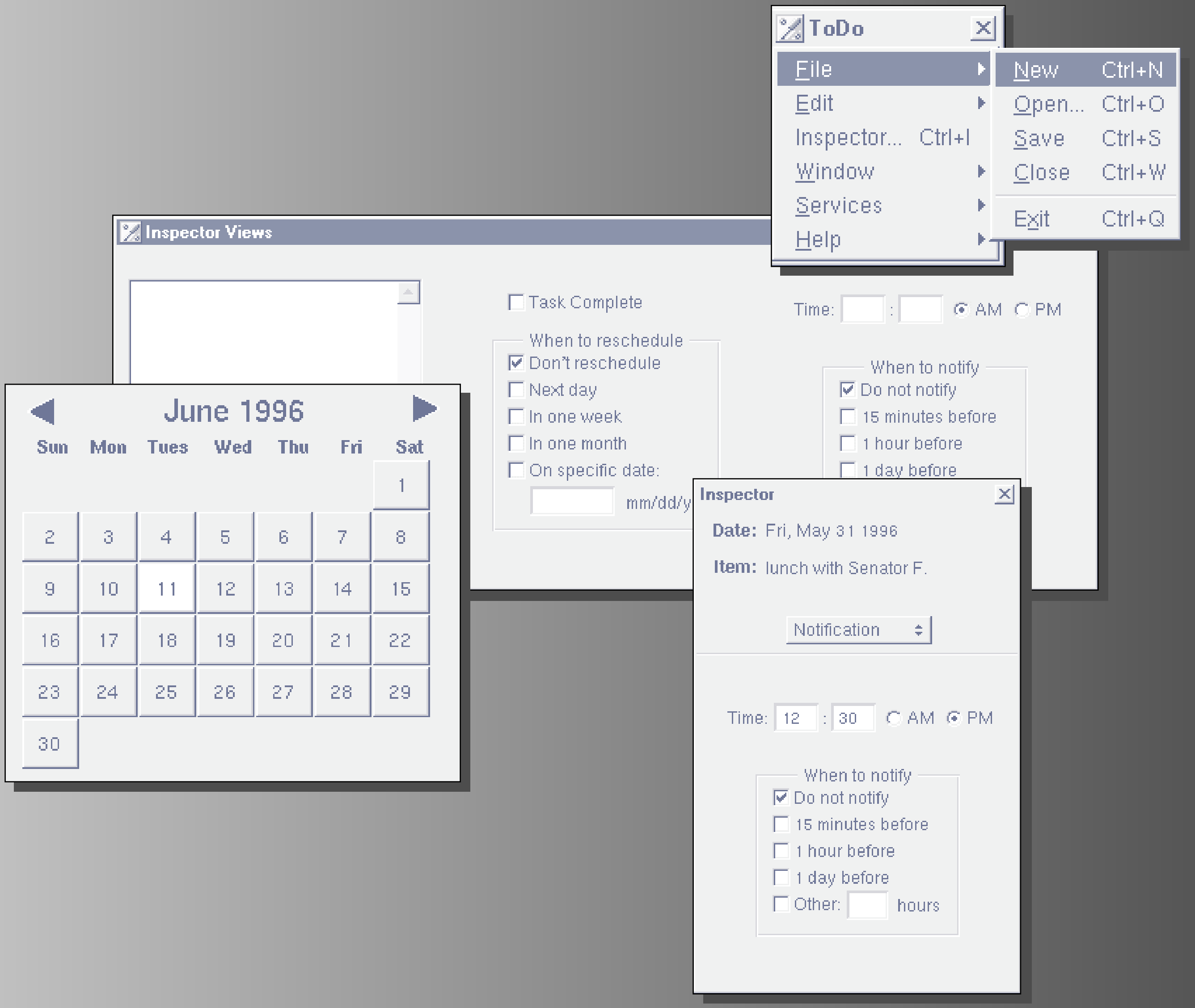
To Do Tutorial Chapter 4



Date: Fri, May 31 1996

Item: lunch with Senator F.

Time: 12 : 30 AM [✓ PM]

When to notify: [✓ Do not notify]

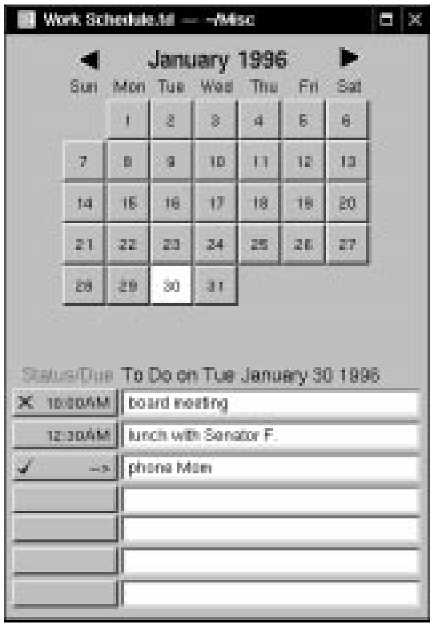
4

Chapter 4
To Do Tutorial

Many kinds of applications—word processors and spreadsheets, to name a couple—are designed with the notion of a *document* in mind. A document is a body of information, usually contained by a window, that is self-contained and repeatable. Users can create, modify, store, and access a document as a discrete unit. Multi-document applications (as these programs are called) can generate an almost unlimited number of documents.

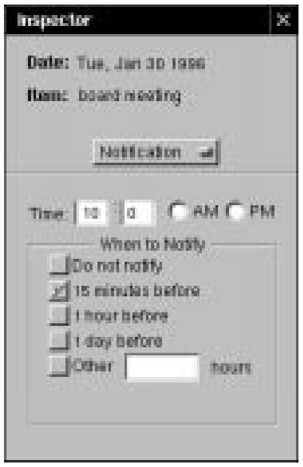
The To Do application presented in this chapter is a multi-document application. It is a fairly simple personal information manager (PIM). Each To Do document captures the daily "must-do" items for a particular purpose. For instance, one could have a To Do list for work and another one for home. To Do allows users to:

- Enter appointments or actions that they must complete on particular days.
- Specify the times those items are due.
- Receive notifications at a specified interval before the due time.
- Associate notes with to-do items.
- Mark items as complete or deferred.



To Do on Tue January 30 1996

✕ 10:00AM	board meeting
12:30AM	lunch with Senator F.
✓ -->	phone Mom



Date: Tue, Jan 30 1996
Item: board meeting
Time: 10 0
Do not notify
[✓ 15 minutes before]
1 hour before
1 day before
Other

As with Travel Advisor, you're going to cover a lot of OpenStep territory by completing this tutorial. It explores two major areas:

- Multi-document architecture: The design of applications that can create multiple documents, save and restore those documents, and do the right thing on certain events, such as application termination.

- Strategies for subclassing: Reuse of existing classes by adding behavior and data, by overriding existing behavior, or by doing both things.

You will also learn about other aspects of OpenStep programming:

- Opening and saving files
- Loading nib files (and other bundles) programmatically
- Creating and managing inspectors
- Programmatic creation and manipulation of user-interface objects
- Time and date manipulation
- Declaring informal protocols
- Using timers

And you'll be introduced to these important OpenStep concepts:

- Event handling
- The core program framework
- Drawing and image composition

When you complete this tutorial, you should be ready to tackle OpenStep programming on your own.

Starting Up — What Happens in NSApplicationMain()

Every OpenStep application project created through Project Builder has the same **main()** function (in the file *ApplicationName*_**main.m**). When users double-click an application or document icon in the File Viewer, **main()** (the entry point) is called first; **main()**, in turn, calls **NSApplicationMain()**—and that's all it does.

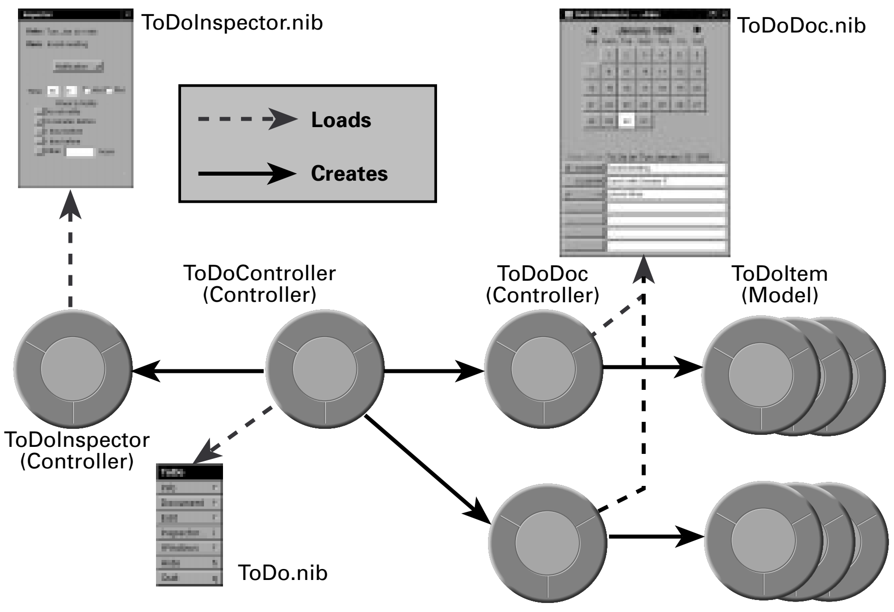
The **NSApplicationMain()** function does what's necessary to get an OpenStep application up and running—responding to events, coordinating the activity of its objects, and so on. The function starts the network of objects in the application sending messages to each other. Specifically, **NSApplicationMain()**:

1. Gets the application's attributes, which are stored in the application wrapper as a property list. From this property list, it gets the names of the main nib file and the principal class (for applications, this is NSApplication or a custom subclass of NSApplication).
2. Gets the Class object for NSApplication and invokes its **sharedApplication** class method, creating an instance of NSApplication, which is stored in the global variable, NSApp. Creating the NSApplication object connects the application to the window system and the Display PostScript server, and initializes its PostScript environment.
3. Loads the main nib file, specifying NSApp as the owner. Loading unarchives and re-creates application objects and restores the connections between objects.
4. Runs the application by starting the main event loop. Each time through the loop, the application object gets the next available event from the Window Server and dispatches it to the most appropriate object in the application. The loop continues until the application object receives a **stop:** or **terminate:** message, after which the application is released and the program exits.

You can add your own code to **main()** to customize application start-up or termination behavior.

The Design of To Do

The To Do application vaults past Travel Advisor in terms of complexity. Instead of Travel Advisor's one nib file, To Do has three nib files. Instead of three custom classes, To Do has seven. This diagram shows the interrelationships among instances of some of those classes and the nib files that they load:



Some of the objects in this diagram are familiar, fitting as they do into the Model-View-Controller paradigm. The ToDoItem class provides the model objects for the application; instances of this class encapsulate the data associated with the items appearing in documents. They also offer functions for computing subsets of that data. And then there's the controller object...actually, there is more than one controller object.

The ToDoInspector instance in the above diagram is an offshoot of the application controller, ToDoController. By breaking down a problem domain into distinct areas of responsibility, and assigning certain types of objects to each area, you increase the modularity and reusability of the object, and make maintenance and trouble-shooting easier. See "Object-Oriented Programming" in the appendix for more on this.

To Do's Multi-Document Design

Two types of controller objects are at the heart of multi-document application design. They claim different areas of responsibility within an application. ToDoController is the *application controller*; it manages events that affect the application as a whole. Each ToDoDoc object is a *document controller*, and manages a single document, including all the ToDoItems that belong to the document. Naturally, it's essential that the application controller be able to communicate with its (potentially) numerous document controllers, and they with it.

Only When Needed: Dynamically Loading Resources and Code

As any developer knows well, performance is a key consideration in program design. One factor is the timing of resource allocation. If an application loads all code and resources that it *might* use when it starts up, it will probably be a sluggish, bloated application—and one that takes awhile to launch.

You can strategically store the resources of an application (including user-interface objects) in several nib files. You can also put code that might be used among one or more *loadable bundles*. When the application needs a resource or piece of code, it loads the nib file or loadable bundle that contains it. This technique of deferred allocation benefits an application greatly. By conserving memory, it improves program efficiency. It also speeds up the time it takes to launch the application.

Auxiliary Nib Files

When more sophisticated applications start up, they load only a minimum of resources in the main nib file—the main menu and perhaps a window. They display other windows (and load other nib files) only when users request it or when conditions warrant it.

Nib files other than an application's main nib file are sometimes called *auxiliary nib files.* There are two general types of auxiliary nib files: special-use and document.

Special-use nib files contain objects (and other resources) that *might* be used in the normal operation of the application. Examples of special-use nib files are those containing inspector panels and Info panels.

Document nib files contain objects that represent some repeatable entity, such as a word-processor document. A document nib file is a template for documents: it contains the UI objects and other resources needed to make a document.

The Owner of an Auxiliary Nib File

The object that loads a nib file is usually the object that owns it. A nib file's owner must be external to the file. Objects unarchived from the nib file communicate with other objects in the application only through the owner.

In Interface Builder, the File's Owner icon represents this external object. The File's Owner is typically the application controller for special-use nib files, and the document controller for document nib files. The File's Owner object is not really appearing twice; it's created in one file and referenced in the other.

The File's Owner object dynamically loads a nib file and makes itself the owner of that file by sending **loadNibNamed:owner:** to NSBundle, specifying **self** as the second argument.

NSBundle and Bundles

A bundle is a location in the file system that stores code and the resources that go with that code, including images, sounds, and archived objects. A bundle is also identified with an instance of NSBundle, which makes the contents of the bundle available to other objects that request it.

The generic notion of bundles is pervasive throughout OpenStep. Applications are bundles, as are frameworks and palettes. Every application has at least one bundle—its main bundle—which is the ".app" directory (or *application wrapper*) where its executable file is located. This file is loaded into memory when the application is launched.

Loadable Bundles

You can organize an application into any number of other bundles in addition to the main bundle and the bundles of linked-in frameworks. Although these loadable bundles usually reside inside the application wrapper, they can be anywhere in the file system. Project Builder allows you to build Loadable Bundle projects.

Loadable bundles differ from nib files in that they don't require you to use Interface Builder to build them. Instead of containing mostly archived objects, they usually contain mostly code. Loadable bundles are especially useful for incorporating extra behavior into an application upon demand. An economic-forecast application, for example, might load a bundle containing the code defining an economic model, but only when users request that model. You could also use loadable bundles to integrate "plug and play" components into an existing framework.

Loadable bundles usually have an extension of ".bundle" (although that's a convention, not a requirement). Each loadable bundle must have a principal class that mediates between bundle objects and external objects.

