Rulers and Paragraph Styles

Cocoa > Text & Fonts

2007-09-04

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 Rulers and Paragraph Styles

Rulers and Paragraph Styles describes paragraph styles and the ruler views and markers that enable users to view and change the paragraph style attributes used in text.

Who Should Read This Document

You should read this document if you need to work directly with paragraph style objects and their associated ruler views.

Organization of This Document

This document contains the following articles:

- "About Paragraph Styles" (page 9) describes paragraph styles and the attributes in them you can change.
- "Breaking Lines by Truncation" (page 11) describes the line break mode attribute of paragraph styles and illustrates how the truncation settings work in the context of table cells.
- "About Ruler Views" (page 13) describes what's in a ruler view and how it works.
- "About Ruler Markers" (page 15) describes what a ruler marker is and how it works.
- "Setting Up a Ruler View" (page 17) describes how to add and set up a ruler view for a particular view.
- "Changing a Ruler's Measurement Units" (page 19) describes how to set a ruler's measurement units and how to define your own measurement units.
- "Using a Ruler View's Client" (page 21) describes how to set a ruler view's client and adjust the ruler view's markers and layout to match the client.
- "Letting Users Manipulate Markers" (page 23) describes how to react when the user adds, removes, and moves markers, or when the user clicks in the ruler view.
- "Updating the Ruler View" (page 25) describes how to change the ruler view to reflect the current selection or changes the user makes in the view.

See Also

For more information, refer to the following documents:

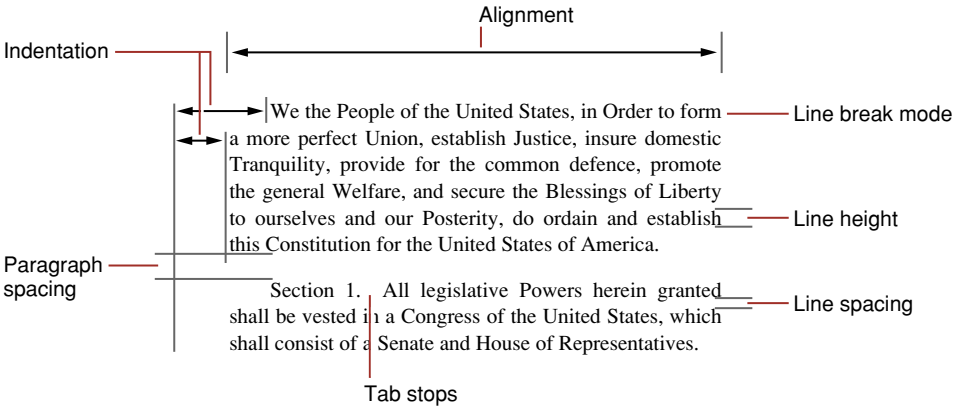
- *Text Attributes* describes the text-related attributes maintained by the Cocoa text system, including paragraph style attributes.
- *Attributed Strings Programming Guide* provides more details about how the text system stores attributes with strings.

About Paragraph Styles

NSParagraphStyle and its subclass NSMutableParagraphStyle encapsulate the paragraph or ruler attributes used by the NSAttributedString classes. Instances of these classes are often referred to as paragraph style objects, or when no confusion will result, as paragraph styles.

A paragraph style object represents a complex attribute value in an attributed string, storing a number of subattributes that affect paragraph layout for the characters of the string. Among these subattributes are alignment, tab stops, and indents. Figure 1 illustrates these and other paragraph style attributes.

Figure 1 Paragraph style attributes



These are the paragraph style attributes the text system uses:

- `alignment` is the text alignment. It is `NSLeftTextAlignment`, `NSRightTextAlignment`, `NSCenterTextAlignment`, `NSJustifiedTextAlignment`, or `NSNaturalTextAlignment`.
- `firstLineHeadIndent` is the distance in points from the leading margin of a text container to the beginning of the paragraph's first line.
- `headIndent` is the distance in points from the leading margin of a text container to the beginning of lines other than the first.
- `tailIndent` is the distance in points from the margin of a text container to the end of lines.
- `tabStops` is an array of NSTextTab objects, sorted by location, that define the tab stops for the paragraph style.
- `lineBreakMode` is the mode that should be used to break lines when laying out the paragraph's text. It can be one of the following:
  - `NSLineBreakByWordWrapping` wraps at word boundaries.
  - `NSLineBreakByCharWrapping` wraps at character boundaries.
  - `NSLineBreakByClipping` clips lines past the edge of the text container.

