Scroll View Programming Guide for Cocoa

Cocoa > Graphics & Imaging

2006-06-28

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

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to Scroll View Programming Guide for Cocoa

A scroll view displays a portion of the contents of a view that's too large to be displayed in a window and allows the user to move the document view within the scroll view. This document describes the `NSScrollView` class and its use.

Who Should Read This Document

You should read this document if your application needs to display content that is too large to fit in a single view. *View Programming Guide for Cocoa* should be considered a prerequisite to this document, as is *Cocoa Event-Handling Guide*.

Organization of This Document

Scroll View Programming Guide for Cocoa consists of the following articles:

- "How Scroll Views Work" (page 9) describes the components of scroll views and how they interact.
- "Creating and Configuring a Scroll View" (page 13) describes how to create and configure scroll views.
- "Scrolling the Document View" (page 17) describes how an application programmatically scrolls the contents of a scroll view.
- "Synchronizing Scroll Views" (page 21) describes how to synchronize scrolling of two scroll views.

See Also

There are other technologies, not fully covered in this document, that are fundamental to using scroll views in your application. Refer to these documents for more details:

- *Cocoa Event-Handling Guide* describes the event model used by Cocoa applications.
- *Cocoa Event-Handling Guide* describes how your application objects can handle the events that they receive and explains the responder chain.
- *View Programming Guide for Cocoa* describes the view hierarchy and how to implement custom views in your application.

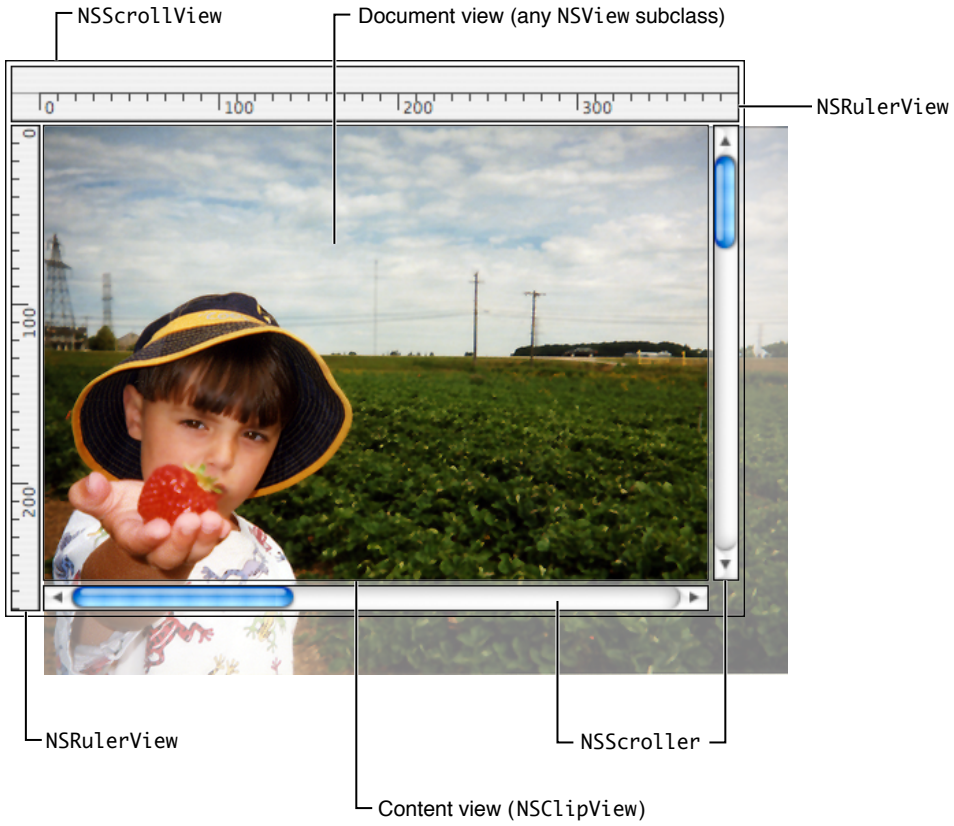
How Scroll Views Work

Scroll views act as the central coordinator for the Application Kit's scrolling machinery, managing instances of scrollers, rulers, and clipping views. A scroll view changes the visible portion of the displayed document in response to user-initiated actions or to programmatic requests by the application. This article describes the various components of a scroll view and how the scrolling mechanism works.