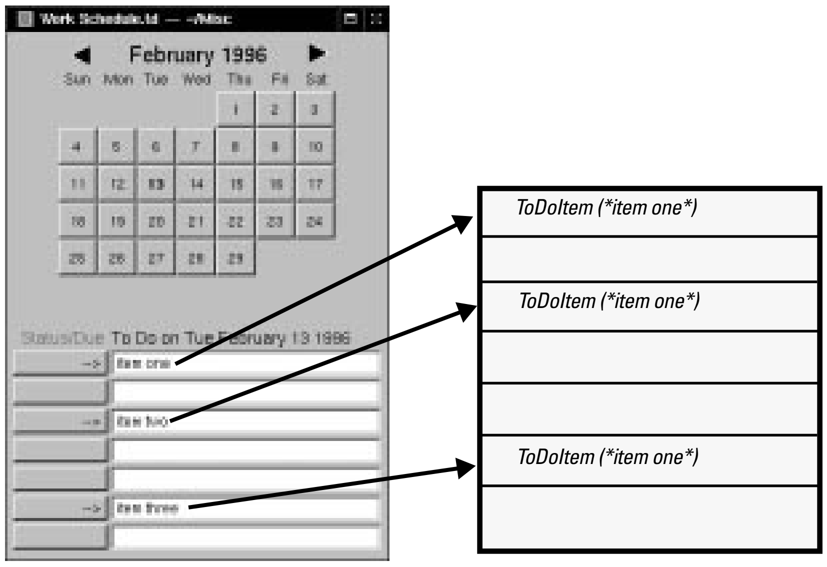
As multi-document applications typically do, To Do includes the Document menu found on Interface Builder's Menus palette. When users choose New from the Document menu, the application controller allocates and initializes an instance of the ToDoDoc class. When the ToDoDoc instance initializes itself, it loads the **ToDoDoc.nib** file. When the user has finished entering items into the document, and chooses Save from the Document menu, a Save panel appears and the user saves the document in the file system under an assigned name. Later, the user can open the document using the Open menu command, which causes the Open panel to be displayed.

The rationale behind, and process of, constructing multi-document applications is discussed in "The Structure of Multi-Document Applications" on page 141.

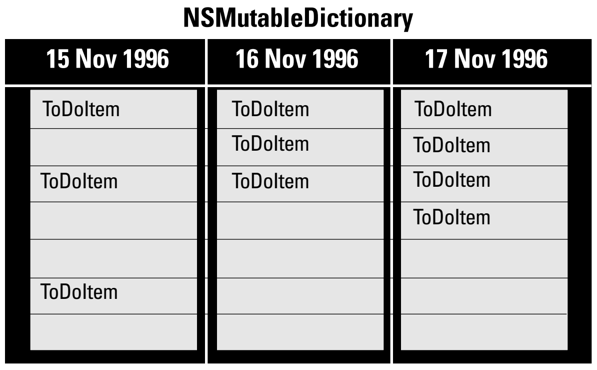
The controller objects of To Do respond to a variety of delegation messages sent when certain events occur—primarily from windows and NSApp—in order to save and store object state. One example of such an event is when the user closes a document window; another is when data is entered into a document. Often when these events happen, one controller sends a message to the other controller to keep it informed.

How To Do Stores and Accesses its Data

The data elements of a To Do document (ToDoDoc) are ToDoItems. When a user enters an item in a document's list, the ToDoDoc creates a ToDoItem and inserts that object in a mutable array (NSMutableArray); the ToDoItem occupies the same position in the array as the item in the matrix's text field. This positional correspondence of objects in the array and items in the matrix is an essential part of the design. For instance, when users delete the first entry in the document's list, the document removes the corresponding ToDoItem (at index 0) from the array.



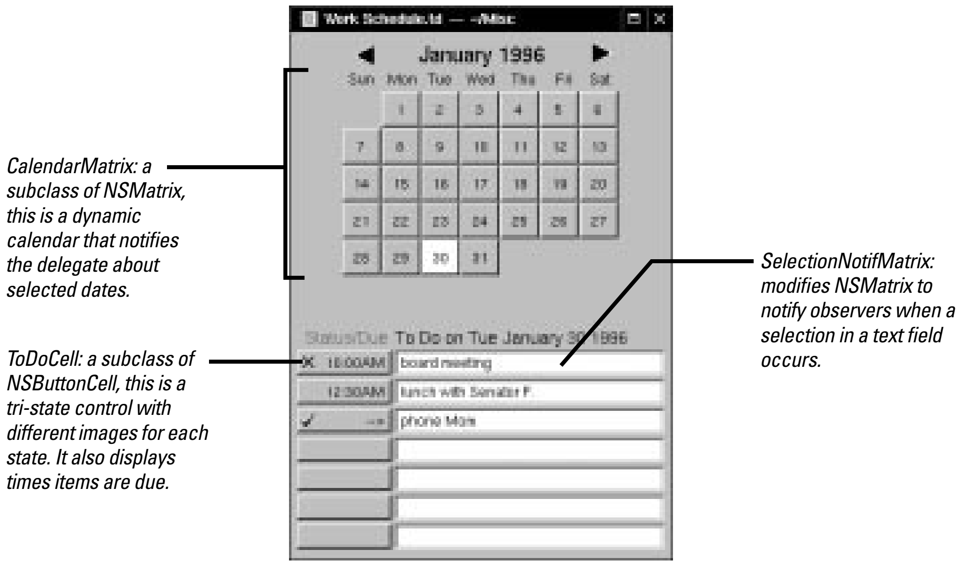
The array of ToDoItems is associated with a particular day. Thus the data for a document consists of a (mutable) dictionary with arrays of ToDoItems for values and dates for keys.



When users select a day in the calendar, the application computes the date, which it then uses as the key to locate an array of ToDoItems in the dictionary.

To Do's Custom Views

The discussion so far has touched on model objects and controller objects, but has said nothing about the second member of the Model-View-Controller triad: view objects. Unlike Travel Advisor, which uses only "off-the-shelf" views, To Do's interface features objects from three custom Application Kit subclasses.


Work Schedule.td — ~/Misc
January 1996
Sun Mon Tue Wed Thu Fri Sat
Status/Due To Do on Tue January 30 1996
10:00AM board meeting
12:30AM lunch with Senator F.
phone Mom

CalendarMatrix: a subclass of NSMatrix, this is a dynamic calendar that notifies the delegate about selected dates.

ToDoCell: a subclass of NSButtonCell, this is a tri-state control with different images for each state. It also displays times items are due.

SelectionNotifMatrix: modifies NSMatrix to notify observers when a selection in a text field occurs.

You'll learn much more about these custom subclasses in the pages that follow.

Setting up the To Do Project

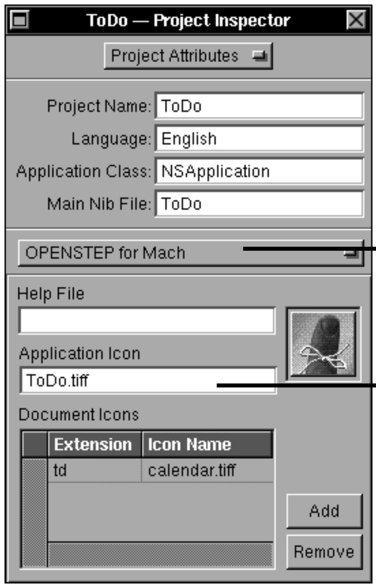
1 **Create the application project.**

Start Project Builder.

Choose New from the Project menu.

Name the application "ToDo."

Create the To Do project almost in the same way you created the Travel Advisor application. There are a few differences; each, of course, has a different name and icon. But the most important difference is that To Do has its own document type.



Project Name: ToDo
Language: English
Application Class: NSApplication
Main Nib File: ToDo
OPENSTEP for Mach
Application Icon: ToDo.tiff

Extension	Icon Name
td	calendar.tiff

You can have different icons and other project attributes for OpenStep for Mach and OpenStep for Windows.

Instead of dragging the image-file icon into the well, you can add the image file to the projedct and then just type the name of the image here.

2 **Add the application icon.**

The ToDo icon (**ToDo.tiff**) is located in the **ToDo** project in the **AppKit** subdirectory of **/NextDeveloper/Examples.**

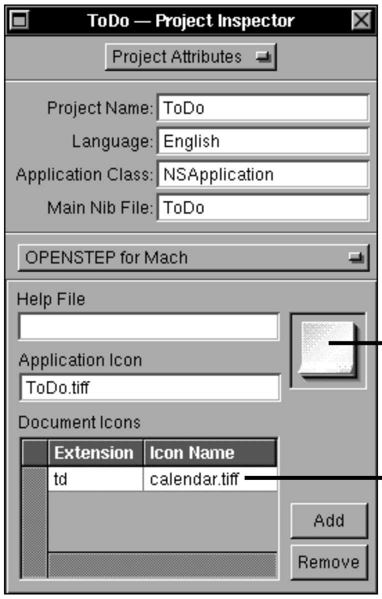
3 **Specify the To Do document type.**

Click Add.

Double-click the new cell under the Extension column.

Type the extension of To Do documents: "td".

Drag into the image well the file **calendar.tiff** from the **ToDo** project in **/NextDeveloper/Examples/ AppKit**.



Project Name: ToDo
Language: English
Application Class: NSApplication
Main Nib File: ToDo
OPENSTEP for Mach
Application Icon: ToDo.tiff

Extension	Icon Name
td	calendar.tiff

Document types specify the kinds of files the application can open and "understand." They appear in the workspace with the assigned icon and may be opened by double-clicking.

As with the applicaation icon, when you drag the document icon into the image well, the image file is added to the project.

Before Project Builder accepts the document icon, you must assign the extension (if the type is new) and select the row.

If the document type is well-known (for example, ".c"), just drag a document of that type into the well.

Creating the Model Class (ToDoItem)

The ToDoItem class provides the model objects for the To Do application. Its instance variables hold the data that defines tasks that should be done or appointments that have to be kept. Its methods allow access to this data. In addition, it provides functions that perform helpful calculations with that data. ToDoItem thus encapsulates both data *and* behavior that goes beyond accessing data.

Since ToDoItem is a model class, it has no user-interface duties and so the expedient course is to create the class without using Interface Builder. We first add the class to the project; Project Builder helps out by generating template source-code files.

1 **Add the ToDoItem class to the project.**

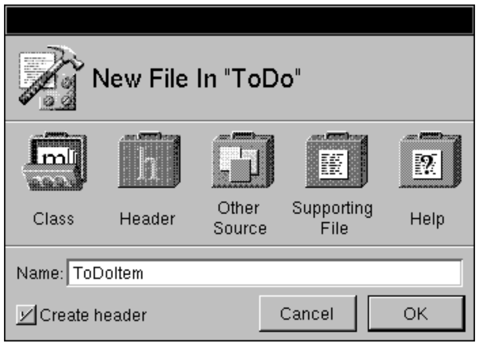
Select Classes in the project browser.

Choose New In Project from the File menu.

In the New File In ToDo panel, type "ToDoItem" in the Name field.

Make sure the "Create header" switch is checked.

Click the OK button.



Name: ToDoItem

☑ Create header

As you've done before with Travel Advisor, start by declaring instance variables and methods in the header file, **ToDoItem.h**.

2 **Declare ToDoItem's instance variables and methods.**

Type the instance variables as shown at right.

Indicate the protocols adopted by this class.

```
@interface ToDoItem:NSObject<NSCoding, NSCopying>
{
    NSCalendarDate *day;
    NSString *itemName;
    NSString *notes;
    NSTimer *itemTimer;
    long secsUntilDue;
    long secsUntilNotif;
    ToDoItemStatus itemStatus;
}
```

You are adopting the NSCopying protocol in addition to the NSCoding protocol because you are going to implement a method that makes "snapshot" copies of ToDoItem instances.

Instance Variable	What it Holds
day	The day (a date resolved to 12:00 AM) of the to-do item
itemName	The name of the to-do item (the content's of a document text field)
notes	The contents of the inspector's Notes display; this could be any information related to the to-do item, such as an agenda to discuss at a meeting.
itemTimer	A timer for notification messages.
secsUntilDue	The seconds after **day** at which the item comes due
secsUntilNotif	The seconds after **day** at which a notification is sent (before **secsUntilDue**)
itemStatus	Either "incomplete," "complete," or "deferToNextDay"

3 **Define enum constants for use in ToDoItem's methods.**

Define these constants before the **@interface** directive.

```
typedef enum _ToDoItemStatus {
    incomplete=0,
    complete,
    deferToNextDay
} ToDoItemStatus;

enum {
    minInSecs = 60,
    hrInSecs = (minInSecs * 60),
    dayInSecs = (hrInSecs * 24),
    weekInSecs = (dayInSecs * 7)
};
```

The first set of constants are values for the **itemStatus** instance variable. The second set of constants are for convenience and clarity in the methods that deal with temporal values.

4 **Declare two time-conversion functions.**

```
BOOL ConvertSecondsToTime(long secs, int *hour, int *minute);
long ConvertTimeToSeconds(int hr, int min, BOOL flag);
```

These functions provide computational services to clients of this class, converting time in seconds to hours and minutes (as required by the user interface), and back again to seconds (as stored by ToDoItem).

Type the method declarations shown at right.

```
- (id)initWithName:(NSString *)name andDate:(NSCalendarDate *)date;
- (void)dealloc;
- (BOOL)isEqual:(id)anObject;
- (id)copyWithZone:(NSZone *)zone;
- (id)initWithCoder:(NSCoder *)coder;
- (void)encodeWithCoder:(NSCoder *)coder;
- (void)setDay:(NSCalendarDate *)newDay;
- (NSCalendarDate *)day;
- (void)setItemName:(NSString *)newName;
- (NSString *)itemName;
- (void)setNotes:(NSString *)notes;
- (NSString *)notes;
- (void)setItemTimer:(NSTimer *)aTimer;
- (NSTimer *)itemTimer;
- (void)setSecsUntilDue:(long)secs;
- (long)secsUntilDue;
- (void)setSecsUntilNotif:(long)secs;
- (long)secsUntilNotif;
- (void)setItemStatus:(ToDoItemStatus)newStatus;
- (ToDoItemStatus)itemStatus;
```

Most of these declarations are for accessor methods. You know what to do.

5 **Implement accessor methods.**

Open **ToDoItem.m** in the code editor.

Implement methods that get and set the values of ToDoItem's instance variables.

Implement the **setItemTimer:** method as shown at right.

```
- (void)setItemTimer:(NSTimer *)aTimer
{
    if (itemTimer) {
        [itemTimer invalidate];
        [itemTimer autorelease];
    }
    itemTimer = [aTimer retain];
}
```

The **setItemTimer:** method is slightly different from the other "set" accessor methods. It sends **invalidate** to **itemTimer** to disable the timer before it autoreleases it.

Timers (instances of NSTimer) are always associated with a run loop (an instance of NSRunLoop). See "Tick Tock Brrrring: Run Loops and Timer" on page 190 for more on timers and run loops.

In this application, you want client objects to be able to copy your ToDoItem objects and test them for equality. You must define this behavior yourself.

6 **Implement the isEqual: method.**

```
- (BOOL)isEqual:(id)anObj
{
    if ([anObj isKindOfClass:[ToDoItem class]] &&
        [itemName isEqualToString:[anObj itemName]] &&
        [day isEqualToDate:[anObj day]])
        return YES;
    else
        return NO;
}
```

The default implementation of **isEqual:** (in NSObject) is based on pointer equality. However, ToDoItem has a different basis for equality; any two ToDoItem objects for the same calendar day and having the same item name are considered equal. The implementation of **isEqual:** overrides NSObject to make these tests. (Note that it invokes NSString's and NSDate's own **isEqual...** methods for the specific tests.)

Before You Go On

There is a specific as well as a general need for the **isEqual:** override. In the To Do application, an NSArray contains a day's ToDoItems. To access them, other objects in the application invoke several NSArray methods that, in turn, invoke the **isEqual:** method of each object in the array.

