

The default implementation of this method returns `NO`. A subclass override should return `YES` only for document types whose reading code can be safely executed concurrently, in non-main threads.

Code executed during the opening of a document triggered by an Apple event cannot get the current Apple event, because the event is suspended until all documents are read, to enable correct reporting of success or failure to the Apple event sender. So, for example, if you are checking the current Apple event for a search string, you should not enable concurrent document opening unless you have another solution for getting the search string.

The `NSUndoManager` mechanism for automatically creating one group per UI event doesn't coexist very well with `NSUndoManager` being used on non-main threads. So, you should disable undo registration during document reading, which is a good idea even in the absence of concurrency. For an example, see `SKTDocument.m` in `/Developer/Examples/Sketch`.

Optional Method Overrides

The functional areas described by items in the following bulleted list require method overrides in some situations.

- Window controller creation

  `NSDocument` subclasses must also create their window controllers. They can do this indirectly or directly. If a document has only one nib file with one window in it, the subclass can override `windowNibName` to return the name of the window nib file. As a consequence, a default `NSWindowController` instance is created for the document, with the document as the nib file's owner. If a document has multiple windows, or if an instance of a custom `NSWindowController` subclass is to be used, the `NSDocument` subclass must override `makeWindowControllers` to create these objects.

- Printing and page layout

  Normally, a document-based application can change the information it uses to define how document data is printed. This information is encapsulated in an `NSPrintInfo` object. If an application is to print document data, subclasses of `NSDocument` must override `printOperationWithSettings:error:`.

  Applications that must be able to run on Mac OS X v10.3 and earlier should override `printShowingPrintPanel:` instead.

  If your application does not support printing, be sure to remove the printing-related menu items from the main menu nib provided by the Cocoa Document-based Application template in Xcode.

- Making backup files

  When it saves a document, `NSDocument` creates a backup of the old file before it writes data to the new one. Backup files have the same name as the new file, but with a tilde just before the extension. Normally, if the write operation is successful, it deletes the backup file. Subclasses can override `keepBackupFile` to return `YES` and thus retain the most recent backup file.

- Modifying the Save panel accessory view

  By default, when `NSDocument` runs the Save panel, and the document has multiple writable document types, it inserts an accessory view near the bottom of the panel. This view contains a pop-up menu of the writable types. If you don't want this pop-up menu, override `shouldRunSavePanelWithAccessoryView` to return `NO`. You can also override `prepareSavePanel:` to do any further customization of the Save panel.

- Validating menu items

  `NSDocument` implements `validateUserInterfaceItem:` to manage the enabled state of the Revert and Save As menu items. If you want to validate other menu items, you can override this method, but be sure to invoke the superclass implementation. For more information on menu item validation, see the description of the `NSUserInterfaceValidations` protocol.

Initialization Issues

The initializers of `NSDocument` are another issue for subclasses. The `init` method is the designated initializer, and it is invoked by the other initializers `initWithType:error:` and `initWithContentsOfURL:ofType:error:`. If you need to perform initializations that must be done when creating new documents but should not be done when opening existing documents, override `initWithType:error:`. If you have any initializations that apply only to documents that are opened, you should override `initWithContentsOfURL:ofType:error:`. If you have general initializations, you should, of course, override `init`. In both cases, be sure to invoke the superclass implementation as the first thing.

If your application must be able to run on Mac OS X v10.3 and earlier, you can override `initWithContentsOfFile:` or `initWithContentsOfURL:` instead.

If you override `init`, make sure that your override never returns `nil`. Doing so could cause a crash (in some versions of the Application Kit) or presentation of a less than useful error message. If, for example, you want to prevent the creation or opening of documents under circumstances unique to your application, override a specific `NSDocumentController` method instead.

Customizing Document Window Titles

Subclasses of `NSDocument` sometimes override `displayName` to customize the titles of windows associated with the document. That is usually not the right thing to do because the document's display name is used in places other than the window title, and the custom value that an application might want to use as a window title is often not appropriate. For example, the document display name is used in the following places:

- Error alerts that may be presented during reverting, saving, or printing of the document
- Alerts presented during document saving if the document has been moved, renamed, or put in the trash
- The alert presented when the user attempts to close the document with unsaved changes
- As the default value shown in the "Save As:" field of Save panels.

To customize a document's window title properly, subclass `NSWindowController` and override `windowTitleForDocumentDisplayName:`. If your application requires even deeper customization, override `synchronizeWindowTitleWithDocumentName`.

Creating a Subclass of NSDocumentController

The `NSDocumentController` class keeps track of the first instance of `NSDocumentController` (or a custom subclass) that is created and returns that instance from its `sharedDocumentController` class method. To get your application to use your custom subclass of `NSDocumentController`, you must ensure your subclass is the first instance of `NSDocumentController` created when the application starts up. There are two ways to do this:

1. Create your subclass in the main nib file.

   The main nib file is loaded by the application when it starts up. If you create an instance of your subclass in the main nib file, the application loads it when it launches and uses it as the shared document controller. If the default `NSDocumentController` object was configured as your application delegate, then be sure to connect the application's delegate outlet to the instance of your subclass.

2. Create an instance of your subclass in the `applicationWillFinishLaunching:` method.

   The application does not ask for its shared document controller until after the `applicationWillFinishLaunching:` message is sent to its delegate. Therefore, you can create an instance of your subclass of `NSDocumentController` in your application delegate's `applicationWillFinishLaunching:` method and that instance will be set as the shared document controller.

Message Flow in the Document Architecture

The objects that form the document architecture interact to perform the activities of document-based applications, and those interactions proceed primarily through messages sent among the objects via public APIs. This provides many opportunities for you to customize the behavior of your application by overriding methods in your `NSDocument` subclass or other subclasses.

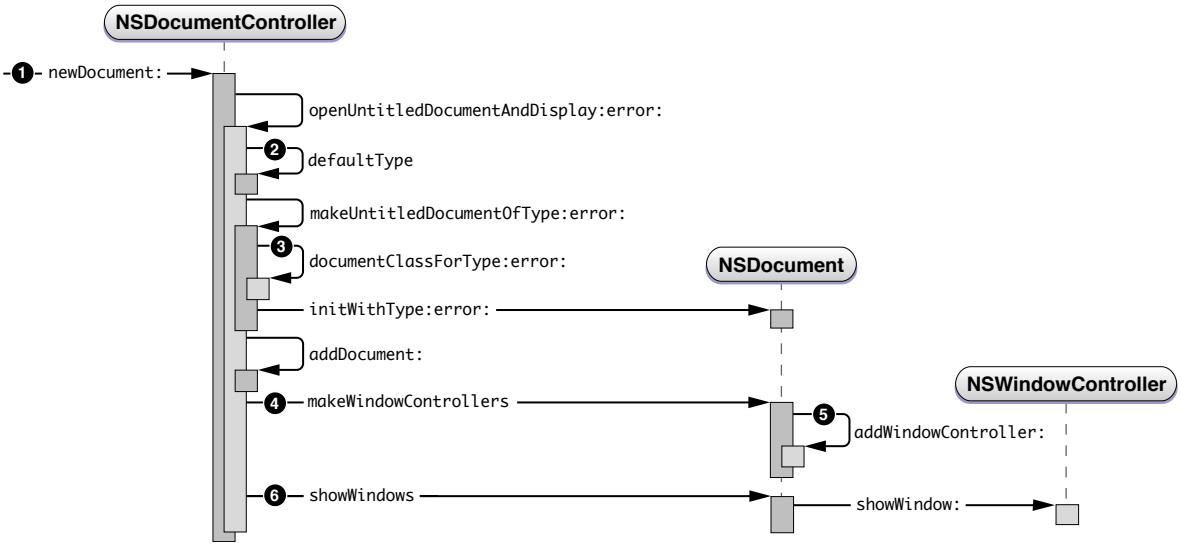
This article describes default message flow among major objects of the document architecture, as defined in Mac OS X v10.4, including objects sending messages to themselves; it leaves out various objects and messages peripheral to the main mechanisms.

This article doesn't cover code paths the document architecture takes to support Java, as well as older, Objective-C document-based applications that override methods deprecated in Mac OS X v10.4. (If those methods are overridden, the document architecture ensures that they are called, so their customizations still work properly. Of course, such applications don't benefit from the improvements introduced in Mac OS X v10.4, such as robust error handling.) Finally, these messages are sent by the default implementations of the methods in question, and the behavior of subclasses may differ.

Creating a New Document

The document architecture creates a new document when the user chooses New from the File menu of a document-based application. This action begins a sequence of messages among the `NSDocumentController` object, the newly created `NSDocument` object, and the `NSWindowController` object, as shown in Figure 2.

