To Do Tutorial - Extended Chapter 5

5

What You'll Learn

Creating and managing an inspector

Responding to user actions

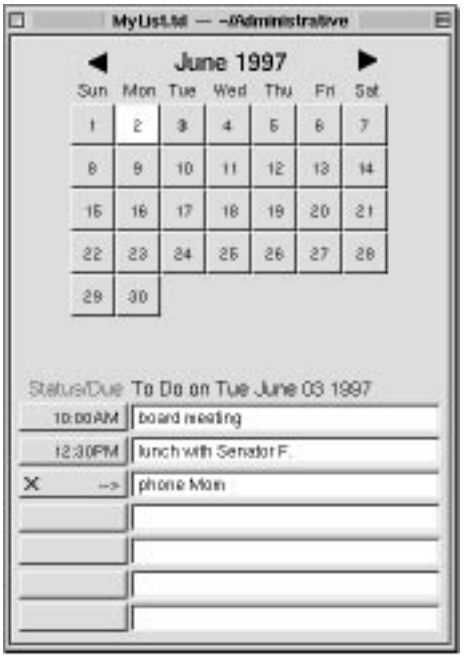
Coordinating events within an application

Overriding behavior of an Application Kit class

Creating a custom NSView subclass

Using timers

Drawing and compositing essentials



*You can find the ToDo project in the **AppKit** subdirectory of **/System/Developer/Examples.***

Chapter 5
Extending the To Do Application

In this tutorial you will add features and functionality to the To Do application you created in the previous tutorial. The finished application will allow users to do much more than entering to-do items into a daily list. In an inspector they will be able to:

- *Specify the times those items are due.*
- *Request that they be notified at a specified interval before the due time.*
- *Associate notes with items.*
- *Mark items as complete or deferred.*
- *Reschedule uncompleted items.*

Moreover, the document interface will have a custom button for each item. The button will display the item's due time. Users can also click the button to change an item's status. Changes users make in the document will be immediately reflected in the inspector, and vice versa.

Events and the Event Cycle

Conceptually, this chapter focuses primarily on *events*—especially events originating from user actions—and how, as a programmer, you intercept, handle, and coordinate them with Rhapsody APIs. Therefore, it's best to begin with a short overview of this topic.

You can depict the interaction between a user and an Rhapsody application as a cyclical process, with the Window Server playing an intermediary role (see illustration below). This cycle—the *event cycle*—usually starts at launch time when the application (which includes all the frameworks it's linked to) sends a stream of PostScript code to the Window Server to have it draw the application interface.

Then the application begins its main event loop and begins accepting input from the user (see next page). When users click or drag the mouse or type on the keyboard, the Window Server detects these actions and processes them, passing them to the application as events. Often the application, in response to these events, returns another stream of PostScript code to the Window Server to have it redraw the interface.

In addition to events, applications can respond to other kinds of input, particularly timers, data received at a port, and data waiting at a file descriptor. But events are the most important kind of input.

Events

The Window Server treats each user action as an event. It associates the event with a window and reported to the application that created the window. Events are objects: instances of NSEvent composed from information derived from the user action.

All event methods defined in NSResponder (such as **mouseDown:** and **keyDown:**) take an NSEvent as their argument. You can query an NSEvent to discover its window, the location of the event within the window, and the time the event occurred (relative to system start-up). You can also find out which (if any) modifier keys were pressed, such as Command, Option (Alternate), and Control), the codes that identify characters and keys, and various other kinds of information.

An NSEvent also divulges the type of event it represents. There are many event types (NSEventType); they fall into four categories:

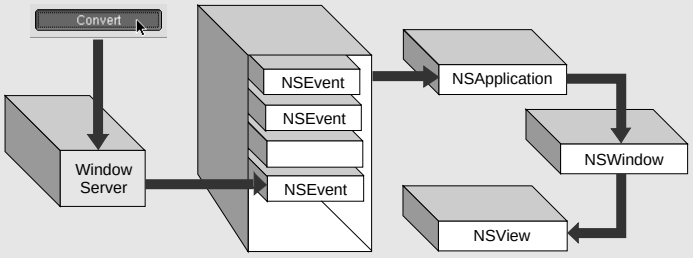
- **Keyboard events**. Generated when a key is pressed down, a pressed key is released, or a modifier key changes. Of these, key-down events are the most useful. When you handle a key-down event, you often determine the character or characters associated with the event by sending the NSEvent a **characters** message.
- **Mouse event**. Mouse events are generated by changes in the state of the mouse buttons (that is, down and up) for both left and right mouse buttons and during mouse dragging. Events are also generated when the mouse simply moves, without any button pressed.
- **Tracking-rectangle events**. If the application has asked the window system to set a tracking rectangle in a window, the window system creates mouse-entered and mouse-exit events when the cursor enters the rectangle or leaves it.
- **Periodic events**. A periodic event notifies an application that a certain time interval has elapsed. An application can request that periodic events be placed in its event queue at a certain frequency. They are usually used during a tracking loop. (These events aren't passed to an NSWindow.)

The Event Queue and Event Dispatching

When an application starts up, the NSApplication object (NSApp) starts the main event loop and begins receiving events from the Window Server. As NSEvents arrive, they're put in the *event queue* in the order they're received. On each cycle of the loop, NSApp gets the topmost event, analyzes it, and sends an *event message* to the appropriate object. (Event messages are defined by NSResponder and correspond to particular events.) When NSApp finishes processing the event, it gets the next event and repeats the process again and again until the application terminates.

The object that is "appropriate" for an event depends on the type of event. NSApp sends most event messages to the NSWindow in which the user action occurred. If the event is a keyboard or mouse event, the NSWindow forwards the message to one of the objects in its view hierarchy: the NSView within which the mouse was clicked or the key was pressed. If the NSView can respond to the event—that is, it accepts first responder status and defines an NSResponder method corresponding to the event message—it handles the event.

If the NSView cannot handle an event, it forwards the message to the next responder in the responder chain (see next section for details). It travels up the responder chain until an object handles it.



NSWindow handles some events itself, and doesn't forward them to an NSView, such as window-moved, window-resized, and window-exposed events. (Since these are handled by NSWindow itself, they are not defined in NSResponder.) NSApp also processes a few kinds of events itself, such as application-activate and application-deactivate events.

First Responder and the Responder Chain

Each NSWindow in an application keeps track of the object in its view hierarchy that has *first responder* status. This is the NSView that currently receives keyboard events for the window. By default, an NSWindow is its own first responder, but any NSView within the window can become first responder when the user clicks it with the mouse.

You can also set the first responder programmatically with the NSWindow's **makeFirstResponder:** method. Moreover, the first-responder object can be a target of an action message sent by an NSControl, such as a button or a matrix. Programmatically, you do this by sending **setTarget:** to the NSControl (or its cell) with an argument of **nil**. You can do the same thing in Interface Builder by making a target/action connection between the NSControl and the First Responder icon in the Instances display of the nib file window.

Recall that all NSViews of the application, as well as all NSWindows and the application object itself, inherit from NSResponder, which defines the default message-handling behavior: events are passed up the responder chain. Many Application Kit objects, of course, override this behavior, so events are passed up the chain until they reach an object that does respond.

The series of next responders in the responder chain is determined by the interrelationships between the application's NSView, NSWindow, and NSApplication objects (see page 152). For an NSView, the next responder is usually its superview; the content view's next responder is the NSWindow. From there, the event is passed to the NSApplication object.

For action messages sent to the first responder, the trail back through possible respondents is even more detailed. The messages are first passed up the responder chain to the NSWindow and then to the NSWindow's delegate. Then, if the previous sequence occurred in the key window, the same path is followed for the main window. Then the NSApplication object tries to respond, and failing that, it goes to NSApp's delegate.

Overriding Behavior of an Application Kit Class: An Example

You can often achieve significant gains in object behavior by making a subclass that adds only a small amount of code to its superclass. Such is the case with the subclass you'll create in this section: SelectionNotifMatrix.

The need for this class is this: An instance of NSMatrix is a control and thus can send action messages to its cell's targets; but when it contains NSTextFieldCells, action messages are sent only when users press the Return key in a cell. You want the inspector (which you'll create in the next section) to synchronize its displays when the user selects a new item by clicking a text field. To do this, you will *override* the method in NSMatrix that is invoked when users click the matrix; in your implementation, you'll invoke the superclass method, detect the selected row, and then post a notification to interested observers.

1 Create template source-code files and add to the project.

Choose File ► New In Project.

In the New File In ToDo panel, select the Class suitcase, turn on the Create header switch, and type "SelectionNotifMatrix" after Name.

1 Add declarations to the header file.

```
#import <AppKit/AppKit.h>

extern NSString *SelectionInMatrixNotification =                    A
        @"SelectionInMatrixNotification";

@interface SelectionNotifMatrix : NSMatrix
{
}

- (void)mouseDown:(NSEvent *)theEvent;                              B

@end
```

A Declares a string constant identifying the notification that will be posted.