Components of a Scroll View

Cocoa provides a suite of classes that allow applications to scroll the contents of a view. Instances of the `NSScrollView` class act as the container for the views that work together to provide the scrolling mechanism. Figure 1 shows the possible components of a scroll view.

Figure 1 Exploded view of a scroll view's components



The Document View

`NSScrollView` instances provide scrolling services to its **document view**. This is the only view that an application must provide a scroll view. The document view is responsible for creating and managing the content scrolled by a scroll view.

The Content View

`NSScrollView` objects enclose the document view within an instance of `NSClipView` that is referred to as the **content view**. The content view is responsible for managing the position of the document view, clipping the document view to the content view's frame, and handling the details of scrolling in an efficient manner. The content view scrolls the document view by altering its bounds rectangle, which determines where the document view's frame lies. You don't normally interact with the `NSClipView` class directly; it is provided primarily as the scrolling machinery for the `NSScrollView` class.

Scroll Bars

The `NSScroller` class provides controls that allow the user to scroll the contents of the document view. Scroll views can have a horizontal scroller, a vertical scroller, both, or none. If the scroll view is configured with scrollers, the `NSScrollView` class automatically creates and manages the appropriate control objects. An application can customize these controls as required. See "How Scrollers Interact with Scroll Views" (page 11) for more information.

Rulers

Scroll views also support optional horizontal and vertical rulers, instances of the `NSRulerView` class or a custom subclass. To allow customization, rulers support accessory views provided by the application. A scroll view's rulers don't automatically establish a relationship with the document view; it is the responsibility of the application to set the document view as the ruler's client view and to reflect cursor position and other status updates. See *Rulers and Paragraph Styles* for more information.

How Scrolling Works

A scroll view's document view is positioned by the content view, which sets its bounds rectangle in such a way that the document view's frame moves relative to it. The action sequence between the scrollers and the corresponding scroll view, and the manner in which scrolling is performed, involve a bit more detail than this.

Scrolling typically occurs in response to a user clicking a scroller or dragging the scroll knob, which sends the `NSScrollView` instance a private action message telling it to scroll based on the scroller's state. This process is described in "How Scrollers Interact with Scroll Views" (page 11). If you plan to implement your own kind of scroller object, you should read that section.

The `NSClipView` class provides low-level scrolling support through the `scrollToPoint:` method. This method translates the origin of the content view's bounds rectangle and optimizes redisplay by copying as much of the rendered document view as remains visible, only asking the document view to draw newly exposed regions. This usually improves scrolling performance but may not always be appropriate. You can turn this behavior off using the `NSClipView` method `setCopiesOnScroll:` passing `NO` as the parameter. If you do leave copy-on-scroll active, be sure to scroll the document view programmatically using the `NSView` method `scrollPoint:` method rather than `translateOriginToPoint:`.

Whether the document view scrolls explicitly in response to a user action or an `NSClipView` message, or implicitly through a `setFrame:` or other such message, the content view monitors it closely. Whenever the document view's frame or bounds rectangle changes, it informs the enclosing scroll view of the change with a `reflectScrolledClipView:` message. This method updates the `NSScroller` objects to reflect the position and size of the visible portion of the document view.

How Scrollers Interact with Scroll Views

`NSScroller` is a public class primarily for developers who decide not to use an instance of `NSScrollView` but want to present a consistent user interface. Its use outside of interaction with scroll views is discouraged, except in cases where the porting of an existing application is more straightforward.