Figure 2 Creating a new document



The sequence numbers in Figure 2 refer to the following steps in the document-creation process:

1. The user chooses New from the File menu, causing the `newDocument:` message to be sent to the document controller (or an Apple event, for example, sends an equivalent message).
2. The `openUntitledDocumentAndDisplay:error:` method determines the default document type (stored in the application's `Info.plist` file) and sends it with the `makeUntitledDocumentOfType:error:` message.
3. The `makeUntitledDocumentOfType:error:` method determines the `NSDocument` subclass corresponding to the document type, instantiates the document object, and sends it an initialization message.
4. The document controller adds the new document to its document list and, if the first parameter passed with `openUntitledDocumentAndDisplay:error:` is `YES`, sends the document a message to create a window controller for its window, which is stored in its nib file. The `NSDocument` subclass can override `makeWindowControllers` if it has more than one window.
5. The document adds the newly created window controller to its list of window controllers by sending itself an `addWindowController:` message.
6. The document controller sends the document a message to show its windows. In response, the document sends the window controller a `showWindow:` message, which makes the window main and key.

If the first parameter passed with `openUntitledDocumentAndDisplay:error:` is `NO`, the document controller needs to explicitly send the document `makeWindowControllers` and `showWindows` messages to display the document window.

Opening a Document

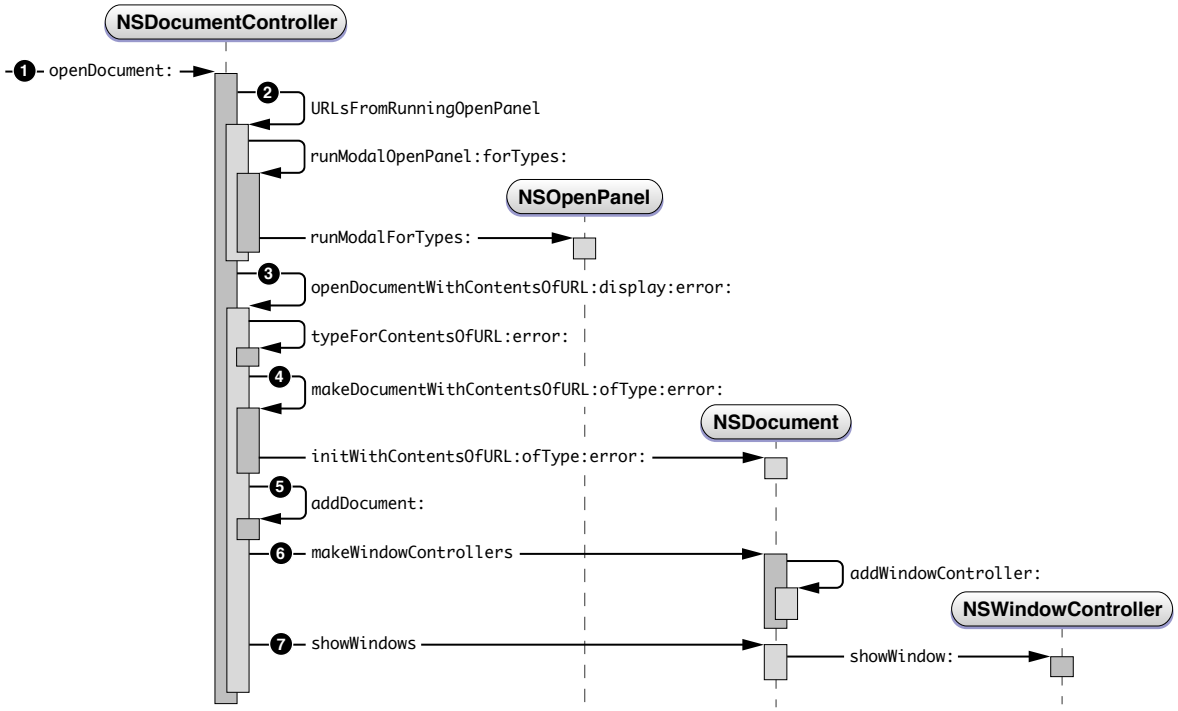
The document architecture opens a document, reading its contents from a file, when the user chooses Open from the File menu. This action begins a sequence of messages among the `NSDocumentController`, `NSOpenPanel`, `NSDocument`, and `NSWindowController` objects, as shown in Figure 3 (page 31).

There are many similarities between the mechanisms for opening a document and creating a new document. In both cases the document controller needs to create and initialize an `NSDocument` object, using the proper `NSDocument` subclass corresponding to the document type, the document controller needs to add the document to its document list, and the document needs to create a window controller and tell it to show its window.

Document Opening Message Flow

Opening a document differs from creating a new document in several ways. If document opening was invoked by the user choosing Open from the File menu, the document controller must run an Open panel to allow the user to select a file to provide the contents of the document. An Apple event can invoke a different message sequence. In any case, the document must read its content data from a file and keep track of the file's meta-information, such as its URL, type, and modification date.

Figure 3 Opening a document



The sequence numbers in Figure 3 refer to the following steps in the document-opening process:

1. The user chooses Open from the File menu, causing the `openDocument:` message to be sent to the document controller.

2. The URL locating the document file must be retrieved from the user, so the `NSDocumentController` object sends itself the `URLsFromRunningOpenPanel` message. After this method creates the Open panel and sets it up appropriately, the document controller sends itself the `runModalOpenPanel:forTypes:` message to present the Open panel to the user. The `NSDocumentController` object sends the `runModalForTypes:` message to the `NSOpenPanel` object.

3. With the resulting URL, the `NSDocumentController` object sends itself the `openDocumentWithContentsOfURL:display:error:` message.

4. The `NSDocumentController` object sends itself the `makeDocumentWithContentsOfURL:ofType:error:` message and sends the `initWithContentsOfURL:ofType:error:` message to the newly created `NSDocument` object. This method initializes the document and reads in its contents from the file located at the specified URL. "Document Initialization Message Flow" (page 32) describes document initialization in this context.

5. When `makeDocumentWithContentsOfURL:ofType:error:` returns an initialized `NSDocument` object, the `NSDocumentController` object adds the document to its document list by sending the `addDocument:` message to itself.

6. To display the document's user interface, the document controller sends the `makeWindowControllers` message to the `NSDocument` object, which creates an `NSWindowController` instance and adds it to its list using the `addWindowController:` message.

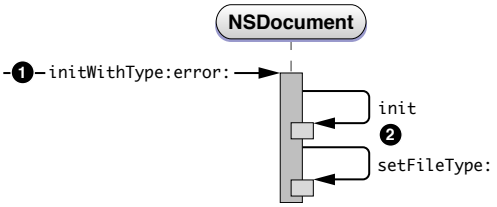
7. Finally, the document controller sends the `showWindows` message to the `NSDocument` object, which, in turn, sends the `showWindow:` message to the `NSWindowController` object, making the window main and key.

8. If the `URLsFromRunningOpenPanel` method returned an array with more than one URL, steps 3 through 7 repeat for each URL returned.

Document Initialization Message Flow

Initialization of the `NSDocument` subclass object is typical. Steps in the document-initialization process for document creation are shown in Figure 4. Document initialization in the context of document opening is noteworthy because it invokes the document's location-based or data-based reading and writing methods, and you must override one of them. Steps in the document-initialization process for document opening are shown in Figure 5.

Figure 4 Document initialization for document creation

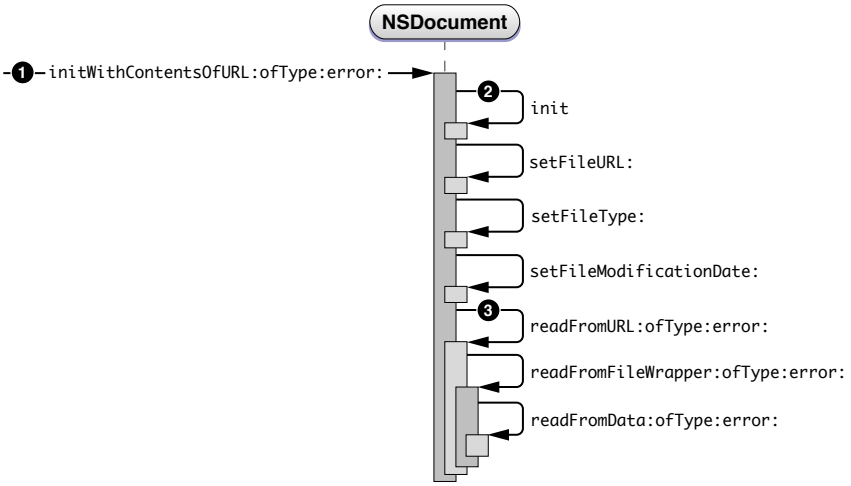


The sequence numbers in Figure 4 refer to the following steps in the document-initialization process:

1. The `NSDocumentController` object begins document initialization by sending the `initWithType:error:` message to the newly created `NSDocument` object.

2. The `NSDocument` object sends the `init` message to itself, invoking its designated initializer, then sets its filetype by sending itself the message `setFileType:`.

Figure 5 Document initialization for document opening



The sequence numbers in Figure 5 refer to the following steps in the document-opening process:

1. The `NSDocumentController` object begins document initialization by sending the `initWithContentsOfURL:ofType:error:` message to the newly created `NSDocument` object.

2. The `NSDocument` object sends the `init` message to itself, invoking its designated initializer, then sets its metadata about the file it is about to open by sending itself the messages `setFileURL:`, `setFileType:`, and `setFileModificationDate:`.