B Declares **mouseDown:**, the method implemented by the superclass but overridden by SelectionNotifMatrix.

Before You Go On

Remember, build the project frequently to catch any errors quickly, to get a sense of how the application is developing, and (just as important) to give yourself a break from coding.

1 Override mouseDown:

In SelectionNotifMatrix.m, implement **mouseDown:** as shown here.

```
- (void)mouseDown:(NSEvent *)theEvent
{
    int row;
    [super mouseDown:theEvent];                                    A

    row = [self selectedRow];                                      B
    if (row != -1) {
        [[NSNotificationCenter defaultCenter]
           postNotificationName:@"SelectionInMatrixNotification"
           object:self userInfo:[NSDictionary
           dictionaryWithObjectsAndKeys:
           [NSNumber numberWithInt:row], @"ItemIndex", nil]];
    }
}
```

This override of **mouseDown:** does the following:

A Invokes NSMatrix's implementation of **mouseDown:** to allow the normal processing of this event.

B Gets the row of the cell clicked and, if it's a valid row, creates a **userInfo** dictionary containing the index of the clicked row, and posts the SelectionInMatrixNotification.

Now that you've created the SelectionNotifMatrix class, you must re-assign the class membership of the object in the interface. You can do this easily in Interface Builder.

1 Assign the new class to the matrix of text fields.

In Interface Builder:

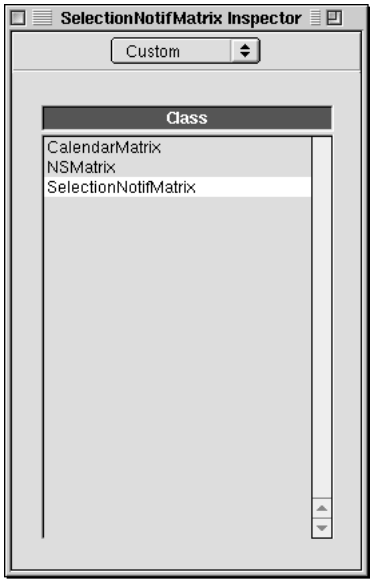
In the Classes display of **ToDoDoc.nib**, select NSMatrix as the superclass.

Choose Read File from the Classes menu.

In the Read File browser, select SelectionNotifMatrix and click OK.

Select the matrix of text cells.

Choose SelectionNotifMatrix in the Custom display of the inspector.



The Custom Classes browser lists the original class of the selected object and all compatible custom subclasses.

Creating and Managing an Inspector (ToDoInspector)

An inspector is a panel of fields and controls that enable users to examine and set an object's attributes. Because objects often have many attributes and because you want to make it easy for users to set those attributes, inspectors usually have more than one display; users typically access these multiple displays using a pop-up list.

The ToDo application has an inspector panel that allows users to inspect and set the attributes of the currently selected ToDoItem. The inspector panel has its own controller: ToDoInspector. While showing you how to create the inspector panel and ToDoInspector, this section focuses on four things:

- Managing displays according to user selections
- Getting the current ToDoItem
- Updating the currently selected display
- Updating the current ToDoItem as users make changes to it

In Interface Builder

1 Create a new nib file named ToDoInspector.nib and add it to the ToDo project.

1 Create the inspector panel.

Drag a panel object from the Windows palette.

Make the title of the panel "Inspector."

Turn on the panel's sizing border and resize it, using the example as a guide.

Turn off the panel's sizing border.

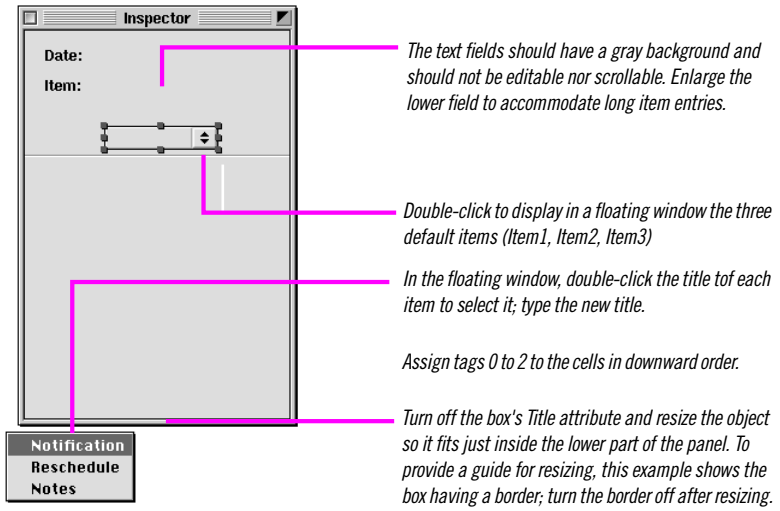
Put labels and fields on the panel and set their attributes (as shown).

Put a pop-up button on the panel and set cell titles (as shown).

Assign tags to the pop-up button cells.

Create a separator line just below the pop-up button.

Put an empty box object in the lower part of the panel.



Before You Go On

You might be wondering about the empty box object in the lower part of the panel. This box by itself may not seem a promising thing for displaying object attributes, but it is critical to the workings of the inspector panel. A box that you drag from the Views palette contains one subview, called the *content view*. NSBox's content view fits entirely within the bounds of the box. NSBox provides methods for obtaining and changing the content view of boxes. You'll use these methods to change what the inspector panel displays.

1 Create an off-screen panel holding the inspector's displays.

Drag a panel object from the Windows palette.

Resize the panel, using the example at right as a guide.

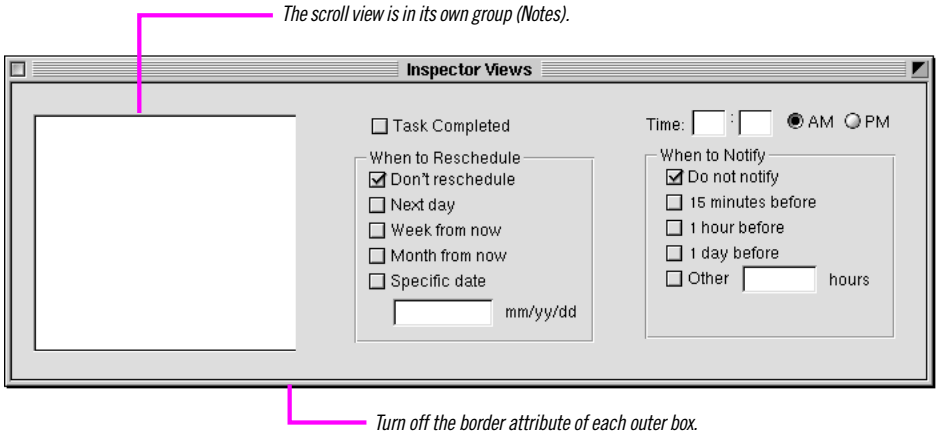
Put the labels, text fields, scroll view, and switch and radio-button matrices on the panel shown in the example at right.

Set the mode attributes of the switch matrices to Radio.

Make the "When to reschedule" and "When to notify" groupings (boxes).

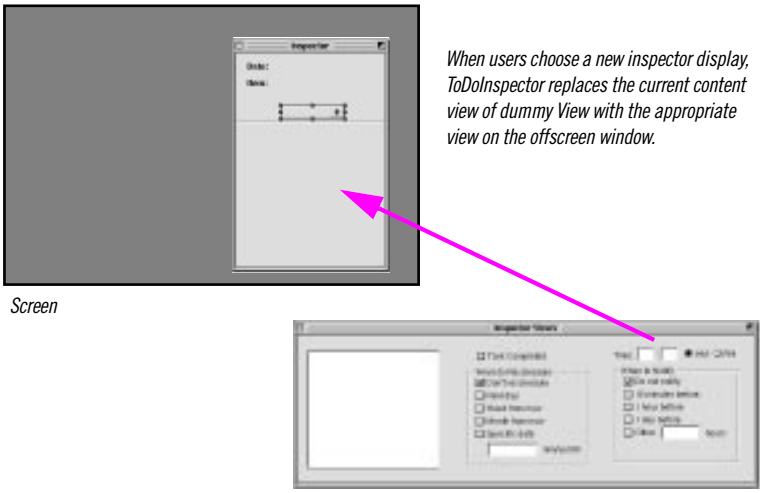
Make three other groupings for the three displays: Notes, Reschedule, and Notification.

Make the resulting outer boxes the same size as the "dummy" view in the inspector panel.



Before You Go On

You probably now see where the inspector panel gets its displays and how it puts them in place. When the inspector panel is first opened (and **ToDoInspector.nib** is loaded) the inspector controller, ToDoInspector, replaces the content view of the inspector's empty box (**dummyView**) with the content view of the Notification box in the off-screen panel. Thereafter, every time the user chooses a new pop-up button in the inspector panel, ToDoInspector replaces the currently displayed content view with the content view of the associated off-screen box.



