To Do Tutorial - The Basics Chapter 4

4

What You'll Learn

Designing a multi-document application

Managing documents

Extending an Application Kit class

Loading code and resources dynamically

Opening and saving files

Manipulating times and dates

Reading and setting user defaults

The core program framework

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

Chapter 4
A Multi-Document Application

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. 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.

This chapter guides you through the steps needed to make To Do a multi-document application. When you finish this tutorial, the completed application will allow users to go to specific dates on a calendar and enter a list appointments or tasks for a particular days.

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. The diagram at the bottom of this page 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.

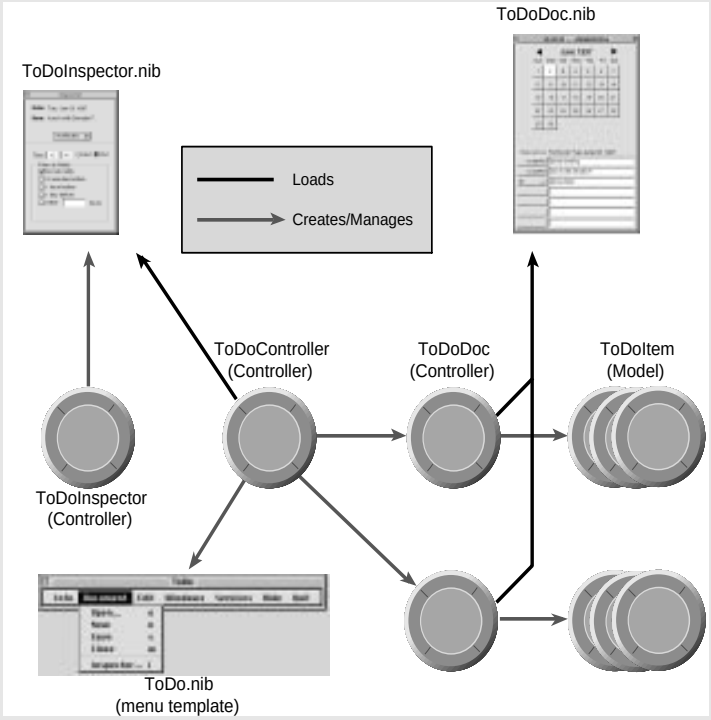
To Do's Multi-Document Design

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.

The File menu, which Interface Builder includes by default on the menu bar, contains the commands that multi-document applications typically need. When users choose New from the File 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 File menu, a Save dialog box 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 dialog box to be displayed.

The controller objects of To Do respond to a variety of delegation messages sent when certain events occur—primarily from windows and the application object (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 or notification to the other controller to keep it informed.

The ToDoInspector instance in this diagram takes on some of the work that the application controller, ToDoController, could do. 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 troubleshooting easier. See "Object-Oriented Programming" in the appendix for more on this.

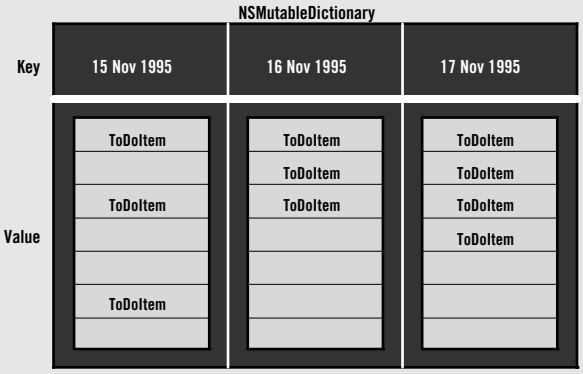


How To Do Stores and Accesses its Data

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 into 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.

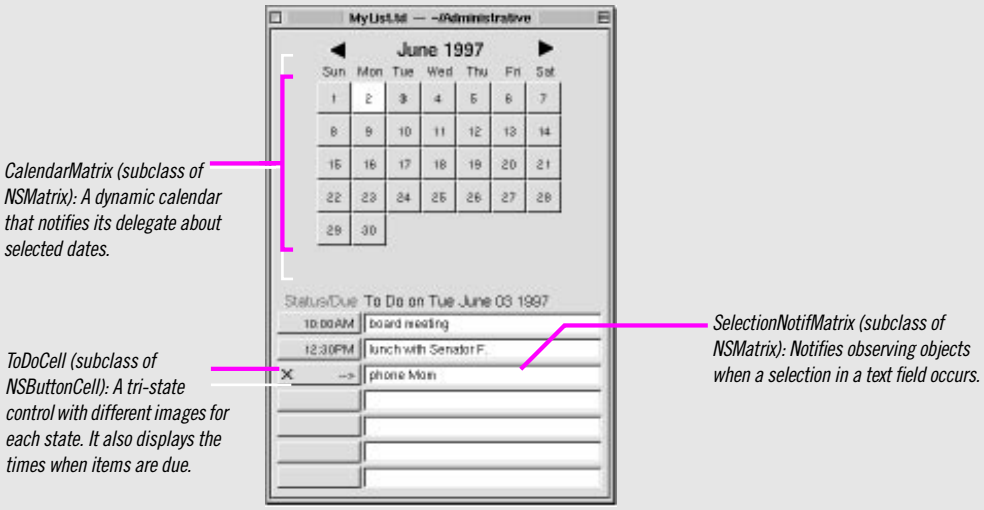
When users select a day in the calendar, the application computes the date, which it then uses as the key to locate an array of ToDoItems in the dictionary.



For further discussion of the architecture of multi-document applications, see page 139.