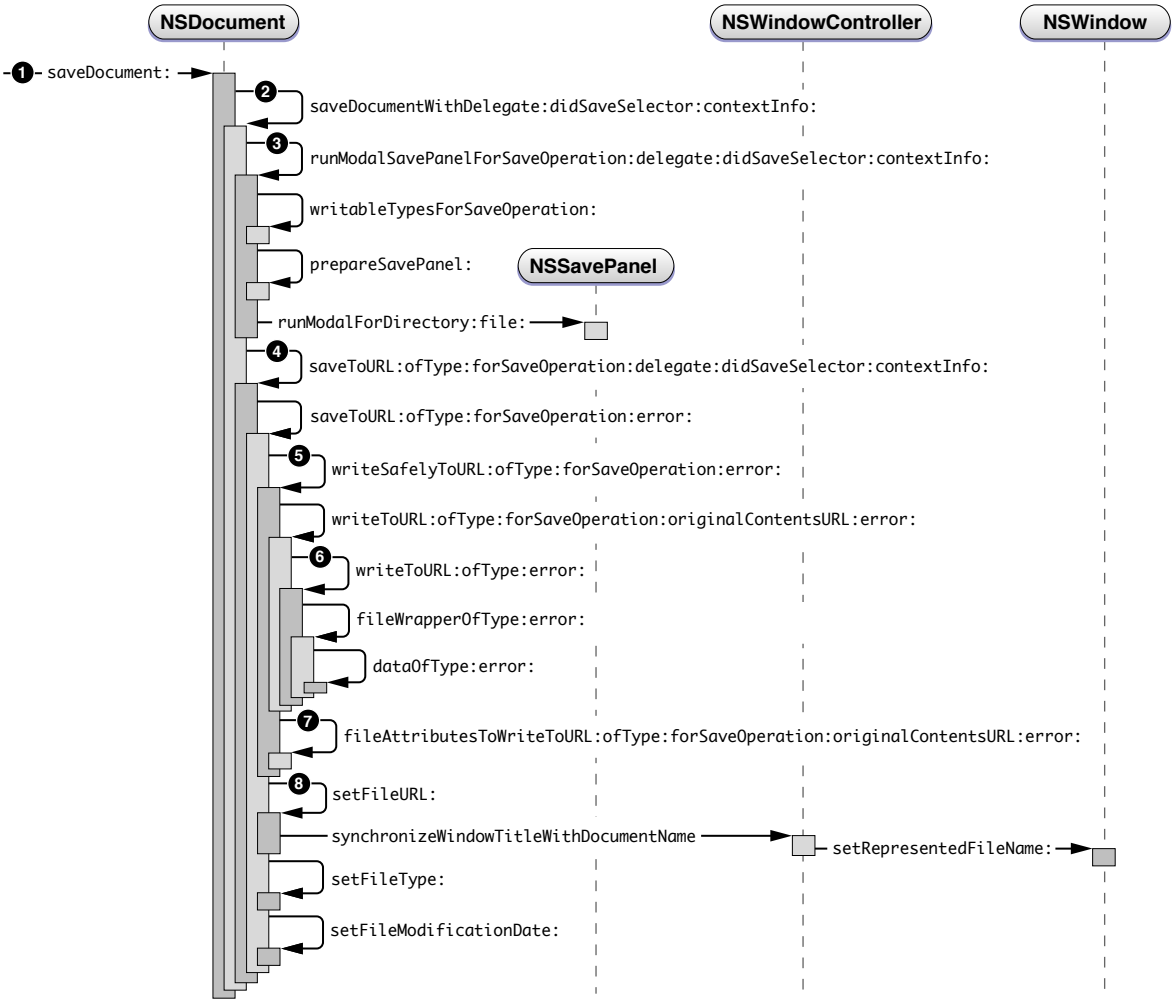
3. The `NSDocument` object reads the contents of the file by sending the `readFromURL:ofType:error:` message to itself. That method gets a file wrapper from disk and reads it by sending the `readFromFileWrapper:ofType:error:` message to itself. Finally, the `NSDocument` object puts the file contents into an `NSData` object and sends the `readFromData:ofType:error:` message to itself.

   Your `NSDocument` subclass *must* override one of the three document-reading methods (`readFromURL:ofType:error:`, `readFromData:ofType:error:`, or `readFromFileWrapper:ofType:error:`) or every method that may invoke `readFromURL:ofType:error:`.

Saving a Document

The document architecture saves a document—writes its contents to a file—when the user chooses one of the Save commands or Export from the File menu. Saving is handled primarily by the document object itself. Steps in the document-saving process are shown in Figure 6.

Figure 6 Saving a document



The sequence numbers in Figure 6 refer to the following steps in the document-saving process:

1. The user chooses Save from the File menu, causing the `saveDocument:` message to be sent to the `NSDocument` object.

2. The `NSDocument` object sends the `saveDocumentWithDelegate:didSaveSelector:contextInfo:` message to itself.

   If the document has never been saved, or if the user has moved or renamed the document file, then the `NSDocument` object runs a modal Save panel to get the file location under which to save the document, as it does immediately if the user chooses Save As or Save To from the File menu.

3. To run the Save panel, the `NSDocument` object sends the `runModalSavePanelForSaveOperation:delegate:didSaveSelector:contextInfo:` message to itself. The document sends `prepareSavePanel:` to itself to give subclasses an opportunity to customize the Save panel, then sends `runModalForDirectory:file:` to the Save panel object.

4. The `NSDocument` object sends the `saveToURL:ofType:forSaveOperation:delegate:didSaveSelector:contextInfo:` and, in turn, `saveToURL:ofType:forSaveOperation:error:` to itself.

5. The `NSDocument` object sends the `writeSafelyToURL:ofType:forSaveOperation:error:` message to itself. The default implementation either creates a temporary directory in which the document writing should be done, or renames the old on-disk revision of the document, depending on what sort of save operation is being done, whether or not there's already a copy of the document on disk, and the capabilities of the file system to which writing is being done. Then it sends the `writeToURL:ofType:forSaveOperation:originalContentsURL:error:` message to the document.

6. To write the document contents to the file, the `NSDocument` object sends itself the `writeToURL:ofType:error:` message, which by default sends the document the `fileWrapperOfType:error:` message. That method, in turn, sends the document the `dataOfType:error:` message to create an `NSData` object containing the contents of the document. (For backward compatibility, if the deprecated `dataRepresentationOfType:` is overridden, the document sends itself that message instead.)

   The `NSDocument` subclass *must* override one of its document-writing methods (`dataOfType:error:`, `writeToURL:ofType:error:`, `fileWrapperOfType:error:`, or `writeToURL:ofType:forSaveOperation:originalContentsURL:error:`).

7. The `NSDocument` object sends the `fileAttributesToWriteToURL:ofType:forSaveOperation:originalContentsURL:error:` message to itself to get the file attributes, if any, which it writes to the file. The method then moves the just-written file to its final location, or deletes the old on-disk revision of the document, and deletes any temporary directories. In addition, the `NSDocument` object sends the `keepBackupFile` message to itself. Subclasses can override this method, which returns `NO` by default, if they want to retain the backup files created before the document writes its data to the file.

8. The `NSDocument` object updates its location, file type, and modification date by sending itself the messages `setFileURL:`, `setFileType:`, and `setFileModificationDate:` if appropriate.

Window Closing Behavior

The document architecture automates memory management for documents and their associated windows and window controllers. The general behavior is as follows:

- When the last window of a document is closed, the document is also closed. The window, window controller, and document are all released.
- When a primary window of a document is closed, the document is also closed, as are any secondary document windows. The document's windows, the document's window controllers, and the document itself are released.
- When a secondary window of a document is closed, only that window is closed. The window controller is removed from the document's list of window controllers. The secondary window and its window controller are released.

One way to think of it is that the document controller works to ensure that a document is open and taking up memory only if it has a visual representation in the user interface (that is, an open window), and that windows are only taking up memory as long as they are being displayed. If a user closes a secondary document window, there is no need to go through the overhead of updating the user interface in that window when it's possible that the user will never again bring up that window for that document.

A primary document window is one that needs to be open if the document is open. You identify a primary document window by telling the custom window controller for that window type that it should close its associated document on close, and you do so by sending it a `setShouldCloseDocument:` message.

Windows work with their associated window controllers, which work with their associated documents, which work with the shared document controller together to implement this behavior. Specifically, when a window closes, it tells its window controller that it's closing. The window controller tells its document that it's closing. The document notes that the window controller is closing, and removes the window controller from the document's list of window controllers. As this is the only place the window controller was retained, the window controller gets released and deallocated as a result.

When a window controller does not have an associated document, the default behavior is different. A window controller not owned by a document must be retained by some other object. When the window closes, that other object is still retaining the window controller. Because of this, the window controller is not deallocated. Because the window controller is not deallocated, neither is the window. This is desired behavior for a window controller that manages a window such as an About box.

