Chapter 15

WORKING WITH NSTEXTVIEW

CHAPTER 15 WORKING WITH NSTEXTVIEW

Goal

To master the basic features of NSTextView for implementing multi-line text objects in graphical interfaces.

Prerequisites

Confidence with using NSString and familiarity with different text types—ASCII and rich text.

Objectives

At the end of this section, you will be able to:

» Write a text view controller that can load, save, append, clear and scroll an NSTextView

» List several of the built-in text operations along with pre-configured menu and panel support provided by the Application Kit and Interface Builder

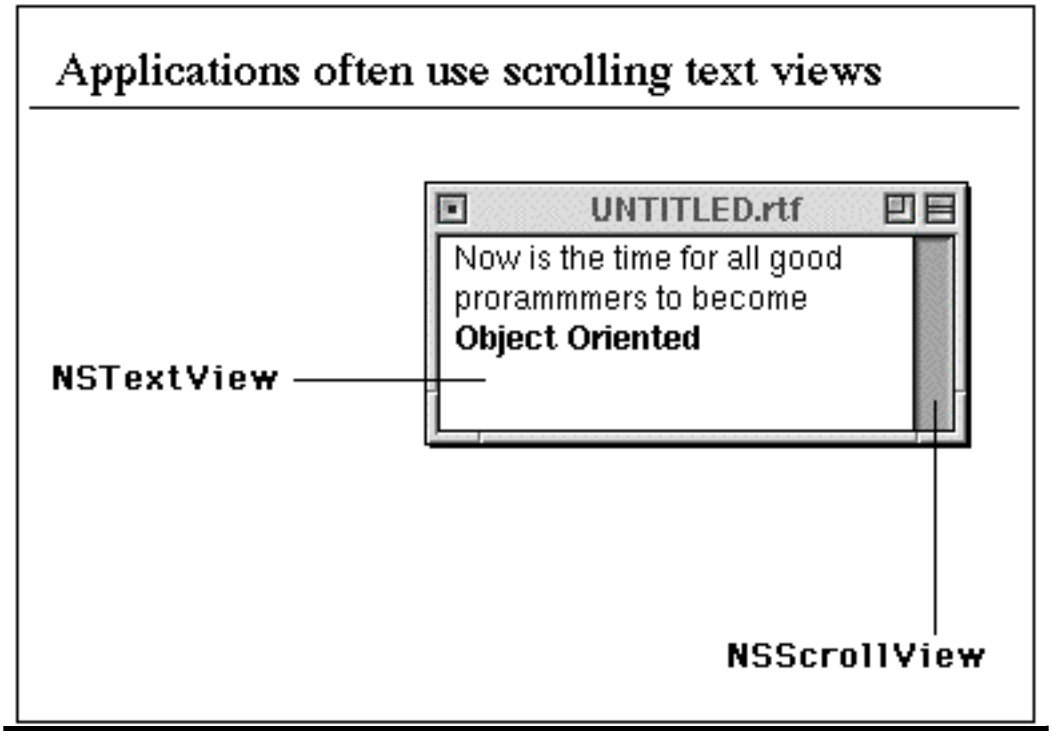
Reading

NSTextView class reference in the Application Kit

NSText class reference in the Application Kit

NSString class reference in the Foundation

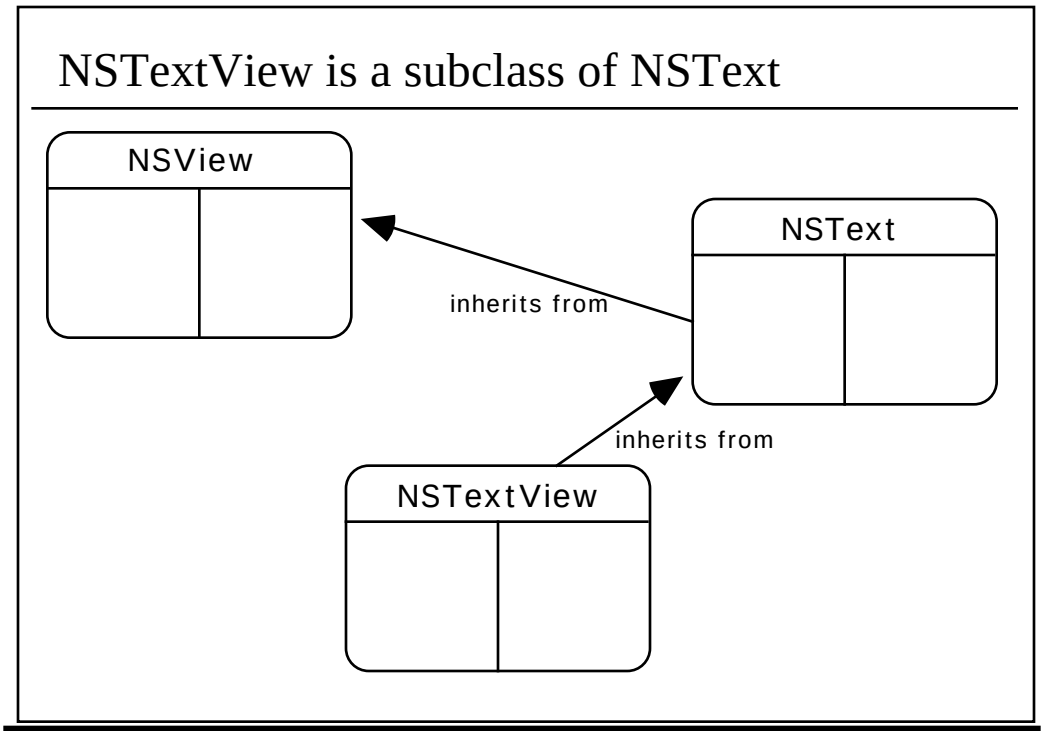
NSData class reference in the Foundation



Applications often use scrolling text views

Many applications need to feature a scrolling multi-line text object. It might be read-only for examining logs or it might be editable, a place for the user to add comments, maintain a log or compose a document. A combination of NSScrollView and NSTextView provides a full featured text editing system ideal for the purpose. While text entry and sophisticated editing is already implemented in the text view, your application will likely need to programmatically manage the text view in a few basic ways. You will want to load the text, from an object or perhaps a file, retrieve the text after the user has modified it, clear and scroll the TextView and possibly operate on sub-ranges within the text.

