;H_ToDo_Subclass2.rtfd;;¬ Previous Section ;J_ToDo_Subclass3.rtfd;linkMarkername ;¬ Next Section

4. To Do Tutorial

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:

SquareBullet.eps ¬ Managing displays according to user selections
461651_SquareBullet.eps ¬ Getting the current ToDoItem
584652_SquareBullet.eps ¬ Updating the currently selected display
706421_SquareBullet.eps ¬ 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.

2 Create the inspector panel.

Drag a panel object from the Windows palette.
Make the title of the panel ªInspector.º
Resize the panel, using the example at right as a guide.
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.

TD_CreatingInspector1.eps ¬

TableRule.eps ¬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.
446655_TableRule.eps ¬

3 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.

Make the When to Reschedule and When to Notify groupings (boxes).

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

Resize the resulting boxes to the same dimensions as the ªdummyº view in the inspector panel.

TD_CreatingInspector2.eps ¬

36226_TableRule.eps ¬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.

ChangingInspectorView.eps ¬

866957_TableRule.eps ¬

4 Define the ToDoInspector class.

Create a subclass of NSObject and name it ªToDoInspector.º

Add the outlets and actions in the tables below to the new class.

Instantiate ToDoInspector.

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

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...**
TableHeadRule.eps ¬	
dummyView	The empty box object in the inspector panel
121729_TableRule.eps ¬	
inspectorViews	The title bar of the off-screen panel
230832_TableRule.eps ¬	
notesView	The box in the off-screen panel containing the scroll view
338492_TableRule.eps ¬	
notifView	The box in the off-screen panel containing the fields and controls related to notification of impending items
447184_TableRule.eps ¬	
reschedView	The box in the off-screen panel containing the fields and controls related to rescheduling items
556137_TableRule.eps ¬	

inspPopUp 664998_TableRule.eps ¬	The pop-up button on the inspector panel
inspDate 774517_TableRule.eps ¬	The uneditable text field next to the ªDateº label
inspItem 884312_TableRule.eps ¬	The uneditable text field next to the ªItemº label
inspNotifHour 992653_TableRule.eps ¬	The first field after the ªTimeº label
inspNotifMinute 100925_TableRule.eps ¬	The second field after the ªTimeº label
inspNotifAMPM 210004_TableRule.eps ¬	The matrix holding the ªAMº and ªPMº radio buttons
inspNotifOtherHours 373753_TableRule.eps ¬	The text field in the ªWhen to Notifyº box
inspNotifSwitchMatrix 485006_TableRule.eps ¬	The matrix of switches in the ªWhen to Notifyº box
inspSchedComplete 659124_TableRule.eps ¬	The ªTask Completedº switch
inspSchedDate 754494_TableRule.eps ¬	The text field in the ªWhen to Rescheduleº box
inspSchedMatrix 849876_TableRule.eps ¬	The matrix of switches in the ªWhen to Rescheduleº box
inspNotes 945383_TableRule.eps ¬	The text object inside the scroll view

Action **42815_TableHeadRule.eps ¬**	**Connection To ToDoInspector From...**
newInspectorView: 152520_TableRule.eps ¬	The pop-up button on the inspector panel
switchChecked: 259392_TableRule.eps ¬	The matrix of switches in the ªWhen to Notifyº box, the AM-PM matrix, the ªTask Completedº switch, and the matrix of switches in the ªWhen to Rescheduleº switches.

In Project Builder

5 Add declarations to ToDoInspector.h.

Open **ToDoInspector.h**.

Type the declarations shown below (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.

6 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];                 /* 1 */
    else
        currentItem = nil;
    [self updateInspector:currentItem];                 /* 2 */
}
```

This implementation of a ªsetº accessor method probably seems familiar to youÐexcept for a couple of things:

1. 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.

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

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

7 Switch inspector displays based on user selections.

Implement **newInspectorView:**.

```
- (void)newInspectorView:(id)sender
```

```
{
    NSBox *newView=nil;
    NSView *cView = [[inspPopUp window] contentView];       /* 1 */
    int selected = [[inspPopUp selectedItem] tag];
    switch(selected){                                       /* 2 */
      case notifTag:
        newView = notifView;
        break;
      case reschedTag:
        newView = reschedView;
        break;
      case notesTag:
        newView = notesView;
    }
    if ([[cView subviews] containsObject:newView]) return;  /* 3 */
    [dummyView setContentView:newView];                     /* 4 */
    if (newView == notifView) [inspNotifHour selectText:self];
    if (newView == notesView) [inspNotes
        setSelectedRange:NSMakeRange(0,0)];
    [self updateInspector:currentItem];                     /* 5 */
    [cView display];
}
```

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:

1. Gets the panel's content view and the tag of the selected pop-up button.
2. Assigns to the **newView** local variable the off-screen box object corresponding to the tag of the selected pop-up button.

3. 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.

4. 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 been retained, the replaced view doesn't disappear.

5. 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 **display** message forces a re-draw of the inspector panel's views.

8 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];                  /* 1 */
    [[inspPopUp window] orderFront:self];
    if (newItem && [newItem isKindOfClass:[ToDoItem class]]) { /* 2 */
        [inspItem setStringValue:[newItem itemName]];
        [inspDate setStringValue:[[newItem day]
            descriptionWithCalendarFormat:@"%a, %b %d %Y"
            timeZone:[NSTimeZone localTimeZone] locale:nil]];
        switch(selected) {
            case notifTag: {                                    /* 3 */
                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) {                              /* 4 */
  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.

1. Gets the tag assigned to the selected pop-up button.
2. 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.
3. 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 an array of NSNumbers (hour, minute, and PM boolean) and then setting the appropriate fields and button matrix with these values.
4. Sets the appropriate switch in the ªWhen to Notifyº 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.

544538_TableRule.eps ¬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 ªnullº range.

Note that tutorial omits the rescheduling logic of the ToDo application, including the code in this method that

would update the ªRescheduleº display. Rescheduling of ToDoItems is reserved as an optional exercise for you at the end of this tutorial.
650971_TableRule.eps ¬

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. This function has a counterpart, **indexOfSetCell()**, that returns the index of the currently selected switch.

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