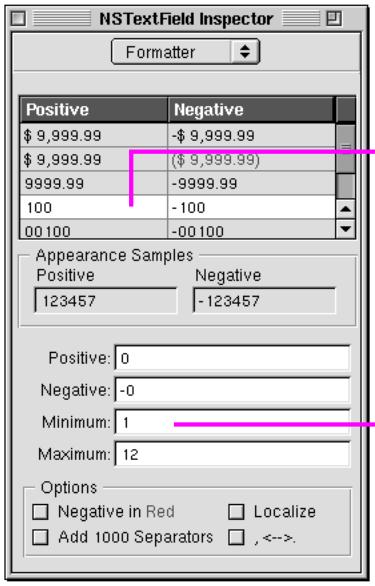
When users choose a new inspector display, ToDoInspector replaces the current content view of dummy View with the appropriate view on the offscreen window.

1 Apply formatters to fields of the inspector.

Drag a number-formatter object from the DataViews palette and drop it on the hours field of the Notification display (the first field after "Time:").

In the inspector's Formatter display, set the field to have a minimum value of 1 and a maximum value of 12 (see example).

Apply a number formatter to the minutes field (the second field after "Time:") and set it to have a minimum value of 0 and a maximum value of 59.



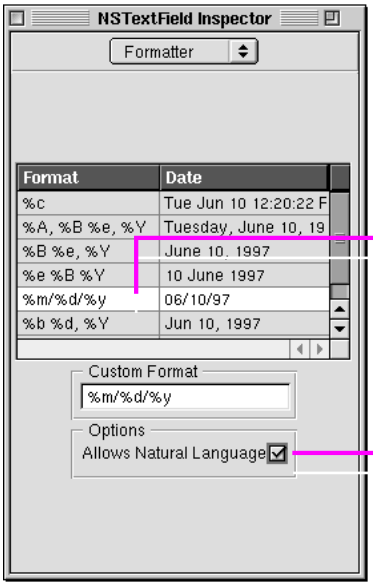
Select a simple integer format for the hour and minute "Time" fields.

Users cannot enter values that are less than this into the field; the cursor will not leave the field until they enter an appropriate value.

Interface Builder provides a palette object that formats dates in addition to the one that formats numbers. You can identify this object on the DataViews palette through its calendar icon.

Drag a date-formatter object from the DataViews palette onto the date field in the Rescheduling display (the "mm/dd/yy" field).

In the inspector's Formatter display, select the "%m/%d/%y" format from the table.



Select this format for the field.

The formatter rejects dates entered in any other format. It also verifies that the individual fields contain proper values (for instance, "13" is disallowed as a month).

Check if you want the formatter to interpret common temporal expressions such as "tomorrow" or "next month."

1 Define the ToDoInspector class.

Create a subclass of NSObject and name it “ToDoInspector.”

Add the outlets and actions in the tables at right to the new class.

Instantiate ToDoInspector.

Connect the ToDoInspector object to its outlets and as the target of action messages (see tables at right).

Connect ToDoInspector and the inspector panel via the panel’s **delegate** outlet.

Close both panels.

Save **ToDoInspector.nib**.

Create source-code files for ToDoInspector and add them to the project.

Outlet	Connection From ToDoInspector To...
dummyView	The empty box object in the inspector panel
inspectorViews	The title bar of the off-screen panel
notesView	The box in the off-screen panel containing the scroll view
notifView	The box in the off-screen panel containing the fields and controls related to notification of impending items
reschedView	The box in the off-screen panel containing the fields and controls related to rescheduling items
inspPopUp	The pop-up button on the inspector panel
inspDate	The uneditable text field next to the “Date” label
inspItem	The uneditable text field next to the “Item” label
inspNotifHour	The first field after the “Time” label
inspNotifMinute	The second field after the “Time” label
inspNotifAMPM	The matrix holding the “AM” and “PM” radio buttons
inspNotifOtherHours	The text field in the “When to Notify” box
inspNotifSwitchMatrix	The matrix of switches in the “When to Notify” box
inspSchedComplete	The “Task Completed” switch
inspSchedDate	The text field in the “When to Reschedule” box
inspSchedMatrix	The matrix of switches in the “When to Reschedule” box
inspNotes	The text object inside the scroll view

Action	Connection To ToDoInspector From...
newInspectorView:	The pop-up button on the inspector panel
switchChecked:	The matrix of switches in the “When to Notify” box, the AM-PM matrix, the “Task Completed” switch, and the matrix of switches in the “When to Reschedule” switches

In Project Builder

1 Add declarations to ToDoInspector.h.

Open **ToDoInspector.h**.

Type the declarations shown at right (ellipses indicate existing declarations).

Import **ToDoItem.h** and **ToDoDoc.h**.

```
@interface ToDoInspector : NSObject
{
    ToDoItem *currentItem;
    /* ... */
}
/* ... */
- (void)setCurrentItem:(ToDoItem *)newItem;
- (ToDoItem *)currentItem;
- (void)updateInspector:(ToDoItem *)item;
@end
```

The ToDoInspector class has a utility function for clearing switches set in a matrix and defines constants for the tags assigned to the pop-up buttons.

Open **ToDoInspector.m**.

Forward-declare **clearButtonMatrix()** at the beginning of the file.

Define **enum** constants for the pop-up button tags.

```
static void clearButtonMatrix(id matrix);
enum { notifTag = 0, reschedTag, notesTag };
```

Using tags to identify cells rather than cell titles is a better localization strategy.

ToDoInspector has two accessor methods, one that gives out the current item and one that sets the current item.

1 Implement the accessor methods for the class.

Implement **currentItem** to return the instance variables it names.

Implement **setCurrentItem:** as shown at right.

```
- (void)setCurrentItem:(ToDoItem *)newItem
{
    if (currentItem) [currentItem autorelease];
    if (newItem)
        currentItem = [newItem retain];                    A
    else
        currentItem = nil;
    [self updateInspector:currentItem];                    B
}
```

The implementation of **setCurrentItem:**'s "set" accessor method probably seems familiar to you—except for a couple of things:

A Instead of copying the new value, this implementation retains it. By retaining, it *shares* the current ToDoItem with the document controller (ToDoDoc) that has sent the **setCurrentItem:** message, enabling both objects to update the same ToDoItem simultaneously.

Later in this section, you'll invoke ToDoInspector's **setCurrentItem:** method in various places in **ToDoDoc.m**.

B Updates the current display of the inspector with the appropriate values of the new ToDoItem.

1 Switch inspector displays based on user selections.

Implement **newInspectorView:**.

```
- (void)newInspectorView:(id)sender
{
    NSBox *newView=nil;
    NSView *cView = [[inspPopUp window] contentView];          A
    int selected = [[inspPopUp selectedItem] tag];             B
    switch(selected){
      case notifTag:
        newView = notifView;
        break;
      case reschedTag:
        newView = reschedView;
        break;
      case notesTag:
        newView = notesView;
        break;
    }
    if ([[cView subviews] containsObject:newView]) return;     C
    [dummyView setContentView:newView];                        D
    if (newView == notifView) [inspNotifHour selectText:self];
    if (newView == notesView) [inspNotes
        setSelectedRange:NSMakeRange(0,0)];
    [self updateInspector:currentItem];                        E
    [cView setNeedsDisplay:YES];
}
```

This method switches the current inspector display according to the pop-up button users select; it does this switching by replacing the **dummyView**'s content view. Toward this end, the method:

A Gets the panel's content view and the tag of the selected pop-up button.

B Assigns to the **newView** local variable the off-screen box object corresponding to the tag of the selected pop-up button.

C Returns if the selected display is already on the inspector panel. The **subviews** message returns an array of all subviews of the inspector panel's control view, and the **containsObject:** message determines if the chosen display is among these subviews.

D Replaces the content view of the inspector panel's **dummyView**. In **awakeFromNib** (which you'll soon implement) you'll retain each original content view. The **setContentView:** method replaces the new view and releases the old one; because it's retained, the replaced view remains visible.

E Updates the inspector with the current item; this item hasn't changed, but the display is new and so the set of instance variables to be displayed is different. The **setNeedsDisplay:** message forces a re-draw of the inspector panel's views.