Configuring an `NSScroller` instance for use with a custom container view class (or a completely different kind of target) involves establishing a target-action relationship as defined by `NSControl`. In the case of the scroll view, the target object is the content view. The target object is responsible for implementing the action method to respond to the scroller, and also for updating the scrollers in reponse to changes in target.

As the scroller tracks the mouse, it sends an action message to its target object, passing itself as the parameter. The target object then determines the direction and scale of the appropriate scrolling action. It does this by sending the scroller a `hitPart` message. The `hitPart` method returns a part code that indicates where the user clicked in the scroller. Table 1 shows the possible codes returned by the `hitPart` method.

Table 1 Part codes returned by `hitPart`

Part code returned by `hitPart`	**Scrolling behavior**
`NSScrollerNoPart`	The scroll view should not scroll at all.
`NSScrollerIncrementLine`	The scroll view should scroll down or to the right by the amount specified by the `verticalLineScroll` or `horizontalLineScroll` methods, dependent on the scroller's orientation.
`NSScrollerDecrementLine`	The scroll view should scroll up or to the left by the amount specified by the `verticalLineScroll` or `horizontalLineScroll` methods, dependent on the scroller's orientation.
`NSScrollerIncrementPage`	The scroll view should scroll down or to the right by the amount specified by the `verticalPageScroll` or `horizontalPageScroll` methods, dependent on the scroller's orientation.
`NSScrollerDecrementPage`	The scroll view should scroll up or to the left by the amount specified by the `verticalPageScroll` or `horizontalPageScroll` methods, dependent on the scroller's orientation.

Part code returned by `hitPart`	Scrolling behavior
`NSScrollerKnob` or `NSScrollerKnobSlot`	The scroll view should scroll directly to the horizontal or vertical location—dependent on the scroller's orientation—returned by the scroller's `floatValue` method.

The target object tracks the size and position of its document view and updates the scroller to indicate the current position and visible proportion of the document view by sending the appropriate scrollers a `setFloatValue:knobProportion:` message, passing the current scroll location. The knob proportion parameter is a floating-point value between 0 and 1 that specifies how large the knob in the scroller should appear. `NSClipView` overrides most of the `NSView` `setBounds...` and `setFrame...` methods to perform this updating.

Creating and Configuring a Scroll View

Scroll views can be created and configured programmatically or in Interface Builder. This article describes both procedures.

Creating a Scroll View in Interface Builder

Creating a scroll view in Interface Builder is straightforward.

1. Create the view, or views, that will be the document view of the scroll view.
2. Select that view, or views, that will be the scroll view's document view.
3. Choose Layout > Make subviews of > Scroll View. This creates a new `NSScrollView` instance with the selected view, or views, as its document view.
4. Open the inspector and configure the visible scrollers, background color, and line and page scroll amounts.

Creating a Scroll View Programmatically

Applications often create scroll views programmatically. Instances of `NSScrollView` are created using the `initWithFrame:` method, specifying the position and size of the scroll view's frame rectangle. After initializing the `NSScrollView` instance you must, at a minimum, set the document view using the method `setDocumentView:`.

The example code in Listing 1 demonstrates how to create a scroll view for an `NSImageView` instance that is large enough to display the entire image.