- `NSLineBreakByTruncatingHead` displays each line so that the end fits in the container and the missing text at the beginning is indicated by an ellipsis glyph.
- `NSLineBreakByTruncatingTail` displays each line so that the beginning fits in the container and the missing text at the end is indicated by an ellipsis glyph.
- `NSLineBreakByTruncatingMiddle` displays each line so that the beginning and end both fit in the container and the missing text in the middle is indicated by an ellipsis glyph.

- `maximumLineHeight` is the maximum height that any line in the receiver can occupy, regardless of the font size or size of any attached graphic.
- `minimumLineHeight` is the minimum height that any line in the receiver can occupy, regardless of the font size or size of any attached graphic.
- `lineHeightMultiple` is a factor by which the default line height (a metric of the font) is multiplied before being constrained by minimum and maximum line height.
- `lineSpacing` is extra space in points added between lines within the paragraph.
- `paragraphSpacing` is space in points added at the end of the paragraph to separate it from the following paragraph.
- `paragraphSpacingBefore` is space in points added between the top of the paragraph and the beginning of its text content.

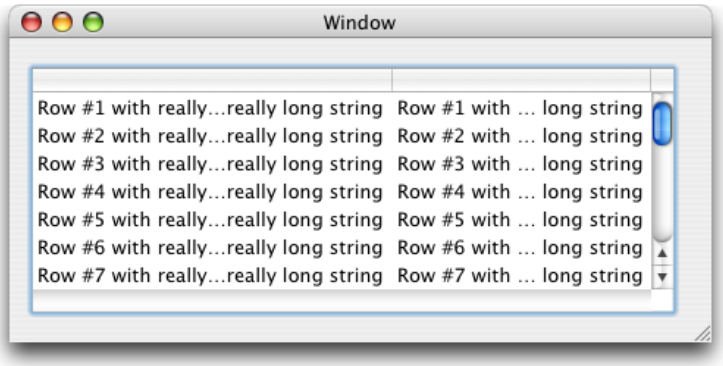
Breaking Lines by Truncation

In Mac OS X version 10.3 and later, paragraph style objects (NSParagraphStyle and NSMutableParagraphStyle) have support for line breaking by truncation. In addition to the previously available line-breaking modes of word wrapping, character wrapping, and clipping, paragraph styles can fit the line into the layout container by truncating the line at its head, tail, or middle. The missing text is indicated by an ellipsis glyph (…). This capability is very useful inside table cells, where space is likely to be limited and users can easily resize column widths.

The technique you use to configure a table view to break lines by truncation is simple but not entirely obvious. First, you configure the table column cells with the message `[cell setWraps:YES]`. Then, for the object value returned by the data source, use an attributed string configured with the desired paragraph style.

Figure 1 shows a table view displaying two columns of strings configured with a paragraph style line-breaking mode of `NSLineBreakByTruncatingMiddle` as shown in Listing 1.

Figure 1 Truncated strings displayed in table cells



The following example has a controller object with an outlet named `_tableView` representing the table view. The controller object is the table's data source. Listing 1 shows the entire controller implementation.

Listing 1 Controller implementation defining a table view using string truncation

```
@implementation Controller
- (void)awakeFromNib {
    NSArray *columns = [_tableView tableColumns];
    int index, count = [columns count];
    for (index = 0; index < count; index++) {
        NSTableColumn *column = [columns objectAtIndex:index];
        [[column dataCell] setWraps:YES];
    }
}

- (int)numberOfRowsInTableView:(NSTableView *)tableView {
    return 20;
```

```
}

- (id)tableView:(NSTableView *)tableView objectValueForTableColumn:(NSTableColumn
 *)tableColumn row:(int)row {
    static NSDictionary *info = nil;

    if (nil == info) {
       NSMutableParagraphStyle *style = [[NSParagraphStyle defaultParagraphStyle]
 mutableCopy];
        [style setLineBreakMode:NSLineBreakByTruncatingMiddle];
        info = [[NSDictionary alloc] initWithObjectsAndKeys:style,
NSParagraphStyleAttributeName, nil];
        [style release];
    }

    return [[[NSAttributedString alloc] initWithString:[NSString
stringWithFormat:@"Row #%d with really, really, really, really long string",
row + 1] attributes:info] autorelease];
}

@end
```