1 Update the current inspector display with the new ToDoItem.

Write the first part of the **updateInspector:** method shown at right.

```
- (void)updateInspector:(ToDoItem *)newItem
{
    int minute=0, hour=0, selected=0;
    selected = [[inspPopUp selectedItem] tag];                          A
    [[inspPopUp window] orderFront:self];
    if (newItem && [newItem isKindOfClass:[ToDoItem class]]) {          B
        [inspItem setStringValue:[newItem itemName]];
        [inspDate setStringValue:[[newItem day]
            descriptionWithCalendarFormat:@"%a, %b %d %Y"
            timeZone:[NSTimeZone localTimeZone] locale:nil]];
        switch(selected) {
            case notifTag: {                                            C
                long notifSecs, dueSecs = [newItem secsUntilDue];
                BOOL ampm = ConvertSecondsToTime(dueSecs, &hour, &minute);
                [[inspNotifAMPM cellAtRow:0 column:0] setState:!ampm];
                [[inspNotifAMPM cellAtRow:0 column:1] setState:ampm];
                [inspNotifHour setIntValue:hour];
                [inspNotifMinute setIntValue:minute];
                notifSecs = dueSecs - [newItem secsUntilNotif];
                if (notifSecs == dueSecs) notifSecs = 0;
                clearButtonMatrix(inspNotifSwitchMatrix);
                switch(notifSecs) {                                     D
                  case 0:
                    [[inspNotifSwitchMatrix cellAtRow:0 column:0]
                        setState:YES];
                    break;
                  case (hrInSecs/4):
                    [[inspNotifSwitchMatrix cellAtRow:1 column:0]
                        setState:YES];
                    break;
                  case (hrInSecs):
                    [[inspNotifSwitchMatrix cellAtRow:2 column:0]
                        setState:YES];
                    break;
                  case (dayInSecs):
                    [[inspNotifSwitchMatrix cellAtRow:3 column:0]
                        setState:YES];
                    break;
                  default:  /* Other */
                    [[inspNotifSwitchMatrix cellAtRow:4 column:0]
                        setState:YES];
                    [inspNotifOtherHours setIntValue:
                        ((dueSecs-notifSecs)/hrInSecs)];
                    break;
                }
                break;
            }
          case reschedTag:
            break;
```

The **updateInspector:** method is a long one, so we'll approach it in stages. This first part updates the common data elements (item name and date) and, if the selected display is for notifications, updates that display.

Ⓐ Gets the tag assigned to the selected pop-up button.

Ⓑ Tests the argument **newItem** to see if it is a ToDoItem. This test is important because if the argument is **nil**, the method clears the display of existing data (next example).

If **newItem** is a ToDoItem, **updateInspector:** first updates the Item and Date fields.

Ⓒ If the tag of the selected pop-up button is **notifTag**, updates the associated inspector display. This task starts by converting the due time from seconds to hour, minute, and PM boolean values and then setting the appropriate fields and button matrix with these values.

Ⓓ Sets the appropriate switch in the "When to Notify" matrix. It starts with the difference (in seconds) between the time the item is due and the time the item notification is sent. It calls **clearButtonMatrix()** to turn all switches off and then, in a switch statement, sets the switch corresponding to the difference in value between seconds from midnight before due and before notification.

Before You Go On

Update the Notes display. Add code to update the inspector's Notes display from the information in the ToDoItem passed into **updateInspector:**. (Check the documentation on NSText to see what method is suitable for this.) The selected pop-up button must have **notesTag** assigned to it. Also put the cursor at the start of the text object by selecting a "null" range.

Note that this tutorial omits the rescheduling logic of the ToDo application, including the code in this method that would update the "Reschedule" display. Rescheduling of ToDoItems is reserved as an optional exercise for you at the end of this tutorial.

Finish the implementation of **updateInspector:** by resetting all displays if the argument is **nil**.

```
    }
    else if (!newItem) { /* newItem is nil */
        [inspItem setStringValue:@""];
        [inspDate setStringValue:@""];
        [inspNotifHour setStringValue:@""];
        [inspNotifMinute setStringValue:@""];
        [[inspNotifAMPM cellAtRow:0 column:0] setState:YES];
        [[inspNotifAMPM cellAtRow:0 column:1] setState:NO];
        clearButtonMatrix(inspNotifSwitchMatrix);
        [[inspNotifSwitchMatrix cellAtRow:0 column:0]
            setState:YES];
        [inspNotifOtherHours setStringValue:@""];
        [inspNotes setString:@""];
     }
}
```

As you've most likely noticed, the **updateInspector:** method calls the function **clearButtonMatrix()**, which resets the states of all button cells in a switch matrix to NO.

Implement the **clearButtonMatrix()** utility function.

```
void clearButtonMatrix(id matrix)
{
    int i, rows, cols;
    [matrix getNumberOfRows:&rows columns:&cols];
    for(i=0; i<rows; i++)
        [[matrix cellAtRow:i column:0] setState:NO];
}
```

The **getNumberOfRows:columns:** message returns, by indirection in the first argument, the number of cells in **itemMatrix**.

Making a Custom View

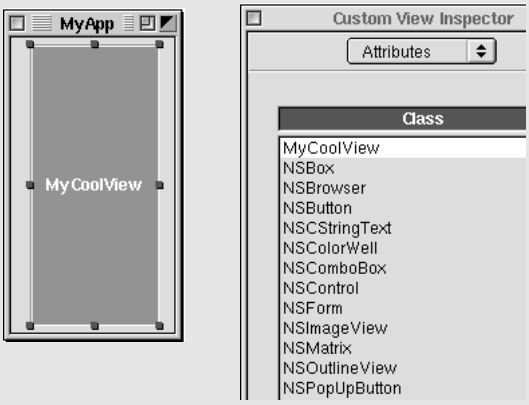
If you want an object that draws itself differently than any other Application Kit object, or responds to events in a special way, you should make a custom subclass of NSView. Your custom subclass should complete at least the steps outlined below.

Note: If you make a custom subclass of any class that inherits from NSView, and you want to do custom drawing or event handling, the basic procedure presented here still applies.

Interface Builder

1 Define a subclass of NSView in Interface Builder. Then generate header and implementation files.

2 Drag a CustomView object from the Views palette onto a window and resize it. Then, with the CustomView object still selected, choose the Custom Class display of the Inspector panel and select the custom class. Connect any outlets and actions.



Initializing Instances

3 Override the designated initializer, **initWithFrame:** to return an initialized instance of **self**. The argument of this method is the frame rectangle of the NSView, usually as set in Interface Builder (see step 2).

Handling Events

In the next section, you'll make a subclass of NSButtonCell that uniquely responds to mouse clicks. The way custom NSViews handle events is different. If you intend your custom NSView to respond to user actions you must do a couple of things:

4 Override **acceptsFirstResponder** to return YES if the NSView is to handle selections. (The default NSView behavior is to return NO.)

5 Override the desired NSResponder event methods (**mouseDown:**, **mouseDragged:**, **keyDown:**, etc.).

```
- (void)mouseDown:(NSEvent *)event {
    if (([event modifierFlags] &
        NSControlKeyMask){
        doSomething();
}
```

You can query the NSEvent argument for the location of the user action in the window, modifier keys pressed, character and key codes, and other information.

Drawing

When you send **display** to an NSView, its **drawRect:** method and each of its subview's **drawRect:** are invoked. This method is where an NSView renders its appearance.

6 Override **drawRect:**. The argument is usually the frame rectangle in which drawing is to occur. This tells the Window Server where the NSView's coordinate system is located. To draw the NSView, you can do one or more of the following:

- Composite an NSImage.
- Call Application Kit functions such as **NSRectFill()** and **NSFrameRect()** (NSGraphics.h).
- Call C functions that correspond to single PostScript operations, such as **PSsetgray()** and **PSfill()**.
- Call custom drawing functions created with **pswrap**.

7 When state changes and you need to have the object redraw itself, invoke **setNeedsDisplay:** with an argument of YES.

See "A Short Guide to Drawing and Compositing" on page 188 for more information on drawing techniques and requirements.

1 Update the current item with new values entered in the inspector.

Implement **switchChecked:** to apply changes made through switches and other controls.

```
- (void)switchChecked:(id)sender
{
    long tmpSecs=0;
    int idx = 0;
    id doc = [[NSApp mainWindow] delegate];
    if (sender == inspNotifAMPM) {                                    A
        if ([inspNotifHour intValue]) {
            tmpSecs = ConvertTimeToSeconds([inspNotifHour intValue],
                [inspNotifMinute intValue],
                [[sender cellAtRow:0 column:1] state]);
            [currentItem setSecsUntilDue:tmpSecs];
            [[NSApp mainWindow] setDocumentEdited:YES];
            [doc updateMatrix];
        }
    } else if (sender == inspNotifSwitchMatrix) {                     B
        idx = [inspNotifSwitchMatrix selectedRow];
        tmpSecs = [currentItem secsUntilDue];
        switch(idx) {
          case 0:
            [currentItem setSecsUntilNotif:0];
            break;
          case 1:
            [currentItem setSecsUntilNotif:tmpSecs-(hrInSecs/4)];
            break;
          case 2:
            [currentItem setSecsUntilNotif:tmpSecs-hrInSecs];
            break;
          case 3:
            [currentItem setSecsUntilNotif:tmpSecs-dayInSecs];
            break;
          case 4:  // Other
            [currentItem setSecsUntilNotif:([inspNotifOtherHours intValue]
                * hrInSecs)];
            break;
          default:
            NSLog(@"Error in selectedRow");
            break;
        }
        [[NSApp mainWindow] setDocumentEdited:YES];
    } else if (sender == inspSchedComplete) {                         C
        [currentItem setItemStatus:complete];
        [[NSApp mainWindow] setDocumentEdited:YES];
        [doc updateMatrix];

    } else if (sender == inspSchedMatrix) {                           D
    } /* left as an exercise */
}
```

When users click a switch button on any inspector display, or when they click one of the AM-PM radio buttons, the **switchChecked:** method is invoked. This method works by evaluating the **sender** argument: the sending object.