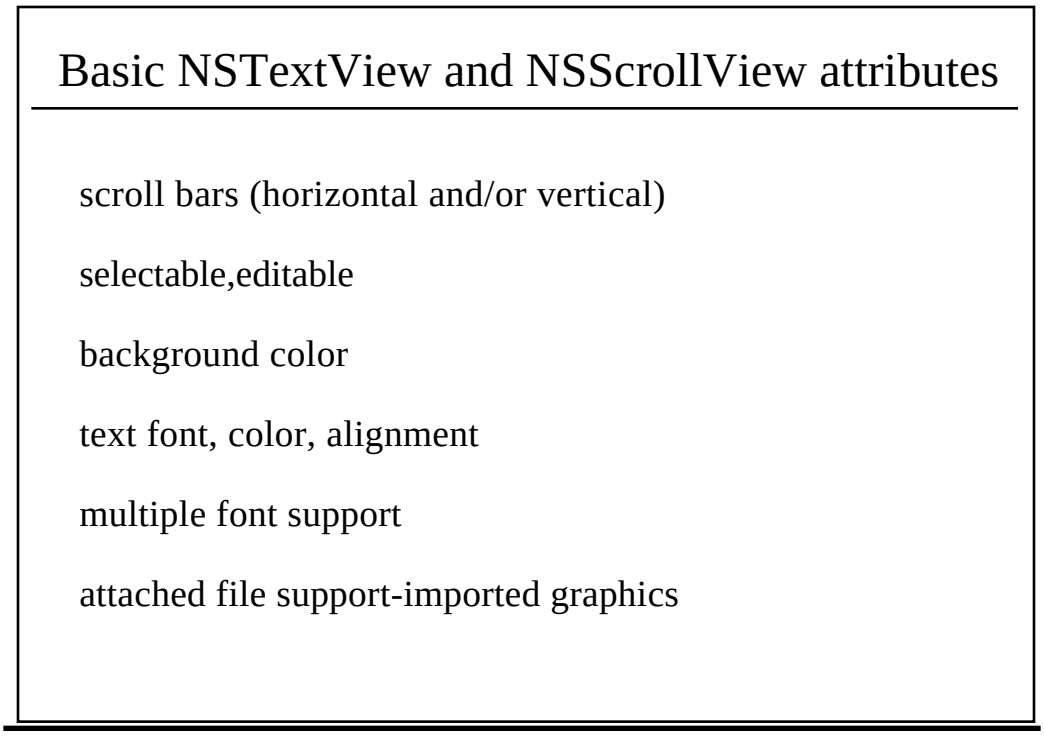
None of this is particularly difficult. It simply requires that you know a few details and the relevant messages to send. Because the text view is so rich in functionality its API seems daunting. Much of it applies to highly customized and sophisticated cases. A much smaller subset is typically all that is needed for more common applications. It is worth highlighting the most practical and frequently used interfaces for the text view situations you are likely to encounter.



NSTextView is a subclass of NSText

Like most sophisticated Application Kit objects, NSTextView is a subclass of another class, NSText. This is worth remembering since much of NSTextView's behavior is inherited and documented elsewhere. Effective handling of NSTextView requires that you are familiar with its less specialized form, NSText.

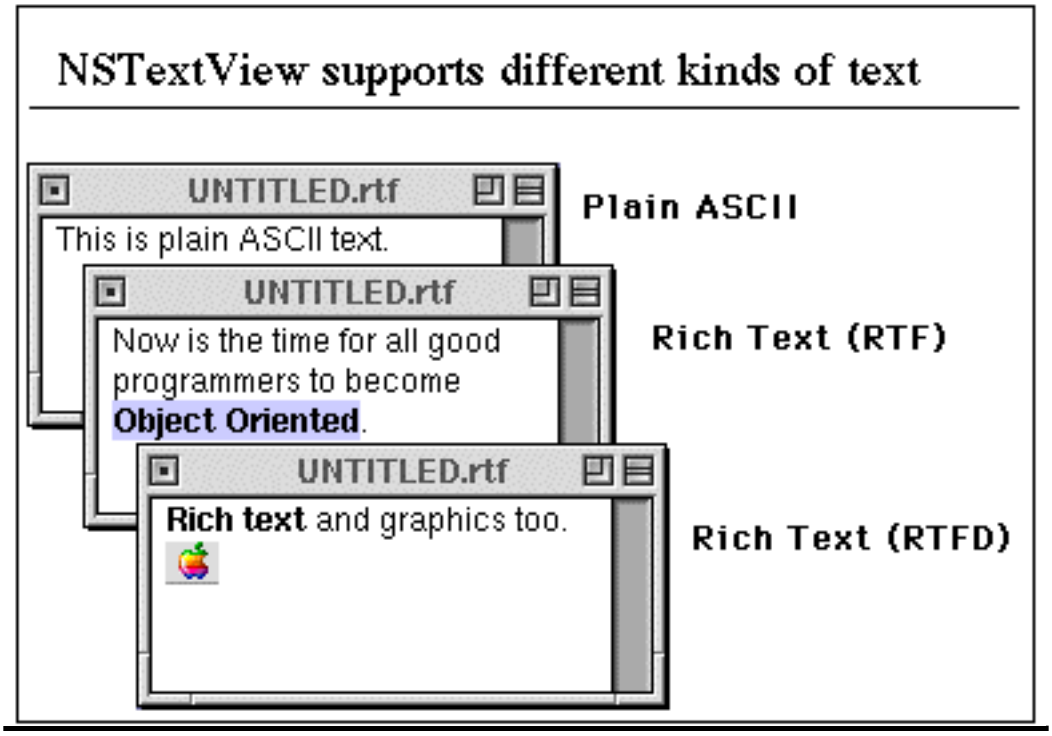
You will also notice that an NSTextView is not an NSControl. It is not a simple target/action object. It provides a much more general container for displaying and editing text. It is almost always used in tandem with other controls. Your controller objects can find out about scrolling and typing actions nonetheless, right down to the granularity of a single keystroke. This is achieved through delegation and notification.



Basic NSTextView and NSScrollView attributes

These are some common attributes you may wish to customize. Because a TextView is a container for text, you can imagine that each character within the text view has a set of attributes such as font and color. The TextView has its own global concept of these as well—what applies to the current state, the current selection or to all the contained text as a whole.

- Scroll bars—can be present or absent for either horizontal or vertical orientations
- Selectable, editable—can the user select for searching and copying? can the user change the contents
- Background color—defaults to white. Might be off-white for read-only, or something else for clarity or aesthetics
- Text properties—font, color and alignment—these are defaults that apply to text which is not specific marked for alternatives
- Multiple font support—whether or not the text view supports rich text
- Attached file support—whether or not the text view supports embedded images, possibly even files and directories

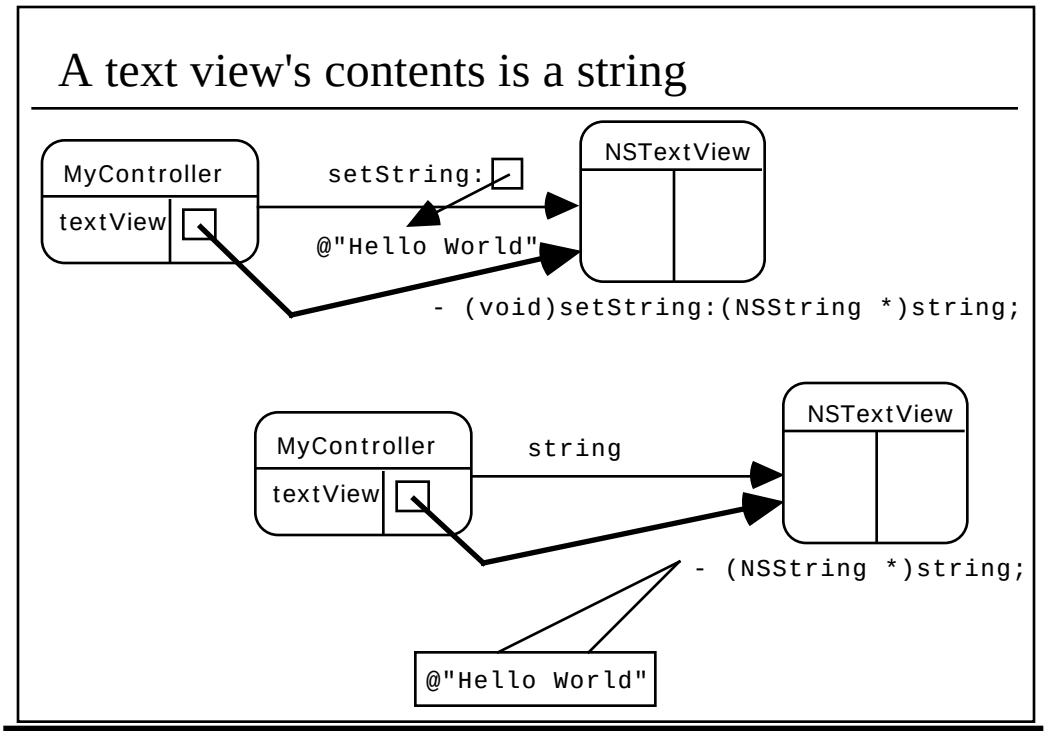


NSTextView supports different kinds of text

NSTextView supports three kinds of text:

- ASCII—all one font, no meta-characters in the text stream
- RTF—rich text. Supports multiple fonts and colors using embedded meta-characters
- RTFD—a further elaboration of rich text, RTFD can include pictures, references to other files and directories

You decide what your application needs. You configure the NSTextView to support the type of text you want it to manage. Saving and loading between the TextView and the file system is also specific to the type of text being used.