The `awakeFromNib` implementation configures the data cell for each column in the table view (the number of columns in the table is set up in Interface Builder). The key message is `[[column dataCell] setWraps:YES]`. This message tells the table to use the line-breaking mode of the attributed string that is the cell's object value. If the value passed with this call were `NO`, the cells' contents would merely be clipped.

The `tableView:objectValueForTableColumn:row:` method sets up a paragraph style object with a line-break mode of `NSLineBreakByTruncatingMiddle`. The method then places the paragraph style in an attributes dictionary. The last line returns an attributed string, with this dictionary, as the object value for each cell in the table.

About Ruler Views

An NSRulerView resides in an NSScrollView, displaying a labeled ruler and markers for its client, the document view of the NSScrollView or a subview of the document view. The client view can add and remove markers representing its contents, such as graphic elements, margins, and text tabs. The NSRulerView tracks user manipulation of the markers and informs the client view of those actions. NSRulerView handles both horizontal and vertical rulers, which are tiled in the scroll view above and to the side of the content view, respectively. NSRulerViews are sometimes called simply ruler views or even rulers.

A ruler view comprises three regions. First is the ruler area, where the ruler's baseline, hash marks, and labels are drawn. The ruler area is largely static, but it scales its hash marks to the document view's coordinate system and can display the hash marks in arbitrary units. The second region is the marker area, where ruler markers (NSRulerMarker objects) representing elements of the client view are displayed. The marker area is more dynamic, changing with the selection and state of the client view. The final region is the accessory view area, which is by default not present but appears when you add an accessory view to the ruler view. An accessory view can contain additional controls for manipulating the ruler's client view, such as alignment buttons or a set of markers that can be dragged onto the ruler.

A ruler view interacts with its client view in several ways. On appropriating the ruler view, the client view usually sets it up according to its needs. The client view can also dynamically update the ruler view's markers as its layout changes. In its turn, the ruler view informs the client view of actions the user takes on the ruler markers, allowing the client view to approve or limit the actions and to update its state based on the results of the actions.

The appearance of a ruler is based on both the document view and the client view. The document view, as the backdrop within the scroll view, serves as the canvas on which any client views are laid. Because of the document view's anchoring role, a ruler always draws its hash marks and labels relative to the document view's coordinate system. A vertical ruler also checks whether the document view is flipped and acts accordingly. However, the ruler view treats subviews of the document view as items laid out within the coordinate system defined by the document view, and so doesn't change its hash marks when a client view other than the document view is moved or scaled. For the client view's convenience it does, however, express marker locations in the client view's coordinate system. A few other operations that ruler views perform are defined in terms of the ruler's own coordinate system. The discussion of a feature or method makes clear which coordinate system applies. Table 1 summarizes the coordinate systems involved in using ruler views.

Table 1 Coordinate systems used with ruler views

Coordinate system	Used for
Client view	Marker locations
Document view	Calculating hash marks based on measurement units and scaling; origin offset for zero marks
Ruler view	Temporary ruler lines; component layout
Marker image	Image origin (which locates the image relative to the marker location)

About Ruler Markers

An `NSRulerMarker` object displays a symbol on an `NSRulerView` object, indicating a location for whatever graphic element it represents in the client of the ruler view (for example, a margin or tab setting, or the edges of a graphic on the page). A ruler marker comprises three primary attributes: the image it displays on the ruler view, the location of that image, and the object it represents. The `setImage:`, `setMarkerLocation:` and `setRepresentedObject:` methods set each of these attributes, respectively. In addition, a ruler marker records an offset for the image, allowing it to be placed relative to the marker location much in the way a cursor's hot spot relates a cursor image to the mouse location; the `setImageOrigin:` method establishes this offset.