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 final interface features objects from three custom Application Kit subclasses. (You'll create only CalendarMatrix in this chapter.)



Setting Up the To Do Project

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.

1 Create the application project.

Start Project Builder.

Choose New from the Project menu.

Set the project type to Application.

Name the application "ToDo."

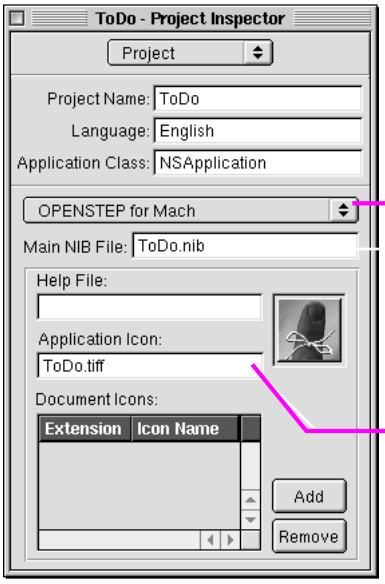
Click OK.

1 Add the application icon.

In the Project Attributes display of the project inspector, drag the application icon (**ToDo.tiff**) into the icon well.

Confirm that you want the image added to the project.

(The icon is in the ToDo project in **/System/Developer/Examples/AppKit**.)



You can have different icons and other project attributes for Rhapsody and Yellow Box for Windows.

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

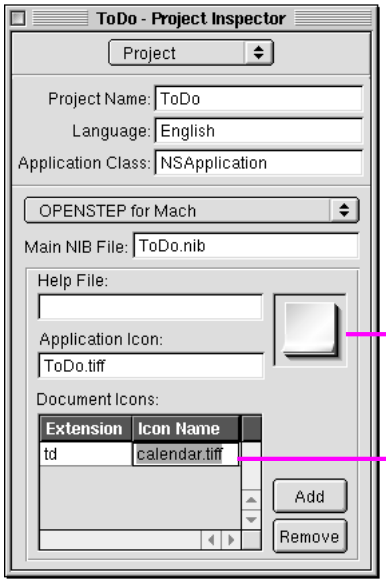
1 Specify the To Do document type.

Click Add.

Double-click the new cell under the Extension column.

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

Drag into the icon well the file **calendar.tiff** from the ToDo project in **/System/Developer/Examples/AppKit**.



Document types specify the kinds of files the application can open and "understand." Documents appear in the desktop with the assigned icon. Double-clicking the icon opens the document.

As with the application icon, when you drag the document icon into the image well (with the document row selected in Document Icons), 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.

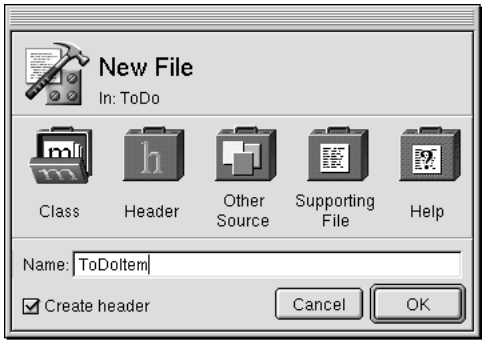
Select Classes in the project browser.

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.



Setting Up the Programmatic Interface

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

1 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"

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;
```

1 **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.

1 **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).

Before You Go On

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

Specifying Basic Object Behavior

Most of the method declarations of this class are for accessor methods. You know from past experience what you must do to implement them.

1 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 Timers" on page 198 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.

Starting Up — What Happens in NSApplicationMain()

Every Rhapsody 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 Manager or Explorer, **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 Rhapsody 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 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.

1 Implement copying and comparing object behavior.

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.

Implement the **copyWithZone:** method.

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

    return newobj;
}
```

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 and initializing it with instance variables held by **self**. Copying is often implemented for *value* objects—objects that represent attributes such as numbers, dates, and to-do items.

> 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.

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.

1 Have the object describe itself during debugging.

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);
}
```

1 Implement ToDoItem's initialization 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.

1 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.

1 Implement ToDoItem's time-conversion functions.

