Control and Cell Programming Topics

Contents

Figures and Listings

Objective-CSwift

Introduction to Control and Cell Programming Topics for Cocoa

Controls and cells implement user-interface objects, like buttons, text fields, and sliders.

This topic contains these subtopics:

Button Programming Topics	A user interface object that sends an action message to a target when clicked.
Image View Programming Topics	A user interface object that displays a single image in a frame, and optionally allow a user to drag an image to it.
Slider Programming Topics	A user interface object that displays a range of values and has an indicator, or knob, which indicates the current setting.
Text Fields	A user interface object that displays text that the user can select or edit.
Box Programming Topics	A user interface object that can draw a border around itself and title itself.
Progress Indicator Programming Topics	A user interface object that shows that a lengthy task is under way.
Status Bar Programming Topics	A user interface object that displays a collection of items that provide interaction with or feedback to the user.
Browser Programming Topics	Provides a user interface for displaying and selecting items from a list of data or from hierarchically organized lists of data, such as directory paths.
Matrix Programming Guide	A user interface object used for creating groups of cells that work together in various ways.
Form Programming Topics	A group of related text fields.
Combo Box Programming Topics	A user interface object that gives the user two ways to enter a value: entering it directly in a text field, or choosing it from a pop-up list of pre-selected values.

Table View Programming Guide for Mac	A user interface object that displays data for a set of related records, with rows representing individual records and columns representing the attributes of those records.
Tab View Programming Topics	A user interface object providing a convenient way to provide information in multiple pages.
Outline View Programming Topics	A type of table which lets the user expand or collapse rows containing hierarchical data.
Text Views	Text views are the main user interface objects of the Cocoa text system.
Stepper Programming Topics	A user interface object consisting of two small arrows that can increment and decrement a value that appears beside it, such as a date or time.
Search Fields	A user interface object that provides a standard user interface for searching.
SegmentedControlProgramming Guide	A user interface object that has the appearance and behavior of a horizontal button divided into multiple segments.

Organization of This Document

Controls and cells implement user-interface objects, like buttons, text fields, and sliders. The control is responsible for

- Displaying itself

- Intercepting user events (such as a clicking a button or moving a slider)

- Sending actions to other objects, usually in response to a user event (such as changing a variable's value as a slider moves or performing a command when a button is pressed.)

A control usually delegates the first two responsibilities to cells. Splitting these responsibilities off makes it easier to create a control with many identical elements (like a spreadsheet table) or with a few different elements (like a pull-down list that lets you enter a string either in a text field or from a menu of pre-elected strings).

Here are the concepts:

- About Cells and Controls (page 9) gives basic information on what the NSCell and NSControl classes do.

- How Controls and Cells Interact (page 11) gives more information on how controls and cells interact and how they operate.

- Cell States (page 13) describes the three states a control can have: on, off, or mixed. Although used primarily by NSButton, states are defined in NSCell so future subclasses can use them.

- Represented Objects (page 12) explains how to associate a cell with the object it represents.

Here are the tasks:

- "Manipulating Cells and Controls (page 14)" discusses various tips and techniques for dealing with cells and controls.

- Changing the Cell for a Control (page 19) describes how to change the NSCell subclass that a control uses.

- Displaying Cell Values (page 18) describes how some cells format and display their values as strings.

- Validating Control Entries (page 17) describes how to validate the contents of some cells, especially cells in a matrix or text field.

- Using a Continuous Control (page 20) describes how to set up a control so it sends its action message repeatedly while being pressed.

- Subclassing NSCell (page 21) and Subclassing NSControl (page 22) describe how to create custom subclasses of NSCell and NSControl.

- Using the System Control Tint (page 25) describes how to use the system-wide control tint in your custom views and control cells.

About Cells and Controls

This topic gives basic information on NSCell and NSControl.