Listing 1 Creating a scroll view instance programmatically

```
// theWindow is an IBOutlet that is connected to a window
// theImage is assumed to be declared and populated already

// determine the image size as a rectangle
// theImage is assumed to be declared elsewhere
NSRect imageRect=NSMakeRect(0.0,0.0,[theImage size].width,[theImage size].height);


// create the image view with a frame the size of the image
NSImageView *theImageView=[[NSImageView alloc] initWithFrame:imageRect];
[theImageView setBounds:imageRect];

// set the image for the image view
```

```
[theImageView setImage:theImage];

// create the scroll view so that it fills the entire window
// to do that we'll grab the frame of the window's contentView
// theWindow is an outlet connected to a window instance in Interface Builder
NSScrollView *scrollView = [[NSScrollView alloc] initWithFrame:
                                        [[theWindow contentView] frame]];

// the scroll view should have both horizontal
// and vertical scrollers
[scrollView setHasVerticalScroller:YES];
[scrollView setHasHorizontalScroller:YES];

// configure the scroller to have no visible border
[scrollView setBorderType:NSNoBorder];

// set the autoresizing mask so that the scroll view will
// resize with the window
[scrollView setAutoresizingMask:NSViewWidthSizable|NSViewHeightSizable];

// set theImageView as the documentView of the scroll view
[scrollView setDocumentView:theImageView];

// setting the documentView retains theImageView
// so we can now release the imageView
[theImageView release];

// set the scrollView as the window's contentView
// this replaces the existing contentView and retains
// the scrollView, so we can release it now
[theWindow setContentView:scrollView];
[scrollView release];


// display the window
[theWindow makeKeyAndOrderFront:nil];
}
```

Warning: A scroll view's frame rectangle and the enclosed clip view's frame rectangle must be pixel aligned. If they are not, redrawing in response to user scrolling is blurred.

When created programmatically, scroll views have no scrollers. You specify that a scroll view should display scrollers by passing an argument of `YES` to the methods `setHasVerticalScroller:` and `setHasHorizontalScroller:`. The scroll view allocates and displays the scrollers automatically. You can configure a scroll view to display its scrollers only when the document view is sufficiently large to require scrolling using the method `setAutohidesScrollers:`, passing `YES` as the parameter.

Calculating the Size of a Scroll View

It is difficult to calculate the frame size of a scroll view if you know only the required size of the content view. `NSScrollView` provides the convenience class method `frameSizeForContentSize:hasHorizontalScroller:hasVerticalScroller:borderType:` to determine the scroll view's frame size based on the content area, presence of scroll bars, and the scroll view's border type. The method returns a rectangle that is suitable for using with `initWithFrame:`.

Setting Scrolling Increments

`NSScrollView` defines two levels of incremental scrolling: a line scroll increment and a page scroll increment. Each of these values can be configured separately for a scroll view's horizontal and vertical scrolling directions.

The line scroll increment moves the document view by a small amount, often in response to the user clicking the scroll buttons of a scroller. The line scroll increment is set for both horizontal and vertical directions using the `setLineScroll:` method. Alternatively, you can specify line scroll increment for the vertical and horizontal scrollers separately using the `setVerticalLineScroll:` and `setHorizontalLineScroll:` methods, respectively.

Scrolling by page moves the document view by a larger amount, typically the size of the document view's visible region. The page scroll amount is specified using the `setPageScroll:` method or separately using the `setVerticalPageScroll:` and `setHorizontalPageScroll:` methods. The page scroll amount differs from the line scroll increment in that the page scroll values specify the amount of visible view content that will remain when the document view scrolls. Setting the page scroll value to 0.0 implies that the entire visible portion of the document view is replaced when a page scroll occurs.

Users scroll directly to a location in the document by dragging a scroller's knob. By default, a scroll view updates the contents dynamically as it scrolls. You disable dynamic scrolling by sending the `setScrollsDynamically:` method to the scroll view, passing `NO` as the parameter. In dynamic scrolling is turned off, updates occur only when the user releases the mouse.

Scrolling the Document View

A scroll view's document view scrolls in response to one of the following actions:

- The user clicks in one of the scrollers or rotates the mouse scroll wheel or scroll ball. These cases are handled automatically by the `NSScrollView` class.
- The scroll view must scroll to a specific location—for example, to display the current selection in response to a user action. The application is responsible for implementing this functionality as described in "Scrolling To a Specific Location" (page 17).
- The user drags the mouse outside the scroll view, causing autoscrolling to occur. The document view is responsible for supporting autoscrolling in response to mouse-drag events as described in "Supporting Automatic Scrolling" (page 18).