```
long ConvertTimeToSeconds(int hr, int min, BOOL flag)                    A
{
    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)             B
{
    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;
}
```

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

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

Breaktime!

Take a break from coding and build the project as it now stands. Go get a coffee, soda, or other beverage while the project is building. When you return, fix any errors that have insinuated themselves into the code. You can stop and build at anytime— a good thing to do because it will help you locate mistakes more easily.

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 application's menus 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 (or About) 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 your application and referenced in your nib file.

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 (a folder) that stores code and the resources that go with that code, including images and archived objects. A bundle is also identified as an instance of NSBundle; this object makes the contents of the bundle available to other objects that request it.

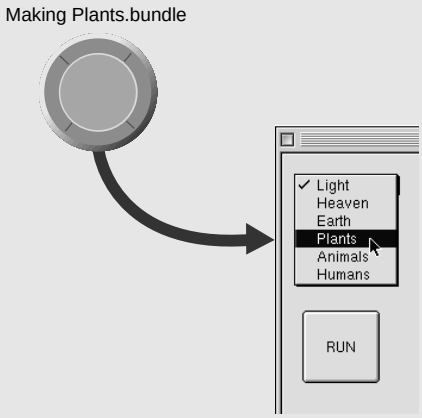
The generic notion of bundles is pervasive throughout Rhapsody. Applications are bundles, as are frameworks and palettes. Every application has at least one bundle—its main bundle—which is the ".app" directory (or *application wrapper*) where its executable file is located. This file is loaded into memory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

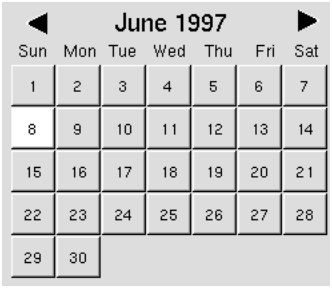
Loadable bundles differ from nib files in that they don't require you to use Interface Builder to build them. Instead of containing mostly archived objects, they usually contain mostly code. Loadable bundles are especially useful for incorporating extra behavior into an application upon demand. An economic-forecast 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.



Extending an Application Kit Class: An Example

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-cell 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 subclass 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 as Superclass?

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.

Composing the Interface

1 Define the CalendarMatrix class in Interface Builder.

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

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

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

Respond Yes when asked if you want the nib file added to the project.

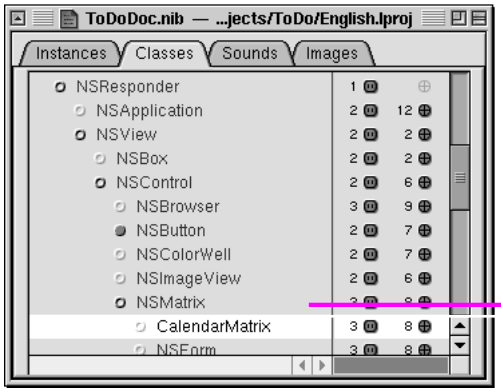
In the Classes display of the nib file window, select NSMatrix.

Choose Subclass from the Classes menu.

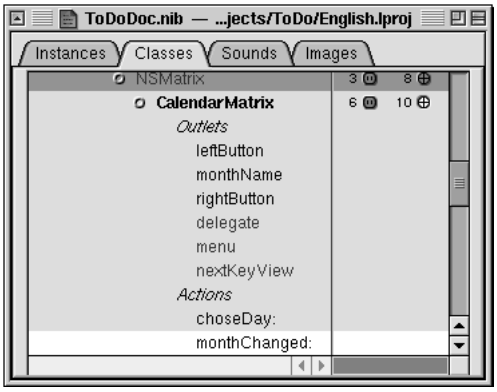
Name the new class “CalendarMatrix”.

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 by its superclasses) 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.

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

Drag a window from the Windows palette.

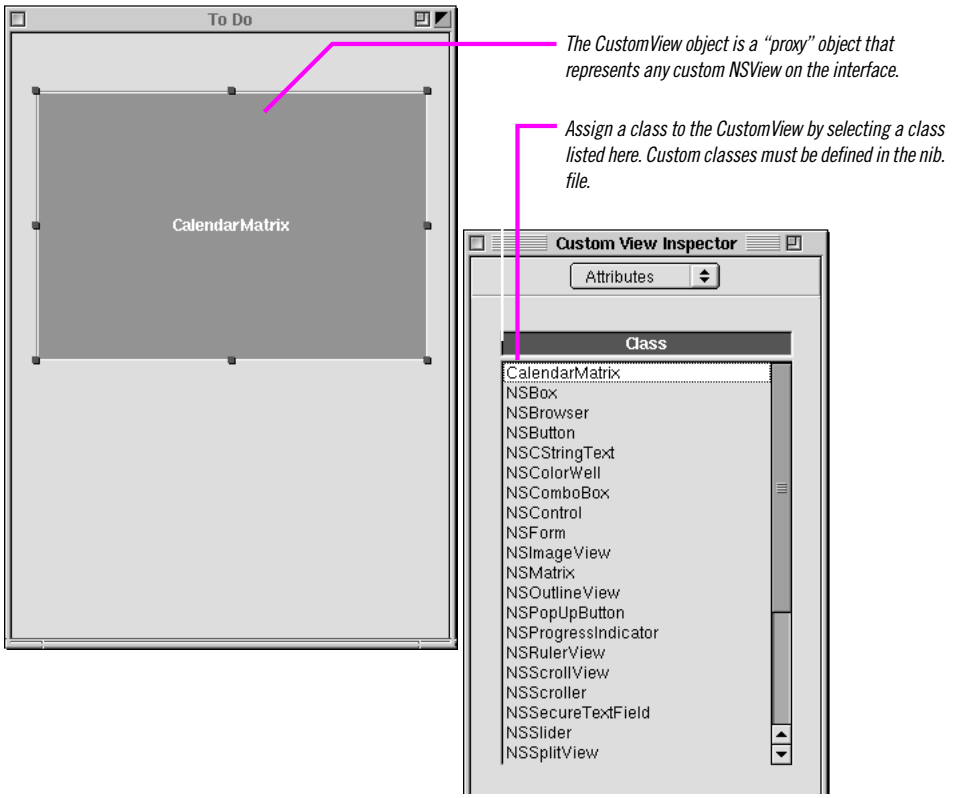
Resize the window, using the example at right as a guide.

Turn off the window's resize bar.

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 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.

1 **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.

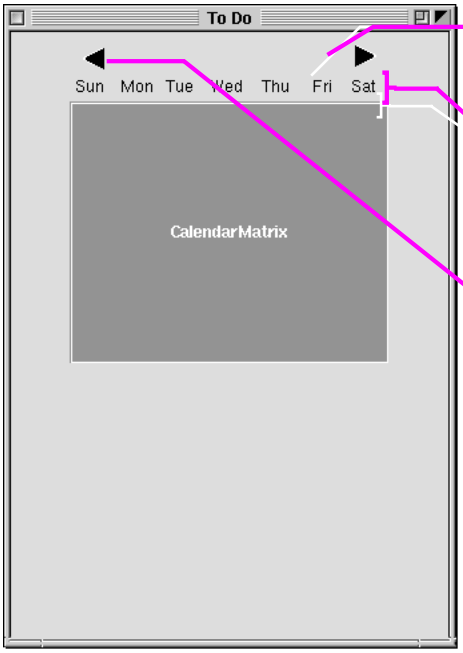
Make a small label for each day of the week.

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

Drag left_arrow.tiff from the ToDo project in **/System/Developer/Examples/AppKit** 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 empty label will display the month and year. Initialize it by typing "September 9999" (the longest possible string). Set the text to Helvetica 18 points, center it, and then delete the text.

Type the days of the week as individual labels, arrange them as a row, and then center the labels over the columns of days. (This latter task could take some trial and error.)

To have the button surround the image as tightly as possible, select the button and choose Format ► Size ► Size To Fit.

Next connect CalendarMatrix to its satellite objects.

1 **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

1 **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.

1 Add declarations to the header file CalendarMatrix.h.

(Existing declarations are indicated by ellipsis.)

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

@interface NSObject(CalendarMatrixDelegate)                     B
 - (void)calendarMatrix:(CalendarMatrix *)object
       didChangeToDate:(NSDate *)date;
 - (void)calendarMatrix:(CalendarMatrix *)object
       didChangeToMonth:(int)month year:(int)year;
@end
```