A text view's contents is a string

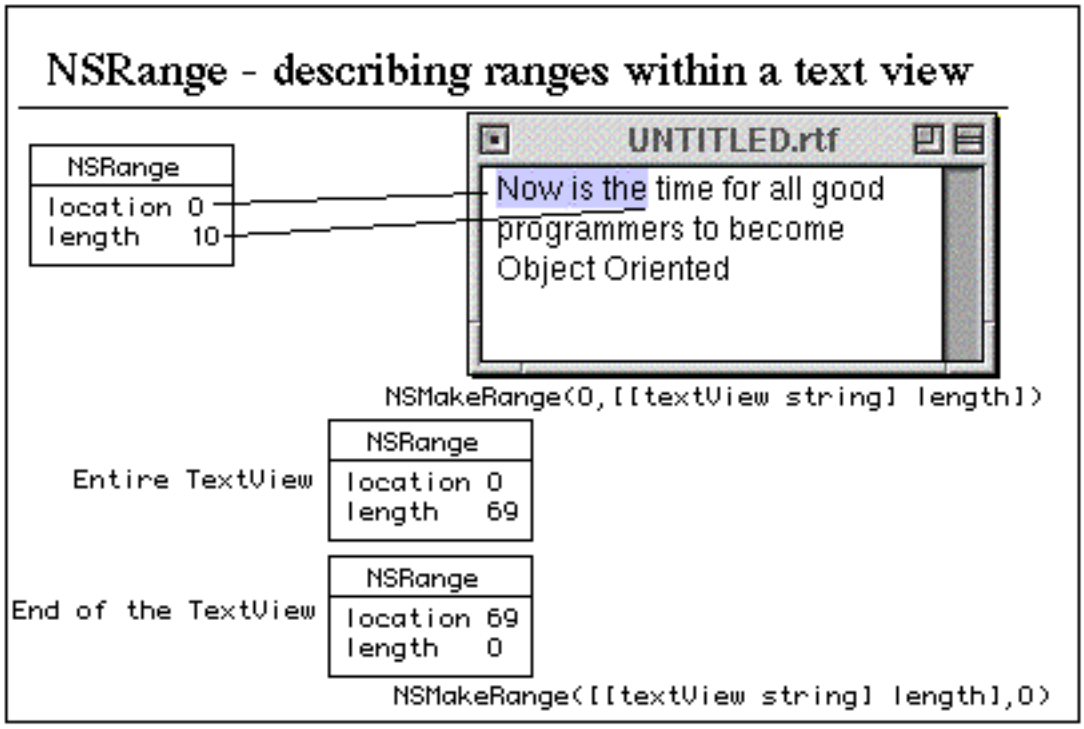
So what about the text within a text view? It can be as simple as a string.

To set the complete contents of the text view—**setString:**.

To retrieve the entire contents of the text view—**string**.

How might you clear a text view?

```
[textView setString: @””];
```



NSRange—describing ranges within a text view

What about appending to text that is already there? What about scrolling the view to the end? How would you get at just the selected portion of text. For these tasks, you need NSRange.

NSRange is a C structure typedef that describes a range with two parameters:

» **location**—the character offset in the string where the range starts

» **length**—the number of characters in the range

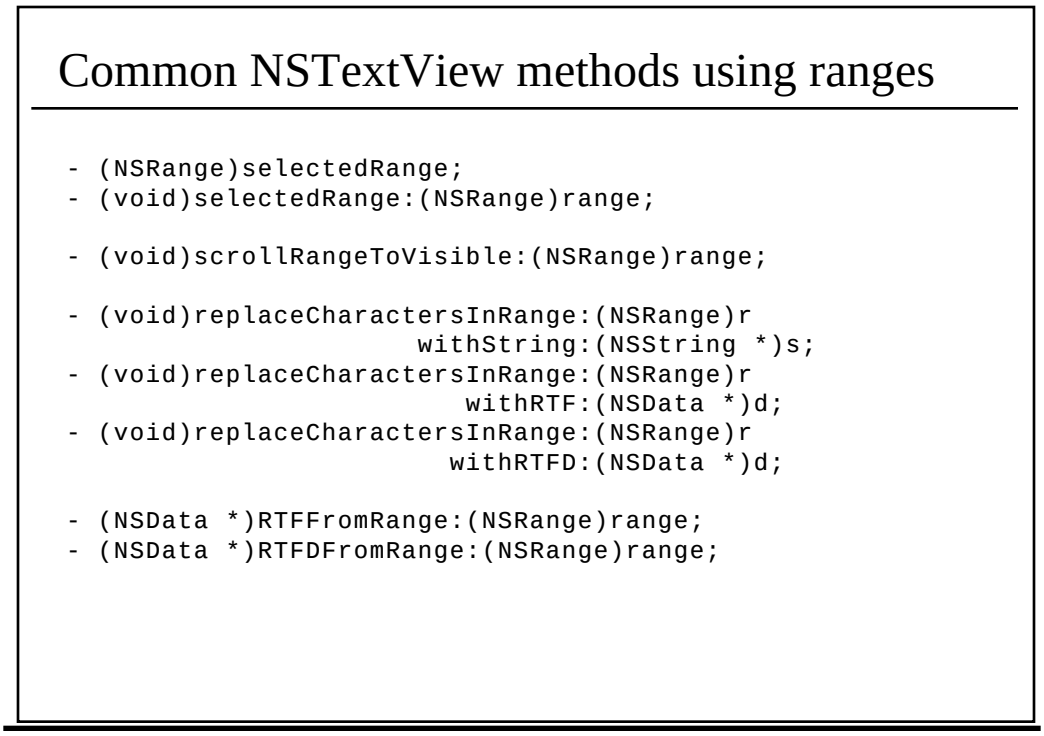
NSMakeRange is a convenience function for creating a valid range structure.

Some commonly used ranges:

» The entire text view.

» The end of the text view—for appending or scrolling. The length should be 0.

