;D_ToDo_Subclass1.rtfd;;¬ Previous Section ;F_ToDo_MgmtByDelegation.rtfd;;¬ Next Section

4. To Do Tutorial

The Basics of a Multi-Document Application

A multi-document application, as described on page 136, has at least one application controller and a document controller for each document opened. The application controller also responds to user commands relating to documents and either creates, opens, closes, or saves a document.

1 Customize the application's main menu.

Open ToDo.nib in Interface Builder.

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

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

Change the title of ªItemº to ªInspector.º

TD_BasicsofMDApp1.eps ¬

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

2 Define the application-controller class.

Create ToDoController as a subclass of NSObject.

Add the outlet and actions (listed below) to the class.

Make the action connections from the appropriate Document menu commands.

TD_BasicsofMDApp2.eps ¬

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

3 Define the document-controller class.

Create ToDoDoc as a subclass of NSObject.

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

Instantiate ToDoController and ToDoDoc.

Save **ToDo.nib**.

TD_BasicsofMDApp3.eps ¬

Now add the remaining objects to the document interface.

4 Complete the document interface.

Open **ToDoDoc.nib**.

Add the matrices of text fields.

Add the labels above the matrices.

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

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

Save **ToDoDoc.nib**.

TD_BasicsofMDApp4.eps ¬

5 Connect the outlets and actions of ToDoDoc.

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

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

Make the connections described in the table below.

Name	Connection	Type
TableHeadRule.eps ¬		
calendar	From File's Owner to the CalendarMatrix object	outlet
TableRule.eps ¬		
dayLabel	From File's Owner to label ªTo Do onº	outlet
830143_TableRule.eps ¬		
itemMatrix	From File's Owner (ToDoDoc) to matrix of long text fields	outlet
156455_TableRule.eps ¬		
markMatrix	From File's Owner to matrix of short text fields	outlet
20358_TableRule.eps ¬		
itemChecked:	From matrix of short text fields to File's Owner	action
115640_TableRule.eps ¬		

Related Concept: ;ToDoConcepts.rtfd;linkMarkername TheStructureofMulti-DocumentApplications;, The Structure of Multi-Document Applications

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 applicationÐthis step is critical to the multi-document design.

Connect ToDoDoc and ToDoController to other objects as their delegates.

Name	Connection

615759_TableHeadRule.eps ¬

textDelegate 729719_TableRule.eps ¬	From the CalendarMatrix object to File's Owner (ToDoDoc)
delegate 822959_TableRule.eps ¬	From the document window's title bar to File's Owner (ToDoDoc)
delegate 920575_TableRule.eps ¬	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.

6 Create source-code files for ToDoDoc and ToDoController.

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

In Project Builder:

Select ToDoDoc.h in the project browser.

Add the declarations at right.

(Ellipses indicate existing declarations.)

```
@interface ToDoDoc:NSObject
{
    /* ... */
    NSMutableDictionary *activeDays;
    NSMutableArray *currentItems;
}
/* ... */
- (NSMutableArray *)currentItems;
- (void)setCurrentItems:(NSMutableArray *)newItems;
- (NSMatrix *)itemMatrix;
- (NSMatrix *)markMatrix;
```

```
- (NSMutableDictionary *)activeDays;
- (void)saveDoc;
- (id)initWithFile:(NSString *)aFile;
- (void)dealloc;
- (void)activateDoc;
- (void)selectItem:(int)item;
@end
```

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

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

8 Write the code that creates documents.

Select ToDoController.m in the project browser.

Implement ToDoController's newDoc: method.

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

The **newDoc:** method is invoked when the user chooses New from the Document menu. The method allocates and initializes an instance of the document controller, ToDoDoc, thereby creating a document. (See the

implementation of **initWithFile:** on the following page to see what happens in this process.) It then updates the document interface by invoking **activateDoc**..

Select **ToDoDoc.m** in the project browser.
Implement ToDoDoc's **initWithFile:** method.

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

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

```
    [[itemMatrix window] makeKeyAndOrderFront:self];
    return self;
}
```

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

1. Restores the document's archived objects if the **aFile** argument is the pathname of a file containing the archived objects (that is, the document is opened). If objects are unarchived, it retains the **activeDays** dictionary; otherwise it displays an attention panel.
2. Initializes the **activeDays** and **currentItems** instance variables. A **aFile** argument with a **nil** value indicates that the user is requesting a new document.
3. Loads the nib file containing the document interface, specifying **self** as owner.
4. Sets the title of the window; this is either the file name on the left of the title bar and the pathname on the right, or ªUNTITLEDº if the document is new.

98132_TableRule.eps ¬Before You Go On

Note the **[itemMatrix window]** message nested in the last message. Every object that inherits from NSView ªknowsº its window and will return that NSWindow object if you send it a **window** message.
293232_TableRule.eps ¬

9 Implement the document-opening method.

Select **ToDoController.m** in the project browser.
Write the code for **openDoc:**.

```
- (void)openDoc:(id)sender
{
```

```
    int result;
    NSString *selected, *startDir;
    NSArray *fileTypes = [NSArray arrayWithObject:@"td"];
    NSOpenPanel *oPanel = [NSOpenPanel openPanel];            /* 1 */

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

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

1. Creates or gets the NSOpenPanel instance (an instance shared among objects of an application). The

previous message specifies the file types (that is, the extensions) of the files that will appear in the Open panel browser. The next message enables selection of multiple file in the panel's browser.

2. Sets the directory at which the NSOpenPanel starts displaying files either to the directory of any document window currently key or , if there is none, to the user's home directory.
3. Runs the NSOpenPanel and obtains the key clicked.
4. If the key is NSOKButton, cycles through the selected files and, for each, creates a document by allocating and initializing a ToDoDoc instance, passing in a file name.

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

10 Write the code that closes documents.

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

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

NSApp, the global NSApplication instance, keeps track of the application's windows, including their status. Because only one window can have main status, the **mainWindow** message returns that NSWindow objectÐ 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, ªManaging Documents Through Delegation,º to learn how the document handles this user event.)

11 Write the code that saves documents.

In ToDoController.m, implement the saveDoc: method.

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

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

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

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

Implement the **saveDoc:** method.

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

    if (![[[itemMatrix window] title] hasPrefix:@"UNTITLED"]) {
        fn = [[itemMatrix window] representedFilename];        /* 1 */
    } else {
        int result;                                            /* 2 */
```

```
        NSSavePanel *sPanel = [NSSavePanel savePanel];
        [sPanel setRequiredFileType:@"td"];
        result = [sPanel runModalForDirectory:NSHomeDirectory() file:nil];
        if (result == NSOKButton) {
            fn = [sPanel filename];
            [[itemMatrix window] setTitleWithRepresentedFilename:fn];
        } else
            return;
    }

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

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

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

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

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

12 Implement the accessor methods for ToDoController and ToDoDoc.

Don't implement **setCurrentItems:** yet. This method does something special for the application that will be covered in ``Managing the Data and Coordinating its Display.''

Related Concepts: ;ToDoConcepts.rtfd;linkMarkername OnlyWhenNeeded:DynamicallyLoadingResourcesandCode;, Only When Needed: Dynamically Loading Resources and Code

;ToDoConcepts.rtfd;linkMarkername TheStructureofMulti-DocumentApplications;, The Structure of Multi-Document Applications