Ⓐ If **sender** is the radio-button matrix (AM-PM), gets the new time due by calling the utility function **ConvertTimeToSeconds()**, sets the current item to have this new value, marks the document as edited, and then sends **updateMatrix** to the document controller to have it display this new time.

Ⓑ If **sender** is the "When to Notify" matrix, gets the index of the selected cell and the seconds until the item is due. It evaluates the first value in a switch statement and uses the second value to set the current item's new **secsUntilNotif** value. It also sets the window to indicate an edited document.

Ⓒ If **sender** is the "Task Completed" switch, sets the status of the current item to "complete," sets the window to indicate an edited document, and has the document controller update its matrices.

Ⓓ As before, implementation of this rescheduling block is left as a final exercise.

Since text fields are controls that send target/action messages, you could also have **switchChecked:** respond when data is entered in the fields. However, users might not press Return in a text field so you can't assume the action message will be sent. Therefore, it's better to rely upon delegation messages.

Update the current item if changes are made to the contents of text fields or the text object of the inspector panel.

```
- (void)textDidEndEditing:(NSNotification *)notif                    A
{
    if ([notif object] == inspNotes)
        [currentItem setNotes:[inspNotes string]];
        [[NSApp mainWindow] setDocumentEdited:YES];
}

- (void)controlTextDidEndEditing:(NSNotification *)notif
{
    long tmpSecs=0;
    if ([notif object] == inspNotifHour ||                           B
        [notif object] == inspNotifMinute) {
        tmpSecs = ConvertTimeToSeconds([inspNotifHour intValue],
            [inspNotifMinute intValue],
            [[inspNotifAMPM cellAtRow:0 column:1] state]);
        [currentItem setSecsUntilDue:tmpSecs];
        [[[NSApp mainWindow] delegate] updateMatrix];
        [[NSApp mainWindow] setDocumentEdited:YES];
    } else if ([notif object] == inspNotifOtherHours) {              C
        if ([inspNotifSwitchMatrix selectedRow] == 4) {
            [currentItem setSecsUntilNotif:([inspNotifOtherHours
                intValue] * hrInSecs)];
            [[NSApp mainWindow] setDocumentEdited:YES];
}
    } else if ([notif object] == inspSchedDate) {                    D
    } /* left as an exercise */
}
```

The **textDidEndEditing:** and **controlTextDidEndEditing:** notification messages are sent to the delegate (and all other observers) when the cursor leaves a text object or text field (respectively) after editing has occurred.

A After editing takes place in the "Notes" text object, this method is invoked, and it responds by resetting the **notes** instance variable of the ToDoItem with the contents of the text object.

B If the object behind the notification is the hour or minute field of the "Notifications" display, **controlTextDidEndEditing:** computes the new due time, sets the current item to have this new value, and then sends **updateMatrix** to the document controller to have it display this new time. (This code is almost the same as that for the AM-PM matrix in the **switchChecked:** method.)

C If the object behind the notification is the "Other...hours" text field in the "When to Notify" box, the method verifies that the "Other" switch is checked and, if it is, sets the ToDoItem with the new value.

D Here is another empty rescheduling block of code that you can fill out in a later exercise.

Now it's time to address two related problems in synchronizing displays of data. The first is the requirement for the inspector to display the ToDoItem currently selected in the document. In **ToDoDoc.m** write code that communicates this object to ToDoInspector through notification.

1 Synchronize the items displayed in the document with the inspector.

Open **ToDoDoc.m**.

Import **ToDoInspector.h**.

Add the code at right to the end of the **controlTextDidEndEditing:** method.

Post identical notifications in the other ToDoDoc methods listed in the table below.

In **ToDoDoc.h** declare as extern the string constant **ToDoItemChangedNotification**.

In **ToDoDoc.m**, declare and initialize the same constant.

```
    id curItem;
/* ... */
        if (curItem = [currentItems objectAtIndex:row]) {
        if (![curItem isKindOfClass:[ToDoItem class]])
            curItem = nil;
        [[NSNotificationCenter defaultCenter] postNotificationName:
            ToDoItemChangedNotification object:curItem
            userInfo:nil];
    }
```

The **controlTextDidEndEditing:** method is where ToDoItems are added, removed, or modified, so it's especially important here to let ToDoInspector know when there's a change in the current ToDoItem. The fragment of code above gets the current item (**row** holds the index of the selected row); if the returned object isn't a ToDoItem, **curItem** is set to **nil**. Then the code posts a ToDoItemChangedNotification, passing in **curItem** as the object related to the notification.

Post an identical notification in other ToDoDoc methods that select a ToDoItem *or* that require the removal of the currently displayed ToDoItem from the inspector's display. In methods of this second type, there is no need to get the current item because the **object** argument of the notification should always be **nil**. This argument is eventually passed to ToDoInspector's **updateInspector:**, to which **nil** means "clear the display."

Other Methods Posting Notifications to ToDoInspector	object: Argument
calendarMatrix:didChangeToDate:	nil
calendarMatrix:didChangeToMonth:year:	nil
windowShouldClose: (for both "Save" and "Close")	nil
selectionInMatrix:	current item or nil

1 Open the inspector panel when users choose the Inspector command.

Implement ToDoController's showInspector: method to load ToDoInspector.nib and make the inspector panel the key window.

1 Update the document and inspector to display initial values.

In ToDoDoc.m, implement **selectItem:**.

Invoke this method at the appropriate places (see table below).

The second data-synchronization problem involves the selection and display of initial values in the document and the inspector when the user:

- Opens the inspector
- Opens a document
- Selects a new day from the calendar

You must return to **ToDoDoc.m** to write code that implements this behavior.

```
- (void)selectItem:(int)item
{
    id thisItem = [currentItems objectAtIndex:item];
    [itemMatrix selectCellAtRow:item column:0];
    if (thisItem) {
        if (![thisItem isKindOfClass:[ToDoItem class]]) thisItem = nil;
        [[NSNotificationCenter defaultCenter]
            postNotificationName:ToDoItemChangedNotification
                          object:thisItem
                        userInfo:nil];
    }
}
```

The **selectItem:** method selects the text field identified in the argument and posts a notification to the inspector with the associated ToDoItem as argument (or **nil** if the text field is empty). Next, invoke **selectItem:** in these methods:

Method	Comment
calendarMatrix:didChangeToDate:	Make it the final message, with an argument of 0 (**ToDoDoc.m**).
openDoc:	Invoke after opening a document, with an argument of 0 (**ToDoController.m**)
showInspector:	Invoke after opening the inspector panel, passing in the index of the selected row in the document. (**ToDoController.m**). Hint: Get the current document by querying for the delegate of the main window, then obtain the selected row from this object.

Before You Go On

Exercise: Make ToDoInspector respond to the notification. Declare a notification method named **currentItemChanged:** and implement it to set the current item with the **object** value of the notification. Then, in **init** or **awakeFromNib**, add ToDoInspector as an observer of the ToDoItemChangedNotification, identifying **currentItemChanged:** as the method to be invoked.

1 Set up the inspector when it is unarchived.

In ToDoInspector.m, implement **awakeFromNib** as shown at right.

```
- (void)awakeFromNib
{
    [inspPopUp selectItemAtIndex:0];                                    A
    [inspNotes setDelegate:self];

    [[notifView contentView] removeFromSuperview];                      B
    notifView = [[notifView contentView] retain];
    [[reschedView contentView] removeFromSuperview];
    reschedView = [[reschedView contentView] retain];
    [[notesView contentView] removeFromSuperview];
    notesView = [[notesView contentView] retain];
    [inspectorViews release];
    [self newInspectorView:self];
}
```

ToDoInspector's **awakeFromNib** method performs some necessary "housekeeping" tasks for the ToDoInspector instance of the application.

A Makes the Notification pop-up display the start-up default, using the index of the "Notification" cell rather than its title to improve localization. Then it sets **self** to be the delegate of the text object.

B Each of the three inspector displays in the off-screen panel (**inspectorViews**) is the content view of an NSBox. This section of code extracts and retains each of those content views, reassigning each to its original NSBox instance variable in the process. This explicit retaining is necessary because, in **newInspectorView:**, each current content view is released when it's swapped out. Once all content views are retained, the code releases the off-screen window and invokes **newInspectorView:** to put up the default display.