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

A 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.

B 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.

Defining the New Behavior

1 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];                    A

    [super initWithFrame:frameRect                                  B
                    mode:NSRadioModeMatrix
               prototype:cell
            numberOfRows:6
         numberOfColumns:7];
    // set cell tags                                                C
    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]                                     D
          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.

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

B 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.

C 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.

D 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 a 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 Rhapsody

In Rhapsody 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.

NSCalendarDate 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.

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

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.

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

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

C 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.

D 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.

1 Specify the behavior that occurs when users select a new month.

Implement the **monthChanged:** action method.

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

    if (sender == rightButton) {                                    A
        if (currentMonth == 12) {
            currentMonth = 1;
            currentYear++;
        } else {
            currentMonth++;
        }
    } else {
        if (currentMonth == 1) {
            currentMonth = 12;
            currentYear--;
        } else {
            currentMonth--;
        }
    }                                                               B
    [self setSelectedDay:[NSCalendarDate dateWithYear:currentYear
                   month:currentMonth
                     day:1 hour:0 minute:0 second:0
                timeZone:[NSTimeZone localTimeZone]]];
    [self refreshCalendar];
    [[self delegate] calendarMatrix:self                            C
        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.

A 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.

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

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

1 Specify the behavior that occurs when users select a day on the calendar.

Implement the **choseDay:** action method.

```
- (void)choseDay:sender
{
    NSCalendarDate *selDate, *thisDate = [self selectedDay];
    unsigned int selDay = [[self selectedCell] tag]-startOffset+1;    A

    selDate = [NSCalendarDate dateWithYear:[thisDate yearOfCommonEra]  B
                                     month:[thisDate monthOfYear]
                                       day:selDay
                                      hour:0
                                    minute:0
                                    second:0
                                  timeZone:[NSTimeZone localTimeZone]];
    [[self cellWithTag:[thisDate dayOfMonth]+startOffset-1]            C
        setFont:[NSFont systemFontOfSize:12]];
    [[self cellWithTag:selDay+startOffset-1] setFont:
        [NSFont boldSystemFontOfSize:12]];

    [self setSelectedDay:selDate];                                     D
    [[self delegate] calendarMatrix:self didChangeToDate:selDate];
}
```

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

A 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.

B Derives an NSCalendarDate that represents the selected date.

C 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.

D Sets the **selectedDay** instance variable to the new date and sends the **calendarMatrix:didChangeToDate:** message to the delegate.

1 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 139, 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 menu.

In Interface Builder:

Open **ToDo.nib**.

Rename the File menu "Document"

Remove unused menu items.

Put a generic menu item ("Item") in the Document menu and rename it "Inspector".

Make sure separator lines are above and below the Inspector command.

Give the Inspector command the key equivalent of Command-i.

Remove all file- or document-related commands from the File menu except for the ones shown here.

The three dots after Inspector indicate that the command displays a modal panel.

Interface Builder gives each new Rhapsody application the following default menus: Apple, File, Edit, Window, Services, and Help. The Windows menu lists windows of the application that are open and allows you to bring them to the top window tier. The Services menu lists other Rhapsody applications on a system and allows you to pass data to, or get data from, those applications.

Note: Disable the Preferences command in the Apple menu. This tutorial does not specifically cover Preferences panels, but it does give you enough information so that you can implement Preferences on your own.

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 choose the Save command, the application displays a Save panel and enables users to save the document in the file system. When users choose 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 the delegate of the application (NSApp) and the document controller the delegate of each 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.