7 **Implement the copyWithZone: method.**

```
- (id)copyWithZone:(NSZone *)zone
{
    ToDoItem *newobj = [[ToDoItem alloc] initWithName:itemName
        andDate:day];
    [newobj setNotes:notes];
    [newobj setItemStatus:itemStatus];
    [newobj setSecsUntilDue:secsUntilDue];
    [newobj setSecsUntilNotif:secsUntilNotif];

    return newobj;
}
```

Copies of objects can be either *deep* or *shallow*. In deep copies (like ToDoItem's) every copied instance variable is an independent replicate, including the values referenced by pointers. In shallow copies, pointers are copied but the referenced objects are the same. For more on this topic, see the description of the NSCopying protocol in the Foundation reference documentation.

This implementation of the **copyWithZone:** protocol method makes a copy of a ToDoItem instance that is an independent replicate of the original (**self**). It does this by allocating a new ToDoItem object and initializing it with the essential instance variables held by **self**. Copying is often implemented for *value* objects—objects that represent attributes such as numbers, dates, and to-do items.

The next method you'll implement—**description**—assists you and other developers in debugging the To Do application with **gdb**. When you enter the **po** (print object) command in **gdb** with a ToDoItem as the argument, this **description** method is invoked and essential debugging information is printed.

8 **Implement the description method.**

```
- (NSString *)description
{
    NSString *desc = [NSString stringWithFormat:@"%@\n\tName: %@\n\tDate:
%@\n\tNotes: %@\n\tCompleted: %@\n\tSecs Until Due: %d\n\tSecs Until
Notif: %d",
        [super description],
        [self itemName],
        [self day],
        [self notes],
        (([self itemStatus]==complete)?@"Yes":@"No"),
        [self secsUntilDue],
        [self secsUntilNotif]];

    return (desc);
}
```

9 **Implement ToDoItem's initializing and deallocation methods.**

Here are some things to remember as you implement **initWithName:andDate:** and **dealloc**:

- If the first argument of **initWithName:andDate:** (the item name) is not a valid string, return **nil**. If the second argument (the date) is **nil**, set the related instance variable to some reasonable value (such as today's date). Also, be sure to invoke **super**'s **init** method.
- The instance variables to initialize are **day**, **itemName**, **notes**, and **itemStatus** (to "incomplete").
- In **dealloc**, release those object instance variables initialized in **initWithName:andDate:** plus any object instance variables that were initialized later. Also invalidate any timer before you release it.

10 **Implement ToDoItem's archiving and unarchiving methods.**

When you implement **encodeWithCoder:** and **initWithCoder:**, keep the following in mind:

- Encode and decode instance variables in the same order.
- Copy the object instance variables after you decode them.
- You don't need to archive the **itemTimer** instance variable since timers are re-set when a document is opened.

The final step in creating the ToDoItem class is to implement the functions that furnish "value-added" behavior.

11 **Implement ToDoItem's time-conversion functions.**

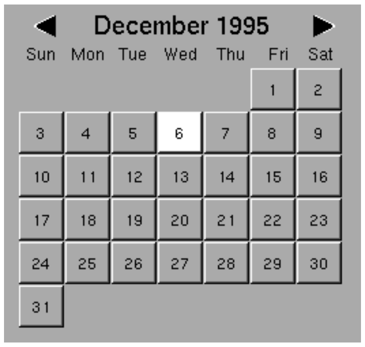
```
long ConvertTimeToSeconds(int hr, int min, BOOL flag)       /* 1 */
{
    if (flag) { /* PM */
        if (hr >= 1 && hr < 12)
            hr += 12;
    } else {
        if (hr == 12)
            hr = 0;
    }
    return ((hr * hrInSecs) + (min * minInSecs));
}

BOOL ConvertSecondsToTime(long secs, int *hour, int *minute) /* 2 */
{
    int hr=0;
    BOOL pm=NO;

    if (secs) {
        hr = secs / hrInSecs;
        if (hr > 12) {
            *hour = (hr -= 12);
            pm = YES;
        } else {
            pm = NO;
            if (hr == 0)
                hr = 12;
            *hour = hr;
        }
        *minute = ((secs%hrInSecs) / minInSecs);
    }
    return pm;
}
```

1. This expression, as well as others in these two methods, uses the **enum** constants for time-values-as seconds that you defined earlier.

2. The **ConvertSecondsToTime()** function uses indirection as a means for returning multiple values and directly returns a Boolean to indicate AM or PM.

Subclass Example: Adding Data and Behavior (CalendarMatrix)

The calendar on To Do's interface is an instance of a custom subclass of NSMatrix. CalendarMatrix dynamically updates itself as users select new months, notifies a delegate when users select a day, and reflects the current day (today) and the current selection by setting button attributes.



Creating a subclass of a class that is farther down the inheritance tree poses more of a challenge for a developer than a simple sublcass of NSObject. A class such as NSMatrix is more specialized than NSObject and carries with it more baggage: It inherits from NSResponder, NSView, and NSControl, all fairly complex Application Kit classes. And since CalendarMatrix inherits from NSView, it appears on the user interface; it is an example of a view object in the Model-View-Controller paradigm, and as such it is highly reusable.

Why NSMatrix?

When you select a specialized superclass as the basis for your subclass, it is important to consider what your requirements are and to understand what the superclass has to offer. To Do's dynamic calendar should:

- Arrange numbers (days) sequentially in rows and columns.
- Respond to and communicate selections of days.
- Understand dates.
- Enable navigation between months.

If you then started to peruse the reference documentation on Application Kit classes, and looked at the section on NSMatrix, you'd read this:

NSMatrix is a class used for creating groups of NSCells that work together in various ways. It includes methods for arranging NSCells in rows and columns.... An NSMatrix adds to NSControl's target/action paradigm by allowing a separate target and action for each of its NSCells in addition to its own target and action.

So NSMatrix has an inherent capability for the first of the requirements listed above, and part of the second (responding to selections). Our CalendarMatrix subclass thus does not need to alter anything in its superclass. It just needs to supplement NSMatrix with additional data and behavior so it can understand dates (and update itself appropriately), navigate between months, and notify a delegate that a selection was made.

1 **Define the CalendarMatrix class in Interface Builder.**

From Project Builder, open **ToDo.nib**.

In Interface Builder, choose Document ► New Module ► New Empty to create a new nib file.

Save the nib file as **ToDoDoc.nib**.

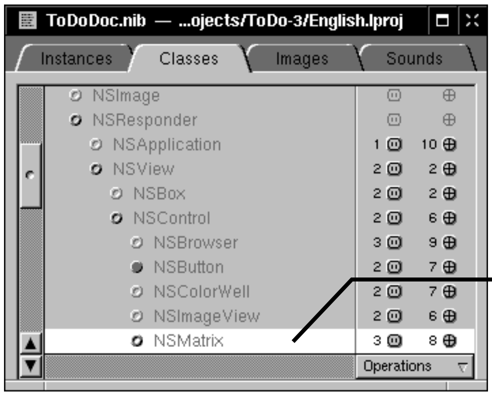
In the Classes display of the nib file window, select NSMatrix.

Choose Subclass from the pull-down list.

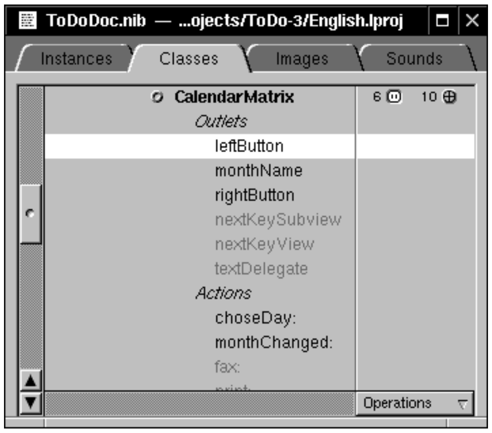
Name the new class "CalendarMatrix".

Select the new class.

Add the outlets and actions shown in the example at right.



Locate NSMatrix several levels down in the class hierarchy.



Outlets and actions already defined by the superclass (or its superclass) appear in gray text. Add the outlets and actions shown in black text.

When you created subclasses of NSObject in the previous two tutorials, the next step was to instantiate the subclass. Because CalendarMatrix is a view (that is, it inherits from NSView), the procedure for generating an instance for making connections is different.

2 **Put a custom NSView object (CalendarMatrix) on the user interface.**

Drag a window from the Windows palette.

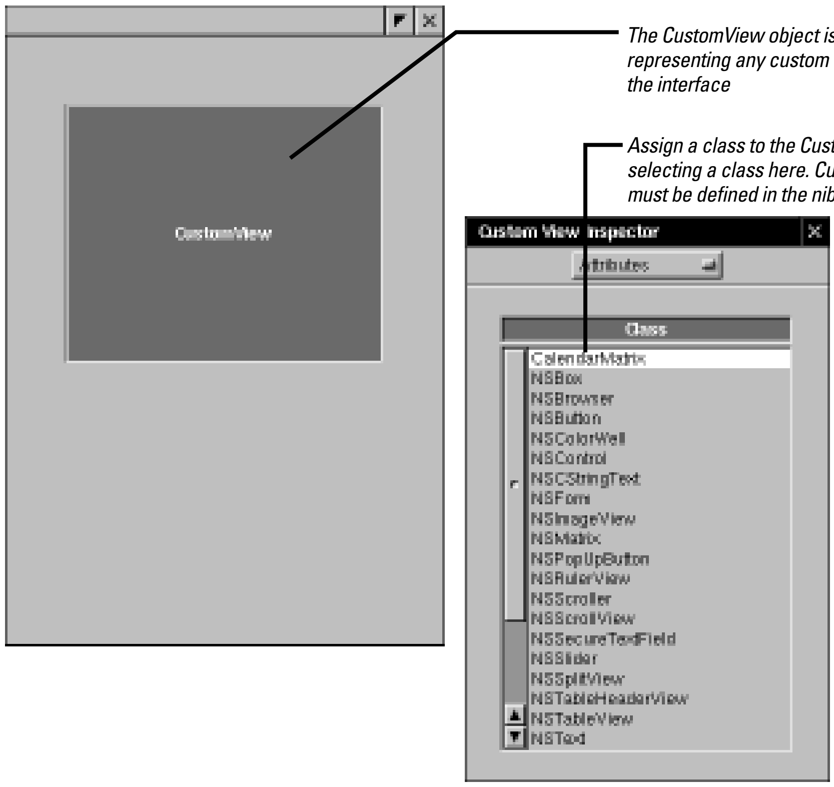
Resize the window, using the example at right as a guide.

Turn off the window's resize handle.

Drag a CustomView from the Views palette onto the window.

Resize and position the CustomView, using the example at right as a guide.

In the Attributes display of the inspector, select CalendarMatrix from the list of available classes.



The CustomView object is a "proxy" object, representing any custom NSView object on the interface

Assign a class to the CustomView by selecting a class here. Custom classes must be defined in the nib file.

The selection of the class for the CustomView creates an instance of it that you can connect to other objects in the nib file. Now put the controls and fields associated with CalendarMatrix on the window.

3 **Put the objects related to CalendarMatrix on the window.**

Drag a label object for the month-year from the Views palette and put it over the CalendarMatrix.

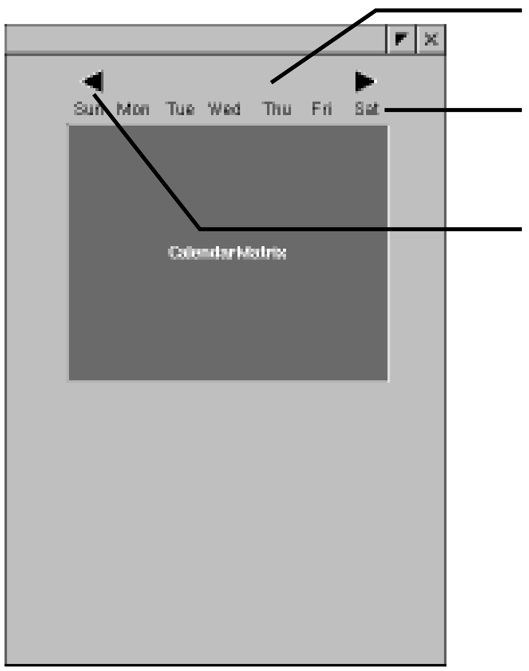
Make seven small labels for each day of the week.

Drag a button onto the interface and set its attributes to unbordered and image only.

Drag **left_arrow.tiff** from **/NextDeveloper/Examples/AppKit/ToDo** and drop it over the button.

To the attention panel that asks "Insert image left_arrow in project?" click Yes.

Repeat the same button procedure for **right_arrow.tiff**.



This label contains the month and year. Initialize by typing "September 9999" (the longest possible string), set text to Helvetica 18, center it, then delete it.

Type the days of the week as individual labels, arrange as a row, then distribute the fields evenly over the columns (this may take some trial and error).

To make the button enclose the image as tightly as possible, select the button and choose Format►Size►Size to fit.

Next connect CalendarMatrix to its satellite objects.

4 **Connect CalendarMatrix to its outlet and to the controls sending action messages.**

Name	Connection	Type
monthName	From CalendarMatrix to the label field above it	outlet
leftButton	From CalendarMatrix to the left-pointing arrow	outlet
rightButton	From CalendarMatrix to the right-pointing arrow	outlet
monthChanged:	From both arrows to CalendarMatrix	action

5 **Finish up in Interface Builder.**

Save **ToDoDoc.nib**.

Select CalendarMatrix and in the Classes display and choose Create Files from the Operations pull-down menu.

Confirm that you want the source-code files added to the project.

You might have noticed that there's an action message left unconnected: **choseDay:**. Because it is impossible in Interface Builder to connect an object with itself, you will make this connection programmatically.

6 **Add declarations to the header file CalendarMatrix.h.**

(Existing declarations are indicted by ellipsis.)

```
@interface CalendarMatrix : NSMatrix
{
    /* ... */
    NSCalendarDate *selectedDay;
    short startOffset;                              /* 1 */
}
   /* ... */
- (void)refreshCalendar;
- (id)initWithFrame:(NSRect)frameRect;
- (void)dealloc;
- (void)setSelectedDay:(NSCalendarDate *)newDay;
- (NSCalendarDate *)selectedDay;
@end

@interface NSObject(CalendarMatrixDelegate)           /* 2 */
 - (void)calendarMatrix:(CalendarMatrix *)obj
      didChangeToDate:(NSDate *)date;
 - (void)calendarMatrix:(CalendarMatrix *)obj
      didChangeToMonth:(int)mo year:(int)yr;
@end
```

There are a couple of interesting things to note about these declarations:

1. The cells in CalendarMatrix are sequentially ordered by tag number, left to right, going downward. **startOffset** marks the cell (by its tag) on which the first day of the month falls.

2. CalendarMatrixDelegate is a category on NSObject that declares the methods to be implemented by the delegate. This technique creates what is called an *informal protocol*, which is commonly used for delegation methods.

7 **Implement CalendarMatrix's initialization methods.**

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

Write the implementation of **initWithFrame:** (at right).