```
void clearButtonMatrix(id matrix)
{
    int i, cnt=[[matrix cells] count];
    for(i=0; i<cnt; i++)
```

```
        [[matrix cellAtRow:i column:0] setState:NO];
}
```

The **cells** message returns the cells of the matrix as an array; the **count** message determines the number of cells.

9 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) {                              /* 1 */
        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) {               /* 2 */
        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) {                /* 3 */
          [currentItem setItemStatus:complete];
          [[NSApp mainWindow] setDocumentEdited:YES];
          [doc updateMatrix];

      } else if (sender == inspSchedMatrix) {                  /* 4 */
      }
  }
```

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.

1. 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.
2. If **sender** is the ªWhen to Notifyº 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.
3. If **sender** is the ªTask Completedº switch, sets the status of the current item to ªcomplete,º sets the window to indicate an edited document, and has the document controller update its matrices.
4. 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          /* 1 */
{
    if ([notif object] == inspNotes)
        [currentItem setNotes:[inspNotes string]];
        [[NSApp mainWindow] setDocumentEdited:YES];
}

- (void)controlTextDidEndEditing:(NSNotification *)notif
{
    long tmpSecs=0;
    if ([notif object] == inspNotifHour ||                 /* 2 */
        [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) {   /* 3 */
        if ([inspNotifSwitchMatrix selectedRow] == 4) {
            [currentItem setSecsUntilNotif:([inspNotifOtherHours
                intValue] * hrInSecs)];
            [[NSApp mainWindow] setDocumentEdited:YES];
}
    } else if ([notif object] == inspSchedDate) {        /* 4 */
    }
}
```

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.

1. After editing takes place in the ªNotesº 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.

2. If the object behind the notification is the hour or minute field of the ªNotificationsº 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.)

3. If the object behind the notification is the ªOther...hoursº text field in the ªWhen to Notifyº box, the method verifies that the ªOtherº switch is checked and, if it is, sets the ToDoItem with the new value.

4. 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.

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

Open **ToDoDoc.m**.

Import **ToDoInspector.h**.

Add the code below 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 ªclear the display.º

Other Methods Posting Notifications to ToDoInspector	**object: Argument**
904507_TableHeadRule.eps ¬	
calendarMatrix:didChangeToDate:	nil
10250_TableRule.eps ¬	
calendarMatrix:didChangeToMonth:year:	nil
116839_TableRule.eps ¬	
windowShouldClose: (for both ªSaveº and ªCloseº)	nil
212049_TableRule.eps ¬	
selectionInMatrix:	current item or nil
415972_TableRule.eps ¬	

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

512834_SquareBullet.eps ¬	Opens the inspector
629746_SquareBullet.eps ¬	Opens a document
739347_SquareBullet.eps ¬	Selects a new day from the calendar

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

11 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.

12 Update the document and inspector to display initial values.

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

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

```
- (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
880971_TableHeadRule.eps ¬	
calendarMatrix:didChangeToDate:	Make it the final message, with an argument of 0 (**ToDoDoc.m**).
992130_TableRule.eps ¬	
openDoc:	Invoke after opening a document, with an argument of 0 (**ToDoController.m**)
100026_TableRule.eps ¬	
showInspector:	Invoke afer 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.

214946_TableRule.eps ¬

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 "The Application Quartet: NSResponder, NSApplication, NSWindow, and NSView." ;ToDoConcepts.rtfd;linkMarkername TheApplicationQuartet:NSResponder,NSApplication,NSWindow,andNSView;¬

369340_TableRule.eps ¬Before You Go On

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.

549506_TableRule.eps ¬

13 Format and validate the contents of inspector text fields.

*In **ToDoInspector.m:***

Implement **awakeFromNib** as shown below.

Implement **control:isValidObject:** to ensure that users can only enter the proper range of numbers in the hour and minute text fields.

```
- (void)awakeFromNib
{
    NSDateFormatter *dateFmt;

    [[inspNotifHour cell] setEntryType:NSPositiveIntType]; /* 1 */
    [[inspNotifMinute cell] setEntryType:NSPositiveIntType];
    dateFmt = [[NSDateFormatter alloc]                     /* 2 */
        initWithDateFormat:@"%m/%d/%y" allowNaturalLanguage:YES];
    [[inspSchedDate cell] setFormatter:dateFmt];
    [dateFmt release];
    [inspPopUp selectItemAtIndex:0];                       /* 3 */
    [inspNotes setDelegate:self];

    [[notifView contentView] removeFromSuperview];         /* 4 */
    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 sets up formatters for the inspector's hour, minute, and date fields. It also performs some necessary ªhousekeepingº tasks.

1. Sets the hour and minute fields to accept only positive integer values.
2. Creates a date formatter (an instance of NSDateFormatter) that accepts and formats dates as (for example) ª12/25/96.º After associating the formatter with the date text-field cell, it releases it (**setFormatter:** retains the formatter).
3. Makes the Notification display the start-up default, using the index of the ªNotificationº cell rather than its title to improve localization. Then it sets **self** to be the delegate of the text object.
4. 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 currrent 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