About NSCell

The NSCell class provides a mechanism for displaying text or images in an NSView without the overhead of a full NSView subclass. In particular, it provides much of the functionality of the NSText class by providing access to a shared NSText object used by all instances of NSCell in an application. NSCells are also extremely useful for placing text or images at various locations in a custom subclass of NSView.

NSCell is used heavily by most of the NSControl classes to implement their internal workings. For example, NSSlider uses an NSSliderCell, NSTextField uses an NSTextFieldCell, and NSBrowser uses an NSBrowserCell. Sending a message to the NSControl is often simpler than dealing directly with the corresponding NSCell. For instance, NSControls typically invoke `updateCell:` (causing the cell to be displayed) after changing a cell attribute; whereas if you directly call the corresponding method of the NSCell, the NSCell might not automatically display itself again.

Some subclasses of NSControl (notably NSMatrix) group NSCells in an arrangement where they act together in some cooperative manner. Thus, with an NSMatrix, you can implement a uniformly sized group of radio buttons without needing an NSView for each button (and without needing an NSText object as the field editor for the text on each button).

The NSCell class provides primitives for displaying text or an image, editing text, setting and getting object values, maintaining state, highlighting, and tracking the mouse. NSCell's method `trackMouse:inRect:ofView:untilMouseUp:` implements the mechanism that sends action messages to target objects. However, NSCell implements target/action features abstractly, deferring the details of implementation to NSActionCell and its subclasses.

About NSControl

NSControl is an abstract superclass that provides three fundamental features for implementing user-interface devices. First, as a subclass of NSView, NSControl draws, or coordinates the drawing of, the on-screen representation of the device. Second, it receives and responds to user-generated events within its bounds by overriding NSResponder's `mouseDown:` method and providing a position in the responder chain. Third, it

implements the `sendAction:to:` method to send an action message to the NSControl's target object. Subclasses of NSControl defined in the Application Kit are NSBrowser, NSButton (and its subclass NSPopUpButton), NSColorWell, NSImageView, NSMatrix (and its subclass NSForm), NSScroller, NSSlider, NSTableView, and NSTextField. Instances of concrete NSControl subclasses are often referred to as, simply, controls.

How Controls and Cells Interact

Controls are usually associated with one or more cells—instances of a subclass of the abstract class NSCell. A control's cell (or cells) usually fit just inside the bounds of the control. Cells are objects that can draw themselves and respond to events, but they can do so only indirectly, upon instruction from their control, which acts as a kind of coordinating backdrop.

Controls manage the behavior of their cells. By inheritance from NSView, controls derive the ability for responding to user actions and rendering their on-screen representation. When users click on a control, it responds in part by sending `trackMouse:inRect:ofView:untilMouseUp:` to the cell that was clicked. Upon receiving this message, the cell tracks the mouse and may have the control send the cell's action message to its target (either upon mouse-up or continuously, depending on the cell's attributes). When controls receive a display request, they, in turn, send their cell (or cells) a `drawWithFrame:inView:` message to have the cells draw themselves.

This relationship of control and cell makes two things possible: A control can manage cells of different types and with different targets and actions (see below), and a single control can manage multiple cells. Most Application Kit controls, like NSButtons and NSTextFields, manage only a single cell. But some controls, notably NSMatrix and NSForm, manage multiple cells (usually of the same size and attributes, and arranged in a regular pattern). Because cells are lighter-weight than controls, in terms of inherited data and behavior, it is more efficient to use a multi-cell control rather than multiple controls.

Many methods of NSControl—particularly methods that set or obtain values and attributes—have corresponding methods in NSCell. Sending a message to the control causes it to be forwarded to the control's cell or (if a multi-cell control) its selected cell. However, many NSControl methods are effective only in controls with single cells (these are noted in the method descriptions).

An NSControl subclass doesn't have to use an NSCell subclass to implement itself—NSScroller and NSColorWell are examples of NSControls that don't. However, such subclasses have to take care of details NSCell would otherwise handle. Specifically, they have to override methods designed to work with a cell. What's more, the lack of a cell means you can't make use of NSMatrix capability for managing multi-cell arrays such as radio buttons.