Implement **dealloc**.

```
- (id)initWithFrame:(NSRect)frameRect
{
    int i, j, cnt=0;
    id cell = [[NSButtonCell alloc] initTextCell:@""];
    NSCalendarDate *now = [NSCalendarDate date];         /* 1 */

    [super initWithFrame:frameRect                       /* 2 */
                    mode:NSRadioModeMatrix
               prototype:cell
            numberOfRows:6
         numberOfColumns:7];
    // set cell tags                                     /* 3 */
    for (i=0; i<6; i++) {
        for (j=0; j<7; j++) {
            [[self cellAtRow:i column:j] setTag:cnt++];
        }
    }
    [cell release];
    selectedDay = [[NSCalendarDate dateWithYear:[now yearOfCommonEra]
        month:[now monthOfYear]                          /* 4 */
          day:[now dayOfMonth]
         hour:0 minute:0 second:0
     timeZone:[NSTimeZone localTimeZone]] copy];

    return self;
}
```

The **initWithFrame:** method is an initializer of NSMatrix, NSControl and NSView.

1. This invocation of **date**, a class method declared by NSDate, returns the current date ("today") as an NSCalendarDate. (NSCalendarDate is a subclass of NSDate.)

2. This message to **super** (NSMatrix) sets the physical and cell dimensions of the matrix, identifies the type of cell using a prototype (an NSButtonCell), and specifies the general behavior of the matrix: radio mode, which means that only one button can be selected at any time.

3. Set the tag number of each cell sequentially left to right and down. Tags are the mechanism by which CalendarMatrix sets and retrieves the day numbers of cells.

4. This NSCalendarDate class method initializes the **selectedDay** instance variable to midnight of the current day, using the year, month, and day elements of the current date. The **localTimeZone** message obtains an NSTimeZone object with an suitable offset from Greenwich Mean Time.

Implement **awakeFromNib** as shown at right.

```
- (void)awakeFromNib
{
    [monthName setAlignment:NSCenterTextAlignment];
    [self setTarget:self];
    [self setAction:@selector(choseDay:)];
    [self setAutosizesCells:YES];
    [self refreshCalendar];
}
```

The **awakeFromNib** method performs additional initializations (some of which could just have easily been done in **initWithFrame:**). Most importantly, it sets **self** as its own target object and specifies an action method for this target, **choseDay:**, something that couldn't be done in Interface Builder. Other methods to note:

- **setAutosizesCells:** causes the matrix to resize its cells on every redraw.
- **refreshCalendar** (which you'll write next) updates the calendar.

The **refreshCalendar** method is fairly long and complex—it is the workhorse of the class—so you'll approach it in sections.

Dates and Times in OpenStep

In OpenStep you represent dates and times as objects that inherit from NSDate. The major advantage of dates and times as objects is common to all objects that represent basic values: they yield functionality that, although commonly found in most operating systems, is not tied to the internals of any particular operating-system.

NSDates hold dates and times as values of type NSTimeInterval and express these values as seconds. The NSTimeInterval type makes possible a wide and fine-grained range of date and time values, giving accuracy within milliseconds for dates 10,000 years apart.

NSDate and its subclasses compute time as seconds relative to an absolute reference date (the first instant of January 1, 2001). NSDate converts all date and time representations to and from NSTimeInterval values that are relative to this reference date.

NSDate provides methods for obtaining NSDate objects (including **date**, which returns the current date and time as an NSDate), for comparing dates, for computing relative time values, and for representing dates as strings.

The NSCalendarDate class, which inherits from NSDate, generates objects that represent dates conforming to western calendrical systems. NSCalendarDate objects also adjust the representations of dates to reflect their associated time zones. Because of this, you can track an NSCalendarDate object across different time zones. You can also present date information from time-zone viewpoints other than the one for the current locale.

Each NSCalendarDate object also has a calendar format string bound to it. This format string contains date-conversion specifiers that are very similar to those used in the standard C library function **strftime()**. NSCalendarDate can interpret user-entered dates that conform to this format string.

NSCalendar has methods for creating NSCalendarDate objects from formatted strings and from component time values (such as minutes, hours, day of week, and year). It also supplements NSDate with methods for accessing component time values and for representing dates in various formats, locales, and time zones.

8 **Implement the code that updates the calendar.**

Initialize the **MonthDays[]** array and write the **isLeap()** macro.

Determine the day of the week at the start of the month and the number of days in the month.

```
static short MonthDays[] =
      {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
#define isLeap(year) (((((year) % 4) == 0 && (((year) % 100) != 0))
                     || ((year) % 400) == 0))
```

```
- (void)refreshCalendar
{
    NSCalendarDate *firstOfMonth, *selDate = [self selectedDay],
        *now = [NSCalendarDate date];
    int i, j, currentMonth = [selDate monthOfYear];
    unsigned int currentYear = [selDate yearOfCommonEra];
    short daysInMonth;
    id cell;

    firstOfMonth = [NSCalendarDate dateWithYear:currentYear /* 1 */
                   month:currentMonth
                     day:1 hour:0 minute:0 second:0
                timeZone:[NSTimeZone localTimeZone]];
    [monthName setStringValue:[firstOfMonth                 /* 2 */
        descriptionWithCalendarFormat:@"%B %Y"]];
    daysInMonth = MonthDays[currentMonth-1]+1;              /* 3 */
    /* correct Feb for leap year */
    if ((currentMonth == 2) && (isLeap(currentYear))) daysInMonth++;
    startOffset = [firstOfMonth dayOfWeek];                 /* 4 */
```

Before it can start writing day numbers to the calendar for a given month, CalendarMatrix must know what cell to start with and how many cells to fill with numbers. The **refreshCalendar** method begins by calculating these values.

1. Creates an NSCalendarDate for the first day of the currently selected month and year (computed from the **selectedDay** instance variable).

2. Writes the month and year (for example, “February 1997”) to the label above the calendar.

3. Gets from the **MonthDays** static array the number of days for that month; if the month is February and it is a leap year, this number is adjusted.

4. Gets the day of the week for the first day of the month and stores this in the **startOffset** instance variable.

Write the **refreshCalendar** code that writes day numbers to the cells and sets cell attributes.

```
        for (i=0; i<startOffset; i++) {
         cell = [self cellWithTag:i];
         [cell setBordered:NO];
         [cell setEnabled:NO];
         [cell setTitle:@""];
         [cell setCellAttribute:NSCellHighlighted to:NO];
     }
     for (j=1; j < daysInMonth; i++, j++) {
         cell = [self cellWithTag:i];
         [cell setBordered:YES];
         [cell setEnabled:YES];
         [cell setFont:[NSFont systemFontOfSize:12]];
         [cell setTitle:[NSString stringWithFormat:@"%d", j]];
         [cell setCellAttribute:NSCellHighlighted to:NO];
     }
     for (;i<42;i++) {
         cell = [self cellWithTag:i];
         [cell setBordered:NO];
         [cell setEnabled:NO];
         [cell setTitle:@""];
         [cell setCellAttribute:NSCellHighlighted to:NO];
     }
```

The first and third for-loops in this section of code clear the leading and trailing cells that aren't part of the month's days. Because the current day is indicated by highlighting, they also turn off the highlighted attribute. The second for-loop writes the day numbers of the month, starting at **startOffset** and continuing until **daysInMonth**, and resets the font (since the selected day is in bold face) and other cell attributes.

Complete the **refreshCalendar** method implementation by resetting the "today" cell attribute.

```
        if ((currentYear == [now yearOfCommonEra])
            && (currentMonth == [now monthOfYear])) {
         [[self cellWithTag:([now dayOfMonth]+startOffset)-1]
       setCellAttribute:NSCellHighlighted to:YES];
         [[self cellWithTag:([now dayOfMonth]+startOffset)-1]
       setHighlightsBy:NSMomentaryChangeButton];
        }
 }
```

This final section of **refreshCalendar** determines if the newly selected month and year are the same as today's, and if so highlights the cell corresponding to today.

9 **Implement the monthChanged: action method.**

```
- (void)monthChanged:sender
{
    NSCalendarDate *thisDate = [self selectedDay];
    int currentYear = [thisDate yearOfCommonEra];
    unsigned int currentMonth = [thisDate monthOfYear];

    if (sender == rightButton) {                            /* 1 */
        if (currentMonth == 12) {
            currentMonth = 1;
            currentYear++;
        } else {
            currentMonth++;
        }
    } else {
        if (currentMonth == 1) {
            currentMonth = 12;
            currentYear--;
        } else {
            currentMonth--;
        }
    }                                                       /* 2 */
    [self setSelectedDay:[NSCalendarDate dateWithYear:currentYear
                   month:currentMonth
                     day:1 hour:0 minute:0 second:0
                timeZone:[NSTimeZone localTimeZone]]];
    [self refreshCalendar];
    [[self delegate] calendarMatrix:self                   /* 3 */
        didChangeToMonth:currentMonth year:currentYear];
}
```

The arrow buttons above CalendarMatrix send it the **monthChanged:** message when they are clicked. This method causes the calendar to go forward or backward a month.

1. Determines which button is sending the message, then increments or decrements the month accordingly. If it goes past the end or beginning of the year, it increments or decrements the year and adjusts the month.

2. Resets the **selectedDay** instance variable with the new month (and perhaps year) numbers and invokes **refreshCalendar** to display the new month.

3. Sends the **calendarMatrix:didChangeToMonth:year:** message to its delegate (which in this application, as you'll soon see, is a ToDoDoc controller object).

10 **Implement the choseDay: action method.**

```
- (void)choseDay:sender
{
    NSCalendarDate *selDate, *thisDate = [self selectedDay];
/* 1 */
    unsigned int selDay = [[self selectedCell] tag]-startOffset+1;
/* 2 */
    selDate = [NSCalendarDate dateWithYear:[thisDate yearOfCommonEra]
                                     month:[thisDate monthOfYear]
                                       day:selDay
                                      hour:0
                                    minute:0
                                    second:0
                                  timeZone:[NSTimeZone localTimeZone]];
/* 3 */
    [[self cellWithTag:[thisDate dayOfMonth]+startOffset-1]
        setFont:[NSFont systemFontOfSize:12]];
    [[self cellWithTag:selDay+startOffset-1] setFont:
        [NSFont boldSystemFontOfSize:12]];
/* 4 */
    [self setSelectedDay:selDate];
    [[self delegate] calendarMatrix:self didChangeToDate:selDate];
}
```

This method is invoked when users click a day of the calendar.

1. Gets the tag number of the selected cell and subtracts the offset from it (plus one to adjust for zero-based indexing) to find the number of the selected day.
2. Derives an NSCalendarDate that represents the selected date.
3. Sets the font of the previously selected cell to the normal system font (removing the bold attribute) and puts the number of the currently selected cell in bold face.
4. Sets the **selectedDay** instance variable to the new date and sends the **calendarMatrix:didChangeToDate:** message to the delegate.

11 **Implement accessor methods for the selectedDay instance variable.**

You are finished with CalendarMatrix. If you loaded **ToDoDoc.nib** right now, the calendar would work, up to a point. If you clicked the arrow buttons, CalendarMatrix would display the next or previous months. The days of the month would be properly set out on the window, and the current day would be highlighted.

But not much else would happen. That's because CalendarMatrix has not yet been hooked up to its delegate.

The Basics of a Multi-Document Application

A multi-document application, as described on page 141, has at least one application controller and a document controller for each document opened. The application controller also responds to user commands relating to documents and either creates, opens, closes, or saves a document.

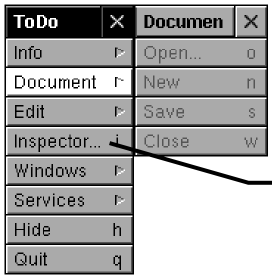
1 **Customize the application's main menu.**

Open **ToDo.nib** in Interface Builder.

Drag the Document item from the Menus palette and drop it between the Info and the Edit submenus.

Drag the Item item from the Menus palette and drop it between the Edit and Windows menus.

Change the title of "Item" to "Inspector."



Customize the document submenu by deleting the Save As, Save To, Save All and Revert To Saved commands.

Append an ellipsis (three dots) to the command name to indicate that the command displays a panel. Also enter "i" as the key equivalent.

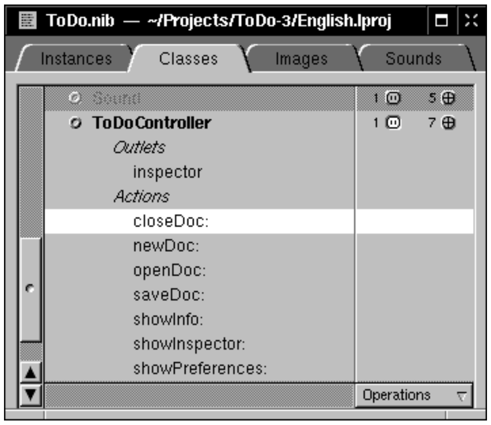
Note: The Info submenu, which you get by default, includes the Info Panel, Preferences, and Help commands. Although this tutorial does not cover implementing Info and Preferences panels specifically, it does give you enough information (which it will supplement with tips) so that you can try to implement these panels on your own. You may delete the Help command from the Info submenu if you wish; if you leave it in and users click it, they get a message informing them that Help is not available.

2 **Define the application-controller class.**

Create ToDoController as a subclass of NSObject.

Add the outlet and actions (listed at right) to the class.

Make the action connections from the appropriate Document menu commands.



Now that you've defined the application-controller class, define the document-controller class, ToDoDoc. Remember, since the ToDoDoc controller must own the nib file containing the document, it must be external to it; although it is defined in the main nib file (**ToDo.nib**) and in **ToDoDoc.nib**, it's instantiated before its nib file is loaded.

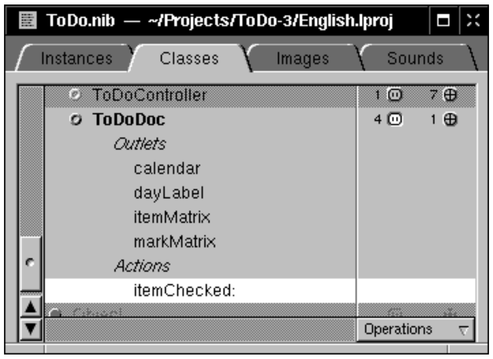
3 **Define the document-controller class.**

Create ToDoDoc as a subclass of NSObject.

Add to the class the outlets and action listed at right.

Instantiate ToDoController and ToDoDoc.

Save **ToDo.nib**.



Now add the remaining objects to the document interface.

4 **Complete the document interface.**

Open **ToDoDoc.nib**.