If you want closing a window to deallocate the window and its window controller when it isn't owned by a document, you should add code to the object that does own the window controller to observe the `NSWindow` notification `NSWindowWillCloseNotification` or, as window delegate, implement the `windowWillClose:` method. When you see that your window controller's window is closing, you can autorelease the window controller, which will also have the effect of releasing the window and any other top-level objects loaded from the window controller's nib file when the application reenters the main event loop.

Another way to think of the distinction between a window controller owned by a document and one owned by some other object is to consider that window controllers for documents are replicated, whereas window controllers for special windows like About boxes or Info windows are typically unique in your application.

Every time your application creates a document object (whether by creating an entirely new document or loading one from a file), that document needs an entirely new set of window controllers. When the document goes away, those window controllers are no longer needed and should also go away. So your application needs to be able to create as many document window controllers as it has documents. Conversely, your application has only one About box. So it needs only a single window controller to manage the About box. If you expect a window to be used again, you should consider leaving the window controller in memory for performance reasons. If you don't expect a window to be used again, you can delete it to reduce your application's memory footprint. For more information on performance, see *Memory Usage Performance Guidelines* and *Code Size Performance Guidelines*.

Window Controllers and Nib Files

When you create a window controller with an associated nib file the window controller assumes responsibility for all aspects of managing that nib file. This includes:

- Loading the nib file

  When the window controller is asked to do something, it first loads its window by loading the window controller's nib file. When it loads the nib file, the window controller sets itself as the nib file's owner.

- Freeing top-level objects when the window controller is deallocated

  As the nib file's owner, the window controller is responsible for freeing any top-level objects instantiated in the nib file. This includes the window itself and any additional objects you might have added to the nib file. The window controller automatically keeps track of these objects when it loads the nib file, and then releases them when the window controller is deallocated.

For a window controller to be able to load and use its window from a nib file, the nib file needs to have certain objects configured properly, with their outlets connected to the appropriate objects.

- File's owner

  When a window controller loads a nib file, it sets itself as the owner of that nib file. To enable connections from the window controller to other objects in the nib file, you need to set the file's owner of the nib file that will be managed by a custom window controller to the class of that custom window controller.

- `window` outlet

  Window controllers keep track of their window using their `window` outlet. The `window` outlet of your window controller (set as the file's owner in your nib file) should be connected to the window your window controller is responsible for.

- `delegate` outlet

  While not required, it's often convenient to set up your window controller as the delegate of the window it manages. In your nib file, connect the `delegate` outlet of the window your window controller is managing to the object that represents your window controller—specifically, the file's owner object.

  Note: `NSWindowController` does not depend on being the controlled window's delegate to do its job, and it doesn't implement any `NSWindow` delegate methods. A subclass of `NSWindowController`, however, is a fine place to put implementations of `NSWindow` delegate methods, and if you do so you'll probably need to connect the delegate outlet of the window to the nib file's owner as described, but you do not have to do so for `NSWindowController` itself to work properly.

For more information, see "How can I make an NSWindowController subclass that automatically uses a particular nib file?" (page 58).

Storing Document Types Information in the Application's Property List

Applications use an information property list file, which is stored in the application's bundle and named, by default, `Info.plist`, to specify various information that can be used at runtime. Document-based applications use this property list to specify the document types the application can edit or view. This information is used by the application and system entities such as the Finder and Launch Services (an API for launching applications in Mac OS X) as well.

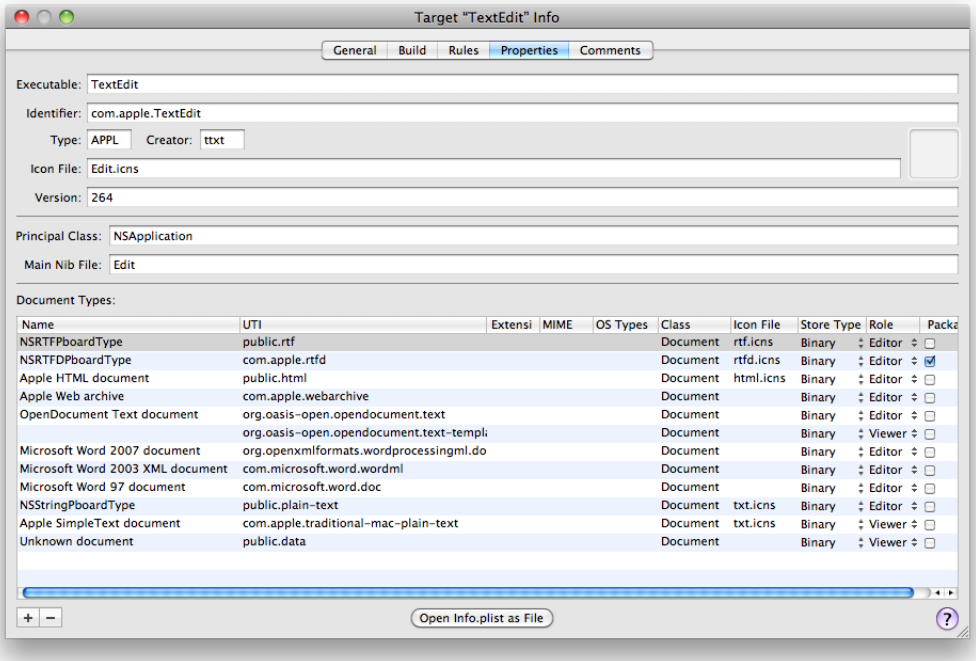
For example, when the `NSDocumentController` object creates a new document or opens an existing document, it searches the property list for such items as the document class that handles a document type, the uniform type identifier (UTI) for the type, and whether the application can edit or only view the type. (Similarly, Launch Services may use information about the icon file for the type.) Supplying this information in a property list allows an application to support the Open and Save panels with little effort.

Document types information is associated with the `CFBundleDocumentTypes` key as an array of dictionaries, each of which contains the key-value pairs that define the document type. For a list of possible keys and their values, see Property List Key Reference.

Although you can create and edit information property lists directly in the Property List Editor application (located in `/Developer/Utilities/Applications`), Xcode also provides convenient options for viewing and editing document types information (and other property list information). Figure 1 shows the Properties pane of the Xcode inspector for the TextEdit target. In this view, you can select and edit any of the document types as well as add and delete entries. You can also edit the property list file in the plain text XML format shown in Listing 1 (page 44). The property list file for the TextEdit application is named `Info-TextEdit.plist`; the name can be specified among the target's build settings. (Complete source code for TextEdit is available in `/Developer/Examples/TextEdit`.)

Note: The Properties pane is visible only for targets that create products with `Info.plist` files, and you cannot configure `Info.plist` entries for legacy targets, such as Jam-based Project Builder targets, in the target inspector. To edit the `Info.plist` entries for Jam-based targets in Xcode, select the target in the Groups & Files list and double-click the target to launch the target editor.

Figure 1 Document types information for TextEdit application in Xcode



TextEdit defines a subclass of `NSDocument` named `Document` to handle all of its document types, as shown in the Class column in Figure 1 (page 42).

The top section of the Properties pane allows you to edit basic information about the product, such as the name of the associated executable, the identifier, type and creator, version information, and an icon to associate with the finished product. The name of the icon here must match the name of an icon file (extension `.icns`) that resides in the Resources folder of the product bundle.

The Principal Class and Main Nib File options are specific to Cocoa applications and bundles, and Automator actions. The Principal Class field corresponds to the information property list key `NSPrincipalClass`. The Main Nib File field specifies the nib file that's automatically loaded when the application is launched. It corresponds to the information property list key `NSMainNibFile`.

The Document Types table allows you to specify which documents your finished product can handle. You can add and remove document types from this list using the plus and minus buttons. You should list the application's primary document type first because the document controller uses that type by default when the user requests a new document. Here is what's in the Document Types table:

- Name. The name of the document type. The corresponding property list key is `CFBundleTypeName`. In Mac OS v10.5 and later, any document type that specifies a UTI (which uses the `LSItemContentTypes` key in XML), the UTI is used as the programmatic type name by `NSDocument` and

`NSDocumentController`, instead of the value of any `CFBundleTypeName` subentry that might also be present. (If the application is linked against Mac OS X v10.4 or earlier, or if a UTI is not specified, the `CFBundleTypeName` subentry specifies the document type in APIs such as the `NSDocumentController` method `makeUntitledDocumentOfType:error:`.)

- UTI. A list of Uniform Type Identifier (UTI) strings for the document. UTIs are strings that uniquely identify abstract types, providing a consistent identifier for data that all applications and services can recognize and rely upon. By using UTIs, applications can avoid the complexity required to handle the disparate kinds of file type information on the system, including file extensions, MIME types, and HFS type codes (OS Types).