Represented Objects

Represented objects are objects an NSCell stands for. (They're not to be confused with an NSCell's object value, which is the value of the cell.) By setting a represented object for an NSCell (using `setRepresentedObject:`) you make an association between the NSCell and that object. For instance, you could have a pop-up list, each cell of which lists a color as its title; when the user selects a cell, the represented NSColor object is displayed in a color well. This feature is solely for the developer's convenience. The cell itself does not use the represented object, except to archive and restore it.

Cell States

For some subclasses of NSCell, such as an NSButtonCell, the object's value is its state. It can have either two states—`NSOnState` and `NSOffState`—or three states—`NSOnState`, `NSOffState`, and `NSMixedState`. A mixed state is useful for a checkbox or radio button that reflects the status of a feature that's true only for some items in your application or the current selection. For example, suppose a checkbox makes the selected text bold. If all the selected text is bold, it's on. If none of the selected text is bold, it's off. If the text has a combination of bold and plain text, it's mixed. Now suppose you click the checkbox. If you turn it on, all the text becomes bold. If you turn it off, all the text becomes plain. If you select the mixed state, the text remains as it is.

By default, an NSCell has two states. You can allow the third state with the method `setAllowsMixedState:`. To set the button's state directly, use `setState:`. To cycle through all available states, use `setNextState`.

Manipulating Cells and Controls

This article contains miscellaneous tips and examples for manipulating cells and controls.

Setting the Size of a Cell or Control

To set the size of a cell (and any enclosing single-cell control) to an optimum size conforming with the human interface guidelines, do the following:

1. If the cell contains text, set the font of the text to be consistent with one of the three standard sizes: regular, small, and mini. To do this use the NSFont class method `systemFontSizeForControlSize:`. The argument to this method is an `NSControlSize` constant declared by the NSControl class.

    ```
    float fontSize = [NSFont systemFontSizeForControlSize:NSMiniControlSize];

    NSCell *theCell = [theControl cell];

    NSFont *theFont = [NSFont fontWithName:[[theCell font] fontName]
    size:fontSize];

    [theCell setFont:theFont];
    ```

2. Set the control size to the same size as given for the font size, using the same constant. Use the NSControl `setControlSize:` method.

    ```
    [theCell setControlSize:NSMiniControlSize];
    ```

3. Finally, send `sizeToFit` to the control to trim the extra width.

    ```
    [theControl sizeToFit];
    ```

Drawing a Focus Ring Inside a Cell's Bounds

The `NSSetFocusRingStyle` sets the style that a focus ring will be drawn in the next time you fill a bezier path. It takes a constant of type of `NSFocusRingPlacement` to tell the Application Kit to draw the focus ring over an image, under text, or (when neither text or image is a consideration) around a shape. You can use this function with a constant of `NSFocusRingOnly` to draw a focus ring just inside a cell's bounds.

Listing 1 (page 15) shows how you draw such a focus ring. It requires you to override the NSCell `drawWithFrame:inView:`. In this method, if the cell is supposed to draw evidence of first-responder status, set the rectangle for the focus ring, call `NSSetFocusRingStyle` with an argument of `NSFocusRingOnly`, and then create and fill a bezier path defining that rectangle. Filling in this case simply draws the ring.

> **Note:** This technique requires you to subclass a cell class. See "Subclassing NSCell (page 21)" for information.