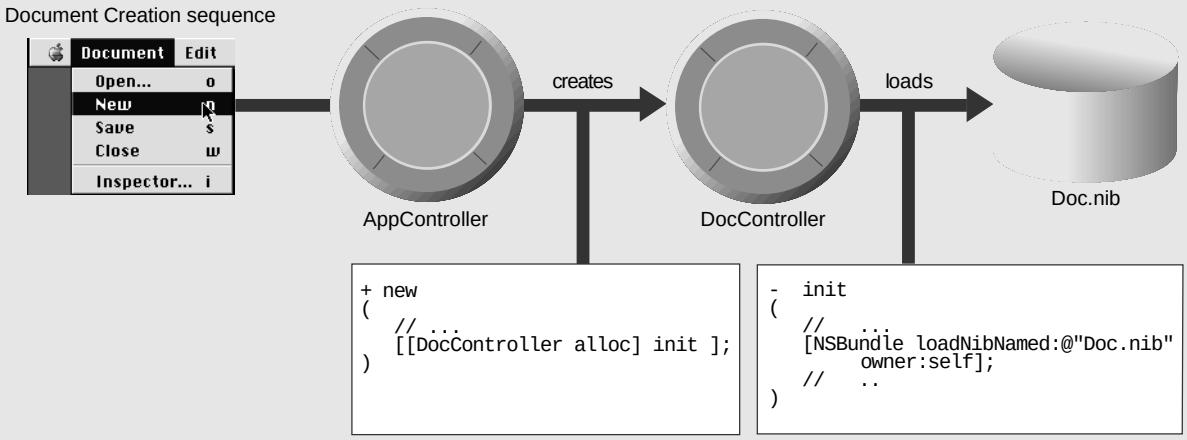
Several NSViews also have delegation messages that facilitate document management, particularly text fields, forms, and other controls with editable text (**controlText**...) and NSText objects (**text**...). One important such message is **textDidChange**: (or **controlTextDidChange**:), which signals that the document's textual content was modified. In responding to this message, controllers can mark a document window as having unsaved data with the **setDocumentEdited**: message (the close button of edited documents is a "broken" X). 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 152) 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]
```



Defining the Controller and User Interfaces

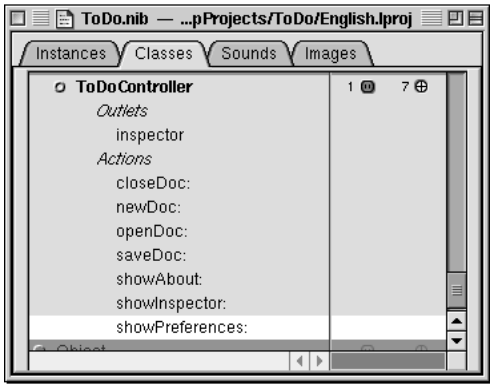
Begin by defining in Interface Builder the object controlling the To Do application.

1 Define the application-controller class.

Create ToDoController as a subclass of NSObject.

Add the outlet and actions shown in the example.

Make the action connections from the appropriate File 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 referenced in the main nib file (**ToDo.nib**) and in **ToDoDoc.nib**, it's instantiated before its nib file is loaded.

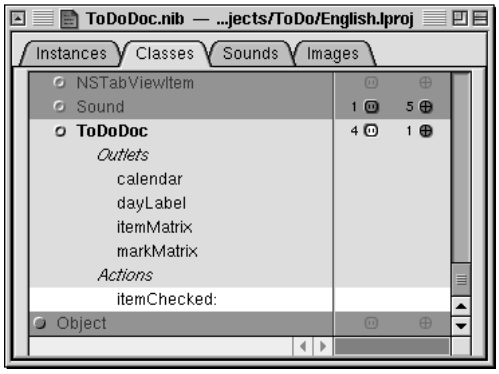
1 Define the document-controller class.

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.

1 Complete the document interface.

Open **ToDoDoc.nib**.

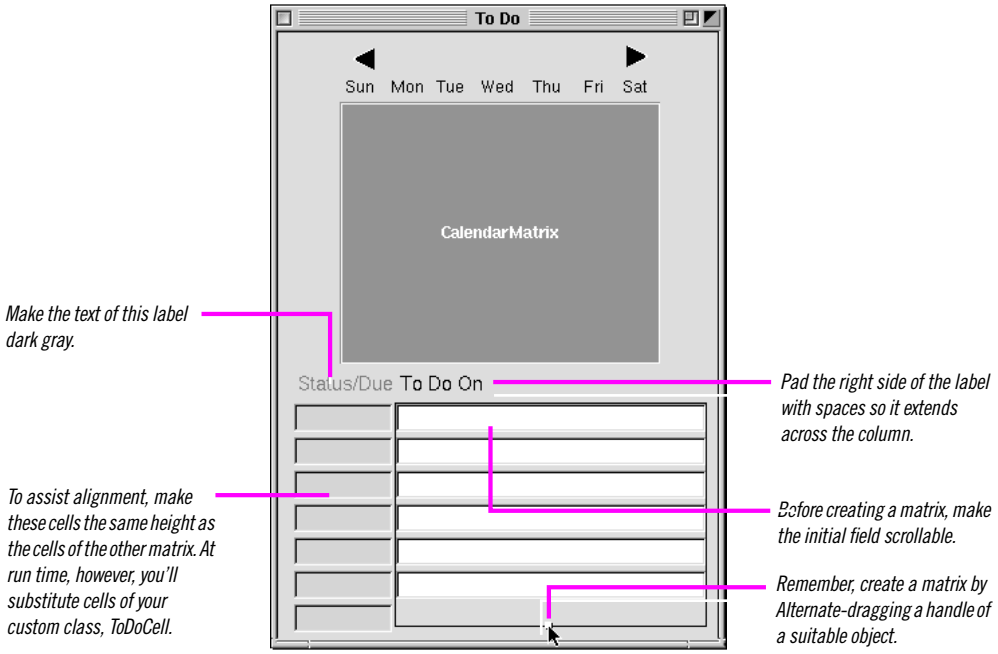
Add the matrices of text fields.

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**.



1 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

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 (or the window icon in the nib file window) to File's Owner (ToDoDoc)
delegate	In **ToDo.nib**, from File's Owner (NSApp) to the ToDoController instance

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

1 Create source-code files for ToDoDoc and ToDoController.

In Project Builder:

1 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.

Creating, Opening, Saving, and Closing Documents

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.

1 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**.

Select ToDoDoc.m in the project browser.

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

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

    [super init];
    if (aFile) {                                                    A
        activeDays = [NSUnarchiver unarchiveObjectWithFile:aFile];
        if (activeDays)
            activeDays = [activeDays retain];
        else
            NSRunAlertPanel(@"To Do", @"Couldn't unarchive file %@",
                nil, nil, nil, aFile);
    } else {                                                        B
        activeDays = [[NSMutableDictionary alloc] init];
        [self setCurrentItems:nil];
    }
    if (![NSBundle loadNibNamed:@"ToDoDoc.nib" owner:self] )        C
          return nil;
    if (aFile)                                                      D
        [[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:

A 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.

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

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

D 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.

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.

1 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];                    A

    [oPanel setAllowsMultipleSelection:YES];
    if ([[[NSApp keyWindow] delegate] isKindOfClass:[ToDoDoc class]])
        startDir = [[[NSApp keyWindow] representedFilename]           B
            stringByDeletingLastPathComponent];
    else
        startDir = NSHomeDirectory();
    result = [oPanel runModalForDirectory:startDir file:nil           C
        types:fileTypes];
    if (result == NSOKButton) {
        NSArray *filesToOpen = [oPanel filenames];