The use of notifications to communicate changes in one object to another object in an application is a good design strategy because it removes the need for the objects to have specific knowledge of each other. It also makes the application more extensible, because any number of objects can also become observers of the changes. However, there is a way for ToDoDoc to locate ToDoInspector reliably using the various relationships established within the program framework. See page 197 to see how this is done.

A Short Guide to Drawing and Compositing

Besides responding to events, all objects that inherit from NSView can render themselves on the screen. They do this rendering through image composition and PostScript drawing.

NSViews draw themselves as an indirect result of receiving the **display** message (or a variant of **display**); this message is sent explicitly or through conditions that cause automatic display. The **display** message leads to the invocation of an NSView's **drawRect:** method and the **drawRect:** methods of all subviews of that NSView. The **drawRect:** method should contain all code needed to redraw the NSView completely.

An NSView can be automatically displayed when:

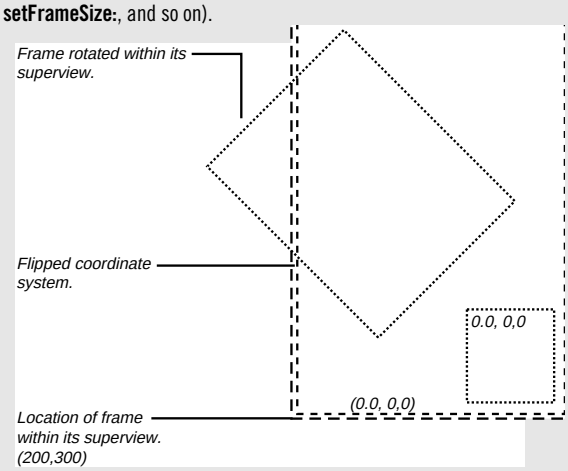
- Users scroll it (assuming it supports scrolling).
- Users resize or expose the NSView's window.
- The window receives a **display** message or is automatically updated.
- For some Application Kit objects, when an attribute changes.

An NSView represents a context within which PostScript drawing can take place. This context has three components:

- A rectangular frame within a window to which drawing is clipped
- A coordinate system
- The current PostScript graphics state

Frame and Bounds

An NSView's *frame* specifies the location and dimensions of the NSView in terms of the coordinate system of the NSView's superview. It is a rectangle that encloses the NSView. You can programmatically move, resize, and rotate the NSView by reference to its frame (**setFrameOrigin:**, **setFrameSize:**, and so on).



To draw efficiently, the NSView must have its frame rectangle translated into its own coordinate system. This translated rectangle, suitable for drawing, is called the *bounds*. The bounds rectangle usually specifies exactly the same area as the frame rectangle, but it specifies that area in a different coordinate system. In the default coordinate system, an NSView's bounds is the same as its frame, except that the point locating the frame becomes the origin of the bounds ($x = 0.0$, $y = 0.0$). The x- and y-axes of the default coordinate system run parallel to the sides of the frame so, for example, if you rotate the frame the default coordinate system rotates with it.

This relationship between frame and bounds has several implications important in drawing and compositing.

- Each NSView's coordinate system is a transformation of its superview's.
- Drawing instructions don't have to account for an NSView's location on the screen or its orientation.
- Changes in a superview's coordinate system are propagated to its subviews.

NSView allows you to flip coordinate systems (so the positive y-axis runs downward) and to otherwise alter coordinate systems.

Focusing

Before an NSView can draw it must *lock focus* to ensure that it draws in the correct window, place, and coordinate system. It locks focus by invoking NSView's **lockFocus** method. Focusing modifies the PostScript graphics state by:

- Making the NSView's window the current device
- Creating a clipping path around the NSView's frame
- Making the PostScript coordinate system match the NSView's coordinate system

After drawing, the NSView should unlock focus (**unlockFocus**).

PostScript Drawing

In Rhapsody, NSViews draw themselves by sending binary-encoded PostScript code to the Window Server. The Application Kit and the Display PostScript frameworks provide a number of C-language functions that send PostScript code to perform common drawing tasks. You can use these functions in combinations to accomplish fairly elaborate drawing.

The Application Kit has functions and constants, declared in **NSGraphics.h**, for (among other things):

- Drawing, filling, highlighting, clipping and erasing rectangles
- Drawing buttons, bezels, and bitmaps
- Computing window depth and related display information

You also call Yellow Box-compliant drawing routines defined in **dpsOpenStep.h**. These routines (such as **DPSDoUserPath()**) draw a specified path. In addition, you can call the functions declared in psops.h. These functions correspond to single PostScript operators, such as **PSsetgray()** and **PSfill()**.

You can also write and send your own custom PostScript code. The **pswrap** program converts PostScript code into C-language functions that you can call within your applications. It is an efficient way to send PostScript code to the Window Server. The following **pswrap** functions draw ovals.:

```
defines PDFramedOval(float x, y, w, h)
    matrix currentmatrix
    w h x y oval
    setmatrix stroke
endps

defines PSFilledOval float x, y, w, h)
    w h x y oval fill
endps
```

Compose the function in a file with a **.psw** extension and add it to the Other Source project "suitcase" in Project Builder. When you next build your project, Project Builder runs the **pswrap** program, generating an object file and a header file (matching the file name of the **.psw**) file, and links these into the application. To use the code, import the header file and call the function when you want to do the drawing:

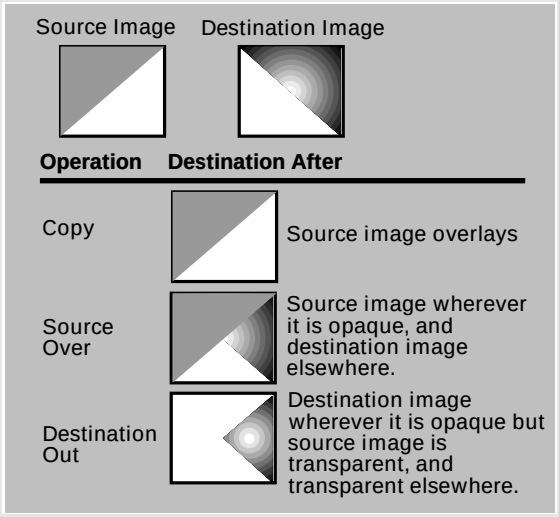
```
PSFilledOval(5.0, 5.0, 1.0, 1.0);
```

Compositing Images

The other technique NSViews use to render their appearance is image compositing. By compositing (with the SOVER operator) NSViews can simply display an image within their frame. You usually composite an image using NSImage's **compositeToPoint:operation:** (or a related method).

NSImage allows you to copy images into your user interface. It uses various subclasses of NSImageRep to store the multiple representations of the same image—color, grayscale, TIFF, EPS, and so on—and choosing the representation appropriate for a given type or display. NSImage can read image data from a bundle (including the application's main bundle), from the pasteboard, or from an NSData object.

Compositing allows you to do more than simply copy images. Compositing builds a new image by overlaying images that were previously drawn. It's like a photographer printing a picture from two negatives, one placed on top of the other. Various compositing operators (NSCompositingOperation, defined in **dpsOpenStep.h**) determine how the source and destination images merge.



You can achieve interesting effects with compositing when the initial images are drawn with partially transparent paint. (Transparency is specified by *coverage*, a indicator of paint opacity.) In a typical compositing operation, paint that's partially transparent won't completely cover the image it's placed on top of; some of the other image will show through. The more transparent the paint is, the more of the other image you'll see.

Overriding and Adding Behavior to a Class: An Example

Buttons in the Application Kit are two-state controls. They have two—and only two—states: 1 and 0 (often expressed as Boolean YES and NO, or ON and OFF). For the To Do application, a three-state button is preferable. You want the button to indicate, with an image, three possible states: not done (no image), done (an "X"), and deferred (a check mark). These states correspond to the possible states of a ToDoItem.

The ToDoCell class, which you will implement in this section, generates cells that behave as three-state buttons. These buttons also display the time an item is due.

1 Add the cell images to the project to the project.

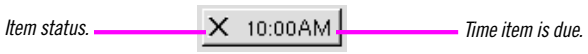
Select the Images "suitcase."

Choose Add Files from the Project menu.

In the Add Images panel, navigate to the ToDo project directory of **/System/Developer/Examples/AppKit** and select file **X.tiff**.

Click OK.

Repeat the same steps for file **checkMark.tiff**, which is in the same location.



The superclass of ToDoCell is NSButtonCell. In creating ToDoCell you will add data and behavior to NSButtonCell, and you will override some existing behavior.

Why Choose NSButtonCell as Superclass?

ToDoCell's superclass is NSButtonCell. This choice prompts two questions:

- Why a button cell and not the button itself?
- Why this particular superclass?