Listing 1   Drawing a focus ring just inside a cell's bounds

```
- (void)drawWithFrame:(NSRect)cellFrame inView:(NSView *)controlView {
    // other stuff might happen here
    if ([self showsFirstResponder]) {
        // showsFirstResponder is set for us by the NSControl that is drawing us.
        NSRect focusRingFrame = cellFrame;
        focusRingFrame.size.height -= 2.0;
        [NSGraphicsContext saveGraphicsState];
        NSSetFocusRingStyle(NSFocusRingOnly);
        [[NSBezierPath bezierPathWithRect: NSInsetRect(focusRingFrame,4,4)] fill];
        [NSGraphicsContext restoreGraphicsState];
    }
     // other stuff might happen here
}
```

Positioning an Image Within a Cell

If a cell is to display an image instead of (or in addition to) text, you can affect the placement of the image within the cell by overriding the `imageRectForBounds:` method, which is declared by both the NSCell and NSMenuItemCell classes. This method returns the rectangle the cell's image is drawn in, which is usually a rectangle slightly offset from the cell's bounds. Listing 2 (page 16) gives an example.

> **Note:** This technique requires you to subclass a cell class. See "Subclassing NSCell (page 21)" for information.

Listing 2 Centering an image in its cell

```
- (NSRect) imageRectForBounds:(NSRect)theBounds {

    NSRect r = theBounds;

    // get size. If no image, result of method returning NSSize is undefined so
assume NSZeroSize

    NSSize imageSize = [self image] != nil ? [[self image] size] : NSZeroSize;

    r.origin.x = floor((r.size.width/2)  - (imageSize.width/2)  + 0.5);

    r.origin.y = floor((r.size.height/2) - (imageSize.height/2) + 0.5);

    r.size     = imageSize;

    return r;

}
```

In this example, the cell centers the image in the cell. Note that it rounds the return values to the nearest pixel to avoid blurring that drawing with partial pixel offsets may cause. The code also sets the size field of the returned rectangle to the size of the image so that it is correctly drawn in the rectangle (assuming the NSImage object uses `drawInRect:fromRect:operation:fraction:` for drawing and not `compositeToPoint:operation:`).

Validating Control Entries

Objective-CSwift

NSControl provides the delegation method `control:isValidObject:` for validating the contents of cells embedded in controls (instances of NSTextField and NSMatrix in particular). In validating you check for values that are permissible as objects, but that are undesirable in a given context, such as a date field in which dates should never be in the future, or zip codes that are valid for a certain state.

The method `control:isValidObject:` is invoked when the insertion point leaves a cell (that is, the associated control relinquishes first-responder status) but before the string value of the cell's object is displayed. Return `YES` to allow display of the string and `NO` to reject display and return the cursor to the cell. The following Objective-C example evaluates an object (an NSDate) and rejects it if the date is in the future:

```
- (BOOL)control:(NSControl *)control isValidObject:(id)obj
{
    if (control == contactsForm) {
        if (![obj isKindOfClass:[NSDate class]]) return NO;
        if ([[obj laterDate:[NSDate date]] isEqual:obj]) {
            NSRunAlertPanel(@"Date not valid",
                @"Reason: date in future", NULL, NULL, NULL);
            return NO;
        }
    }
    return YES;
}
```

NSControl provides several delegate methods for its subclasses that allow text editing, such as NSTextField and NSMatrix. Some are invoked when formatters for a control's cells cannot format a string (`control:didFailToFormatString:errorDescription:`) or reject a partial string entry (`control:didFailToValidatePartialString:errorDescription:`). NSControl also provides `control:textView:doCommandBySelector:`, which allows delegates the opportunity to detect and respond to key bindings, such as `complete:` (name completion). Note that although NSControl defines delegate methods, it does not itself have a delegate. Any subclass that uses the delegate methods must contain a delegate and the methods to get and set it.

Displaying Cell Values

Every `NSCell` that displays text has a value associated with it. The `NSCell` object stores that value as an object of potentially any type, displays it as an `NSString` object, and returns it as a primary value or string object, according to what's requested (`intValue`, `floatValue`, `stringValue`, and so on). Formatters are objects associated with `NSCell` objects (through `setFormatter:`) that translate a cell's object value to its it textual representation and convert what users type into the underlying object. `NSCell` objects have built-in formatters to handle common string and numeric (`int`, `float`, `double`) translations. In addition, you can implement your own formatters to provide specialized object translation; see *Data Formatting Guide*.