  In Mac OS X v10.5 and later, if a document type specifies a UTI, the Cocoa and LaunchServices frameworks ignore sibling entries specifying file extensions, MIME types, and HFS type codes (OS Types). For more information on UTIs, see *Uniform Type Identifiers Overview* and the header file `UTType.h`, available as part of `LaunchServices.framework` in Mac OS X v10.3 and later.
- Extensions. A list of the filename extensions for this document type. The period is not included with the extension. Using UTIs eliminates the need to list extensions in the application's property list file. The corresponding property list key is `CFBundleTypeExtensions`.
- MIME Types. A list of the Multipurpose Internet Mail Extensions (MIME) types for the document. MIME types identify content types for Internet applications. Using UTIs eliminates the need to list MIME types in the application's property list file. The corresponding property list key is `CFBundleTypeMIMETypes`.
- OS Types. A list of four-letter codes for the document. These codes are stored in the document's resources or information property list files. Using UTIs eliminates the need to list OS Types in the application's property list file. The corresponding property list key is `CFBundleTypeOSTypes`.
- Class. The subclass of `NSDocument` that this document type uses. Specifying the document class is important because when opening a document, the `NSDocumentController` object can use the information to create instances of the `NSDocument` subclass appropriate to a data type. As a result, you don't have to allocate and initialize your subclass of `NSDocument` explicitly in your code; it is done automatically for you.
- Icon File. The name of the file that contains the document type's icon.
- Store Type. The format of persistent data store used for the document.
- Role. A description of how the application uses the documents of this type. You can choose from four values:
  - Editor. The application can display, edit, and save documents of this type.
  - Viewer. The application can display, but not write, documents of this type. It may also specify this role to to indicate that it is not a primary handler of this type, even though it can write it.
  - Shell. The application provides runtime services for other processes—for example, a Java applet viewer.
  - None. The application can neither display nor edit documents of this type but instead uses them in some other way. For example, an application may not understand the data but can declare other information about the type, as when the Finder declares the icon for fonts, or it can use this role to declare types it can export but not read.
- Package. Specifies whether the document is a single file or a file package—that is, a directory that is treated as a single file by certain applications, such as the Finder.

To edit a document type, click the type's line in the Document Types list and double-click the individual fields in each column to add or change the document type information.

Listing 1 shows some of the information for one of the document types from the information property list file for the TextEdit application in text-only XML format. The property list is considerably simplified by the use of the UTI for the document type, which is listed under the key `LSItemContentTypes`.

This listing omits additional document types handled by the TextEdit application, which are shown in the Xcode inspector view in Figure 1 (page 42).

Listing 1 Document types information for TextEdit application in XML format

```
<key>CFBundleDocumentTypes</key>
    <array>
        <dict>
            <key>CFBundleTypeIconFile</key>
            <string>rtf.icns</string>
            <key>CFBundleTypeName</key>
            <string>NSRTFPboardType</string>
            <key>LSItemContentTypes</key>
            <array>
                <string>public.rtf</string>
            </array>
            <key>CFBundleTypeRole</key>
            <string>Editor</string>
...
            <key>NSDocumentClass</key>
            <string>Document</string>
        </dict>
...
```

Saving HFS Type and Creator Codes

Hierarchical file system (HFS) type and creator codes are used by applications such as the Finder and Launch Services to identify document files with their associated applications, icons, and so on. By default, applications based on `NSDocument` do not save HFS type and creator codes in documents. To set the type and creator codes, your `NSDocument` subclass can override `fileAttributesToWriteToURL:ofType:forSaveOperation:originalContentsURL:error:` to add type and creator codes for the `NSFileHFSTypeCode` and `NSFileHFSCreatorCode` attributes, respectively.

If you want to set the type and creator codes for a file, independent of `NSDocument`, use the `NSFileManager` method `changeFileAttributes:atPath:`.

Listing 1 shows the `fileAttributesToWriteToURL:ofType:forSaveOperation:originalContentsURL:error:` method of an `NSDocument` subclass. This implementation assumes that the `NSDocument` subclass has previously set its type and creator code constants in a manner such as:

```
const OSType kMyAppCreatorCode = 'Blah';
```

You can modify this fragment to achieve the type and creator code behavior you want in your application.

Listing 1 Saving HFS type and creator information

```
- (NSDictionary *)fileAttributesToWriteToURL:(NSURL *)absoluteURL
    ofType:(NSString *)typeName
    forSaveOperation:(NSSaveOperationType)saveOperation
    originalContentsURL:(NSURL *)absoluteOriginalContentsURL
    error:(NSError **)outError
{
    NSMutableDictionary *fileAttributes =
            [[super fileAttributesToWriteToURL:absoluteURL
             ofType:typeName forSaveOperation:saveOperation
             originalContentsURL:absoluteOriginalContentsURL
             error:outError] mutableCopy];
   [fileAttributes setObject:[NSNumber numberWithUnsignedInt:kMyAppCreatorCode]
        forKey:NSFileHFSCreatorCode];
   [fileAttributes setObject:[NSNumber numberWithUnsignedInt:kMyDocumentTypeCode]
        forKey:NSFileHFSTypeCode];
    return [fileAttributes autorelease];
}
```

Creating Multiple-Document-Type Applications

The document architecture provides support for applications that handle multiple types of documents, each type using its own subclass of `NSDocument`. For example, AppleWorks allows you to create text documents, spreadsheets, and other types of documents, all in a single application. Such different document types each require a different user interface encapsulated in a unique `NSDocument` subclass. To integrate multiple subclasses into your document-based application, you can configure your nib files, `Info.plist` file, and document controller as described in this article.

You create a document-based application in Xcode by selecting the Cocoa Document-based Application template in the New Project panel. Xcode provides an `NSDocument` subclass with a window already configured in a nib file to display the document. If you create additional nib files and additional `NSDocument` subclasses, however, you must configure them manually.

To create a new `NSDocument` subclass, create the class interface and implementation of primitive reading and writing methods as described in "Creating a Subclass of NSDocument" (page 23).

Use Interface Builder to design the user interface for your document, which is stored in the nib file for the document window. Implement the `windowNibName` method in your `NSDocument` subclass to return the name of the nib file.

In a new window nib file created in Interface Builder, the file's owner is not automatically configured to the the `NSDocument` subclass, but is instead set to `NSObject`. So, you must configure the file's owner of the nib file to be your `NSDocument` subclass or, if you subclass `NSWindowController` to manage your user interface, configure the file's owner of the nib file to be your `NSWindowController` subclass. Then you must set the window outlet of the file's owner object to be your window. Otherwise, the window won't display when you create a new document instance from your subclass. Additionally, set the window's delegate outlet to the file's owner.

You also must configure the `Info.plist` file to describe your `NSDocument` subclass under the `CFBundleDocumentTypes` key, as described in "Storing Document Types Information in the Application's Property List" (page 41).

If your multiple-document-type application opens only existing documents, you can use the default `NSDocumentController` instance that is automatically created by the Application Kit, because the document type is determined from the file being opened. However, if your application creates new documents, it needs to choose the correct type.

The `NSDocumentController` action method `newDocument:` creates a new document of the first type listed in the application's array of document types (as configured in the `Info.plist` file). But this does not work for applications that want to support several distinct types of document. If your application cannot determine which type to create depending on circumstances, you must provide a user interface allowing the user to choose which type of document to create.

You can create your own new actions, either in your application's delegate or in an `NSDocumentController` subclass. You could create several action methods and have several different New menu items, or you could have one action that asks the user to pick a document type before creating a new one.

Once the user selects a type, your action method can use the `NSDocumentController` method `makeUntitledDocumentOfType:error:` to create a document of the correct type. After creating the document, your method should add it to the document controller's list of documents, and it should send the document `makeWindowControllers` and `showWindows` messages, as described in "Message Flow in the Document Architecture" (page 29).

Alternatively, if you subclass `NSDocumentController`, you can override the `defaultType` method to determine the document type and return it, when the user chooses New from the File menu.

Autosaving in the Document Architecture

In Mac OS X v10.4 and later, the document-based application architecture supports autosaving documents. The support takes the form of application behavior and `NSDocument` and `NSDocumentController` methods that you can call or override to customize the behavior.

Autosaving Behavior

Autosaving is the mechanism by which the document architecture automatically saves a document to disk, at some regular interval, while it is being edited by the user. Autosaving protects the user from data loss in case of power failure, application crashes, and so on. By default, autosaving is not turned on in the document architecture, but you can easily turn it on by sending a single message to your application's `NSDocumentController` object.

To turn on autosaving in a document-based application, you need only send a `setAutosavingDelay:` message to the `NSDocumentController` object with a parameter value greater than `0`. You can customize autosaving behavior by overriding autosaving-related methods in `NSDocument` and `NSDocumentController`.

During normal operation of an application with autosaving turned on, the document architecture autosaves untitled documents, by default, in the folder `~/Library/Autosave Information/` until the user saves the document. Thereafter, by default, autosaved documents go in the same folder where the user saved the document. When a document is saved by the user or the application quits normally, autosaved documents are deleted.

When an application with autosaving turned on crashes or otherwise quits abnormally, its autosaved documents retain on disk all of the content changes made to the time of the last autosave. When the application is relaunched, it automatically reopens its autosaved documents.

NSDocumentController Autosaving Methods

`NSDocumentController` has four methods related to autosaving. One method turns autosaving on and sets its interval. By invoking or overriding the other methods, you can customize autosaving behavior. Two methods relate to when documents are autosaved. Two other methods enable you to customize what is done when autosaved documents are reopened at application launch time.