Most of these attributes are set upon initialization by the `initWithRulerView:markerLocation:image:imageOrigin:` method. New ruler markers don't have represented objects; the client typically establishes the represented object in its `rulerView:didAddMarker:` method. A new ruler marker can be moved around in its ruler view, but not removed from it. The `setMovable:` and `setRemovable:` methods alter these default settings.

Represented objects allow the ruler view's client to distinguish among different attributes of the selection. In the ruler view client methods, the client can retrieve the marker's represented object to determine what attribute to alter. Generic attributes can be represented by string or other value objects, such as the edge names "Left", "Right", "Top", and "Bottom". Attributes already implemented as objects can be represented by those objects. For example, the text system records tab stops as `NSTextTab` objects, which include the tab location and its alignment. When an `NSTextView` object is the client view of a ruler, it simply makes the `NSTextTab` objects the represented objects of the ruler markers.

Setting Up a Ruler View

Adding a ruler view to a scroll view can be as simple as invoking the `NSScrollView` method `setHasHorizontalRuler:` and `setHasVerticalRuler:` methods. These create instances of the default ruler view class, which you can change using the `NSScrollView` class method `setRulerViewClass:`. You can also set ruler views directly on a per-instance basis using `setHorizontalRulerView:` and `setVerticalRulerView:`. Once you've added rulers to a scroll view, you can hide and reveal them using `setRulersVisible:`.

Beyond creating the rulers, you need take only two steps to set them up properly for use by the views contained within the scroll view: locate the zero marks of the rulers and reserve room for accessory views. You normally perform these steps only once, when setting up the `NSScrollView` object with rulers. However, if you allow the user to reset document attributes such as margins, you should change the zero mark locations as well. Also, if you reuse the scroll view by swapping in a new document view you may need to set up the rulers again with different settings.

The first step is to determine where you want the zero marks of the rulers to be located relative to the bounds origin of the document view. The zero marks are coincident with the bounds origin by default, but you can change this with the method `setOriginOffset:`. This method takes an offset specified in the document view's coordinate system. If you need to set the origin offset based on a point in a subview of the document view, such as a text view that's inset on a page, use `convertPoint:fromView:` to realize it in the document view's coordinate system. This Objective-C code fragment places the zero marks at the bounds origin of a client view, which lies somewhere inside the document view:

```
zero = [docView convertPoint:[clientView bounds].origin fromView:clientView];
[horizRuler setOriginOffset:zero.x - [docView bounds].origin.x];
```

After placing the zero marks, you should set up your rulers so that they don't change in size as the user works within the document view. For example, if two different subviews of the document view use different accessory views, the ruler view enlarges itself as necessary each time you change the accessory view. Such changes are at best unsightly and at worst confusing to the user. To avoid this problem, calculate ahead of time the sizes of the largest accessory view and the largest markers, and set the ruler view's required thickness for these elements using `setReservedThicknessForAccessoryView:` and `setReservedThicknessForMarkers:`. For example, if you have two accessory views for the horizontal ruler, one `16.0` PostScript units high and the other `24.0`, invoke `setReservedThicknessForAccessoryView:` with an argument of `24.0`.

Changing a Ruler's Measurement Units

A new ruler view automatically uses the user's preferred measurement units for drawing hash marks and labels, as stored in the user defaults system under the key `NSMeasurementUnit`. If your application allows the user to change his preferred measurement units, you can change them at runtime using `setMeasurementUnits:`, which takes the name of the units to use, such as `Inches` or `Centimeters`, and causes the ruler view to use that unit definition in spacing its hash marks and labels.

NSRulerView supports the units `Inches`, `Centimeters`, `Points`, and `Picas` by default. If your application uses other measurement units, your application should define and register them before creating any ruler views. To register a unit, use the class method `registerUnitWithName:abbreviation:unitToPointsConversionsFactor:stepUpCycle:stepDownCycle:`. Your application can register these wherever it's most convenient, such as in the NSApplication delegate method `applicationDidFinishLaunching:`.