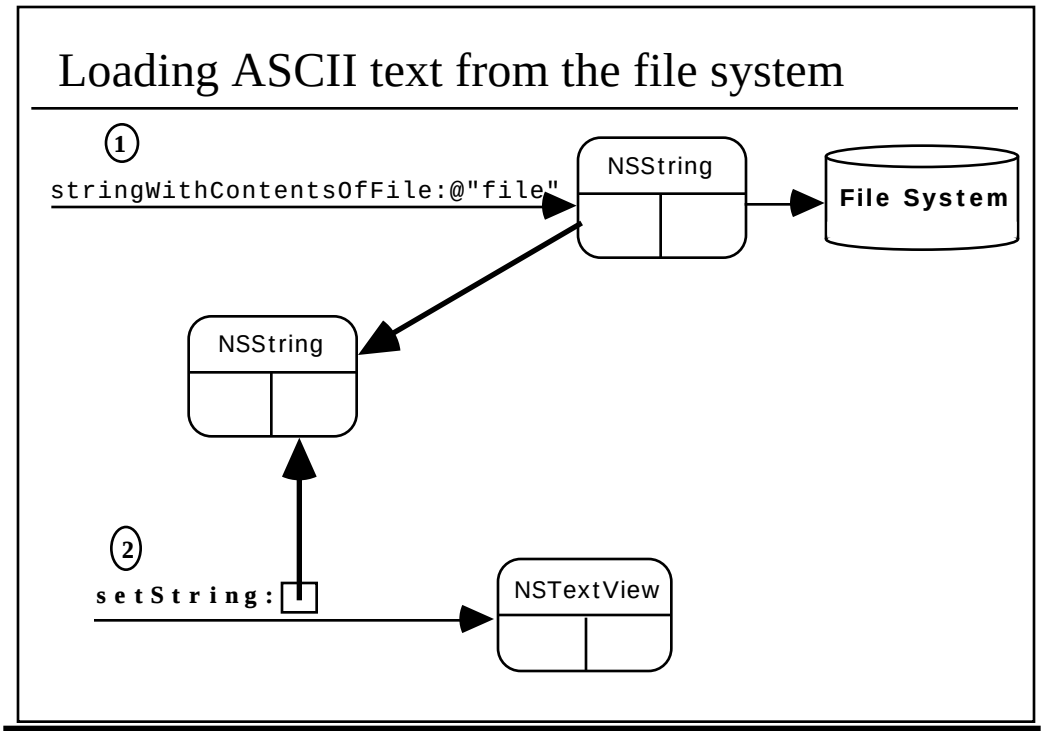
» The selected range. You can ask the text view for the range of text the user has selected for editing. The current insertion point is always indicated by the location of this range. The length may be 0 if nothing is selected.



Common NSTextView methods using ranges

You can programmatically get and set the selected range. This is useful for implementing your own text processing features, search and replace components and the like. You can programmatically scroll a certain range into view, handy again for searches and for appending and auto-tracking a scrolling log. You can replace an arbitrary range.

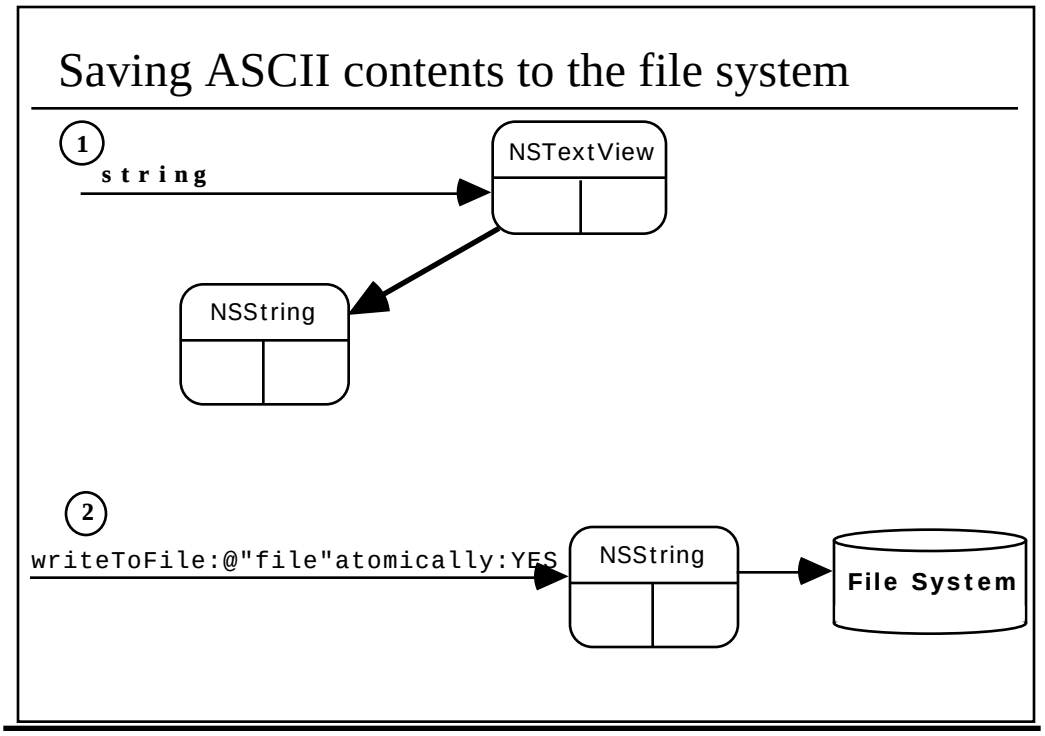
When your speak purely in terms of strings, you are not including any meta-data used to encode the rich text formatting. Although the text view supports rich text, it might be fine for your application to examine the text as pure ASCII. When handling rich text, either getting or setting or moving between the view and file system you need to use specific API. Notice that there are separate interfaces for RTF and RTFD type text.



Loading ASCII text from the file system

If one of your business objects contains a multi-line text attribute, you can load from and save to the object instance using accessor methods. The issue there is how you get at your business objects. Other times you may need to deal directly with files as the source and destination of text. ASCII text with an NSString is straightforward:

```
NSString *string = [NSString stringWithContentsOfFile:@”file”];

// error checking

[textView setString: string];
```

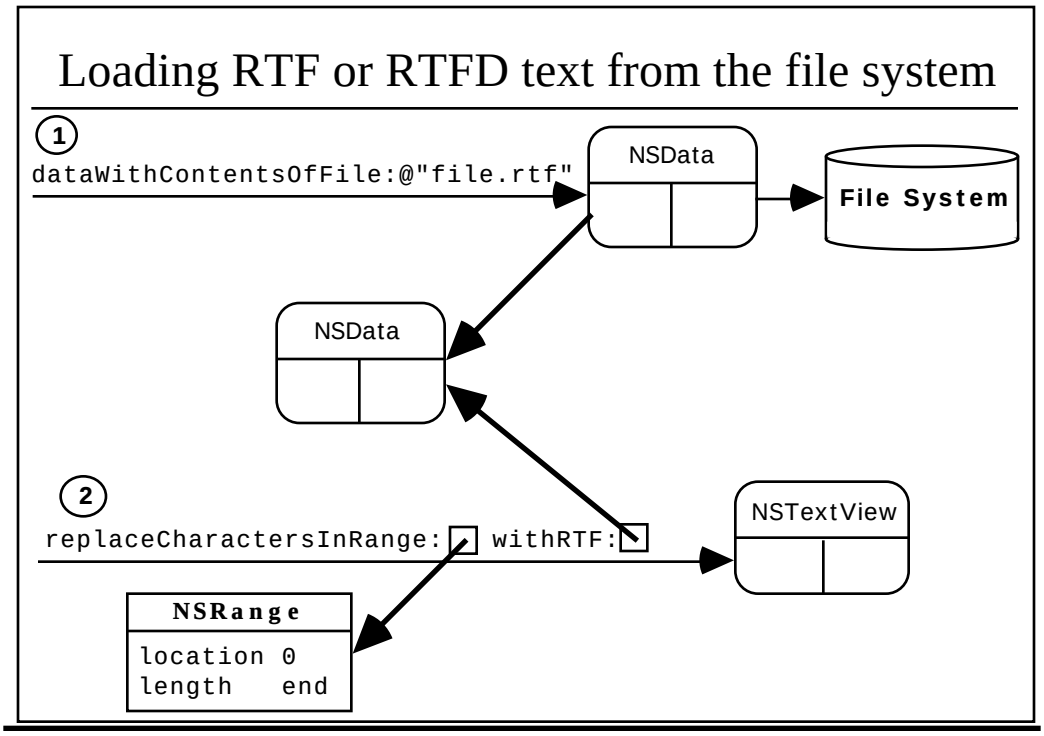


Saving ASCII contents to the file system

Writing back to the file is just the reverse:

```
NSString *string = [textView string];

[string writeToFile: @”file” atomically: YES];

// error checking
```

Atomically means that the file I/O should not partially write the file then fail—it either writes it all or none at all.