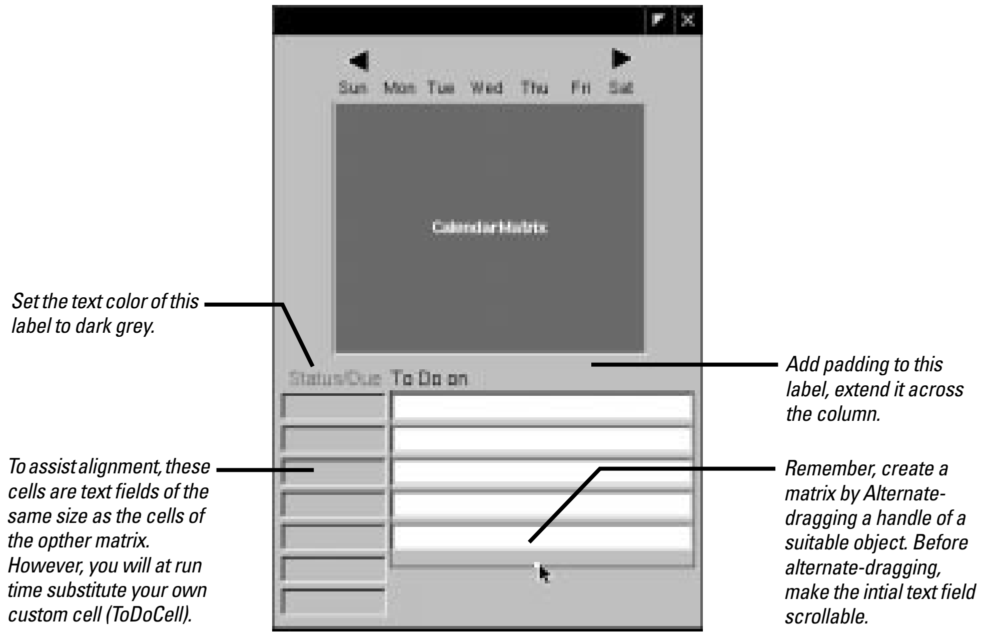
Add the matrices of text fields.

Add the labels above the matrices.

Make the labels 14 points in the user's application font.

Make the item text 12 points in the user's application font.

Save **ToDoDoc.nib**.


Sun Mon Tue Wed Thu Fri Sat
CalendarMatrix
Status/Due
To Do on
Set the text color of this label to dark grey.
Add padding to this label, extend it across the column.
To assist alignment, these cells are text fields of the same size as the cells of the opther matrix. However, you will at run time substitute your own custom cell (ToDoCell).
Remember, create a matrix by Alternate-dragging a handle of a suitable object. Before alternate-dragging, make the intial text field scrollable.

5 **Connect the outlets and actions of ToDoDoc.**

Select File's Owner in the Instances display of **ToDoDoc.nib**.

Choose ToDoDoc from the list of classes in the Attributes display of the inspector.

Make the connections described in the table at right.

Name	Connection	Type
calendar	From File's Owner to the CalendarMatrix object	outlet
dayLabel	From File's Owner to label "To Do on"	outlet
itemMatrix	From File's Owner (ToDoDoc) to matrix of long text fields	outlet
markMatrix	From File's Owner to matrix of short text fields	outlet
itemChecked:	From matrix of short text fields to File's Owner	action

The Structure of Multi-Document Applications

From a user's perspective, a document is a unique body of information usually contained by its own window. Users can create an unlimited number of documents and save each to a file. Common documents are word-processing documents and spreadsheets.

From a programming perspective, a document comprises the objects and resources unarchived from an auxiliary nib file and the controller object that loads and manages these things. This *document controller* is the owner of the auxiliary nib file containing the document interface and related resources.To manage a document, the document controller makes itself the delegate of its window and its "content" objects. It tracks edited status, handles window-close events, and responds to other conditions.

When users choose the New (or equivalent) command, a method is invoked in the application's controller object. In this method, the application controller creates a document-controller object, which loads the document nib file in the course of initializing itself. A document thus remains independent of the application's "core" objects, storing state data in the document controller. If the application needs information about a document's state, it can query the document controller.

When users chose the Save command, the application displays a Save panel and enables users to save the document in the file system. When users chose the Open command, the application displays an Open panel, allowing users to select a document file and open it.

Document Management Techniques

When you make the application controller and the document controller delegates of the application (NSApp) and the document window, they can receive messages sent at critical moments of a running application. These moments include the closure of windows (**windowShouldClose:**), window selection (**windowDidResignMain:**), application start-up (**applicationWillFinishLaunching:**) and application termination (**applicationShouldTerminate:**). In the methods handling these messages, the controllers can then do the appropriate thing, such as saving a document's data or displaying an empty document.

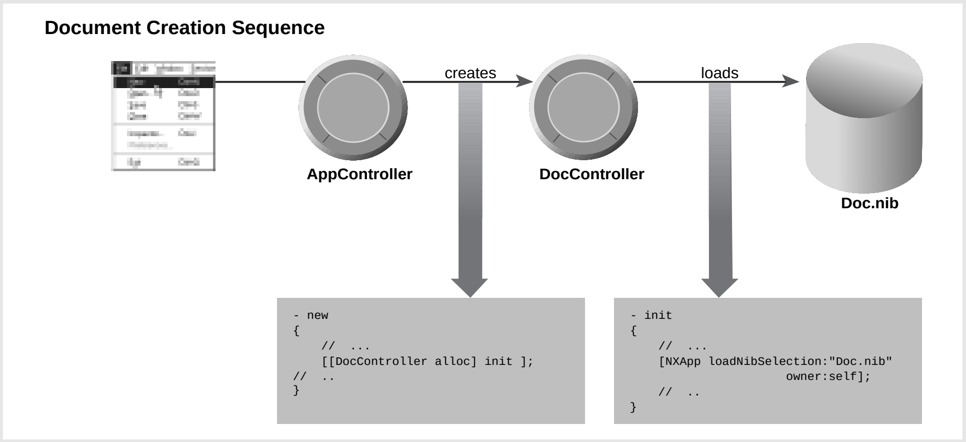
Several NSViews also have delegation messages that facilitate document management, particularly text fields, forms, and other controls with editable text (**controlText...**) and NSText objects (**text...**). One important such message is **textDidChange:** (or **controlTextDidChange:**), which signals that the document's textual content was modified. In responding to this message, controllers can set the window's close button to have a "broken" X with the **setDocumentEdited:** message; later, they can determine whether the document needs to be saved by sending **isDocumentEdited** to the window.

Document controllers often need to communicate with the application controller or other objects in the application. One way to do this is by posting notifications. Another way is to use the key relationships within the core program framework (see page 149) to find the other object (assuming it's a delegate of an Application Kit object). For example, the application controller can send the following message to locate the current document controller:

```
[[NSApp mainWindow] delegate]
```

The document controller can find the application controller with:

```
[NSApp delegate]
```



```
- new
{
    //  ...
    [[DocController alloc] init ];
//  ..
}
```

```
- init
{
    //  ...
    [NXApp loadNibSelection:"Doc.nib"
                     owner:self];
    //  ..
}
```

Text fields in a matrix, just like a form's cells, are connected for inter-field tabbing when you create the matrix. But you must also connect ToDoDoc and ToDoController to the delegate outlets of other objects in the application—this step is critical to the multi-document design.

Connect ToDoDoc and ToDoController to other objects as their delegates.

Name	Connection
textDelegate	From the CalendarMatrix object to File's Owner (ToDoDoc)
delegate	From the document window's title bar to File's Owner (ToDoDoc)
delegate	In **ToDo.nib**, from File's Owner (NSApp) to the ToDoController instance

6 **Create source-code files for ToDoDoc and ToDoController.**

The ToDoDoc class needs supplemental data and behavior to get the multi-document mechanism working right.

In Project Builder:

7 **Add declarations of methods and instance variables to the ToDoDoc class.**

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

Add the declarations at right.

(Ellipses indicate existing declarations.)

```
@interface ToDoDoc:NSObject
{
    /* ... */
    NSMutableDictionary *activeDays;
    NSMutableArray *currentItems;
}
/* ... */
- (NSMutableArray *)currentItems;
- (void)setCurrentItems:(NSMutableArray *)newItems;
- (NSMatrix *)itemMatrix;
- (NSMatrix *)markMatrix;
- (NSMutableDictionary *)activeDays;
- (void)saveDoc;
- (id)initWithFile:(NSString *)aFile;
- (void)dealloc;
- (void)activateDoc;
- (void)selectItem:(int)item;
@end
```

The **activeDays** and **currentItems** instance variables hold the collection objects that store and organize the data of the application. (You'll deal with these instance variables much more in the next section of this tutorial.) Many of the methods declared are accessor methods that set or return these instance variables or one of the matrices of the document.

You'll be switching between **ToDoDoc.m** and **ToDoController.m** in the next few tasks. The intent is not to confuse, but to show the close interaction between these two classes.

8 **Write the code that creates documents.**

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

Implement ToDoController's **newDoc:** method.

```
- (void)newDoc:(id)sender
{
    id currentDoc = [[ToDoDoc alloc] initWithFile:nil];
    [currentDoc activateDoc];
}
```

The **newDoc:** method is invoked when the user chooses New from the Document menu. The method allocates and initializes an instance of the document controller, ToDoDoc, thereby creating a document. (See the implementation of **initWithFile:** on the following page to see what happens in this process.) It then updates the document interface by invoking **activateDoc.**.

Coordinate Systems in OpenStep

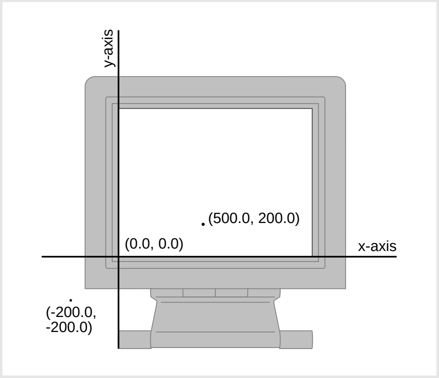
The screen's coordinate system is the basis for all other coordinate systems used for positioning, sizing, drawing, and event handling. You can think of the entire screen as occupying the upper-right quadrant of a two-dimensional coordinate grid. The other three quadrants, which are invisible to users, take negative values along their x-axis, their y-axis, or both axes. The screen's quadrant has its origin in the lower left corner; the positive x-axis extends horizontally to the right and the positive y-axis extends vertically upward. A unit along either axis is expressed as a pixel.

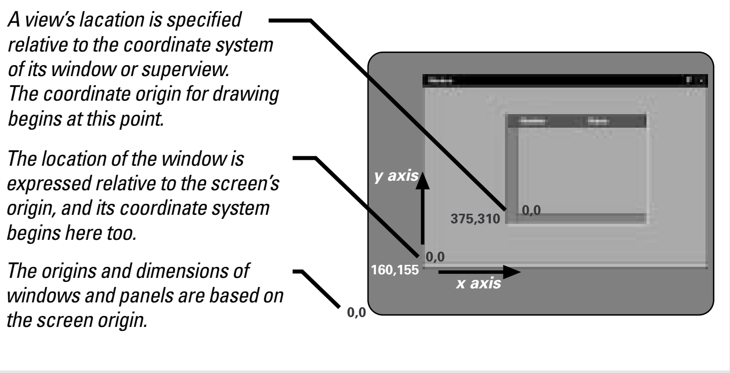
The screen coordinate system has just one function: to position windows on the screen. When your application creates a new window, it must specify the window's initial size and location in screen coordinates.You can "hide" windows by specifying their origin points well within one of the invisible quadrants. This technique is often used in off-screen rendering in buffered windows.

The reference coordinate system for a window is known as the *base* coordinate system. It differs from the screen coordinate system in only two ways:

- It applies only to a particular window; each window has its own base coordinate system.
- Its origin is at the lower left corner of the window, rather than the lower left corner of the screen. If the window moves, the origin and the entire coordinate system move with it.

For drawing, each NSView uses a coordinate system transformed from the base coordinate system or from the coordinate system of its superview. This coordinate system also has it origin point at the lower-left corner of the NSView, making it more convenient for drawing operations. NSView has several methods for converting between base and local coordinate systems. When you draw, coordinates are expressed in the application's *current* coordinate system, the system reflecting the last coordinate transformations to have taken place within the current window.




A view's lacation is specified relative to the coordinate system of its window or superview. The coordinate origin for drawing begins at this point.
The location of the window is expressed relative to the screen's origin, and its coordinate system begins here too.
The origins and dimensions of windows and panels are based on the screen origin.
y axis
375,310
0,0
0,0
160,155
x axis
0,0

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

Implement ToDoDoc's **initWithFile:** method.

```
- initWithFile:(NSString *)aFile
{
    NSEnumerator *dayenum;
    NSDate *itemDate;

    [super init];
    if (aFile) {                                               /* 1 */
        activeDays = [NSUnarchiver unarchiveObjectWithFile:aFile];
        if (activeDays)
            activeDays = [activeDays retain];
        else
            NSRunAlertPanel(@"To Do", @"Couldn't unarchive file %@",
                nil, nil, nil, aFile);
    } else {                                                   /* 2 */
        activeDays = [[NSMutableDictionary alloc] init];
        [self setCurrentItems:nil];
    }
    if (![NSBundle loadNibNamed:@"ToDoDoc.nib" owner:self] )    /* 3 */
          return nil;
    if (aFile)                                                 /* 4 */
        [[itemMatrix window] setTitleWithRepresentedFilename:aFile];
    else
        [[itemMatrix window] setTitle:@"UNTITLED"];
    [[itemMatrix window] makeKeyAndOrderFront:self];
    return self;
}
```

This method, which initializes and loads the document, has the following steps:

1. Restores the document's archived objects if the **aFile** argument is the pathname of a file containing the archived objects (that is, the document is opened). If objects are unarchived, it retains the **activeDays** dictionary; otherwise it displays an attention panel.

2. Initializes the **activeDays** and **currentItems** instance variables. A **aFile** argument with a **nil** value indicates that the user is requesting a new document.

3. Loads the nib file containing the document interface, specifying **self** as owner.

4. Sets the title of the window; this is either the file name on the left of the title bar and the pathname on the right, or "UNTITLED" if the document is new.

Before You Go On

Note the **[itemMatrix window]** message nested in the last message. Every object that inherits from NSView "knows" its window and will return that NSWindow object if you send it a **window** message.

9 **Implement the document-opening method.**

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

Write the code for **openDoc:**.

```
- (void)openDoc:(id)sender
{
    int result;
    NSString *selected, *startDir;
    NSArray *fileTypes = [NSArray arrayWithObject:@"td"];
    NSOpenPanel *oPanel = [NSOpenPanel openPanel];            /* 1 */

    [oPanel setAllowsMultipleSelection:YES];
    if ([[[NSApp keyWindow] delegate] isKindOfClass:[ToDoDoc class]])
        startDir = [[[NSApp keyWindow] representedFilename]   /* 2 */
            stringByDeletingLastPathComponent];
    else
        startDir = NSHomeDirectory();
    result = [oPanel runModalForDirectory:startDir file:nil   /* 3 */
        types:fileTypes];
    if (result == NSOKButton) {
        NSArray *filesToOpen = [oPanel filenames];
        int i, count = [filesToOpen count];
        for (i=0; i<count; i++) {                             /* 4 */
              NSString *aFile = [filesToOpen objectAtIndex:i];
              id currentDoc = [[ToDoDoc alloc] initWithFile:aFile];
              [currentDoc activateDoc];
        }
    }
}
```

The **openDoc:** method displays the modal Open panel, gets the user's response (which can be multiple selections) and opens the file (or files) selected.

1. Creates or gets the NSOpenPanel instance (an instance shared among objects of an application). The previous message specifies the file types (that is, the extensions) of the files that will appear in the Open panel browser. The next message enables selection of multiple file in the panel's browser.
2. Sets the directory at which the NSOpenPanel starts displaying files either to the directory of any document window currently key or , if there is none, to the user's home directory.
3. Runs the NSOpenPanel and obtains the key clicked.
4. If the key is NSOKButton, cycles through the selected files and, for each, creates a document by allocating and initializing a ToDoDoc instance, passing in a file name.