        int i, count = [filesToOpen count];
        for (i=0; i<count; i++) {                                     D
              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.

A 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 files in the panel's browser.

B Sets the directory at which the NSOpenPanel starts displaying files either to the directory of any document window that is currently key or, if there is none, to the user's home directory.

C Runs the NSOpenPanel and obtains the key clicked.

D 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.

1 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.)

1 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];          A
    } else {                                                     B
        int result;
        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])   C
        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.

A 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**.

B 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.

C Archives the document under the chosen directory path and file name and, with the **setDocumentEdited:** message, puts an asterisk next to the window's title (more on this in the next section).

1 **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 ToDo's Data and Coordinating its Display" on page 158.

Coordinate Systems in Rhapsody

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.

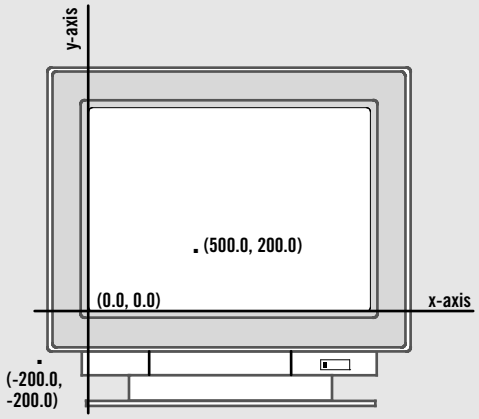
The screen coordinate system has just one function: to position windows on the screen. When your application creates a new window, it must specify the window's initial size and location in screen coordinates.You can "hide" windows by specifying their origin points well within one of the invisible quadrants.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

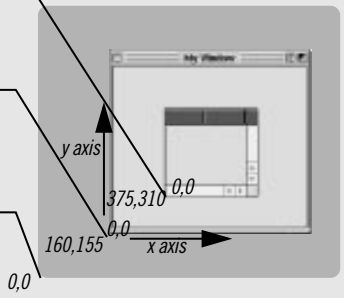
These coordinate systems are the inverse of several other operating systems, which put the origin point at the upper left of the window or screen and extend dimensions downward and to the right. NSView provides means for "flipping" coordinate systems to conform to those other systems.



A view's location 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.



Managing Documents Through Delegation

At certain points while an application is running you want to ensure that a document's data is preserved, that a document's edited status is tracked, or that the application otherwise does "the right thing" for a given circumstance. These events occur when users:

- Edit a document.
- Close a window.
- Launch the application.
- Quit the application by choosing the Exit command.
- Quit the application by closing the last window.
- 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 (with an argument of YES) inserts an asterisk to the right of the window's title, thereby marking it as "dirty" (containing modified, unsaved data).


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

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

Note: The 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.

Assuming that you've completed certain steps (see "Opening Documents by Double-Clicking," below), when users select or double-click a To Do document icon in the Start menu, in Explorer, or elsewhere on the desktop, To Do will launch itself and open the document. But what happens when users simply launch the application, without specifying a document? Rhapsody applications have several alternatives (see side bar on page 155). To Do lends itself well to the user-defaults technique:

- At first, open an "UNTITLED" document.
- When the user saves a document, save the document path in user defaults.
- Thereafter open the last-saved document when the user launches To Do.

1 Customize the launch behavior for your multi-document application.

Initialize the ToDoController class.