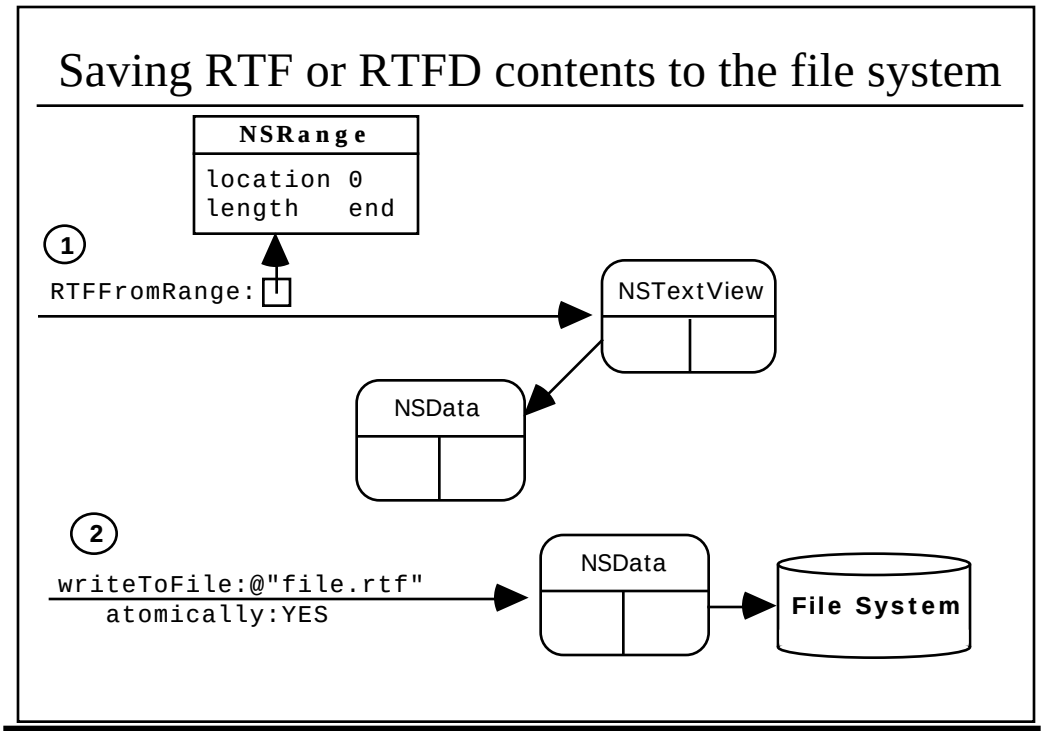


Loading RTF or RTFD text from the file system

Rich text is only slightly more complicated due to the fact that it is not just a single stream of ASCII text. Because RTF data is more that just the literal string, you must to RTF-specific API that involves two different pieces:

- NSData—an object wrapper for arbitrary chunks of binary data. You can create an instance bound to a file with **dataWithContentsOfFile:**.
- NSRange—because you are replacing rather than setting, specify the desired range. Since you are replacing everything, use:

```
NSMakeRange(0, [[textView string] length]);
```



Saving RTF or RTFD contents to the file system

Again, the situation is the same in reverse. Ask the text view for a range of RTF and ask the returned NSData instance to write itself to the file.

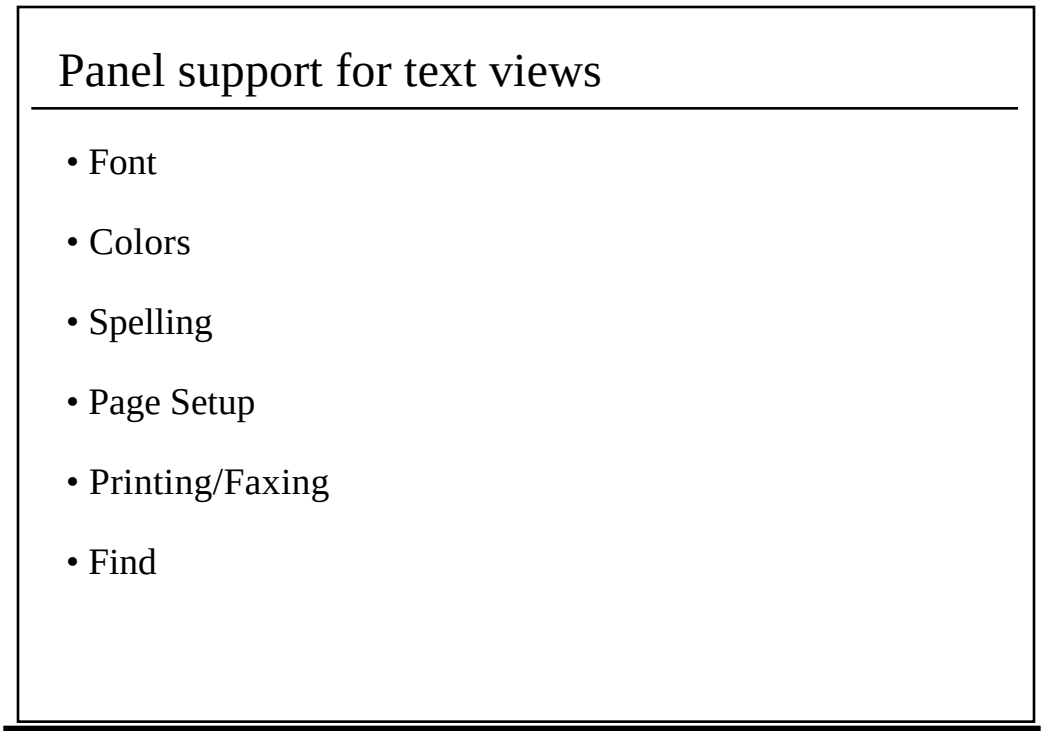
NSTextView delegation and notification

```
- (BOOL)textShouldBeginEditing:(NSText *)text;

- (void)textDidBeginEditing:(NSNotification *)notification;

- (void)textDidChange:(NSText *)text;

- (BOOL)textShouldEndEditing:(NSText *)text;

- (void)textDidBeginEditing:(NSNotification *)notification;
```

NSTextView delegation and notification

Your controller can find out about user interactions with the text view down to the granularity of a keystroke. NSTextView can take a delegate which it will automatically register for a number of notifications if it responds to them. Non-delegate objects can register for the desired notifications from a specific text view instance.