Scrolling To a Specific Location

Applications often need to scroll to a specific location in a document view because of some user action unrelated to scrolling. For example, when a user searches a document they expect the document to scroll to show the found item. The `NSView` class provides high-level scrolling methods that automatically update the scrollers and redisplay the document view as required. The `NSClipView` and `NSScrollView` classes methods provide low-level scrolling support that requires the application to update the scrollers and mark the document view for display.

Two `NSView` methods support scrolling to reveal a specific location: `scrollPoint:` and `scrollRectToVisible:`. These high-level methods scroll the specified point or rectangle to the origin of the content view. You send these methods to the scroll view's document view or to one of its descendants. Scroll messages are passed up through the view hierarchy to the nearest enclosing `NSClipView` instance. `NSView` also provides a convenience method, `enclosingScrollView`, that returns the `NSScrollView` instance that contains the receiver, allowing the view to interact directly with a parent scroll view. If the receiver is not contained in a scroll view, `enclosingScrollView` returns `nil`.

The code fragment in Listing 1 illustrates how to scroll to the top and bottom of the document view. The orientation of the document view determines where the origin of the content view lies and you must allow for this when calculating the top and bottom locations.

Listing 1 Scrolling to the bottom or top of the document view

```
- (void)scrollToTop:sender;
{
    NSPoint newScrollOrigin;

    // assume that the scrollview is an existing variable
    if ([[scrollview documentView] isFlipped]) {
        newScrollOrigin=NSMakePoint(0.0,0.0);
    } else {
        newScrollOrigin=NSMakePoint(0.0,NSMaxY([[scrollview documentView] frame])
```

```
                                        -NSHeight([[scrollview contentView] bounds]));
    }

    [[scrollview documentView] scrollPoint:newScrollOrigin];

}

- (void)scrollToBottom:sender;
{
    NSPoint newScrollOrigin;

    // assume that the scrollview is an existing variable
    if ([[scrollview documentView] isFlipped]) {
        newScrollOrigin=NSMakePoint(0.0,NSMaxY([[scrollview documentView] frame])
                                       -NSHeight([[scrollview contentView] bounds]));
    } else {
        newScrollOrigin=NSMakePoint(0.0,0.0);
    }

    [[scrollview documentView] scrollPoint:newScrollOrigin];

}
```

The low-level scrolling methods bypass the `adjustScroll:` mechanism described in "Constraining Scrolling."

Supporting Automatic Scrolling

Scroll views scroll automatically as the user drags the mouse outside of the scroll view's content area if the document view, or a descendent, calls the `NSView` method `autoscroll:` as it handles the mouse-drag event. Supporting autoscrolling allows the user to drag the mouse, moving or selecting items, and have the scroll view continually display the active portion of the document.

View subclasses should send `autoscroll:` messages as part of their mouse-drag handling code, passing the current `NSEvent` object as the parameter. The `autoscroll:` method does nothing if the receiver is not contained within a scroll view. Listing 2 shows a typical `NSView` subclass's implementation of a `mouseDragged:` method that supports automatic scrolling.

Listing 2 Supporting automatic scrolling in a `mouseDragged:` implementation

```
-(void)mouseDragged:(NSEvent *)event
{
    NSPoint dragLocation;
    dragLocation=[self convertPoint:[event locationInWindow]
                           fromView:nil];

    // support automatic scrolling during a drag
    // by calling NSView's autoscroll: method
    [self autoscroll:event];

    // act on the drag as appropriate to the application
}
```

Determining the Current Scroll Location

You can determine the current visible location in a scroll view by examining the bounds of its clip view. The origin of the clip view's bounds is suitable for using with `scrollPoint:` to restore the scroll location later. The code fragment in Listing 3 demonstrates getting a scroll view's current scroll location and restoring it.