The methods invoked by the Document menu's Close and Save commands both simply send a message to another object. How they locate these objects exemplify important techniques using the core program framework.

10 **Write the code that closes documents.**

In **ToDoController.m**, implement the **closeDoc:** method.

```
- (void)closeDoc:(id)sender
{
    [[NSApp mainWindow] performClose:self];
}
```

NSApp, the global NSApplication instance, keeps track of the application's windows, including their status. Because only one window can have main status, the **mainWindow** message returns that NSWindow object— which is, of course, the one the user chose the Close command for. The **closeDoc:** method sends **performClose:** to that window to simulate a mouse click in the window's close button. (See the following section, "Managing Documents Through Delegation," to learn how the document handles this user event.)

11 **Write the code that saves documents.**

In **ToDoController.m**, implement the **saveDoc:** method.

```
- (void)saveDoc:(id)sender
{
    id currentDoc = [[NSApp mainWindow] delegate];
    if (currentDoc)
        [currentDoc saveDoc];
}
```

As did **closeDoc:**, this method sends **mainWindow** to NSApp to get the main window, but then it sends **delegate** to the returned window to get its delegate, the ToDoDoc instance that is managing the document. It then sends the ToDoDoc-defined message **saveDoc** to this instance.

Note: You could implement **closeDoc:** and **saveDoc:** in the ToDoDoc class, but the ToDoController approach was chosen to make the division of responsibility clearer.

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

Implement the **saveDoc:** method.

```
- (void)saveDoc
{
    NSString *fn;

    if (![[[itemMatrix window] title] hasPrefix:@"UNTITLED"]) {
        fn = [[itemMatrix window] representedFilename];        /* 1 */
    } else {
        int result;                                             /* 2 */
        NSSavePanel *sPanel = [NSSavePanel savePanel];
        [sPanel setRequiredFileType:@"td"];
       result = [sPanel runModalForDirectory:NSHomeDirectory() file:nil];
        if (result == NSOKButton) {
            fn = [sPanel filename];
            [[itemMatrix window] setTitleWithRepresentedFilename:fn];
        } else
            return;
    }

    if (![NSArchiver archiveRootObject:activeDays toFile:fn])  /* 3 */
        NSRunAlertPanel(@"To Do", @"Couldn't archive file %@",
            nil, nil, nil, fn);
    else
        [[itemMatrix window] setDocumentEdited:NO];
}
```

ToDoDoc's **saveDoc** method complements ToDoController's **openDoc:** method in that it runs the modal Save panel for users.

1. The **title** method returns the text that appears in the window's title bar. If the title doesn't begin with "UNTITLED" (what new document windows are initialized with), then a file name and directory location has already been chosen, and is stored as the **representedFilename**.

2. If the window title begins with "UNTITLED" then the document needs to be saved under a user-specified file name and directory location. This part of the code creates or gets the shared NSSavePanel instance and sets the file type, which is the extension that's automatically appended. Then it runs the Save panel, specifying the user's home directory as the starting location.

3. Archives the document under the chosen directory path and file name and, with the **setDocumentEdited:** message, changes the window's close button to an "unbroken X" image (more on this in the next section).

12 **Implement the accessor methods for ToDoController and ToDoDoc.**

Don't implement **setCurrentItems:** yet. This method does something special for the application that will be covered in "Managing the Data and Coordinating its Display (ToDoDoc)" on page 154.

The Application Quartet: NSResponder, NSApplication, NSWindow, and NSView

Many classes of the Application Kit stand out in terms of relative importance. NSControl, for example, is the superclass of all user-interface devices, NSText underlies all text operations, and NSMenu has obvious significance. But four classes are at the core of a running application: NSResponder, NSApplication, NSWindow, and NSView. Each of these classes plays a critical role in the two primary activities of an application: drawing the user interface and responding to events. The structure of their interaction is sometimes called the core program framework.

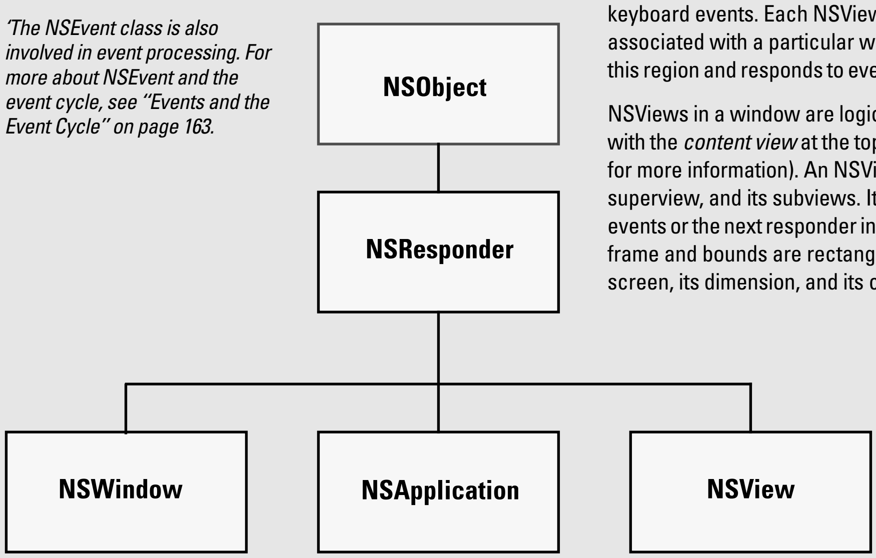
NSWindow

An NSWindow object manages each physical window (that is, each window created by the Window Server) on the screen. It draws the title bar and window frame and responds to user actions that close, move, resize, and otherwise manipulate the window.

The main purpose of an NSWindow is to display an application's user interface (or part of it) in its *content area*: that space below the title bar and within the window frame. A window's content is the NSViews it encloses, and at the root of this *view hierarchy* is the *content view*, which fills the content area. Based on the location of a user event, NSWindows assigns an NSView in its content area to act as *first responder*.

An NSWindow allows you to assign a custom object as its delegate and so participate in its activities.

'The NSEvent class is also involved in event processing. For more about NSEvent and the event cycle, see ''Events and the Event Cycle'' on page 163.



NSResponder

NSResponder is an abstract class, but it enables event handling in all classes that inherit from it. It defines the set of messages invoked when different mouse and keyboard events occur. It also defines the mechanics of event processing among objects in an application, especially the passing of events up the *responder chain* to each *next responder* until the event is handled. See the ''Events and the Event Cycle'' on page 163 for more on the responder chain and a description of *first responder.*

NSApplication

Every application must have one NSApplication object to act as its interface with the Window Server and to supervise and coordinate the overall behavior of the application. This object receives events from the Window Server and dispatches them to the appropriate NSWindows (which, in turn, distribute them to their NSViews). The NSApplication object manages its windows and detects and handles changes in their status as well as in its own status: hidden and unhidden, active and inactive. The NSApplication object is represented in each application by the global variable NSApp. To coordinate your own code with NSApp, you can assign your own custom object as its delegate.

NSView

Any object you see in a window's content area is an NSView. (Actually, since NSView is an abstract class, these objects are instances of NSView subclasses.) NSView objects are responsible for drawing and for responding to mouse and keyboard events. Each NSView owns a rectangular region associated with a particular window; it produces images within this region and responds to events occurring within the rectangle.

NSViews in a window are logically arranged in a *view hierarchy*, with the *content view* at the top of the hierarchy (see facing page for more information). An NSView references its window, its superview, and its subviews. It can be the first responder for events or the next responder in the responder chain. An NSView's frame and bounds are rectangles that define its location on the screen, its dimension, and its coordinate system for drawing.

The View Hierarchy

Just inside each window's content area—the area enclosed by the title bar and the other three sides of the frame—lies the content view. The content view is the root (or top) NSView in the window's view hierarchy. Conceptually like a tree, one or more NSViews may branch from the content view, one one or more other NSViews may branch from these subordinate NSViews, and so on. Except for the content view, each NSView has one (and only one) NSView above it in the hierarchy. An NSView's subordinate views are called its subviews; its superior view is known as the superview.

On the screen *enclosure* determines the relationship between superview and subview: a superview encloses its subviews. This relationship has several implications for drawing:

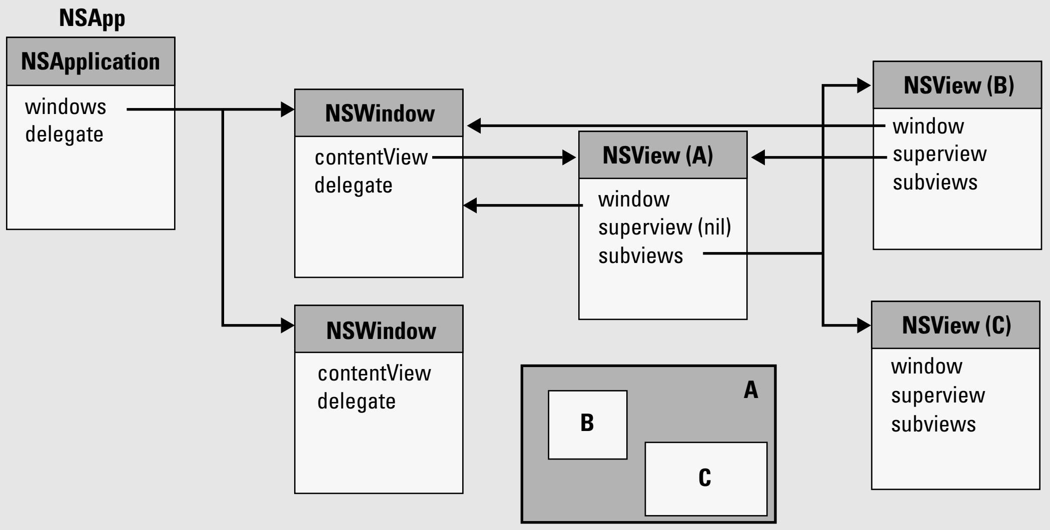
- It permits construction of a superview simply by arrangement of subviews. (An NSBrowser is an instance of a compound NSView.)
- Subviews are positioned in the coordinates of their superview, so when you move an NSView or transform its coordinate system, all subviews are moved and transformed in concert.
- Because an NSView has its own coordinate system for drawing, its drawing instructions remain constant regardless of any change in position in itself or of its superview.

Fitting Your Application In

The core program framework provides ways for your application to access the participating objects and so to enter into the action.

- The global variable NSApp identifies the NSApplication object. By sending the appropriate message to NSApp, you can obtain the application's NSWindow objects (**windows**), the key and main windows (**keyWindow** and **mainWindow**), the current event (**currentEvent**), the main menu (**mainMenu**), and the application's delegate (**delegate**).
- Once you've identified an NSWindow object, you can get its content view (by sending it **contentView**) and from that you can get all subviews of the window. By sending messages to the NSWindow object you can also get the current event (**currentEvent**), the current first responder (**firstResponder**), and the delegate (**delegate**).
- You can obtain from an NSView most objects it references. You can discover its **window**, its **superview**, and its **subviews**. Some NSView subclasses can also have delegates, which you can access with **delegate**.

By making your custom objects delegates of the NSApplication object, your application's NSWindows, and NSViews that have delegates, you can integrate your application into the core program framework and participate in what's going on.



Managing Documents Through Delegation

At certain points while an application is running you want to ensure that a document's data is preserved or that a document's edited status is tracked. These events occur when users:

- Edit a document.
- Close a window.
- Quit the application.
- Hide the application.
- Switch to another application or window.

Several classes of the Application Kit send messages to their delegates when these events occur, giving the delegate the opportunity to do the appropriate thing, whether that be saving a document to the file system or marking a document as edited.

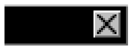
1 **Mark a document as edited.**

Open **ToDoDoc.m.**

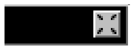
Implement the **controlTextDidChange:** method to mark the document.

```
- (void)controlTextDidChange:(NSNotification *)notif
{
    [[itemMatrix window] setDocumentEdited:YES];
}
```

When a control that contains editable text—such as a text field or a matrix of text fields—detects editing in a field, it posts the **controlTextDidChange:** notification which, like all notifications, is sent to the control's delegate as well as to all observers. The **setDocumentEdited:** message causes the document's window to change the image in its close button to a broken X.



```
[window setDocumentEdited:NO];
```



```
[window setDocumentEdited:YES];
```

Note: The ToDo object that, by notification, invokes the **controlTextDidChange:** method is **itemMatrix**, the matrix of to-do items (text fields). You will programmatically set ToDoDoc to be the delegate of this object later in this tutorial.

2 **Save edited documents when windows are closed.**

Implement the delegation method **windowShouldClose:**.

```
- (BOOL)windowShouldClose:(id)sender
{
    int result;
/* 1 */
   if (![[itemMatrix window] isDocumentEdited]) return YES;
/* 2 */
    [[itemMatrix window] makeFirstResponder:[itemMatrix window]];
    result = NSRunAlertPanel(@"Close", @"Document has been edited.
        Save changes before closing?", @"Save", @"Don't Save",
        @"Cancel");
/* 3 */
    switch(result) {
      case NSAlertDefaultReturn: {
          [self saveDocItems];
          [self saveDoc];
          return YES;
      }
        case NSAlertAlternateReturn: {
          return YES;
        }
        case NSAlertOtherReturn: {
          return NO;
        }
    }
    return NO;
}
```

When users click a window’s close button, the window sends **windowShouldClose:** to its delegate. It expects a response directing it either to close the window or leave it open.

1. Returns YES (meaning: go ahead, close the window) if the document hasn’t been edited.
2. Makes the window its own first responder. This has the effect of forcing the validation of cells, flushing currently entered text to the method that handles it (more on this in the next section).
3. Identifies the clicked button by evaluating the constant returned from **NSRunAlertPanel()** and returns the appropriate boolean value. If the user clicks the Save button, this method also updates internal storage with the currently displayed items (**saveDocItems**) and then sends **saveDoc** to itself to archive application data to a file. (**saveDocItems** is described in the following section.)

Note: Do you recall the **performClose:** method that ToDoController sends the document window when the user chooses the Close command? This method simulates a mouse click on the window's close button, causing **windowShouldClose:** to be invoked.

3 **Save edited documents when the user quits the application.**

In **ToDoController.m**, implement the delegation method **applicationShouldTerminate:**.

```
- (BOOL)applicationShouldTerminate:(id)sender
{
    while ([NSApp keyWindow]) {
        int result;
        id doc = [[NSApp keyWindow] delegate];

        if (![[NSApp keyWindow] isDocumentEdited]) {
            [[NSApp keyWindow] close];
            if (doc) [doc autorelease];
            continue;
        }
        if ([doc isKindOfClass:[ToDoDoc class]]) {
            NSString *repfile = [[NSApp keyWindow] representedFilename];
            result = NSRunAlertPanel(@"To Do", @"Save %@?", @"Save",
                @"Don't Save", @"Cancel",
                ([repfile isEqualToString:@""]?@"UNTITLED":repfile));
            switch(result) {
              case NSAlertDefaultReturn:
                [doc saveDocItems];
                [doc saveDoc];
                break;
              case NSAlertAlternateReturn:
                [[NSApp keyWindow] close];
                break;
              case NSAlertOtherReturn:
                return NO;
            }
            if (doc) [doc autorelease];
        }
        else
            [[NSApp keyWindow] close];
    }
    return YES;
}
```

NSApplication sends several message to its delegate. One of these messages—**applicationShouldTerminate:**—notifies the delegate that the application is about to terminate. The implementation of this method is similar to that for **windowShouldClose:**. What's different is that this method cycles through all windows of the application and, if the window is managed by ToDoDoc, puts up an attention panel and responds according to the user's choice.