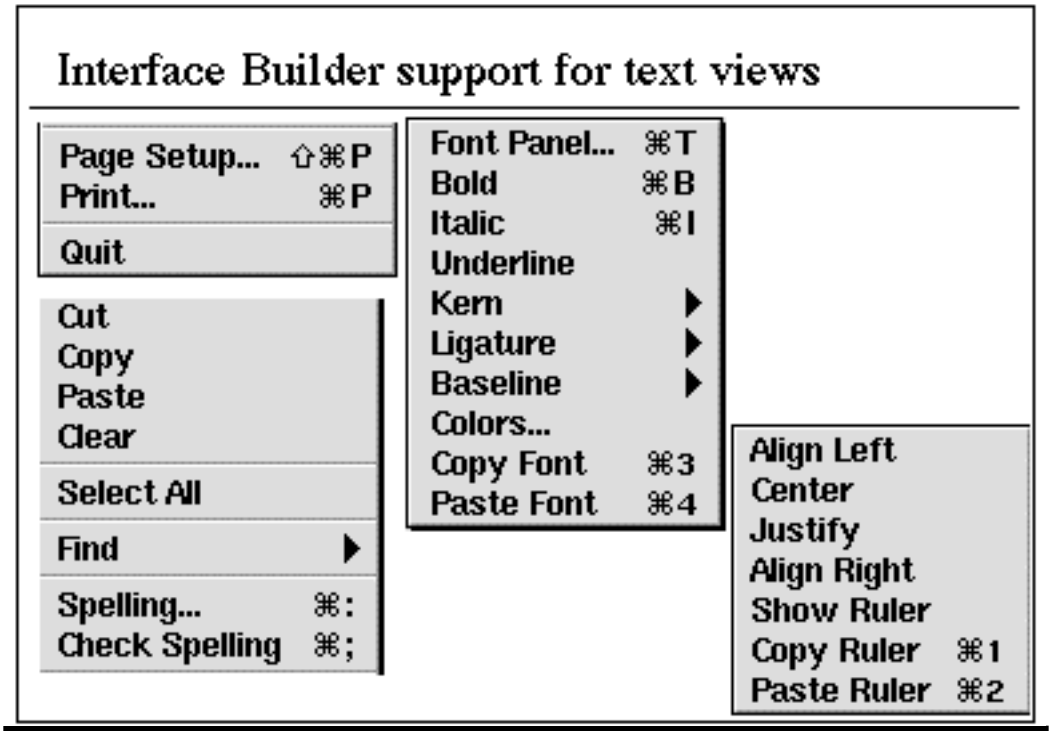
The most simple and common delegate method, also available as a notification, is **textDidChange:**. Your document controller can use this to mark the document as edited with all that implies. Other delegate methods and notifications and be used for pre and post-editing operations, as well as character by character monitoring if necessary.



Panel support for text views

Text is the focus of many if not most software applications. Text handling is old, has evolved, and should provide a wide range of standard features that users have come to expect. The Application Kit provides a number of auxiliary panels with sophisticated functionality that can work with text views with very little effort on your part:

- Font
- Colors
- Spelling
- Page setup
- Printing/Faxing
- Find

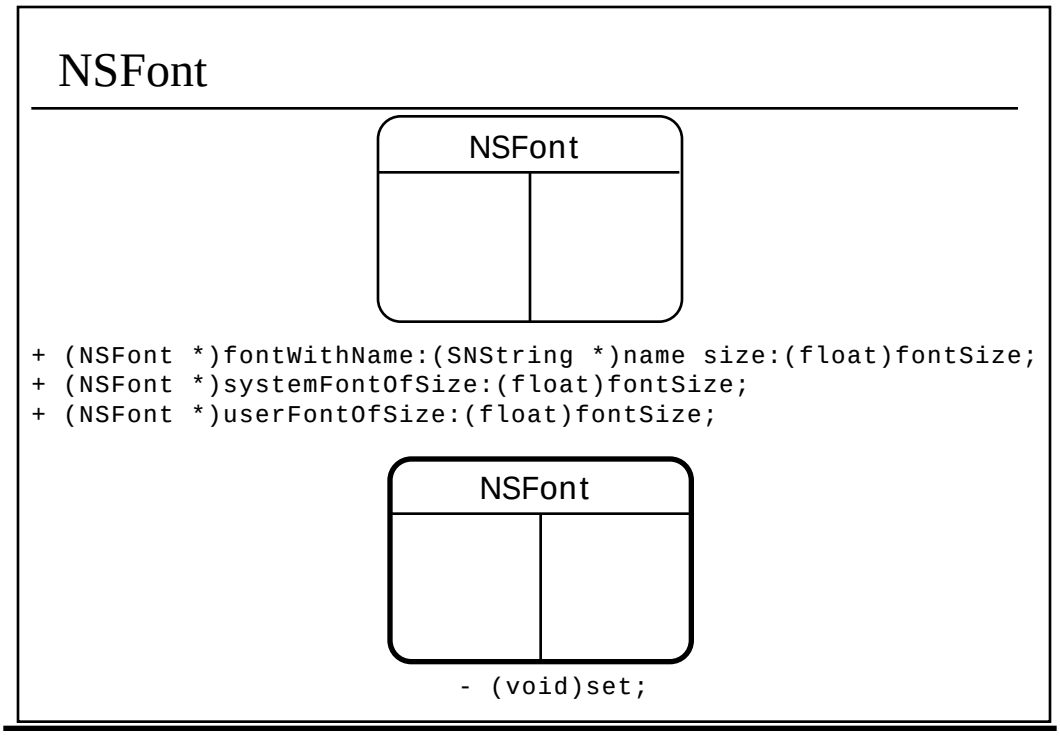


Interface Builder menu support for text views

Access to these as well as a large number of fully functional text-based operations are available as menu items and completely assembled submenus on the Interface Builder views palette. You can pick and choose what you want, deleting or disabling options that don't apply. Some menu items such as Find are non-functional placeholders, disabled because they are not connected to anything. To enable their functionality, you must typically implement some custom objects and actions that are specific to the workings of your application.

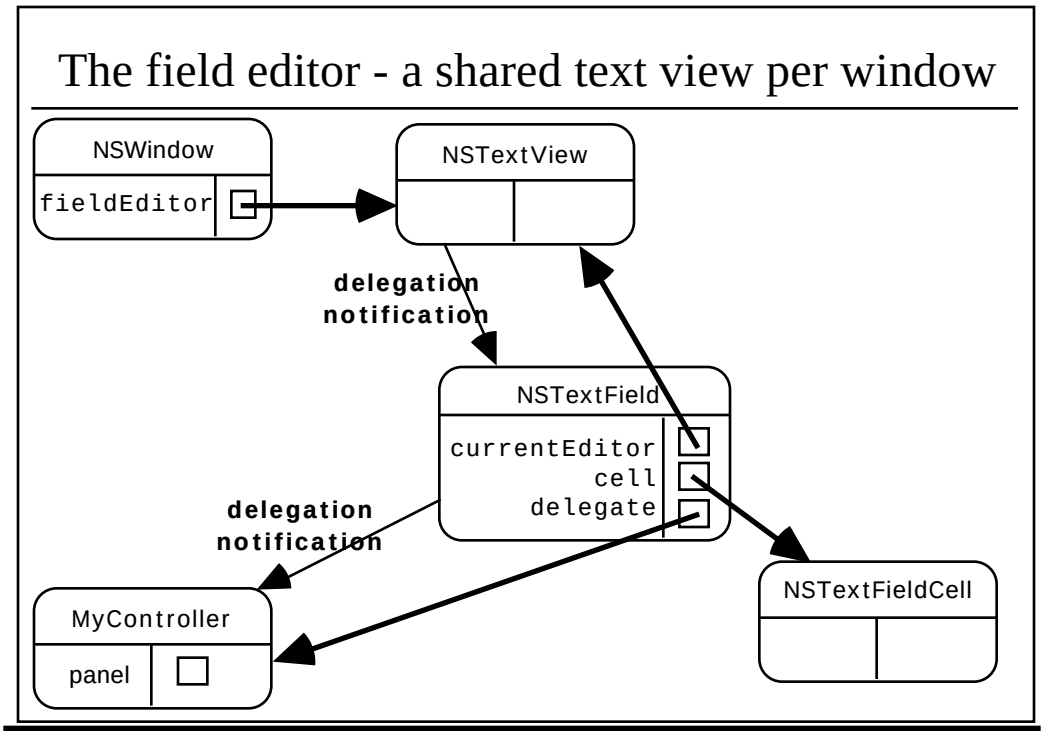
Like the text support panels, many of these functions apply not only to text views but any text bearing cell—text fields, form cells, table view cells, and so on. With any kind of text-based control, your application will be more usable if it includes at least common editing functions such as cut, copy and paste.