Listing 3 Getting and setting the current scroll location of a view

```
// get the current scroll position of the document view
NSPoint currentScrollPosition=[[theScrollView contentView] bounds].origin;

// restore the scroll location
[[theScrollView documentView] scrollPoint:currentScrollPosition];
```

Constraining Scrolling

Subclasses of `NSView` override the `adjustScroll:` method to provide a view fine-grained control of its position during scrolling. A custom view subclass can quantize scrolling into regular units—to the edges of a spreadsheet's cells, for example—or simply limit scrolling to a specific region of the view. The `adjustScroll:` method provides the proposed rectangle that the scrolling mechanism will make visible and expects the subclass to return the passed rectangle or an altered rectangle that will constrain the scrolling.

Listing 4 shows an implementation of `adjustScroll:` that constrains scrolling of the view to 72 pixel increments, even when dragging the scroll knob.

Listing 4 Constraining scrolling with `adjustScroll:`

```
- (NSRect)adjustScroll:(NSRect)proposedVisibleRect
{
    NSRect modifiedRect=proposedVisibleRect;

    // snap to 72 pixel increments
    modifiedRect.origin.x = (int)(modifiedRect.origin.x/72.0) * 72.0;
    modifiedRect.origin.y = (int)(modifiedRect.origin.y/72.0) * 72.0;

    // return the modified rectangle
    return modifiedRect;
}
```

The `adjustScroll:` method is not used when scrolling is initiated by lower-level scrolling methods provided by `NSClipView` (`scrollToPoint:`) and `NSScrollView` (`scrollRectToVisible:`).

Synchronizing Scroll Views

Some application user interfaces require that scrolling in a scroll view causes synchronized scrolling in another scroll view. You synchronize the scrolling of two scroll view's by having the views register to receive notifications of the changes in the other's content view bounds.

For example, an interface has two adjacent scroll views: right and left scroll views. The right scroll view registers to receive bounds notifications sent by the content view of the left scroll view. Similarly, the left scroll view registers for bounds notification changes sent by the right scroll view's content view. When a scroll view receives a bounds change notification, it determines which content view changed by examining the object returned by sending the notification an object method. It then checks to see if its origin is already at the origin of the content view bounds and if it is not, it scrolls to that location. This check ensures that the scroll view doesn't scroll if it is already at the correct location, preventing notifications being sent to the other scroll view and causing an infinite loop.

Implementing this synchronizing in a generic way is best done by subclassing `NSScrollView`. The `SynchroScrollView` class example synchronizes scrolling only in the vertical plane and requires the following:

- An instance variable that tracks the synchronized view (`synchronizedScrollView`)
- A method to set the view to synchronize with (`setSynchronizedScrollView:`)
- A method to break synchronization (`stopSynchronizing`)
- A method that receives the bounds change notifications (`synchronizedViewContentBoundsDidChange:`)

The `SynchroScrollView` implementation relies on the scroll view being retained by some other object, typically the view hierarchy itself. It is the application's responsibility to break the synchronization between the views before the views are deallocated.

Listing 1 shows the class declaration.

Listing 1 `SynchroScrollView` class declaration

```
@interface SynchroScrollView : NSScrollView {
    NSScrollView* synchronizedScrollView; // not retained
}

- (void)setSynchronizedScrollView:(NSScrollView*)scrollview;
- (void)stopSynchronizing;
- (void)synchronizedViewContentBoundsDidChange:(NSNotification *)notification;

@end
```

The `synchronizedScrollView` variable is used to track the synchronized scroll view. Both scroll views that are synchronized are set as each other's `synchronizedScrollView`. As a result, the `synchronizedScrollView` is a weak reference and it is not retained.

An application initiates synchronization by sending both scroll view's a `setSynchronizedScrollView:` message, passing the other scroll view as the parameter. Listing 2 shows the `SynchroScrollView` implementation of `setSynchronizedScrollView:`.