Managing the Data and Coordinating its Display (ToDoDoc)

If you recall the discussion on To Do's design earlier in this chapter ("How To Do Stores and Accesses its Data" on page 119), you'll remember that the application's real data consists of instances of the model class, ToDoItem. To Do stores these objects in arrays and stores the arrays in a dictionary; it uses dates as the keys for accessing specific arrays. (Both the dictionary and its arrays are mutable, of course.) You might also recall that this design depends on a positional correspondence between the text fields of the document interface and the "slots" of the arrays.

To lend clarity to this design's implementation, this section follows the process from start to finish through which the ToDoDoc class handles entered data, and organizes, displays, and stores it. It also shows how the display and manipulation of data is driven by the selections made in the CalendarMatrix object.

Start by revisiting a portion of code you wrote earlier for ToDoDoc's **initWithFile:** method.

```
- initWithFile:(NSString *)aFile
{
    /* ... */
    if (aFile) {
        activeDays = [NSUnarchiver unarchiveObjectWithFile:aFile];
        if (activeDays)
            activeDays = [activeDays retain];
        else
            NSRunAlertPanel(@"To Do", @"Couldn't unarchive file %@",
                nil, nil, nil, aFile);
    } else {
        activeDays = [[NSMutableDictionary alloc] init];
        [self setCurrentItems:nil];
    }
/* ... */
}
```

Assume the user has chosen the New command from the Document menu. Since there is no archive file (**aFile** is **nil**), the **activeDays** dictionary is created but is left empty. Then **initWithFile:** invokes its own **setCurrentItems:** method, passing in **nil**.

1 **Set the current items or, if necessary, create and prepare the array that holds them.**

Implement **setCurrentItems:**.

```
- (void)setCurrentItems:(NSMutableArray *)newItems
{
    if (currentItems) [currentItems autorelease];

    if (newItems)
        currentItems = [newItems mutableCopy];
    else {
        int numRows = [[itemMatrix cells] count];
        currentItems = [[NSMutableArray alloc]
            initWithCapacity:numRows];
        while (--numRows >= 0)
            [currentItems addObject:@""];
    }
}
```

This "set" accessor method is like other such methods, except in how it handles a **nil** argument. In this case, **nil** signifies that the array does not exist, and so it must be created. Not only does **setCurrentItems:** create the array, but it "initializes" it with empty string objects. It does this because NSMutableArray's methods cannot tolerate **nil** objects within the bounds of the array.

So there's now a **currentItems** array ready to accept ToDoItems. Imagine yourself using the application. What are the user events that cause a ToDoItem to be added to the **currentItems** array? To Do allows entry of items "on the fly," and thus does not require the user to click a button to add a ToDoItem to the array. Specifically, items are added when users type something and then:

- Press the Tab key.
- Press the Enter key.
- Click outside the text field.

The **controlTextDidEndEditing:** delegation method makes these scenarios possible. The matrix of editable text fields (**itemMatrix**) invokes this method when the cursor leaves a text field that has been edited.

2 **As items are entered in the interface, add ToDoItems to internal storage, delete them, or modify them, as appropriate.**

Implement **controlTextDidEndEditing:**.

```
- (void)controlTextDidEndEditing:(NSNotification *)notif
{
    id curItem, newItem;
    int row = [itemMatrix selectedRow];
    NSString *selName = [[itemMatrix selectedCell] stringValue];
/* 1 */
    if (![[itemMatrix window] isDocumentEdited] ||
        (row >= [currentItems count])) return;
    if (!currentItems)
        [self setCurrentItems:nil];
/* 2 */
    if ([selName isEqualToString:@""] &&
        ([[currentItems objectAtIndex:row] isKindOfClass:
            [ToDoItem class]]) &&
        (![[[currentItems objectAtIndex:row] itemName]
            isEqualToString:@""]))
        [currentItems replaceObjectAtIndex:row withObject:@""];
/* 3 */
    else if ([[currentItems objectAtIndex:row] isKindOfClass:
            [ToDoItem class]] &&
            (![[[currentItems objectAtIndex:row] itemName]
            isEqualToString:selName]) )
        [[currentItems objectAtIndex:row] setItemName:selName];
/* 4 */
   else if (![selName isEqualToString:@""]) {
        newItem = [[ToDoItem alloc] initWithName:selName
            andDate:[calendar selectedDay]];
        [currentItems replaceObjectAtIndex:row withObject:newItem];
        [newItem release];
    }
/* 5 */
    [self updateMatrix];
}
```

A control sends **controlTextDidEndEditing:** to its delegate when the cursor *leaves* a text field. In addition to creating new ToDoItems, this implementation of **controlTextDidEndEditing:** removes ToDoItems from arrays and modifies item text. What it does is appropriate to what the user does.

1. If the document hasn't been edited (see **controlTextDidChange:**) or if the selected row exceeds the array bounds, it returns because there's no reason to proceed. It initializes a **currentItems** array if one doesn't exist.

2. If the user deletes the text of an existing item, it removes the ToDoItem that positionally corresponds to the row of that deleted text.

3. It changes the name of an item if the text entered in a field doesn't match the name of the corresponding item in the **currentItems** array.

4. If either of the two previous conditions don't apply, and text has been entered, it creates a new ToDoItem and inserts it in the **currentItems** array.

5. Updates the list of items in the document interface.

3 **Update the document interface with the current items.**

Implement **updateMatrix:**.

```
- (void)updateMatrix
{
    int i, cnt = [currentItems count], rows = [[itemMatrix cells] count];
    ToDoItem *thisItem;

    for (i=0; i<cnt, i<rows; i++) {
        NSDate *due;
        thisItem = [currentItems objectAtIndex:i];
        if ([thisItem isKindOfClass:[ToDoItem class]]) {        /* 1 */
            if ( [thisItem secsUntilDue] )
                due = [[thisItem day] addTimeInterval:
                          [thisItem secsUntilDue]];
            else
                due = nil;
            [[itemMatrix cellAtRow:i column:0] setStringValue:
                [thisItem itemName]];
            [[markMatrix cellAtRow:i column:0] setTimeDue:due];
            [[markMatrix cellAtRow:i column:0] setTriState:
                [thisItem itemStatus]];
        }
        else  {                                                /* 2 */
            [[itemMatrix cellAtRow:i column:0] setStringValue:@""];
            [[markMatrix cellAtRow:i column:0] setTitle:@""];
            [[markMatrix cellAtRow:i column:0] setImage:nil];
        }
    }
}
```

The **updateMatrix** method writes the names of the items (ToDoItems) in the **currentItems** array to the text fields of **itemMatrix**. It also updates the visual appearance of the cells in the matrix (**markMatrix**) next to **itemMatrix**. These cells are instances of a custom subclass of NSButtonCell that you will create later in this tutorial. For now, just type all the code above; later, when you create the cell class, ToDoCell, you can refer back to this example to see what is happening.

Basically, this method cycles through the array of items, doing the following:

1. If an object in the array is a ToDoItem, it writes the item name to the text field corresponding to the array slot and updates the button cell next to the field.

2. If an object isn't a ToDoItem, it blanks the corresponding text field and cell.

4 **Respond to user actions in the calendar.**

Implement CalendarMatrix's delegation methods.

```
- (void)calendarMatrix:(CalendarMatrix *)matrix         /* 1 */
        didChangeToDate:(NSDate *)date
{
    [[itemMatrix window] makeFirstResponder:[itemMatrix window]];
    [self saveDocItems];

    [self setCurrentItems:[activeDays objectForKey:date]];
    [dayLabel setStringValue:[date descriptionWithCalendarFormat:
        @"To Do on %a %B %d %Y" timeZone:[NSTimeZone defaultTimeZone]
        locale:nil]];
    [self updateMatrix];
}

- (void)calendarMatrix:(CalendarMatrix *)matrix         /* 2 */
        didChangeToMonth:(int)mo year:(int)yr
{
    [self saveDocItems];
    [self setCurrentItems:nil];
    [self updateMatrix];
}
```

As you recall, CalendarMatrix declared two methods to allow delegates to "hook into" its behavior. Its delegate for this application is ToDoDoc.

1. The calendar sends **calendarMatrix:didChangeToDate:** when users click a new day of the month. This implementation saves the current items to the **activeDays** dictionary. It then sets the current items to be those corresponding to the selected date (if there are no items for that date, the **objectForKey:** message returns **nil** and the **currentItems** array is initialized with empty strings). Finally it updates the matrix with the new data.

2. The calendar sends **calendarMatrix:didChangeToMonth:year:** when users go to a new month and (possibly) a new year. This implementation responds by saving the current items to internal storage and presenting a blank list of items.

5 **Save the data to internal storage.**

Implement **saveDocItems:**.

6 **Archive and unarchive the document's data.**

Implement **encodeWithCoder:** and **initWithCoder:** to archive and unarchive the dictionary holding the arrays of ToDoItems.

```
- (void)saveDocItems
{
    ToDoItem *anItem;
    int i, cnt = [currentItems count];
    // save day's current items (array) to document dictionary
    for (i=0; i<cnt; i++) {
        if ( (anItem = [currentItems objectAtIndex:i])  &&
             ([anItem isKindOfClass:[ToDoItem class]]) ) {
            [activeDays setObject:currentItems forKey:
                [anItem day]];
            break;
        }
    }
}
```

This method inspects the **currentItems** array and, if it contains at least one ToDoItem, puts the array in the **activeDays** dictionary with a key corresponding to the date.

Now that you've completed the methods for saving and archiving the collection objects holding ToDoItems, assume that the user has saved his document and then opens it.

7 **Perform set-up tasks when the document's nib file is unarchived.**

Implement **awakeFromNib** as shown at right.

```
- (void)awakeFromNib
{
    int i;
    NSDate *date;

    date = [calendar selectedDay];
    [self setCurrentItems:[activeDays objectForKey:date]];
    /* set up self as delegates */
    [[itemMatrix window] setDelegate:self];
    [itemMatrix setDelegate:self];
    [[itemMatrix window] makeKeyAndOrderFront:self];
}
```

When the **ToDoDoc.nib** file is completely unarchived, **awakeFromNib** is invoked. It sets the current items for today, sets a couple of delegates, and puts the document window in front of all other windows.

Note: This method sets some delegates programmatically, which is redundant since you set these delegates in Interface Builder. However, this code demonstrates the programmatic route—and no harm done.

8 **Set up the document once it's created or opened.**

Implement **activateDoc** as shown at right.

```
- (void)activateDoc
{
    if ([currentItems count]) [self updateMatrix];
    [dayLabel setStringValue:[[calendar selectedDay]
         descriptionWithCalendarFormat:@"To Do on %a %B %d %Y"
         timeZone:[NSTimeZone defaultTimeZone] locale:nil]];
}
```

The **activateDoc** method is invoked right after a ToDo document is created or opened. It starts the ball rolling by updating the list matrices of the document and writing the current date to the "To Do on <*date*>" label.

Subclass Example: Overriding Behavior (SelectionNotifMatrix)

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 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.

2 **Add declarations to the header file.**

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

extern NSString *SelectionInMatrixNotification = /* 1 */
       @"SelectionInMatrixNotification";

@interface SelectionNotifMatrix : NSMatrix
{
}

- (void)mouseDown:(NSEvent *)theEvent; /* 2 */

@end
```

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

2. Declares **mouseDown:**, the method implemented by the superclass that SelectionNotifMatrix overrides.

3 **Override mouseDown:**

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

```
- (void)mouseDown:(NSEvent *)theEvent
{
    int row;
    [super mouseDown:theEvent];                              /* 1 */