Additional Points To Ponder



NSFont

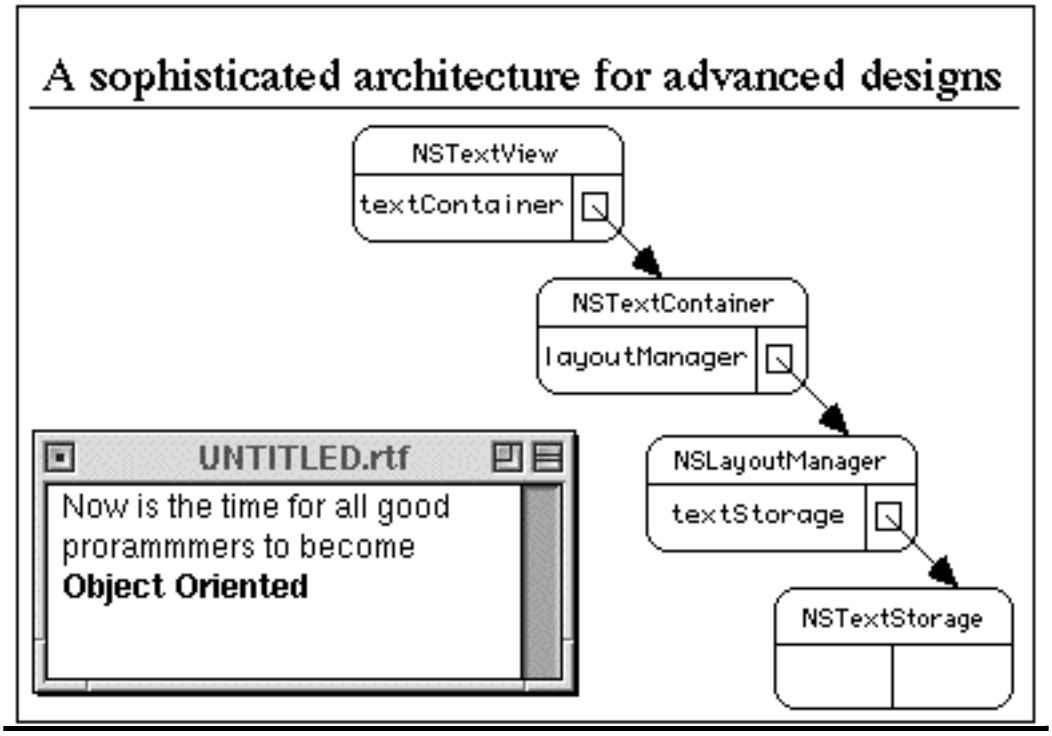
Fonts enjoy an object-oriented interface. You can create an instance of any specific font by name and point size or access system-wide fonts sized to your needs. Font objects can be passed to text view and text cell methods to programmatically configure the appearance of your user interface. A certain font can be set as the current font in the PostScript context when drawing in your custom views. For more information on this, see the section on Custom Views.



The field editor—a shared text view per window

NSTextView is highly useful, reusable, and also heavyweight. Each window has its own instance of a NSTextView which it uses for a variety of text display and editing. It is called the field editor. NSTextField and NSTextFieldCell instances do not encapsulate their own redundant text editing capabilities but make use of this shared NSTextView instance to carry out their functions. When you connect yourself as the delegate or receive notifications from an NSTextField control, you are getting the relevant field editor messages and notifications indirectly.

This is all transparent and typically of little concern to you. It is pleasing to note the efficiency, the reuse, and the cooperation among objects. Perhaps this will inspire some design ideas that apply to your problem domain. You might need to get at the field editor for some advanced operations. You can even replace the field editor if you have a custom object with specialized display or editing capabilities and you wish all text cells to utilize these capabilities.

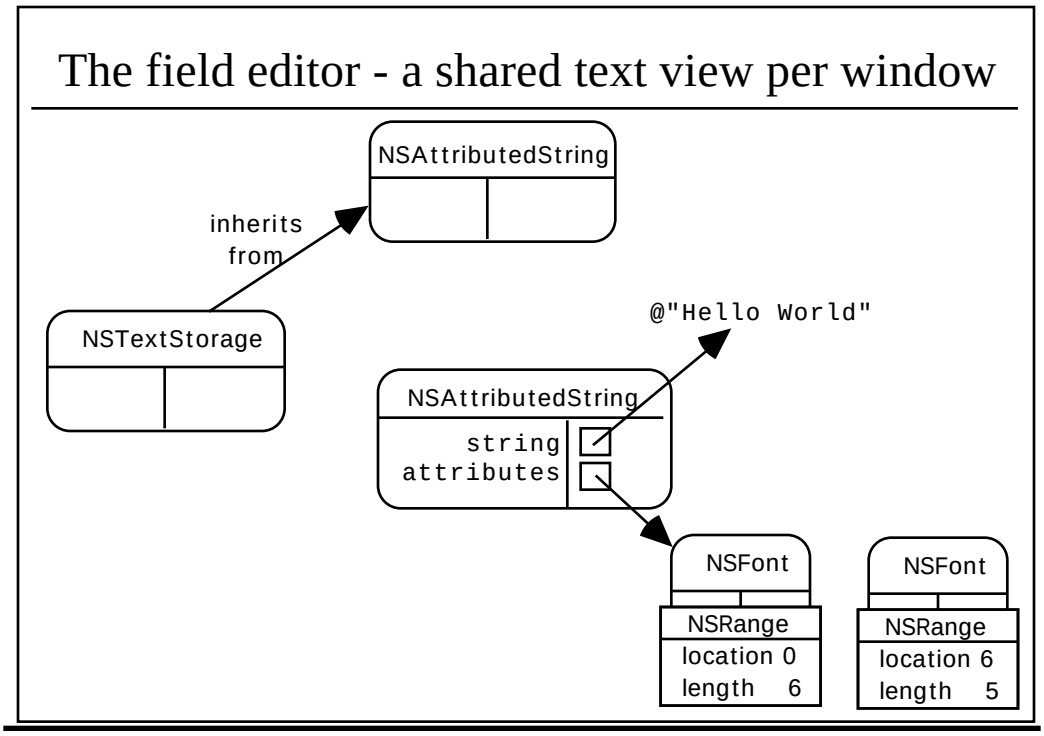


A sophisticated architecture for advanced designs

NSTextView is really just the tip of an iceberg. It cooperates with a number of objects designed to decompose the problem of sophisticated text handling into a framework of objects to facilitate diverse and flexible designs. Among other things, it is possible to have multiple different text views windowing in on different parts of a common text store. While this course does not have the time for a complete investigation, you now know where to turn for more information. With its most common uses and interfaces, NSTextView beautifully demonstrates encapsulation and the pleasing fact that, often, you don't need to sweat the details. It does it for you.

- NSTextView—the highest level class and featured in this section. NSTextView provides the user interface layer.
- NSTextContainer—defines a region within a view where text can be laid out in a rectangular area. Actual display and user event handling does not happen here.
- NSLayoutManager—defines a mapping between the character encoding and the visual glyph used to render it in the user interface.
- NSTextStorage—the character string data repository for a group of one or more text handling objects. It associates attributes—font, color, kerning and so on—with the character string.

There are additional objects and protocols for complete text handling. See the text sections in Programming Topics for more information.



Fancy strings with NSAttributedString

At the core of the text system is NSAttributedString, a class that can represent an arbitrary string of text along with associated formatting attributes. Not a subclass of NSString, NSAttributedString uses an NSString to store the essential sequence of characters. In addition, the attributed string contains a set of zero or more attribute specifications—a range of the string and a set of attributes that apply to that range. The attribute set is rich and includes not only fundamentals like fonts and colors, but text formatting specifics like superscripts, ligatures and the like.

NSAttributedStrings can be used for a variety of purposes besides the text view system. Formatters can provide attributed strings to text cells for rich display of value classes. Attributed strings know how to draw themselves in the PostScript context of a custom view.