Listing 2 `SynchroScrollView` implementation of `setSynchronizedScrollView:`

```
- (void)setSynchronizedScrollView:(NSScrollView*)scrollview
{
    NSView *synchronizedContentView;

    // stop an existing scroll view synchronizing
    [self stopSynchronizing];

    // don't retain the watched view, because we assume that it will
    // be retained by the view hierarchy for as long as we're around.
    synchronizedScrollView = scrollview;

    // get the content view of the
    synchronizedContentView=[synchronizedScrollView contentView];

    // Make sure the watched view is sending bounds changed
    // notifications (which is probably does anyway, but calling
    // this again won't hurt).
    [synchronizedContentView setPostsBoundsChangedNotifications:YES];

    // a register for those notifications on the synchronized content view.
    [[NSNotificationCenter defaultCenter] addObserver:self
                        selector:@selector(synchronizedViewContentBoundsDidChange:)
                        name:NSViewBoundsDidChangeNotification
                          object:synchronizedContentView];
}
```

This method first stops any existing synchronization by calling `stopSynchronizing`. It then keeps a reference to the new scroll view in `synchronizedScrollView`. Finally, it registers to receive bounds change notifications for the content view of the `synchronizedScrollView`. The change notifications are sent to the `synchronizedViewContentBoundsDidChange:` method, shown in Listing 3.

Listing 3 `SynchroScrollView` implementation of `synchronizedViewContentBoundsDidChange:`

```
- (void)synchronizedViewContentBoundsDidChange:(NSNotification *)notification
{
    // get the changed content view from the notification
    NSView *changedContentView=[notification object];

    // get the origin of the NSClipView of the scroll view that
    // we're watching
    NSPoint changedBoundsOrigin = [changedContentView bounds].origin;

    // get our current origin
    NSPoint curOffset = [[self contentView] bounds].origin;
    NSPoint newOffset = curOffset;

    // scrolling is synchronized in the vertical plane
    // so only modify the y component of the offset
    newOffset.y = changedBoundsOrigin.y;

    // if our synced position is different from our current
```

```
    // position, reposition our content view
    if (!NSEqualPoints(curOffset, changedBoundsOrigin))
    {
    // note that a scroll view watching this one will
    // get notified here
    [[self contentView] scrollToPoint:newOffset];
    // we have to tell the NSScrollView to update its
    // scrollers
    [self reflectScrolledClipView:[self contentView]];
    }
}
```

The view that triggered the change notification is determined by sending the notification an `object` message. The origin of that content view is determined and the new origin that the receiving scroll view should reflect is determined. If the two locations differ, the receiver scrolls to the new origin. This test is necessary to prevent the scroll view that originated the scrolling action from acting on the change in the other scroll view's content view, causing an infinite loop.

The `stopSynchronizing` method, shown in Listing 4, causes the receiver to stop listening for bounds-change notifications in the synchronized scroll view's content view and sets the `synchronizedScrollView` instance variable to `nil`.

Listing 4 `SynchroScrollView` implementation of `stopSynchronizing`

```
- (void)stopSynchronizing
{
    if (synchronizedScrollView != nil) {
    NSView* synchronizedContentView = [synchronizedScrollView contentView];

    // remove any existing notification registration
    [[NSNotificationCenter defaultCenter] removeObserver:self
                            name:NSViewBoundsDidChangeNotification
                              object:synchronizedContentView];

    // set synchronizedScrollView to nil
    synchronizedScrollView=nil;
    }
}
```

Document Revision History

This table describes the changes to *Scroll View Programming Guide for Cocoa*.

Date	Notes
2006-06-28	Corrected typos.
2006-05-23	Corrected minor typos.
2006-04-04	Updated synchronized scroll example.
2006-03-08	New document that describes how to use scroll views in Cocoa applications. Some of the information in this document previously appeared in "Drawing and Views."