The text that an `NSCell` object displays and stores can be an attributed string. Several methods help to set and get attributed-string values, including `setAttributedStringValue:` and `setImportsGraphics:`.

Changing the Cell for a Control

Because NSControl uses objects derived from the NSCell class to implement most of its functionality, you can usually implement a unique user interface device by creating a subclass of NSCell rather than NSControl. As an example, let's say you want all your application's NSSliders to have a type of cell other than the generic NSSliderCell. First, you create a subclass of NSCell, NSActionCell, or NSSliderCell. (Let's call it MyCellSubclass.) Then, you can simply invoke NSSlider's `setCellClass` class method:

```
[NSSlider setCellClass:[MyCellSubclass class]];
```

All NSSliders created thereafter will use MyCellSubclass, until you call `setCellClass:` again.

If you want to create generic NSSliders (ones that use NSSliderCell) in the same application as the customized NSSliders that use MyCellSubclass, there are two possible approaches. One is to invoke `setCellClass:` as above whenever you're about to create a custom NSSlider, resetting the cell class to NSSliderCell afterwards. The other approach is to create a custom subclass of NSSlider that automatically uses MyCellSubclass, as explained in the `NSControl` class reference.

Using a Continuous Control

A continuous control sends its action message at regular intervals as the user holds the mouse button down. For example, a continuous slider sends its action message repeatedly as the user moves the knob, and a continuous button sends its action message repeatedly as the user presses the button. If a control isn't continuous, it sends its action message only after the user releases the mouse.

To find out or change whether a control is continuous: send `isContinuous` or `setContinuous:` to its cell. By default, sliders are continuous and other controls are not.

To find out how often a continuous control sends its action message, use the method `getPeriodicDelay:interval:`, which returns the following:

- The periodic delay is the amount of time (in seconds) that a continuous control will pause before starting to periodically send action messages to the target object. It's taken from the user's defaults. If the user hasn't set it, it's 0.4 seconds.

- The interval is the amount of time (in seconds) between messages. By default, it's taken from the user's defaults. If the user hasn't set it, it's 0.075 seconds.

(In Java, use the two methods `periodicDelay` and `interval`.)

If you're using a button, use `setPeriodicDelay:interval:` to change these values programmatically. If you're using another type of control, you must subclass the control's cell class to change them programmatically.

Subclassing NSCell

The `initImageCell:` method is the designated initializer for NSCells that display images. The `initTextCell:` method is the designated initializer for NSCells that display text. Override one or both of these methods if you implement a subclass of NSCell that performs its own initialization. If you need to use target and action behavior, you may prefer to subclass NSActionCell or one of its subclasses, which provide the default implementation of this behavior.

If you want to implement your own mouse-tracking or mouse-up behavior, consider overriding `startTrackingAt:inView:`, `continueTracking:at:inView:`, and `stopTracking:at:inView:mouseIsUp:`. If you want to implement your own drawing, override `drawWithFrame:inView:` or `drawInteriorWithFrame:inView:`.

If the subclass contains instance variables that hold pointers to objects, consider overriding `copyWithZone:` to duplicate the objects. The default version copies only pointers to the objects.

Subclassing NSControl

If you are going to create a custom `NSControl` class that performs its own initialization, you should override the designated initializer (`initWithFrame:`). Be aware, however, that this method is not called when an instance of your subclass of an Application Kit control class is unarchived from a nib.