Important ideas from this section

» NSTextView and its companion objects provide your application with a very sophisticated text display and editing object completely functional and ready for use

» NSTextView has a variety of configurable attributes and handles three different text types:

ASCII

RTF

RTFD

» Effective programmatic handling of NSTextView requires some basic familiarity with the following:

NSString

NSRange

NSData

» NSTextView can take a delegate and provides a set of useful notifications for your controllers to track user actions

» Working together, the Application Kit and Interface Builder provide you with a rich set of panels, menus and text-based operations that you can easily incorporate into an application for full-featured text-based controls and views.

Classes featured in this section

» NSTextView

» NSData

» NSString

» NSFont

» NSAttributedString

REVIEW WORKING WITH NSTEXTVIEW

1. How would you programmatically clear the contents of an NSTextView?

2. How would you programmatically scroll to the bottom of an NSTextView?

3. Name at least two other classes commonly used in conjunction with NSTextView.

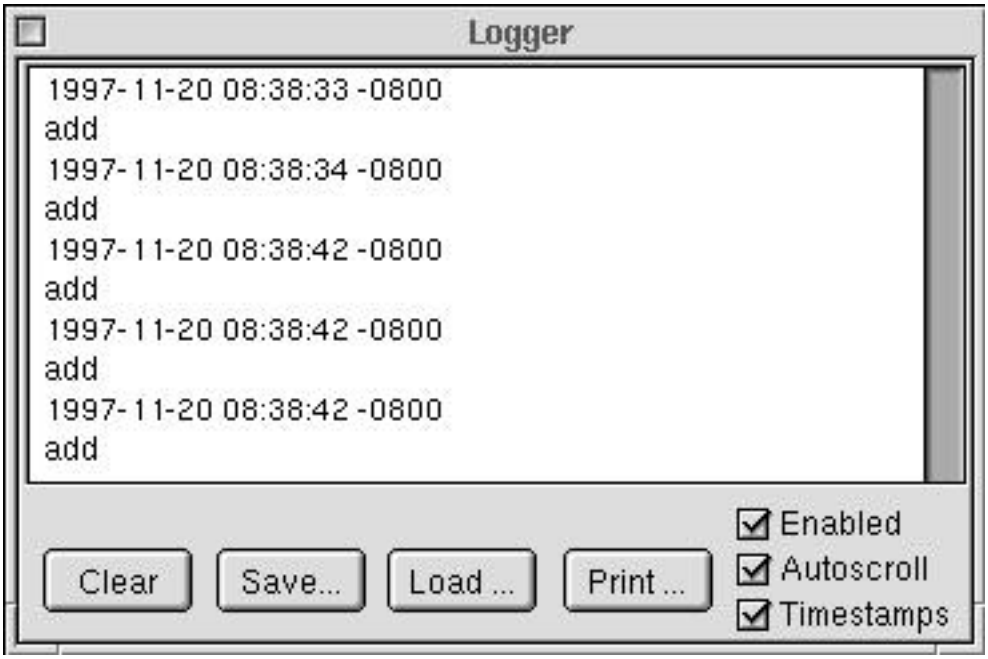
4. You are attempting to connect the NSTextView's delegate in Interface Builder. How can you tell if you have selected the NSTextView rather than the surrounding NSScrollView?

5. You can group multiple text views inside an NSSplitView. How might you configure a Print menu item to work with both of them? How might you programmatically access the one that the user is currently typing in?

EXERCISE 15.1 REUSABLE LOG PANEL COMPONENT

Many applications can make use of a logging component. Data is written to the log at various points in the application. Like a console or detail inspector, a log view can be displayed and the contents cleared, saved to a file, printed and so on. While some systems provide a console or terminal window, a log component is portable, dedicated and arguably more sophisticated. It lends itself to a wide range of uses—debugging, transaction monitoring, low level detail capture, such as SQL statements, and other behind the scenes mechanics. The ability to time stamp the log entries may prove especially useful.

In this exercise you construct a reusable log panel component that contains a text view object, which is responsible for recording various actions during the life of the application. The user is able to access this log at any time via the main menu and can control various properties thereof.



Objectives

After completing this exercise, you will be able to:

- Programmatically manage a text view object and provide cut, copy and paste functionality in an application
- Store and retrieve text to and from the file system using the text view

Exercise

1. As usual, you continue to extend the existing application. If you want to keep a working backup copy of the previous version, do it now. You may also develop this in the context of a new project whose main user interface is a single window with a text field for entering data to be logged.

2. Looking at the Log panel interface, you can see that there are many features. You should implement the basic logging features first and incrementally add as you go. Consider implementing the features in the following order:

    - Appending messages to the log—the text view
    - Clearing the log
    - Auto-scrolling
    - Printing
    - Saving to and Loading from a file
    - Enabling/disabling
    - Timestamping

3. The tasks involved in adding a log panel are threefold:

    - Creating the user interface.
    - Designing and implementing the controller object that connects the user interface and the rest of the application. In addition to controlling the user interface, the Logger object provides programmatic interfaces for logging and configuring Logger behavior.
    - Adding the logging entry points to various parts of the application. There are two ways to provide access to logging services. One requires other objects to message the Logger directly; it must be accessible via the application delegate. The other approach is more loosely coupled. It requires objects to post notifications which theLogger is registered to receive. The string to be logged can be passed using the userInfo attribute of the notification. Choose one method. You can implement the other as an enhancement later on in the exercise.

4. In the interest of time, the Logger panel user interface is already built for you. The various controls are there to demonstrate the capabilities of the text object and you can implement them in stages, testing each as you go.

    - Create a new component subproject called Logger and add the new class and nib files found in **ExerciseMaterials** to that subproject.
    - The **ExerciseMaterials** also provide skeleton **Logger.h** and **Logger.m**. Use these as the basis of your Log Panel controller object.
    - Open the pre-built **Logger.nib** file. Read in the **Logger.h**. Configure File's Owner and connect all outlets and target/actions.

- You should load the Logger nib in theLogger's **init** method so you can use the text view to store the log messages. Otherwise you will have no buffer to stash the logged messages until the text view is instantiated.

5. The Logger panel should appear on request from the user. Add a command to the Tools submenu to launch the logger panel.

   - Decide whether you want to include the logger controller in the main nib or whether you want to create it dynamically. This decision will be based in part on how you implement the logging API.
   - You will need to add an action and outlet to the AppController in order to connect the command if you create it dynamically. Remember to re-read **AppController.h** into Interface Builder so you can connect to the new outlet and action.
   - If you put the Logger controller in the main nib, you will still require an outlet in AppController to connect it and an accessor method to dispense it, so it can be reached via the application delegate.

   Note: If you want to message the Logger class from another part of the application, you will have to make its interface file a Project Header so it can be included by other classes.

6. After configuring and loading the nib, your biggest job is to implement the methods in Logger.m. Read through the methods to see what is already there and to get a feel for the suggested design. Implement features one by one following the list above. One common question is how to append a newline to a string passed in for logging? You can easily generate a new autoreleased string like so:

```
[aString stringByAppendingString:@"\n"]
```

7. Try out the application. Check that when a message is logged it appears in the view. What happens when the panel is not visible?

8. Select the text in the panel and try copying and pasting it to another application. Should the text view be read only or editiable?

Enhancements

- Add all the features you can see in the example picture, if you haven't already. Note that to **print** the log file you just need to connect the print: action of the text view to the button you see in the above picture.
- Provide a default to switch logging on or off on startup.
- Implement the second logging interface—use notifications, if you previously used direct messaging, or messaging if you used notifications.
- Add rich text capability to the logger so that the text can be saved and loaded with different fonts.