This Objective-C code fragment registers a new unit called `Grummets`, with the abbreviation `gt`:

```
NSArray *upArray;
NSArray *downArray;

upArray = [NSArray arrayWithObjects:[NSNumber numberWithFloat:2.0], nil];
downArray = [NSArray arrayWithObjects:[NSNumber numberWithFloat:0.5],
    [NSNumber numberWithFloat:0.2], nil];
[NSRulerView registerUnitWithName:@"Grummets"
    abbreviation:NSLocalizedString(@"gt", @"Grummets abbreviation string")
    unitToPointsConversionFactor:100.0
    stepUpCycle:upArray stepDownCycle:downArray];
```

This Java code fragment does the same thing:

```
NSArray upArray;
NSArray downArray;

upArray = new NSArray (2.0, nil);
downArray = new NSArray (0.5, 0.2, nil);
[NSRulerView.registerUnit ("Grummets", "gt", 100.0, upArray, downArray]);
```

A `Grummet` is 100.0 PostScript units (points) in length, so a ruler view using it draws a major hash mark every 100.0 points when its document view is unscaled. If the document view is scaled, the ruler view spaces its hash marks accordingly.

The arguments `stepUpCycle` and `stepDownCycle` control how hash marks are drawn for fractions and multiples of units. NSRulerView attempts to place hash marks so that they're neither too crowded nor too sparse based on the current scale of the document view. It does so by drawing smaller hash marks for fractions of units where possible and by removing hash marks for whole units where necessary.

The value of `stepDownCycle` determines the fractional units checked by the ruler view. For example, with the unit `Grummets` defined above, the step down cycle is 0.5, then 0.2. With this cycle, the ruler view first checks to see if there's room for marks every half `Grummet`, placing them if there is. Then, it checks every fifth of the remaining space, or a tenth of a full `Grummet`, placing further hash marks if there's room. Then it returns to the first step in the cycle to further subdivide the ruler, and so on.

The value of `stepUpCycle` determines how many full unit marks get dropped when there isn't room for each one. The example uses a single-step cycle of 2.0, which means that each second `Grummet` hash mark is displayed if there isn't room for every one, then every fourth if there still isn't room, and so on.

Using a Ruler View's Client

Once you've set up a ruler view, as described in "Setting Up a Ruler View" (page 17), the scroll view's document view, or any subview of the document view, can become its client by sending it a `setClientView:` message. This method notifies the prior client that it's losing the ruler view using the `rulerView:willSetClientView:` method, removes all of the ruler view's markers, and sets the new client view. A client view normally appropriates the ruler when it becomes first responder and keeps it until some other view appropriates it. After appropriating the ruler view, the client needs to set up its layout and markers.

Adjusting the Layout

If the client has a custom accessory view, it sets that using `setAccessoryView:`. Clients without accessory views should avoid removing the ruler view's accessory view when appropriating the ruler, as this can cause unsightly screen flicker as the ruler is redrawn. It's better in this case for a client view that has an accessory view to implement `rulerView:willSetClientView:`, disabling the controls in the accessory view so that they're not active when other clients are using the ruler. Then, when the client view with the accessory view appropriates the ruler, it should set its accessory view again in case another client swapped the accessory view out, and reenable the controls.

Setting Ruler Markers

Aside from the layout of the ruler view itself, the client can also add markers to indicate the positions of its graphic elements, such as tabs and margins in text or the bounding boxes of drawn shapes or images. Each marker is an NSRulerMarker object, which displays a graphic image on the ruler at its given location and can be associated with an object that identifies the attribute indicated by the marker. You initialize an NSRulerMarker using its `initWithRulerView:markerLocation:image:imageOrigin:` method, which takes as arguments the NSRulerView where the marker is displayed, its location on the ruler in the client view's coordinate system, the image to display, and the point within the image that lies on the ruler's baseline. Once you've created the markers, you can use the NSRulerView methods `addMarker:` or `setMarkers:` to put them on the ruler. This Objective-C code fragment, for example, sets up markers denoting the left and right edges of the selected object's frame rectangle:

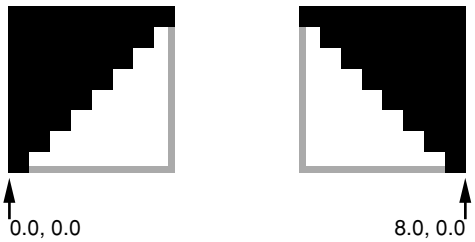
```
NSRulerMarker *leftMarker;
NSRulerMarker *rightMarker;

leftMarker = [[NSRulerMarker alloc] initWithRulerView:horizRuler
    markerLocation:NSMinX([selectedItem frame]) image:leftImage
    imageOrigin:NSMakePoint(0.0, 0.0)];

rightMarker = [[NSRulerMarker alloc] initWithRulerView:horizRuler
    markerLocation:NSMaxX([selectedItem frame]) image:rightImage
    imageOrigin:NSMakePoint(8.0, 0.0)];
```

```
[horizRuler setMarkers:[NSArray arrayWithObjects:leftMarker, rightMarker, nil]];
```

This Java code fragment does the same thing:

```
NSRulerMarker leftMarker;
NSRulerMarker rightMarker;

NSRect frame = selectedItem.frame();

leftMarker = new NSRulerMarker (horizRuler, frame.origin(), leftImage, new
Point(0.0, 0.0));

rightMarker = new NSRulerMarker (horizRuler, frame.maxX(), rightImage, new
Point(8.0, 0.0));

[horizRuler.setMarkers (new NSArray (leftMarker, rightMarker, nil));
```

The images used for this example are 8 pixels square and lie just inside of their relevant positions. The figure below shows the left and right marker images, enlarged and with gray bounding boxes. Thus, the left marker's image must be placed with its lower left corner, or (0.0,0.0), at the marker location, while the lower right corner of the right marker, at (8.0,0.0), is used. The image origin is always expressed in the coordinate system of the image itself, just as an NSCursor's hot spot is.



A new NSRulerMarker allows the user to drag it around on its ruler but not to remove it. You can change these defaults by sending it `setMovable:` and `setRemovable:` messages. For example, you might make markers representing tabs in text removable to allow the user to edit the paragraph settings.

Markers bear one additional attribute, which allows you to distinguish among multiple markers, specifically markers that share the same image. This is the represented object, set with the NSRulerMarker method `setRepresentedObject:`. A represented object can simply be a string identifying a generic attribute, such as "Left Margin" or "Right Margin". It can also be an object stored in the client view or in the selection; for example, the text system records tab stops as NSTextTab objects, which include the tab location and its alignment. When the user manipulates a tab marker, the client can simply retrieve its represented object to get the tab being affected.

Letting Users Manipulate Markers

While a ruler's client view must perform the work of determining marker locations and placing them on the ruler, the ruler itself handles all the work of tracking user manipulations of the markers, sending messages to the client view that inform it of the changes before they begin, as they occur, and after they finish. The client view can use these messages to update its own state. The following sections describe the individual processes of moving, removing, and adding markers, along with a special method for handling mouse events in the ruler area.

Moving Markers

When the user presses the mouse button over a ruler marker, NSRulerView sends the marker a `trackMouse:adding:` message. If the marker isn't movable this method does nothing and immediately returns `NO`. If it is movable, then it sends the client a series of messages allowing it to determine how the user can move the marker around on the ruler.

The first of these messages is `rulerView:shouldMoveMarker`, which allows the client view to prevent an otherwise movable marker from being moved. Normally, whether a marker can be moved should be set on the marker itself, but there are situations, such as where items can be locked in place, where this is more properly tracked by the client view instead. If the client view returns `YES`, allowing the movement, then it receives a series of `rulerView:willMoveMarker:toLocation:` messages as the user drags the marker around. Each message identifies the marker being moved and its proposed new location in the client view's coordinate system. The client view can return an altered location to restrict the marker's movement, or update its display to reflect the new location. Finally, when the user releases the mouse button, the client receives `rulerView:didMoveMarker:`, on which it can update its state and clean up any information it may have used while tracking the marker's movements.

Removing Markers

Removal of markers is handled by a similar set of messages. However, these are always sent during a movement operation, as the user must first be dragging a marker within the ruler to be able to drag it off the ruler. If a marker isn't set to be removable, the user simply can't drag it off. If the marker is removable, then when the user drags the mouse far enough away from the ruler's baseline, it sends the client view a `rulerView:shouldRemoveMarker:` message, allowing the client to approve or veto the removal. No messages are necessary for new locations, of course, but if the user returns the marker to the ruler then it resumes sending `rulerView:willMoveMarker:toLocation:` messages as before. If the user releases the mouse with the marker dragged away from the ruler, the marker sends the client view a `rulerView:didRemoveMarker:` message, so the user can delete the item or attribute represented by the marker.