If you create a custom control subclass that is paired with a custom cell subclass—for example, a custom subclass of `NSSlider` and a custom subclass of `NSSliderCell`—you have two ways of associating an instance of that custom cell with an instance of the custom control. The first approach requires that you have version 3 of Interface Builder. In Interface Builder when you place a control on a window, the control and its cell are instantiated and, when you save the user interface, these objects (along with other placed objects) are encoded and serialized to a nib file. Interface Builder also helps you define custom subclasses, including subclasses of framework control classes such as `NSButton` and `NSSlider`. Interface Builder also allows you to change the class of a placed control object to a custom control class, but earlier versions of the application give you no way to do the same with custom cells associated with the control object.
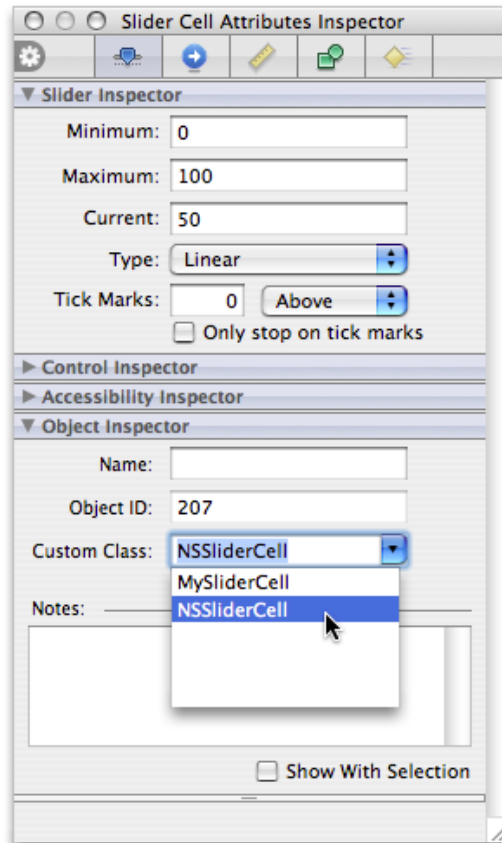
But version 3.0 of Interface Builder lets you set the class of a control's cell. To do this, click the control to select it and right-click the mouse (Control-click on single-button mice). Then click again in the upper-right corner of the pop-up list that appears. A submenu lists the objects under the mouse pointer—including the control's cell. In the case of an `NSSliderCell` object, as shown in Figure 1, the submenu includes a "Slider Cell" item. Select the cell item.

Figure 1   Selecting the cell of a control in Interface Builder
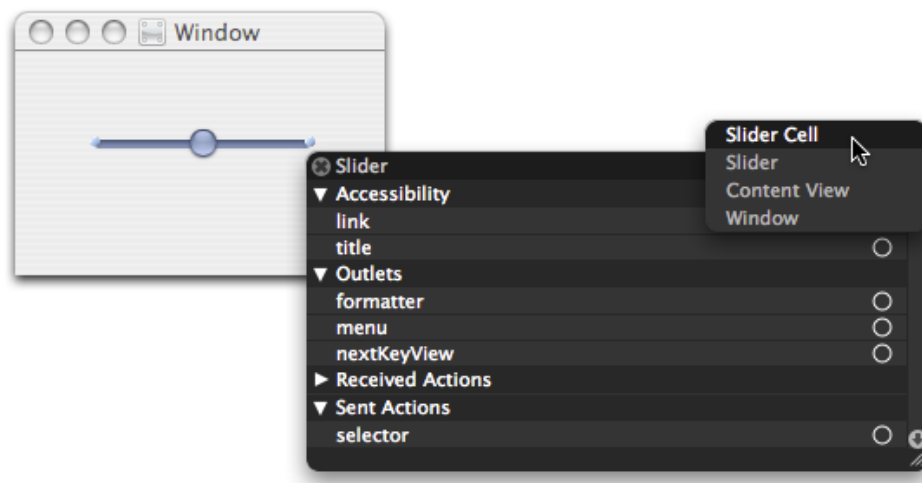
Next open the Inspector window for the selected cell object. Find the "Object inspector" section and, in the Custom Class combo box select (or enter) the name of your custom cell class.