NSCell defines state as an instance variable, and thus all cells inherit it. Cells instead of controls hold state information for reasons of efficiency—one control (a matrix) can manage a collection of cells, each cell with its own state setting. NSButton does provide methods for getting and setting state values, but it accesses the state value of the cell (usually NSButtonCell) that it contains.

NSButtonCell is ToDoCell's superclass because button cells already have much of the behavior you want. By virtue of inheritance from NSActionCell, button cells can hold target and action information. Button cells also have the unique capability to display an image and text simultaneously. These are all aspects of behavior needed for ToDoCell.

When you think that you need a specialized subclass of a Yellow Box class, you should first spend some time examining the header files and reference documentation on not only that class, but its superclasses and any "sibling" classes.

1 Add header and implementation files to the project.

Choose New in Project from the File menu.

In the New File In ToDo panel, select the Class suitcase, click Create header, type "ToDoCell" after Name, and click OK.

1 Complete ToDoCell.h.

Make the superclass NSButtonCell.

Add the instance-variable and method declarations shown at right.

Add the **enum** constants for state values (as shown).

```
enum ToDoButtonState {notDone=0, done, deferred} ToDoButtonState;

@interface ToDoCell : NSButtonCell
{
    ToDoButtonState triState;
    NSImage *doneImage, *deferredImage;
    NSDate *timeDue;
}
- (void)setTriState:(ToDoButtonState)newState;
- (ToDoButtonState)triState;
- (void)setTimeDue:(NSDate *)newTime;
- (NSDate *)timeDue;
@end
```

The **triState** instance variable will be assigned ToDoButtonState constants as values. The NSImage variables hold the "X" and check mark images that represent statuses of completed and deferred (that is, rescheduled for the next day). The **timeDue** instance variable carries the time the item is due as an NSDate; for display, this object will be converted to a string.

1 Initialize the allocated ToDoCell instance (and deallocate it).

Select ToDoCell.m in the project browser.

Implement **init** as shown at right.

Implement **dealloc**.

```
- (id)init
{
    NSString *path;
    [super initTextCell:@""];

    triState = notDone;
    [self setType:NSToggleButton];                                    A
    [self setImagePosition:NSImageLeft];
    [self setBezeled:YES];
    [self setFont:[NSFont userFontOfSize:12]];
    [self setAlignment:NSRightTextAlignment];
                                                                      B
    path = [[NSBundle mainBundle] pathForImageResource:@"X.tiff"];
    doneImage = [[NSImage alloc] initByReferencingFile:path];
    path = [[NSBundle mainBundle]
        pathForImageResource:@"checkMark.tiff"];
    deferredImage = [[NSImage alloc] initByReferencingFile:path];

    return self;
}
```

A Sets some superclass (NSButtonCell) attributes, such as button type, image and text position, font of text, and border.

B Through NSBundle's **pathForImageResource:**, gets the pathname for the cell images and creates and stores the images using the pathname.

1 Implement the accessor methods related to state.

Write the methods that get and set the **triState** instance variable.

Override the superclass methods that get and set state.

```
- (void)setTriState:(ToDoButtonState)newState                    A
{
    if (newState == deferred+1)
        triState = notDone;
    else
        triState = newState;
    [self TD_setImage:triState];
 }

- (ToDoButtonState)triState {return triState;}

- (void)setState:(int)val                                        B
{
}

- (int)state                                                     C
{
    if (triState == deferred)
        return (int)done;
    else
        return (int)triState;
}
```

Accessing state information is a dual-path task in ToDoCell. It involves not only setting and getting the new state instance variable, **triState**, but properly handling the inherited instance variable by overriding the superclass accessor methods for state.

A If the new value for **triState** is one greater than the limit (**deferred**), reset it to zero (**notDone**); otherwise, assign the value. The reason behind this logic is that (as you'll soon learn) when users click a ToDoCell, **setTriState:** is invoked with an argument one more than the current value. This way users can cycle through the three states of ToDoCell.

B Overrides **setState:** to be a null method. The reason for this override is that NSCell intervenes when a button is clicked, resetting state to zero (NO). This override nullifies that effect.

C Overrides **state** to return a reasonable value to client objects that invoke this accessor method.

1 Set the cell image.

Declare the private method **TD_setImage:**.

Implement the **TD_setImage:** method.

```
@interface ToDoCell (PrivateMethods)
- (void)TD_setImage:(ToDoButtonState)aState;                    A
@end
/* ... */
- (void)TD_setImage:(ToDoButtonState)aState
{
    switch(aState) {                                            B
       case notDone: {
           [self setImage:nil];
           break;
       }
       case done: {
           [self setImage:doneImage];
           break;
       }
       case deferred: {
           [self setImage:deferredImage];
           break;
       }

     [(NSControl *)[self controlView] updateCell:self];         C
}
```

This portion of code handles the display of the cell's image by doing the following:

A In a category of ToDoCell in **ToDoCell.m**, it declares the private method **TD_setImage:**. Private methods are methods that you don't want clients of your object to invoke, and thus you don't "publish" them by declaring them in public header files. In this case, you don't want the image to be set independently from the cell's **triState** value.

B In a switch statement, evaluates the tri-state argument and sets the cell's image appropriately (**setImage:** is an NSButtonCell method).

C Sends **updateCell:** to the control view of the cell's control (a matrix) to force a re-draw of the cell.

1 Track mouse clicks on a ToDoCell and reset state.

Override two NSCell mouse-tracking methods as shown in this example.

```
- (BOOL)startTrackingAt:(NSPoint)startPoint inView:                    A
  (NSView *)controlView
{
    return YES;
}

- (void)stopTracking:(NSPoint)lastPoint at:(NSPoint)stopPoint
  inView:(NSView *)controlView mouseIsUp:(BOOL)flag                    B
{
    if (flag == YES) {
        [self setTriState:([self triState]+1)];
    }
}
```

When you create your own cell subclass, you might want to override some methods that are intrinsic to the behavior of the cell. Mouse-tracking methods, inherited from NSCell, are among these. You can override these methods to incorporate specialized behavior when the mouse clicks the cell or drags over it. ToDoCell overrides these methods to increment the value of **triState**.

A Overrides **startTrackingAt:inView:** to return YES, thus signalling to the control that the ToDoCell will track the mouse.

B Overrides **stopTracking:at:inView:mouseIsUp:** to evaluate flag and, if it's YES, to increment the **triState** instance variable. The **setTriState:** method "wraps" the incremented value to zero (**notDone**) if it is greater than 2 (**deferred**).

1 Get and set the time due, displaying the time in the process.

Implement setTimeDue: as shown in this example.

Implement **timeDue** to return the NSDate.

```
- (void)setTimeDue:(NSDate *)newTime
{
    if (timeDue)
        [timeDue autorelease];
    if (newTime) {
        timeDue = [newTime copy];
        [self setTitle:[timeDue descriptionWithCalendarFormat:
            @"%I:%M %p" timeZone:[NSTimeZone localTimeZone]
            locale:nil]];
    }
    else {
        timeDue = nil;
        [self setTitle:@"-->"];
    }
}
```

The **setTimeDue:** method is similar to other "set" accessor methods, except that it handles interpretation and display of the NSDate instance variable it stores. If **newTime** is a valid object, it uses **descriptionWithCalendarFormat:timeZone:locale:**,an NSDate method, to interpret and format the date object before displaying the result with

setTitle:. If **newTime** is **nil**, no due time has been specified, and so the method sets the title to "-->".

You've now completed all code required for ToDoCell. However, you must now "install" instances of this class in the To Do interface.

1 **At launch time, create and install your custom cells in the matrix.**

Select **ToDoDoc.m** in the project browser.

Insert the code at right in **awakeFromNib**.

```
- (void)awakeFromNib
{
    int i;
/* ... */
    i = [[markMatrix cells] count];
    while (i--) {
        ToDoCell *aCell = [[ToDoCell alloc] init];
        [aCell setTarget:self];
        [aCell setAction:@selector(itemChecked:)];
        [markMatrix putCell:aCell atRow:i column:0];
        [aCell release];
    }
}
```

This block of code substitutes a ToDoCell for each cell in the left matrix (**markMatrix**) you created for the To Do interface. It creates a ToDoCell, sets its target and action message, then inserts it into the **markMatrix** by invoking NSMatrix's **putCell:atRow:column:** method.

Finally, you must implement the action message sent when the matrix of ToDoCells is clicked. (This response to mouse-down is for objects external to ToDoCell, while the mouse-tracking response sets state internally.)

1 **Respond to mouse clicks on the matrix of ToDoCell's.**

In **ToDoDoc.m**, implement itemChecked:.

```
- (void)itemChecked:sender
{
    int row = [sender selectedRow];
    ToDoCell *cell = [sender cellAtRow:row column:0];
    if (cell && [currentItems count]) {
        id item = [currentItems objectAtIndex:row];
        if (item && [item isKindOfClass:[ToDoItem class]]) {
            [item setItemStatus:[cell triState]];
            [[sender window] setDocumentEdited:YES];
        }
    }
}
```

This method gets the ToDoCell that was clicked and the object in the corresponding text field. If that object is a ToDoItem, the method updates its status to reflect the state of the ToDoCell. It then marks the window as containing an edited document.