The method that controls when (and whether) documents are autosaved is `setAutosavingDelay:`, which sets the time interval, in seconds, for periodic autosaving. This time interval is the time the document controller waits between detecting that a document has unsaved changes and sending the document an `autosaveDocumentWithDelegate:didAutosaveSelector:contextInfo:` message. A value of `0` indicates that autosaving is not done at all. By default, the value is `0`, so autosaving is off. The `autosavingDelay` method returns the current autosaving time interval.

The `NSDocumentController` methods you can use to customize reopening of autosave documents are `reopenDocumentForURL:withContentsOfURL:error:`, which the document controller invokes to reopen its autosaved documents, and `makeDocumentForURL:withContentsOfURL:ofType:error:`, which determines the class of document to instantiate, allocates a document object, and sends it an `initForURL:withContentsOfURL:ofType:error:` message.

NSDocument Methods

`NSDocument` also has methods related to autosaving. You don't need to use them to enable or configure autosaving in your application, but you can invoke or override them if you want to customize autosaving behavior.

The `initForURL:withContentsOfURL:ofType:error:` method does the actual initialization of the document object during reopening of an autosaved document. This method initializes the document located by a specified URL but reads the contents from another URL, where the autosaved file is located.

You can use the `setAutosavedContentsFileURL:` method to set the location to which documents are autosaved, and `autosavedContentsFileURL` returns that location.

The `autosaveDocumentWithDelegate:didAutosaveSelector:contextInfo:` method autosaves the document. The default implementation of this method figures out where the document should be autosaved and calls the `NSDocument` method `saveToURL:ofType:forSaveOperation:delegate:didSaveSelector:contextInfo:` to perform the save operation. If you override that method or any `NSDocument` method that writes to disk and contains a URL parameter, take care to write the file just where the URL parameter specifies. You should not assume that there is a relationship between the document and the value returned by `fileURL`.

You can test the current state of autosaving for a document by sending it the `hasUnautosavedChanges` message, which returns `YES` if the document has changes that have not been autosaved.

The `autosavingFileType` method returns the document type that should be used for an autosave operation. You can override this method to return `nil` to completely disable autosaving of an individual document. You can also override this method to return a special document type if, for example, your application defines a document type specifically designed for autosaving, such as one that efficiently represents document changes instead of complete document contents.

Error Handling in the Document Architecture

In Mac OS X v10.4 and later, the document architecture has greatly improved support for error handling.

Many `NSDocument` and `NSDocumentController` methods introduced in Mac OS X v10.4 include as their last parameter an indirect reference to an `NSError` object. These are methods that create a document, write a file, access a resource, or perform a similar operation. In most cases, those methods replace older methods that did not take error arguments and which are now deprecated.

Two examples of `NSDocumentController` methods that take error parameters are `openUntitledDocumentAndDisplay:error:`, which creates a new untitled document, and `openDocumentWithContentsOfURL:display:error:`, which opens a document located by an URL. In case of failure, these methods directly return `nil` and, in the last parameter, indirectly return an `NSError` object that describes the error. Before calling such a method, client code that is interested in a possible error declares an `NSError` object variable and passes the address of the variable in the error parameter. If they are not interested in the error, clients just pass `NULL` in the error parameter.

Using `NSError` objects gives Cocoa applications the capability to present much more useful error messages to the user, including detailed reasons for the error condition, suggestions for recovery, and even a mechanism for attempting programmatic recovery. In addition, the Application Kit handles presenting the error to the user.

For detailed information about `NSError` handling see *Error Handling Programming Guide*.

To take advantage of `NSError` handling in the document architecture, override methods that take `NSError` parameters such as `readFromURL:ofType:error:` instead of their deprecated counterparts such as `readFromURL:ofType:`. You should also remove from your code overrides of the deprecated `NSDocument` and `NSDocumentController` methods because, for backward compatibility, the document architecture invokes those methods if they are overridden, so the error-handling methods are not called.

Important: Cocoa methods that take error parameters in the Cocoa error domain are guaranteed to return `NSError` objects. So, if you override such a method, you must adhere to the following rule: A method that takes an `error:(NSError **)outError` argument must set the value of `*outError` to point to an `NSError` object whenever the method returns a value that signals failure (typically `nil` or `NO`) and `outError != NULL`.

If you override such a method to prevent some action, but you don't want an error alert to be presented to the user, return an error object whose domain is `NSCocoaErrorDomain` and whose code is `NSUserCancelledError`. The Application Kit presents errors through the `NSApplication` implementations of the `presentError:` and `presentError:modalForWindow:delegate:didPresentSelector:contextInfo:` methods declared by `NSResponder`. Those implementations silently ignore errors whose domain is `NSCocoaErrorDomain` and whose code is `NSUserCancelledError`. So, for example, if your override of `openDocumentWithContentsOfURL:display:error:` wanted to avoid presenting an error to the user, it could set error object as shown in the following fragment:

```
if (outError) {
```

```
    *outError = [NSError errorWithDomain:NSCocoaErrorDomain
        code:NSUserCancelledError userInfo:nil];
}
```

Cocoa memory management rules dictate that the invoker of this method is not responsible for releasing the `NSError` object, but the `errorWithDomain:code:userInfo:` class factory method returns an autoreleased object, so this fragment is correct. If, on the other hand, you call the superclass implementation, you don't need to set `outError` because the `NSDocumentController` default implementation follows the rules as well. Simply pass it the error argument your override received when invoked.

Frequently Asked Questions

This article answers commonly asked questions about the document-handling classes in the Application Kit. This includes the `NSDocument` class as well as `NSDocumentController` and `NSWindowController`.

How do I start?

To get started writing an `NSDocument`-based application, use the Cocoa Document-based Application project template when creating a new Xcode project. When you do this, you get a new application project that already contains a subclass of `NSDocument` and a document nib file.

The `NSDocument` subclass is pre-set to load the document nib file. Empty methods are provided for loading and saving; comments explain what you need to fill in. A method is also provided for you to add code that is called after the document nib file is loaded.

Without writing any additional code, you should be able to compile and run the application. You will see an untitled document with an empty window created when you first launch the application, and the File menu commands all do something reasonable, such as bringing up a Save panel or Open panel. Because you have not yet defined any types or implemented loading and saving, you can't actually open or save anything.

From here, you should start by defining a document type (see "How do I define types?" (page 53)) and by implementing the load and save methods (see "How do I implement saving and loading for simple files?" (page 54)).

Beyond defining types and implementing loading and saving, you will certainly need to implement other capabilities in your document subclass. The document subclass should contain and own the contents of the document. This means that your `NSDocument` subclass should provide methods for maintaining and managing the document contents (that is, the document's model objects).

For more information on writing a document-based application, see "Implementing a Document-Based Application" (page 17). For more information on subclassing `NSDocument`, see "Creating a Subclass of NSDocument" (page 23). For an example of an application that follows the application design suggestions, see the Sketch example in `/Developer/Examples/AppKit/Sketch`.

How do I define types?

Types are defined in a file called `Info.plist`, which is managed for you by Xcode. For details on information property lists, see "Storing Document Types Information in the Application's Property List" (page 41), as well as *Property List Programming Guide*.

You can define types for your application in the Target inspector in Xcode. The Target inspector provides an editable list of document types, as shown in "Storing Document Types Information in the Application's Property List" (page 41).

For a new application, you should create a type with a name and extension that make sense for your application. You can add more types as well.

The application's "main" or most important document type should be listed first in the list of types. This is the type `NSDocumentController` will use by default when the user asks for a new document.

How do I implement saving and loading for simple files?

A new document-based application project comes with empty method implementations for `dataRepresentationOfType:` and `loadDataRepresentation:ofType:` in the custom subclass that is automatically created for you. You should implement these methods to support reading and writing of simple files if your application must be able to run on Mac OS X v10.3 or earlier.

For applications that can require Mac OS X v10.4 and later, override `dataOfType:error:` and `readFromData:ofType:error:`, respectively, instead. For example implementations of those overrides, see "Implementing a Document-Based Application" (page 17) and the Sketch example application at `/Developer/Examples/AppKit/Sketch/`.

The `dataOfType:error:` method should provide the contents of the document as an `NSData` object, formatted as the requested type.

The `readFromData:ofType:error:` method should be able to read in the document contents from the given `NSData`, interpreting the data as the given type.

If your document saves document as file wrappers or has other more sophisticated needs, see "How do I implement document packages?" (page 54) and "How do I implement loading and saving when the simple data or file wrapper API won't do?" (page 55) for more information.

How do I implement document packages?

Document packages are really folders, but they appear in the Finder to be opaque documents. Usually, an application that wants its documents to be document packages uses the `NSFileWrapper` class to construct and access its documents. If this is true for some or all of the types that your document class supports, then instead of overriding `dataOfType:error:` and `readFromData:ofType:error:`, you should override `fileWrapperOfType:error:` and `readFromFileWrapper:ofType:error:`.

These methods are much the same as the data-based methods, but they use `NSFileWrapper` objects instead.