Figure 2   Setting the custom cell class of a control

If version 3.0 of Interface Builder is not available, you still have a programmatic way to assign an instance of a custom cell to a custom control, which is illustrated in Listing 1. In the custom control subclass, when all objects are unarchived from the nib file, create an instance of the custom cell and assign it all pertinent attributes of the current cell. Then set the custom cell as the cell of the control using the `setCell:` method of `NSControl`.

Listing 1   Creating and setting a custom cell for a custom control

```
- (void)awakeFromNib {
    MySliderCell *newCell = [[MySliderCell alloc] init];
    id oldCell = [self cell];
    [newCell setImage:[oldCell image]];
    [newCell setMinValue:[oldCell minValue]];
    [newCell setMaxValue:[oldCell maxValue]];
```

```
    [newCell setSliderType:[oldCell sliderType]];

    // ....  set other slider cell attributes

    [self setCell:newCell];

    [newCell release];

}
```

You can override the `cellClass` method whenever a control needs to make a new cell for itself—for example if it is instantiated with `initWithFrame:`. The `initWithFrame:` method uses the return value of `cellClass` to allocate and initialize an `NSCell` object of the correct type.

Listing 2   Overriding `cellClass`

```
+ (Class) cellClass
{
    return [MySliderCell class];
}
```

Note that overriding the `cellClass` class method of `NSControl` does not change the class of a cell object unarchived from a nib file.

Using the System Control Tint

OS X allows a user to set the color used in the display of windows, menus and controls using the Appearance pane in System Preferences. This color is referred to as the control tint. User interface elements provided by the Application Kit automatically modify their appearance based on the current control tint.

Custom controls and other aspects of your application's user interface can also use the control tint as shown in Figure 1.

Figure 1   Examples of control tint aware custom control cells

Custom Control using Blue Tint   Custom Control using Graphite Tint
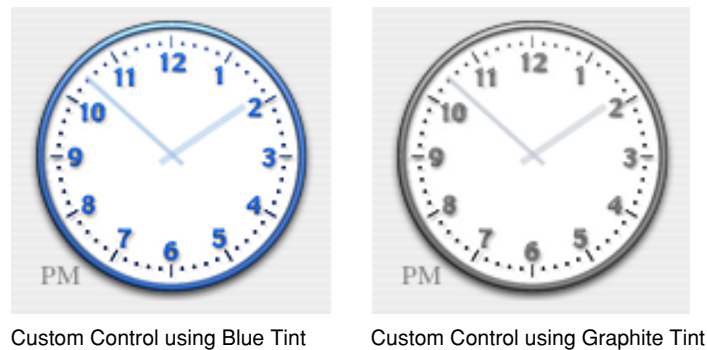
Getting the System Control Tint and Color

You can get the current system-wide control tint using the NSColor class method `currentControlTint`. This method returns an `NSControlTint` that represents the currently selected appearance color. Currently `NSBlueControlTint` or `NSGraphiteControlTint` are the possible return values representing the Aqua and Graphite appearances respectively.

Having determined the current `NSControlTint` value, you can now get the corresponding color using the `NSColor` class method `colorForControlTint:`.

An application can detect when the user changes the system-wide appearance setting, by registering as an observer of the `NSControlTintDidChangeNotification` notification.

Interaction with Views and Images

Your application may want to modify the appearance of the contents of a NSView subclass, or change the type of image used by an NSControl subclass in response to the system-wide appearance being changed. To do this your application must register for the NSControlTintDidChangeNotification, update the appropriate elements, and redraw the view.

The example in Listing 1 demonstrates how to determine the current control tint and change the image displayed in an NSButton and an NSImageView. The object that implements this method has previously been registered as an observer of the NSControlTintDidChangeNotification notification, with this method as the selector.