Setting Up Timers

One of To Do's features is the capability for notifying users of items with impending due times. Users can specify various intervals before the due time for these notifications, which take the form of a message in an attention panel. In this section you will implement the notification feature of To Do. In the process you'll learn the basics of creating, setting, and responding to timers.

Here's how it works: Each ToDoItem with a "When to Notify" switch (other than "Do not notify") selected in the inspector panel—and hence has a positive **secsUntilNotif** value—has a timer set for it. If a user cancels a notification by selecting "Do not notify," the document controller invalidates the timer. When a timer fires, it invokes a method that displays the attention panel, selects the "Do not notify" switch, and sets **secsUntilNotif** to zero.

Implementing the timer feature takes place entirely in Project Builder, but extends across several classes.

1 Add the timer as an instance variable to ToDoItem.

Open **ToDoItem.h**.

Add the instance variable **itemTimer** of class NSTimer.

Write accessor methods to get and set this instance variable.

1 Create and set the timer, or invalidate it.

Open **ToDoDoc.m**.

Implement the **setTimerForItem:** method, which is shown at right.

```
- (void)setTimerForItem:(ToDoItem *)anItem
{
    NSDate *notifDate;
    NSTimer *aTimer;
    if ([anItem secsUntilNotif]) {                                    A
        notifDate = [[anItem day] addTimeInterval:[anItem
            secsUntilNotif]];
        aTimer = [NSTimer scheduledTimerWithTimeInterval:             B
                    [notifDate timeIntervalSinceNow]
            target:self
          selector:@selector(itemTimerFired:)
          userInfo:anItem
           repeats:NO];
        [anItem setItemTimer:aTimer];
    } else
        [[anItem itemTimer] invalidate];                              C
}
```

This method sets or invalidates a timer, depending on whether the ToDoItem passed in has a positive **secsUntilNotif** value.

A Tests the ToDoItem to see if it has a positive **secsUntilNotif** value and, if it has, composes the time the notification should be sent.

B Creates a timer and schedules it to fire at the right time, directing it to invoke **itemTimerFired:** when it fires. It also sets the timer in the ToDoItem.

C If the **secsUntilNotif** variable is zero, invalidates the item's timer.

1 Respond to timers firing.

Implement **itemTimerFired:** as shown at right.

```
- (void)itemTimerFired:(id)timer
{
    id anItem = [timer userInfo];
    ToDoInspector *inspController = [[[NSApp delegate]          A
        inspector] delegate];
    NSDate *dueDate = [[anItem day] addTimeInterval:            B
        [anItem secsUntilDue]];
    NSBeep();
    NSRunAlertPanel(@"To Do", @"%@ on %@", nil, nil, nil,
        [anItem itemName], [dueDate
        descriptionWithCalendarFormat:@"%b %d, %Y at %I:%M %p"
        timeZone:[NSTimeZone defaultTimeZone] locale:nil]);
    [anItem setSecsUntilNotif:0];
    [inspController resetNotifSwitch];
}
```

When a ToDoItem's timer goes off, it invokes the **itemTimerFired:** method (remember, you designated this method when you scheduled the timer).

A This method communicates with ToDoInspector in a more direct manner than notification. It gets the ToDoInspector object through this chain of association: the delegate of the application object is ToDoController, which holds the **id** of the inspector panel as an instance variable, and the delegate of the inspector panel is ToDoInspector.

B Composes the notification time (as an NSDate), beeps, and displays an attention panel specifying the name of a ToDoItem and the time it is due. It then sets the ToDoItem's **secsUntilNotif** instance variable to zero, and sends **resetNotifSwitch** to ToDoInspector to have it reset the "When to Notify" switches to "Do not Notify."

Before You Go On

Exercise: You haven't written ToDoInspector's **resetNotifSwitch** method yet, so do it now as an exercise. It should select the "Do not Notify" switch after turning off all switches in the matrix, and then force a redisplay of the switch matrix.

Next you must send **setTimerForItem:** at the right place and time, which is ToDoInspector, when the user alters a "When to Notify" value.

1 Send the message that sets the timer at the right times.

Open **ToDoInspector.m**.

In **switchChecked:**, insert the **setTimerForItem:** message at right *after* the switch statement evaluating which "When to Notify" switch was checked.

In **controlTextDidEndEditing:**, insert the same message at the end of the block related to the **inspNotifOtherHours** variable.

```
[[[NSApp mainWindow] delegate] setTimerForItem:currentItem];
```

Instead of archiving an item's NSTimer, To Do re-creates and resets it when the application is launched.

1 When the application is launched, reset item timers.

Add the code shown at right to ToDoDoc's **initWithFile:** method.

```
if ([self activeDays]) {
     dayenum = [[self activeDays] keyEnumerator];
     while (itemDate = [dayenum nextObject]) {
         NSEnumerator *itemenum;
         ToDoItem *anItem=nil;
         NSArray *itemArray = [[self activeDays]
            objectForKey:itemDate];
         itemenum = [itemArray objectEnumerator];
         while ((anItem = [itemenum nextObject]) &&
                [anItem isKindOfClass:[ToDoItem class]] &&
                [anItem secsUntilNotif]) {
             [self setTimerForItem:anItem];
         }
     }
 }
```

This block of code traverses the **activeDays** dictionary, evaluating each ToDoItem within the dictionary. If the ToDoItem has a positive **secsUntilNotif** value, it invokes **setTimerForItem:** to have a timer set for it.

Tick Tock Brrrring: Run Loops and Timers

A run loop—an instance of NSRunLoop—manages and processes sources of input. These sources include mouse and keyboard events from the window system, file descriptor, inter-thread connections (NSConnection), and timers (NSTimer).

Applications typically won't need to either create or explicitly manage NSRunLoop objects. When a thread is created, an NSRunLoop object is automatically created for it. The NSApplication object creates a default thread and therefore creates a default run loop.

NSTimer creates timer objects. A timer object waits until a certain time interval has elapsed and then fires, sending a specified message to a specified object. For example, you could create an NSTimer that periodically sends messages to an object, asking it to respond if an attribute changes.

NSTimer objects work in conjunction with NSRunLoop objects. NSRunLoops control loops that wait for input, and they use NSTimers to help determine the maximum amount of time they should wait. When the NSTimer's time limit has elapsed, the NSRunLoop fires the NSTimer (causing its message to be sent), then checks for new input.

Build, Run, and Extend the Application

Although you probably have been building the ToDo project frequently now, as it's been taking shape, build it one more time and check out what you've created. Go through the following sequence and observe To Do's behavior.

1. When you choose New from the Document menu, the application creates a new To Do document and selects the current day.

2. Enter a few items. Click a new day on the calendar and enter a few more items. Click the previous day and notice how the items you entered reappear.

3. Choose Inspector from the main menu. When the inspector appears, click an item and notice how the name and date of the item appears in the top part of the inspector. Enter due times for a couple items, and some associated notes. Note how the times, as you enter them, appear in the Status/Due column of the To Do document. Click among a few items again and note how the Notifications and Notes displays change.

4. Click a Status/Due button; the image toggles among the three states. Then, with an item that has a due time, select a notification time that has already passed. The application immediately displays an attention panel with a notification message. When you dismiss this panel, To Do sets the notification option to "Do not notify."

5. Click the document window and respond to the attention panel by clicking Save. In the Save panel, give the document a location and name. When the window has closed, chose Open from the Document menu and open the same document. Observe how the items you entered are redisplayed.

Optional Exercises

You should be able now to supplement the To Do application with other features and behaviors. Try some of the following suggestions.

Make Your Own Info Panel

Make your own Info panel. Define a method that responds to a click on the Info panel button by loading a nib file containing the panel. The owner of the panel can be the application controller. You can customize this panel however you wish. For instance, put the application icon in a toggled button (the main image) and make the alternate image a photo (yourself, your significant other, your dog). When users click the button, the image changes between the two.

Implement Application Preferences

Make a Preferences panel for the application, with a new controller object (or the application controller) as the owner of the nib file containing the panel. Follow what you've done for ToDoInspector, especially if the panel has multiple displays. Some ideas for Preferences: how long to keep expired ToDoItems before logging and purging them (see below); the default document to open upon launch; the default rescheduling interval (see below). Store and retrieve specified preferences as user defaults; for more information, see the NSUserDefaults specification.

Implement Rescheduling

ToDo's Inspector panel has a Rescheduling display that does almost nothing now. Implement the capability for rescheduling items by the period specified.

Implement Logging and Purging

After certain period (set via Preferences), append expired ToDoItems (as formatted text) to a log, and expunge the ToDoItems from the application.