```
+ (void)initialize
{
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    NSDictionary *regdom = [NSDictionary dictionaryWithObject:@"UNTITLED"
            forKey:@"ToDoDocumentLastSaved"];
    [defaults registerDefaults:regdom];
 }
```

The **initialize** message is sent to each class before it receives any other message, giving it an opportunity to do something having global effect on all future instances. In ToDoController's case, the **initialize** method specifies a "catch-all" default in the registration domain of user defaults. To Do applications that are launched the first time on a system will take this default.

Opening Documents by Double-Clicking

To let users of your application open documents by double-clicking the document icon in the file system, you must complete the following steps:

1 Specify an icon and a type (file extension) for your document in the Project Attributes display of Project Builder's Project Inspector (see page 118 for an example).

2 Implement the NSApplication delegation method **application:openFile:**. This method is invoked when users double-click or select a document in the file system (for instance, using File Manager or Explorer). In your implementation, you should attempt to create your document using the path given in the second argument. If you succeed, return YES; otherwise, return NO.

3 After building the application, install it in the conventional file-system locations for applications, such as **/LocalApps** and **~/Apps**.

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

```
- (BOOL)applicationOpenUntitledFile:(NSApplication *)sender
{                                                                    A
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    NSString *docToOpen = [defaults stringForKey:
        @"ToDoDocumentLastSaved"];
                                                                     B
    if (![docToOpen isEqualToString:@"UNTITLED"] &&
       [[NSFileManager defaultManager] fileExistsAtPath:docToOpen] &&
        [[docToOpen pathExtension] isEqualToString:@"td"]) {
        ToDoDoc *thisDoc = [[ToDoDoc alloc] initWithFile:docToOpen];
        [thisDoc activateDoc];
        return YES;
    }
    [self newDoc:self];                                              C
    return YES;
}
```

An NSApp's delegate can implement the **applicationOpenUntitledFile:** method to display an appropriate starting document when an Yellow Boxfor Windows application is launched. This specific implementation does the following:

A The class method **standardUserDefaults** returns the NSUserDefaults representing the current user's defaults. From this object, it gets the path of the To Do document that was last saved (more soon on how this was done).

B If the default is not the registration-domain one and the path references a real To Do document, it re-creates and activates the document.

C Otherwise, it creates a new document, which has a title of "UNTITLED."

User Defaults and the Defaults System

User defaults denotes information about a user's preferences that an Rhapsody program keeps between sessions. Also recorded in user defaults are initial values for applications (such as the position of windows), default values that apply globally, and defaults specific to a language (for example, the way in which time is expressed). An application typically allows its users to enter their choices into users defaults through a Preferences panel.

User defaults belong to *domains*. The most common domain consists of individual applications, but there are other domains. For example, NSGlobalDomain holds values common to all applications; there is also a language-specific domain and NSRegistrationDomain (temporary default values).

Each domain has a dictionary of keys and values representing its defaults. Keys are always strings, but values can be property lists: complex data structures comprising arrays, dictionaries, strings, and binary data. Searches for a default proceed through a search list, in which the application's domain typically comes before the global, language-specific, and registration domains.

The defaults system, which implements user defaults, includes a framework component and a command-line component. You can specify, read, and manage user defaults with the methods of NSUserDefaults and with the **defaults** utility.

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

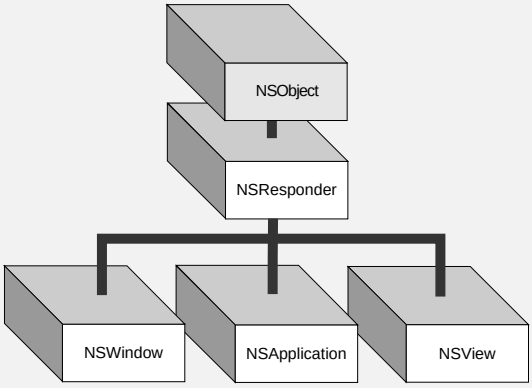
Many classes of the Application Kit stand out in terms of relative importance. NSControl, for example, is the superclass of all user-interface devices, NSText underlies all text operations, and NSMenu has obvious significance. But four classes are at the core of a running application: NSResponder, NSApplication, 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 on the screen. It draws the window's content area 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 menu 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.



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 "First Responder and the Responder Chain" on page 169 for more on the responder chain and a description of *first responder*.

NSApplication

Every application must have one NSApplication object to supervise and coordinate the overall behavior of the application. This object dispatches events 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 active and inactive status. 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 next 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 NSEvent class is also involved in event processing. For more about NSEvent and the event cycle, see "Events and the Event Cycle" on page 168.

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 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:

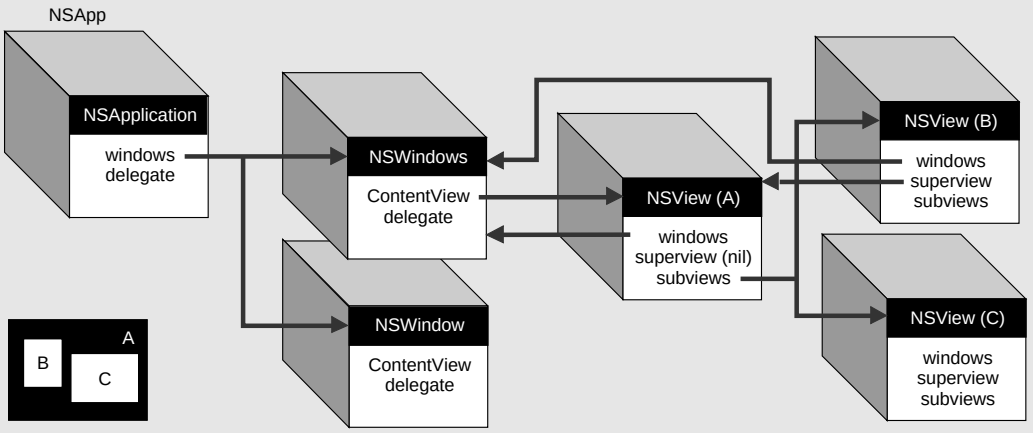
- It permits construction of a superview simply by arrangement of subviews. (An NSBrowser is an instance of a compound NSView.)
- Subviews are positioned in the coordinates of their superview, so when you move an NSView or transform its coordinate system, all subviews are moved and transformed in concert.
- 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.



For To Do, we want the last-saved document to be opened when the user launches the application. Accordingly, in the method that saves documents, we store the document's path in user defaults.

In **ToDoDoc.m**'s **saveDoc** method, add code to write the path of the saved document to user defaults.

(See comments in example for code to add.)