You should also specify, in the document types section of your application's information property list, that documents of the specified type are packages (as described in "Storing Document Types Information in the Application's Property List" (page 41)).

How do I implement loading and saving when the simple data or file wrapper API won't do?

If, for some reason, neither the `NSData` nor the `NSFileWrapper` loading and saving API work for you, you can override the three `NSDocument` methods that load and save a document from a URL. These methods are:

```
readFromURL:ofType:error:
writeToURL:ofType:error:
writeToURL:ofType:forSaveOperation:originalContentsURL:error:
```

You should override the reading method and one of the writing methods if you are going to use these as your only reading and writing methods.

You might also override these methods not to provide the loading and saving behavior, but rather to do something immediately before or after actually loading or saving. In this case you would add your code before or after a call to the superclass implementation. This type of overriding is discussed in "How can I support reading one type and automatically converting (internally) to another?" (page 57).

Should I subclass NSWindowController?

The default Document-based Application project template does not subclass `NSWindowController`. You do not need to subclass `NSWindowController` if you are writing a simple application. However, if you are writing an application with more advanced requirements, you will almost certainly want to do so. Here are some common situations that would make subclassing `NSWindowController` desirable:

- You need multiple windows to display your application's document. If you are writing an application whose documents are complex enough to require multiple kinds of windows (such as a CAD program that wants to present front, top, and side views, as well as a rendered 3D view), then you will probably want to have one or more subclasses of `NSWindowController` to manage the different kinds of windows that your document needs.
- You want to support multiple views onto a document. If you are writing an application where you want the user to be able to create multiple views onto a document (such as a drawing program that would like the user to be able to open multiple views so they can see different parts of the same document or see the same parts but with different view settings, such as scales), then you should subclass `NSWindowController`.
- Your application's controller layer is complex enough to make splitting it into a back-end controller (your `NSDocument` subclass) and a front-end controller (an `NSWindowController` subclass) worthwhile. Especially for larger applications, splitting the controller duties between two classes makes a lot of sense. This strategy allows you to have documents that are open, but not onscreen, to avoid having to allocate memory and other resources of a front end that may not be used in some circumstances.

For more information on the roles of `NSDocument` and `NSWindowController` in the document architecture, see The Roles of Key Objects in Document-Based Applications (page 13).

How do I subclass NSWindowController?

Once you've decided to subclass `NSWindowController`, you need to make a couple of changes to the default document-based application setup. First, you should add any Interface Builder outlets and actions for your document's user interface to the `NSWindowController` subclass instead of the `NSDocument` subclass. This is because, now, the `NSWindowController` subclass is the nib file's owner. Some menu actions can still be implemented in the `NSDocument` subclass. For example, save and revert are implemented by `NSDocument`, and you might add other menu actions of your own such as an action for creating new views on a document.

Second, instead of overriding `windowNibName` in your `NSDocument` subclass, override `makeWindowControllers`. In `makeWindowControllers` you should create at least one instance of your custom `NSWindowController` subclass and use `addWindowController:` to add it to the document. If your document always needs multiple controllers, create them all here. If your document will support multiple views, but by default has one, create the controller for the default view here and provide user actions for creating other views.

You should not force the windows to be visible in `makeWindowControllers`. `NSDocument` will do that for you if it's appropriate.

See "How can I make an NSWindowController subclass that automatically uses a particular nib file?" (page 58) and subsequent questions for more information about `NSWindowController`.

When can I do setup operations on my user interface objects?

Many applications need to perform setup operations on user interface objects, such as setting the content of a view, after the application's model data has been loaded. In this case, you must remember that the `NSDocument` data-reading methods, such as `readFromData:ofType:error:`, are called *before* the document's user interface objects contained in its nib file are loaded. Of course, you cannot send messages to user interface objects until after the nib file loads.

After you load the data, you must store it temporarily and set up your user interface objects after the document's nib file loads. If you do not subclass `NSWindowController`, then you can override the `NSDocument` method `windowControllerDidLoadNib:` instead. Or if you do subclass `NSWindowController`, you should instead override the `NSWindowController` method `windowDidLoad`.

If, on the other hand, you need to do some operation immediately before the nib file loads, you can override the `NSDocument` method `windowControllerWillLoadNib:` (if you don't subclass `NSWindowController`) or the `NSWindowController` method `windowWillLoad` (if you do subclass `NSWindowController`).

For objects that are instantiated in the nib file, you can implement the `awakeFromNib` method for this purpose. The Application Kit sends `awakeFromNib` to the nib file's objects after they have all been loaded and all their connections set up. However, the order in which the message is sent is not guaranteed.

How can I support read-only types?

If your application has some types that it can read but not write, you can declare this by setting the role for those types to "Viewer" instead of "Editor" in Xcode, as shown in "Storing Document Types Information in the Application's Property List" (page 41).

How can I support write-only types?

If your application has some types that it can write, but not read, you can declare this by using the `NSExportableAs` key. You can include the `NSExportableAs` key in the type dictionary for another type that your document class supports. Usually this key would go in the type dictionary for the most native type for your document class. Its value is an array of type names that your document class can write, but not read.

The Sketch example uses this key to allow it to export TIFF and EPS images even though it cannot read those types.

Write-only types can be chosen only when doing Save As operations. They are not allowed for Save operations.

How can I support reading one type and automatically converting (internally) to another?

Sometimes an application might understand how to read a type, but not how to write it, and when it reads documents of that type, it should automatically convert them to another type that you can write. An example of this would be an application that can read documents from an older version or from a competing product. It might want to read in the old documents and automatically convert them to the new native format.

The first step is to add the old type as a read-only type (see "How can I support read-only types?" (page 57)). By doing this, your application is able to open the old files, but they come up as untitled files.

If you want to automatically convert them to be saved as your new type, you can override the `readFrom...` methods in your `NSDocument` subclass to call `super` and then reset the filename and type afterwards. You should use `setFileType:` and `setFileURL:` to set an appropriate type and name for the new document. When setting the filename make sure to strip the filename extension of the old type from the original filename, if it is there, and add the extension for the new type.

How can I customize the Save panel?

You can control whether the default accessory view (which contains a pop-up menu allowing the user to choose what type to save) appears in the Save panel by overriding `shouldRunSavePanelWithAccessoryView`. The default accessory view is used if that method returns `YES` and the document supports writing multiple types.

You can customize the panel more completely by overriding `prepareSavePanel:` and modifying the panel before calling `super`. For example, you could replace the accessory view entirely.

How do I implement printing?

Subclasses of `NSDocument` that wish to support printing should override `printShowingPrintPanel:`. Usually this is implemented to create an `NSPrintOperation` object with the document's print info and run it.

A document should be prepared to print itself even if it currently has no window controllers.

Should I do anything about print info?

Ideally you should treat a document's print info as part of the document, to be saved and loaded along with the rest of the contents. This is not always possible if your document format is already defined and is not flexible enough to allow saving the print info.

Apart from using it to create your print operations and possibly saving and loading it, you should not have to do anything else about print info.

How can I make an NSWindowController subclass that automatically uses a particular nib file?

An `NSWindowController` object expects to be told what nib file to load (through its `initWithWindowNib...` methods), because it is a generic implementation of the default behavior for all window controllers. However, when you write a subclass of `NSWindowController`, it is almost always designed to control the user interface contained in a particular nib file, and your subclass would not work with a different nib file. It is therefore inconvenient and error-prone for the client of the subclass to have to tell it which nib file to load.

This is easily solved by overriding the `init` method to simply call the superclass's `initWithWindowNibName:` method with the correct nib name. Now clients just use `init` and the controller has the correct nib file. You can also override the `initWithWindowNib...` methods to log an error, because no clients should ever try to tell your subclass which nib file to use. This is a good idea for any `NSWindowController` subclass designed to work with a specific nib file. You should do otherwise only if you are extending the basic functionality of `NSWindowController` in your subclass and have not tied that functionality to any particular nib file.

How can I use NSWindowController for shared panels (inspectors, find panels, etc.)?

An `NSWindowController` object without an associated `NSDocument` object is useful all by itself. `NSWindowController` can be used as the base class for auxiliary panel controllers in order to gain the use of its nib management abilities.

One common standalone use of `NSWindowController` subclasses is as controllers for shared panels such as find panels, inspectors, or preferences panels. In this case, you can make an `NSWindowController` subclass that implements a shared instance method. For example, you could create a `PreferencesController` subclass with a `sharedPreferenceController` class method that creates a single instance the first time it is called and returns that same instance on all subsequent calls.

Because your subclass derives from `NSWindowController`, you can just tell it the name of your Preferences nib file and it will handle loading the nib file and managing the window automatically. You add your own outlets and actions, as usual, to hook up the specific user interface for your panel and add methods to manage the panel's behavior.

The Sketch application uses `NSWindowController` subclasses for its various secondary panels.

How can I use multiple NSWindowControllers for a single document?

If you use `makeWindowControllers` to create your window controllers (see "How do I subclass NSWindowController?" (page 56)), you can create more than one, possibly of different subclasses of `NSWindowController`, from the beginning. Another possibility is to allow the application to create new controllers later as it needs them. In any event, multiple controllers can be added to a document with `addWindowController:`.