    row = [self selectedRow];                                /* 2 */
    if (row != -1) {
        [[NSNotificationCenter defaultCenter]
           postNotificationName:@"SelectionInMatrixNotification"
           object:self userInfo:[NSDictionary dictionaryWithObjectsAndKeys:
           [NSNumber numberWithInt:row], @"ItemIndex", nil]];
    }
}
```

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

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

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

4 **Replace the class of the matrix object.**

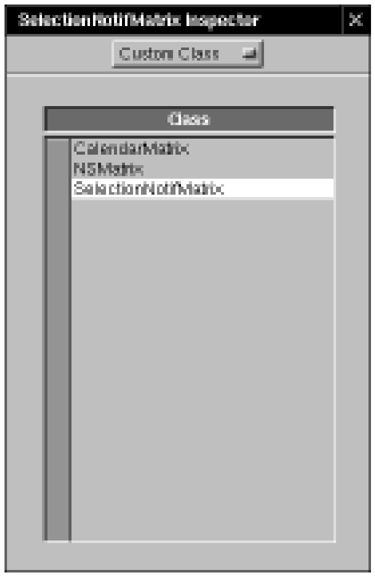
In Interface Builder:

Open **ToDoDoc.nib**.

Select the matrix of editable text cells.

Open the inspector and choose Custom Class from the pop-up menu.

Select SelectionNotifMatrix in the browser of compatible classes.



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

Events and the Event Cycle

You can depict the interaction between a user and an OpenStep 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 OpenStep 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 facing 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 reports it 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, Alternate, and Control), the codes identifying 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 five categories:

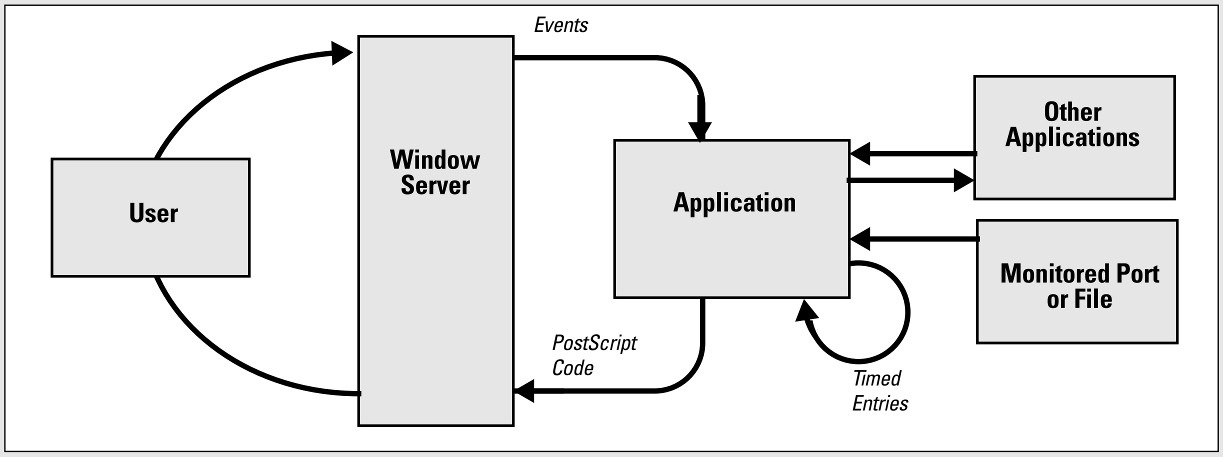
- **Keyboard events** Generated when a key is pressed down, a pressed key is released, or a modifier key changes. Of these, key-down events are the most useful. When you handle a key-down event, you often determine the character or characters associated with the event by sending the NSEvent a **characters** message.

- **Mouse events** 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.)

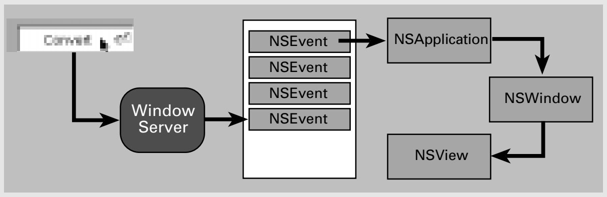
- **Cursor-update events** An cursor-update event is generated when the cursor has crossed the boundary of a predefined rectangular area.



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 (see page 116). 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 below). 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; these include cursor-update, and application-activate and -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 149). 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.

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.**

2 **Create the inspector panel.**

Drag a panel object from the Windows palette.

Make the title of the panel "Inspector."

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

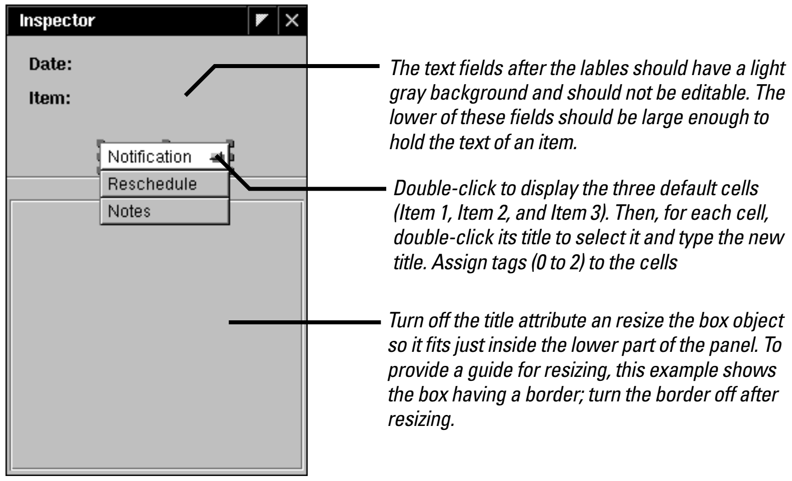
Put labels and fields on the panel and set their attributes (as shown).

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.



The text fields after the lables should have a light gray background and should not be editable. The lower of these fields should be large enough to hold the text of an item.

Double-click to display the three default cells (Item 1, Item 2, and Item 3). Then, for each cell, double-click its title to select it and type the new title. Assign tags (0 to 2) to the cells

Turn off the title attribute an resize the box object so it fits just inside the lower part of the panel. To provide a guide for resizing, this example shows the box having a border; turn the border off after resizing.

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.

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

Drag a panel object from the Windows palette.

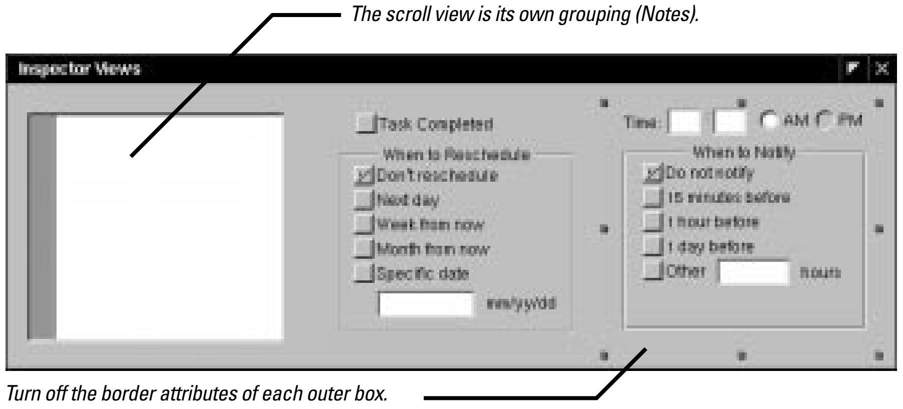
Resize the panel, using the example at right as a guide.

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).

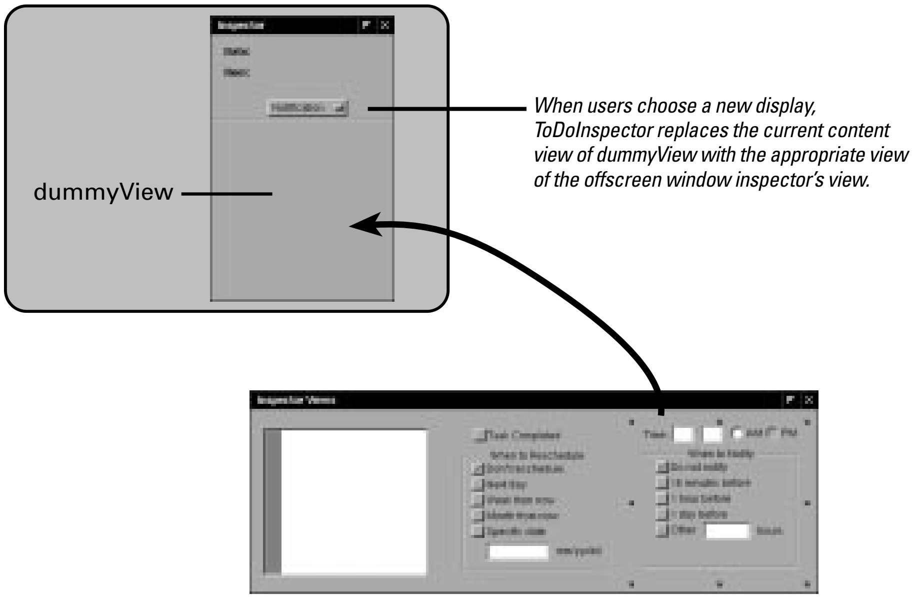
Make three other groupings for the three displays: Notes, Reschedule, and Notification.

Resize the resulting boxes to the same dimensions as the "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.


Inspector
Notification
dummyView
When users choose a new display, ToDoInspector replaces the current content view of dummyView with the appropriate view of the offscreen window inspector's view.
Inspector Views

4 **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

5 **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.

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 "set" accessor method probably seems familiar to you—except 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 hour, minute, and PM boolean values and then setting the appropriate fields and button matrix with these values.
4. 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 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. 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 "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.

3. 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.

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 “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.

2. 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.)

3. 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.

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 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

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

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 below).

- 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.

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 189 to see how this is done.

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.

13 **Format and validate the contents of inspector text fields.**

*In **ToDoInspector.m**:*

Implement **awakeFromNib** as shown at right.

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 "housekeeping" 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) "12/25/96." 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 "Notification" 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 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.

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:

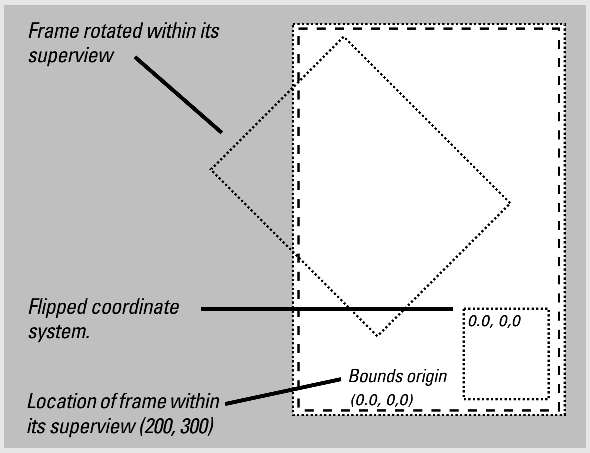
- Users scroll it (assuming it supports scrolling).
- Users resize or expose the NSView's window.
- The window receives a **display** message or is automatically updated.
- For some Application Kit objects, when an attribute changes.

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, scale, and rotate the NSView by reference to its frame (**setFrameOrigin:**, **setFrameSize:**, and so on).


Frame rotated within its superview
Flipped coordinate system.
0.0, 0,0
Bounds origin (0.0, 0,0)
Location of frame within its superview (200, 300)

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 OpenStep, 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 OpenStep-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. **pswrap** is a program (in **/usr/bin**) that 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** function draws grid lines:

```
defineps DrawGrid(float width, height, every)
    5 6 div setgray
    0 every width {
        0 moveto 0 height rlineto stroke
    } for
    0 every height {
        0 exch moveto width 0 rlineto stroke
   } for
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:

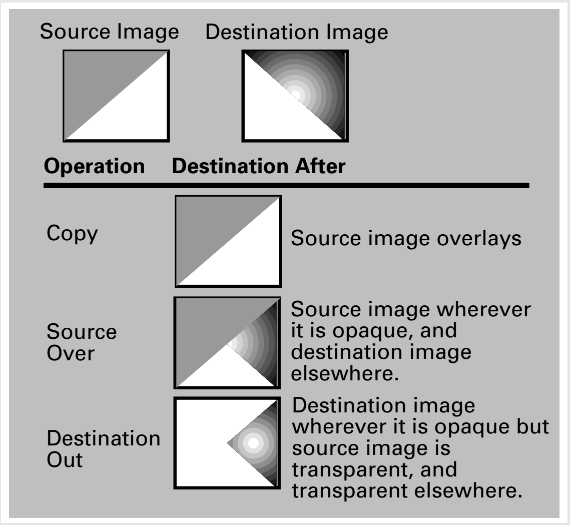
```
DrawGrid(5.0, 5.0, 1.0);
```

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.



Source image overlays

Source image wherever it is opaque, and destination image elsewhere.

Destination image wherever it is opaque but source image is transparent, and transparent elsewhere.

You can achieve interesting effects with compositing when the initial images are drawn with partially transparent paint. (Transparency is specified by *coverage*, a PostScript 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.

Making a Custom View

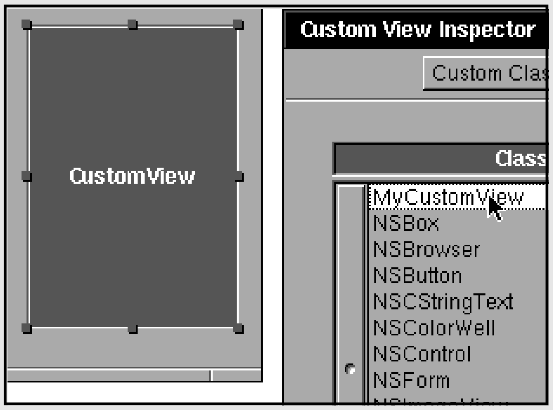
If you want an object that draws itself differently than any other Application Kit object, or responds to events in a special way, you should make a custom subclass of NSView. Your custom subclass should complete at least the steps outlined below.

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). You might want to **display** the custom view at this point.

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:

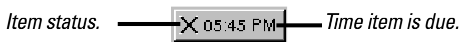
- Composite an NSImage.
- Call Application Kit functions such as **NSRectFill()** and **NSFrameRect ()** (NSGraphics.h).
- Call C functions that correspond to single PostScript operations, such as **PSsetgray()** and **PSfill()**.
- Call custom drawing functions created with **pswrap**.

See "A Short Guide to Drawing and Compositing" on page 179 for more information on drawing techniques and requirements.

Subclass Example: Overriding and Adding Behavior (ToDoCell)

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: notDone (no image), done (an "X"), and deferred (a check mark). These states correspond to the possible statues 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.



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 Chose 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 an OpenStep 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.**

Chose 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.

2 **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.

3 **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];                         /* 1 */
    [self setImagePosition:NSImageLeft];
    [self setBezeled:YES];
    [self setFont:[NSFont userFontOfSize:12]];
    [self setAlignment:NSRightTextAlignment];
                                                           /* 2 */
    path = [[NSBundle mainBundle] pathForImageResource:@"X.tiff"];
    doneImage = [[NSImage alloc] initByReferencingFile:path];
    path = [[NSBundle mainBundle]
        pathForImageResource:@"checkMark.tiff"];
    deferredImage = [[NSImage alloc] initByReferencingFile:path];

    return self;
}
```

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

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

4 **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           /* 1 */
{
    if (newState == deferred+1)
        triState = notDone;
    else
        triState = newState;
    [self _setImage:triState];
 }

- (ToDoButtonState)triState {return triState;}

- (void)setState:(int)val                               /* 2 */
{
}

- (int)state                                            /* 3 */
{
    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.

1. 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.

2. 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.

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

5 **Set the cell image.**

Declare the private method **_setImage:**.

Implement the **_setImage:** method.

```
@interface ToDoCell (PrivateMethods)
- (void)_setImage:(ToDoButtonState)aState;                /* 1 */
@end
/* ... */
- (void)_setImage:(ToDoButtonState)aState
{
    switch(aState) {                                       /* 2 */
       case notDone: {
           [self setImage:nil];
           break;
       }
       case done: {
           [self setImage:doneImage];
           break;
       }
       case deferred: {
           [self setImage:deferredImage];
           break;
       }
     }
     [(NSControl *)[self controlView] updateCell:self];    /* 3 */
}
```

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

1. In a category of ToDoCell in **ToDoCell.m**, it declares the private method **_setImage:**. Private methods, which by convention begin with an underscore, are methods that you don't want clients of your object to invoke. In this case, you don't want the image to be set independently from the cell's **triState** value.
2. In a switch statement, evaluates the tri-state argument and sets the cell's image appropriately (**setImage:** is an NSButtonCell method).
3. Sends **updateCell:** to the control view of the cell's control (a matrix) to force a re-draw of the cell.

6 **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:
  (NSView *)controlView
{
    return YES;
}

- (void)stopTracking:(NSPoint)lastPoint at:(NSPoint)stopPoint
  inView:(NSView *)controlView mouseIsUp:(BOOL)flag
{
    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**.

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

7 **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 NSDate's **descriptionWithCalendarFormat:timeZone:locale:** method to interpret and format the date object, then displays 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.

8 **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.)

9 **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 for Notification Messages

The To Do application includes as a feature 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.

2 **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]) {                        /* 1 */
        notifDate = [[anItem day] addTimeInterval:[anItem
            secsUntilNotif]];
        aTimer = [NSTimer scheduledTimerWithTimeInterval:  /* 2 */
                   [notifDate timeIntervalSinceNow]
            target:self
          selector:@selector(itemTimerFired:)
          userInfo:anItem
           repeats:NO];
        [anItem setItemTimer:aTimer];
    } else
        [[anItem itemTimer] invalidate];                 /* 3 */
}
```

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

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

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

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

3 **Respond to timers firing.**

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

```
- (void)itemTimerFired:(id)timer
{
    id anItem = [timer userInfo];
    ToDoInspector *inspController = [[[NSApp delegate]     /* 1 */
        inspector] delegate];
    NSDate *dueDate = [[anItem day] addTimeInterval:       /* 2 */
        [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).

1. 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.

2. 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

Implement resetNotifSwitch: 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.

4 **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.

5 **When the application is launched, reset item timers.**

Add the code at right, below, 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 Timer

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 have wrought. 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 pane 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.