```
/* ... */
        if (result == NSOKButton) {
            fn = [sPanel filename];
            [[itemMatrix window] setTitleWithRepresentedFilename:fn];
            /* add the code below ==========> */
            if (fn && ![fn isEqualToString:@""]) {
                NSUserDefaults *defaults =
                    [NSUserDefaults standardUserDefaults];
                [defaults setObject:fn forKey:@"ToDoDocumentLastSaved"];
                [defaults synchronize];
            }
            /* <========== add the code above */
        }
/* ... */
```

The new section of code gets the NSUserDefaults object for the current user and stores the document path (**fn**) in user defaults for that user under the key ToDoDocumentLastSaved. The **synchronize** method saves this default to disk.

1 Save edited documents when windows are closed.

Implement the delegation method **windowShouldClose:**.

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

When an Rhapsody Application Is Launched

When the user launches an application, the default behavior is to display the contents of the main nib file. This initial presentation could be one or more windows, but often it is just the application's menu. Often with document-centric applications, this behavior is what you want. But you aren't restricted to this behavior.

With Rhapsody applications you have a number of alternatives. The alternative that is best for an application depends on that application's nature and purpose.

Put up an untitled document. The application displays a content-less document with a window title of "UNTITLED" (or something similar). The user can start adding content immediately or can open an existing document. This is the course adopted by the TextEdit application. A variation of this approach always displays an initial window with some standard content, such as a product logo (see the Preview application).

How: The application's delegate must implement the **applicationOpenUntitledFile:** method and, in that method, create a new document or open a standard document.

Display the document that the user last saved. The first time a user launches an application, the application creates and displays an untitled document. When the user saves that document, the application stores the full path of the saved file in user defaults. The next time the user launches the application the application restores the document from the file. This is the approach taken by the To Do application.

How: Implement **applicationOpenUntitledFile**:, as before, but this time first check user defaults to see if it contains a path for a document file. If it does, verify that the file exists (it could have been moved or deleted since the last session) before opening and displaying it. Otherwise, display an untitled document. When the user closes a document or terminates the application, store the full path of the last-saved document file in user defaults.

Display an opened-document window. The opened-document window (typically small) contains a list of documents that the user currently has created or opened. Users can get a document to appear by clicking an item in the list. When users choose the Exit command, the application can terminate after closing (and, if necessary, saving) all listed documents. As a variation, the application can, when it's next launched, restore to the project window (via user defaults) the documents opened when the last session was terminated.

How: In the application's main nib file create a small window that contains a table view or browser. The project window's menu bar can contain the complete set of menus or an appropriate subset. When the application is launched, the project window is automatically displayed. When users open or create a document, create and insert an appropriate entry in the table view or browser. When users click (or double-click) on an entry, display the document.

When users click a window's close button, the window sends **windowShouldClose:** to the window's delegate. The window expects a response directing it either to close the window or leave it open. This implementation does the following:

Ⓐ Returns YES (close the window) if the document hasn't been edited.

Ⓑ 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).

Ⓒ 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**, described in the following section) and then sends **saveDoc** to itself to archive application data to a file.

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.

The NSApplication object sends **applicationShouldTerminate:** to its delegate to give it notice that the application is about to terminate. In this method you should first let the user save any edited document.

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

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

```
- (BOOL)applicationShouldTerminate:(id)sender
{
    NSString *repfile = nil;
    NSArray *appWindows = [NSApp windows];                    A
    NSEnumerator *enumerator = [appWindows objectEnumerator];
    id object;

    while (object = [enumerator nextObject]) {                B
        int result;
        id doc;
        if ((doc = [object delegate]) &&
            [doc isKindOfClass:[ToDoDoc class]] &&
            [object isDocumentEdited] ) {
            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];                    C
                break;
              case NSAlertOtherReturn:
                return NO;
            }
        }
    }
    return YES;
}
```

Much of the code in this method is similar to that for **windowShouldClose:**; if a window is managed by ToDoDoc, the **applicationShouldTerminate:** method puts up an attention panel and responds according to the user's choice. However, there are some significant differences:

A Returns all open windows of the application in an NSArray. Remember, one of the jobs of an NSApplication is to track and manage all windows.

B Enumerates and processes the NSWindow objects in this NSArray, as noted.

C If the user clicks "Don't Save," the **close** message forces the window to close (without sending the **windowShouldClose:** delegate message).

Managing ToDo's Data and Coordinating its Display

If you recall the discussion on To Do's design earlier in this chapter ("How To Do Stores and Accesses its Data" on page 117), 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, numCols;
        [itemMatrix getNumberOfRows:&numRows columns:&numCols];
        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** 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.

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

Implement **controlTextDidEndEditing:**. as shown.

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

A control sends **controlTextDidEndEditing:** to its delegate when the insertion point *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.

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

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

C 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.

D 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.

E Updates the list of items in the document interface.

1 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]]) {          A
            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  {                                                   B
            [[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.

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

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

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

1 Respond to user actions in the calendar.

Implement CalendarMatrix's delegation methods.

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

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

A 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.

B 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.

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

Implement **saveDocItems:**.

1 **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 or her document and then opens it.

1 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.

1 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 To Do 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.