By default, a document closes when its last remaining window controller closes. Specific window controllers can also be set to close the document when they close even if there are other controllers still open. An example of where this happens is Interface Builder. There is a main window for a nib document with a tab view and top-level instances in it, and there are a number of other windows, which are the window editors for the windows in the nib. If the user closes the main window, all the other windows close as well and the document itself closes. Interface Builder calls `setShouldCloseDocument:YES` on the main window's controller to implement this behavior.

How can I customize the window title for a document's NSWindowController?

You can override the `NSWindowController` method `windowTitleForDocumentDisplayName:` to modify the title for each view. For instance, a CAD program might have the titles of the different windows it uses for a document named "Airplane" as "Airplane – Top," "Airplane – Side," and so on.

How do I implement undo?

Undo is not always easy to implement, but at least the mechanism for implementing it is straightforward. By default, an `NSDocument` object has its own `NSUndoManager` object. The `NSUndoManager` class enables you to easily construct invocations that do the opposite of a preceding change.

The key is to have well-defined primitives for changing your document. Each model object, plus the `NSDocument` subclass itself, should define the set of primitive methods that can change it. Each primitive method is then responsible for using the undo manager to enqueue invocations that undo the action of the primitive method. For example, if you decide that `setColor:` is a primitive method for one of your model objects, then inside of `setColor:` your object would do something like the following:

```
[[[myDocument undoManager] prepareWithInvocationTarget:self] setColor:oldColor]
```

This call causes the undo manager to construct an invocation and save it away. If the user later chooses Undo, the saved invocation is invoked and your model object receives another `setColor:` message, this time with the old color. (If you're wondering if you have to keep track of whether things are being undone and avoid doing the undo manager stuff again, you don't. In fact, the way redo works is by watching what invocations get registered as the undo is happening and recording them on the redo stack.)

Another piece of good undo implementation is to provide action names so the Undo and Redo menu items can have more descriptive titles. Undo action names are usually best set in action methods instead of the change primitives in your model objects because many primitive changes might go into one user action, or different user actions might result in the same primitives being called in different ways. The Sketch example application implements undo in action methods.

For more information on supporting undo in your application, see *Undo Architecture*.

How do I implement partial undo?

Because `NSUndoManager` does multiple-level undo, it is not a good idea to implement undo for only a subset of the possible changes to your document. The undo manager relies on being able to reliably take the document back through history with repeated undos. If some changes get skipped, the undo stack state is no longer synchronized with the contents of the document. Depending on your architecture, that can cause problems that range from merely annoying to fatal.

If there are some changes that you just can't undo, there are two possibilities for handling the situation when a user makes such a change. If you can be absolutely sure that the change has no relationship to any other changes that can happen to the document (that is, something totally independent of all the rest of the contents of the document has changed), then you can safely just not register any undo action for that change. On the other hand, if the change does have some relationship to the rest of the document contents, you should remove all actions from the undo manager when such a change takes place. Such changes then mark points of no return in your user experience. When designing your application and document format, you should strive to avoid the need for these “point of no return” operations.

What if I don't want to support undo?

If you don't wish to support undo at all, the first thing you should do is call `setHasUndoManager:NO` on your document. This causes the document never to get an undo manager.

Without an undo manager (and undo support from your model objects), the document cannot automatically track its dirty state. So, if you aren't implementing undo, you need to call `updateChangeCount:` by hand whenever your document is edited.

What's all this change count stuff?

Because of undo support, the document must keep more information than just whether the document is dirty or clean. If a user opens a file, makes five changes, and then chooses Undo five times, the document should once again be clean. But if the user chooses Undo only four times, the document is still dirty.

The `NSDocument` object keeps a change count to deal with this. The change count can be modified by calling `updateChangeCount:` with one of the supported change types. The supported changes are `NSChangeDone`, `NSChangeUndone`, and `NSChangeCleared`. The `NSDocument` object itself clears the change count whenever the user saves or reverts the document. If the document has an undo manager, it observes the undo manager and automatically updates the change count when changes are done, undone, or redone.

If your document subclass does not support undo, then you need to inform `NSDocument` of edits with `updateChangeCount:` yourself (see "What if I don't want to support undo?" (page 60)).

Should I subclass NSDocumentController?

Usually, you should not need to subclass `NSDocumentController`. Almost anything that can be done by subclassing can be done just as easily by the application's delegate. However, it is possible to subclass `NSDocumentController` if you need to.

For example, if you need to customize the Open panel, an `NSDocumentController` subclass is clearly needed. You can override the `NSDocumentController` method `runModalOpenPanel:forTypes:` to customize the panel or add an accessory view. The `addDocument:` and `removeDocument:` methods are provided for subclassers that want to know when documents are opened or closed.

How can I subclass NSDocumentController?

There are two ways to subclass `NSDocumentController`:

- You can make an instance of your subclass in your application's main nib file. This instance becomes the shared instance.
- You can create an instance of your subclass in your application delegate's `applicationWillFinishLaunching:` method.

The first `NSDocumentController` object to be created becomes the shared instance. The Application Kit itself creates the shared instance (using the `NSDocumentController` class) during the "finish launching" phase of application startup. So if you need a subclass instance, you must create it before the Application Kit does.

How can I create new documents other than through user-action methods?

You can use the `NSDocumentController` methods `openUntitledDocumentAndDisplay:error:` and `openDocumentWithContentsOfURL:display:error:`, which create a document and, if the display parameter is `YES`, also create the document's window controller (or controllers) and add the document to the list of open documents. These methods also check file paths and return an existing document for the file path if one exists.

You can also use the `NSDocumentController` methods `makeUntitledDocumentOfType:error:`, `makeDocumentWithContentsOfURL:ofType:error:`, and `makeDocumentForURL:withContentsOfURL:ofType:error:`, which just create the document, or you can simply create a document yourself with any initializer the subclass supports. In either case, you usually need to call the `NSDocumentController` method `addDocument:` to add the document to the document controller's list of documents

How can I keep my application from creating an untitled document at launch?

Implementing the `applicationShouldOpenUntitledFile:` method to return `NO` in your application delegate prevents the application from opening an untitled file when launched or activated. If you do want to open an untitled file when launched, but don't want to open an untitled file when already running and activated from the dock, you can instead implement `applicationShouldHandleReopen:hasVisibleWindows:` to return `NO`.

Document Revision History

This table describes the changes to *Document-Based Applications Overview*.

Date	Notes
2010-03-24	Made slight alteration to document types XML listing. Corrected method name and document-writing method in "Implementing a Document-Based Application."
2009-11-17	Added guidance in "Creating a Subclass of NSDocument" to call breakUndoCoalescing when saving a document containing an NSTextView object.
2009-10-19	Expanded information about UTIs and subclassing NSDocumentController.
2009-07-27	Added information about using UTIs and concurrent document reading. Fixed typo.
2009-01-12	Removed private constant from document types listing.
2009-01-06	Added advice for setting up nibs with NSWindowController as file's owner.
2007-09-04	Corrected typos.
2007-01-08	Added information about using openUntitledDocumentAndDisplay:error: when the display flag is NO and fixed awkward phrase in "What is a Document?" article.
2006-05-23	Added note that the Name field of the document type in the Info.plist is used to specify the document type in APIs.
2006-03-08	Added several new articles and revisions throughout. Updated for Mac OS Xv10.4. Made editorial revisions, rewrote introduction, added an index, and fixed errors. Changed title from "Document-Based Applications."
	New articles added are "Message Flow in the Document Architecture" (page 29), "Creating Multiple-Document-Type Applications" (page 47), "Autosaving in the Document Architecture" (page 49), and "Error Handling in the Document Architecture" (page 51).
2004-08-31	Added a table of contents (a list of the questions) to "Frequently Asked Questions" (page 53) for HTML version. Added annotated list of articles to introduction.
2003-11-12	Clarified that HTML is not fully supported for reading and writing in "Storing Document Types Information in the Application's Property List" (page 41).
2003-11-06	Corrected references and links to documents referring to additional property list information in "Frequently Asked Questions" (page 53).

Date	Notes
	Updated description and art for editing document types in "Storing Document Types Information in the Application's Property List" (page 41).
2003-06-11	Property list samples in "Storing Document Types Information in the Application's Property List" (page 41) have been updated to current format.
2003-06-06	Clarified in first table of "Implementing a Document-Based Application" (page 17) that there is one NSDocument per document, rather than saved file (since a document may not have been saved yet.
	Added links to Performance programming topics in "Window Closing Behavior" (page 37).
2003-04-21	First-responder action method for "Save As" and "Save To" corrected.
2003-02-07	First-responder action method for "Revert" corrected.
2003-01-17	Changed title and made minor corrections to "Storing Document Types Information in the Application's Property List" (page 41) (formerly "The Document Types Information Property List").
	In "Frequently Asked Questions" (page 53), removed the section "How can I support stationery documents?" pending revision.
2002-12-03	Corrected typo in method name in "Creating a Subclass of NSDocumentController" task.
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.