Listing 1   Example that sets the contents of an NSButton and NSImageView using the currentControlTint

```
- (void)systemTintChangedNotification:(NSNotification *)notification;
{

    NSString *tintImageName;


    // compare the result of [NSColor currentControlTint]
    // with the supported tint values, defaulting to Aqua
    if ([NSColor currentControlTint] == NSGraphiteControlTint)
        tintImageName=@"GraphiteImage";
    else
        tintImageName=@"AquaImage";


    [exampleButton setImage:[NSImage imageNamed:tintImageName]];

    [exampleImageView setImage:[NSImage imageNamed:tintImageName]];


}
```

Interaction with NSCell Subclasses

When the system-wide appearance is changed, any NSCell objects are automatically redrawn. It is the responsibility of an NSCell subclass to determine the current control tint as part of its implementation of drawWithFrame:inView:.

The example in Listing 2 demonstrates how to implement a tint-aware `drawWithFrame:inView:` method. It determines if the cell is using the system-wide appearance tint, or if its tint has been set explicitly and the appropriate image is then selected for display. The appropriate control tint color is also determined for drawing the clock's hands.

Listing 2   Example of a tint-aware NSCell drawWithFrame:inView: implementation

```
- (void)drawWithFrame:(NSRect)cellFrame
            inView:(NSView *)controlView
{
    // if we're not the front window, we'll resort to using the
    // special NSClearControlTint value
    NSControlTint currentTint;
    if ([[controlView window] isKeyWindow])
        currentTint = [self controlTint];
    else
        currentTint= NSClearControlTint;


    // If the NSCell's control tint has been overridden
    // using the setControlTint: method we should use
    // the value returned by [self controlTint] as this
    // controls authoritative tint. If the tint is
    // NSDefaultControlTint then this cell should use the
    // system-wise appearance value, and we use the value
    // returned by the NSColor +currentControlTint method.
    if ([self controlTint] == NSDefaultControlTint)
        currentTint=[NSColor currentControlTint];
    else
        currentTint=[self controlTint];




    // and change the image used in drawing according to
    // the currentTint
    // Use the Aqua image as the default image
    NSImage *clockFaceImage;
```

```
    switch (currentTint) {

        case NSGraphiteControlTint:

            clockFaceImage = [NSImage imageNamed: @"ClockFace-Graphite"];

            break;

        case NSClearControlTint:

            clockFaceImage = [NSImage imageNamed: @"ClockFace-Clear"];

            break;

        case NSBlueControlTint:

        default:

            clockFaceImage = [NSImage imageNamed: @"ClockFace-Aqua"];

            break;

    }


    float clockRadius = MIN(NSHeight(cellFrame), NSWidth(cellFrame));


    // Draw the clock face (draw it flipped

    // if we are in a flipped view, like NSMatrix).

    [clockFaceImage setFlipped:[controlView isFlipped]];

    [clockFaceImage drawInRect:NSMakeRect(NSMinX(cellFrame),

                                        NSMinY(cellFrame),

                                        clockRadius,clockRadius)

                fromRect:NSMakeRect(0,0,

                                        [clockFaceImage size].width,

                                        [clockFaceImage size].height)

                operation:NSCompositeSourceOver

                fraction:1.0];


    // get the color for the currentTint and use it for

    // drawing the hands on the clock face

    NSColor *tintColor=[NSColor colorForControlTint:currentTint];


    // Draw the clock hour and minute hands.

    [self drawClockHandsForTime:time

                    withFrame:cellFrame
```

```
                    inView:controlView
            usingColor:tintColor];
}
```

Document Revision History

This table describes the changes to *Control and Cell Programming Topics* .

Date	Notes
2008-10-15	Made minor corrections.
2006-10-16	Clarified how to set a custom cell of a custom control defined in Interface Builder; also removed reference to deprecated method.
2006-03-08	Added a link to NSControl reference.
2005-08-11	Added "Manipulating Cells and Controls" and fixed typos. Changed title from "Controls and Cells."
2003-10-13	Updated the list of topics in the "Child Topics" section of the Introduction.
2003-10-11	Added the Using the System Control Tint (page 25)article.
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.