Adding Markers

When the user wants to add a marker, the addition must be initiated by the application, of course, since there is no marker yet for the ruler to track. The first step in adding a marker, then, is to create one, using the NSRulerMarker`initWithRulerView:markerLocation:image:imageOrigin:` method. Once the new marker is created, you instruct the ruler view to handle dragging it onto itself by sending it a `trackMarker:withMouseEvent:` message. One means of doing this is to use the mouse event from the client view method `rulerView:handleMouseDown:`, as described in "Handling Mouse Events in the Ruler Area" (page 24). Another is to create a custom view object—which typically resides in the ruler's accessory view—that displays prototype markers, and that handles a mouse-down event by creating a new marker for the ruler and invoking `trackMarker:withMouseEvent:` with the new marker and that mouse-down event.

Once you've initiated the addition process, things proceed in the same manner as for moving a marker. The ruler view sends the new marker a `trackMouse:adding:` message, with `YES` as the second argument to indicate that the marker isn't merely being moved. The marker being added then sends the client view a `rulerView:shouldAddMarker:` message, and if the client approves, then it repeatedly sends `rulerView:willAddMarker:atLocation:` messages as the user moves the marker around on the ruler. The user can drag the marker away to avoid adding it, or release the mouse button over the ruler, in which case the client receives a `rulerView:didAddMarker:` message.

As with moving a marker, you should consider enabling and disabling in a more immediate fashion than by the client view method if possible. If the user shouldn't be able to drag a marker from the accessory view, for example, the view containing the prototype marker should disable itself and indicate this in its appearance, rather than allowing the user to drag a marker out only to discover that the ruler won't accept it.

Handling Mouse Events in the Ruler Area

In addition to handling user manipulation of markers, a ruler informs its client view when the user presses the mouse button while the mouse is inside the ruler area (where hash marks are drawn) by sending it a `rulerView:handleMouseDown:` message. This information allows the client view to take some special action, such as adding a new marker to the ruler, as described in "Adding Markers" (page 24). This approach works well when it's quite clear what kind of marker will be created. The client view can also use this message as a cue to change its display in some way—for example, to add or remove a guideline that assists the user in laying out and aligning items in the view.

Updating the Ruler View

A single client view may contain many selectable items, such as graphic shapes or paragraphs of text with different ruler settings. When the selection changes, the client must reset the ruler view's markers based on the new selection. This kind of updating is fairly straightforward and can be performed as described in "Using a Ruler View's Client" (page 21) for situations where the client view itself changes.

Another kind of updating is needed when you want to support dynamic updating of ruler markers as the user manipulates the elements of the client view. For example, when the user moves a shape, you want the ruler markers to relocate when the user finishes moving it. Any method that changes relevant attributes of the selection should update the ruler markers, whether by replacing them as a set or by checking each one present and updating its location.

You can even put such updating code within a modal loop that handles dragging items around in the client view, so that the markers track the position of the selected item. This can be a fairly heavyweight operation to perform while also handling movement of the selected item, however. In support of a lighter weight means of showing this information, NSRulerView allows you to draw temporary ruler lines that can be drawn and erased very quickly. One method, `moveRulerlineFromLocation:toLocation:`, controls the drawing of ruler lines. It takes two locations expressed in the NSRulerView's coordinate system, erasing the ruler line at the old location and redrawing it at the new. To create a new ruler line, specify `-1.0` as the old location; to erase one completely, specify `-1.0` as the new location. Although you're responsible for keeping track of the locations to erase and redraw, this isn't as cumbersome or inefficient as sifting through or replacing the markers themselves.

Document Revision History

This table describes the changes to *Rulers and Paragraph Styles*.

Date	Notes
2007-09-04	Corrected method names in "Setting Up a Ruler View."
2004-04-16	Copyedited all articles. Added an article, "Breaking Lines by Truncation" (page 11), and related information in other articles.
2004-02-13	Rewrote introduction and added an index.
2002-11-12	Revision history added to existing topic.

Index

P

R

S

T

U