


```

    } else {
        activeDays = [[NSMutableDictionary alloc] init];
        [self setCurrentItems:nil];
    }
    /* ... */
}

```

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

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

Implement **setCurrentItems:**.

```

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

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

```

This ^aset^o 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 ^ainitializes^o it with empty string objects. It does this because `NSMutableArray`'s methods cannot tolerate `nil` objects within the bounds of the array.

So there's now a `currentItems` array ready to accept `ToDoItems`. Imagine yourself using the application. What are the user events that cause a `ToDoItem` to be added to the `currentItems` array? To Do allows entry of items ^aon the fly,^o 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:

SquareBullet.eps → Press the Tab key.
953169_SquareBullet.eps → Press the Enter key.
75306_SquareBullet.eps → Click outside the text field.

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

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

Implement `controlTextDidEndEditing:`.

```
- (void)controlTextDidEndEditing:(NSNotification *)notif
{
    id curItem, newItem;
    int row = [itemMatrix selectedRow];
    NSString *selName = [[itemMatrix selectedCell] stringValue];
/* 1 */
    if (![itemMatrix window] isDocumentEdited) ||
        (row >= [currentItems count])) return;
    if (!currentItems)
        [self setCurrentItems:nil];
/* 2 */
```

```

    if ([selName isEqualToString:@""] &&
        ([[currentItems objectAtIndex:row] isKindOfClass:
         [ToDoItem class]]) &&
        (![currentItems objectAtIndex:row] itemName]
         isEqualToString:@""))
        [currentItems replaceObjectAtIndex:row withObject:@""];
/* 3 */
    else if ([[currentItems objectAtIndex:row] isKindOfClass:
             [ToDoItem class]] &&
             (![currentItems objectAtIndex:row] itemName]
              isEqualToString:selName))
        [[currentItems objectAtIndex:row] setName:selName];
/* 4 */
    else if (![selName isEqualToString:@""]) {
        newItem = [[ToDoItem alloc] initWithName:selName
              andDate:[calendar selectedDay]];
        [currentItems replaceObjectAtIndex:row withObject:newItem];
        [newItem release];
    }
/* 5 */
    [self updateMatrix];
}

```

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

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

2. If the user deletes the text of an existing item, it removes the `ToDoItem` that positionally corresponds to the row of that deleted text.
3. It changes the name of an item if the text entered in a field doesn't match the name of the corresponding item in the **currentItems** array.
4. If either of the two previous conditions don't apply, and text has been entered, it creates a new `ToDoItem` and inserts it in the **currentItems** array.
5. Updates the list of items in the document interface.

3 Update the document interface with the current items.

Implement **updateMatrix**:

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

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

```

        [[markMatrix cellAtRow:i column:0] setTimeDue:due];
        [[markMatrix cellAtRow:i column:0] setTriState:
            [thisItem itemStatus]];
    }
    else {
        /* 2 */
        [[itemMatrix cellAtRow:i column:0] setStringValue:@""];
        [[markMatrix cellAtRow:i column:0] setTitle:@""];
        [[markMatrix cellAtRow:i column:0] setImage:nil];
    }
}
}

```

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

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

1. If an object in the array is a ToDoItem, it writes the item name to the text field corresponding to the array slot and updates the button cell next to the field.
2. If an object isn't a ToDoItem, it blanks the corresponding text field and cell.

4 Respond to user actions in the calendar.

Implement CalendarMatrix's delegation methods.

```

- (void)calendarMatrix:(CalendarMatrix *)matrix
    didChangeToDate:(NSDate *)date
{
    /* 1 */
}

```

```

[[itemMatrix window] makeFirstResponder:[itemMatrix window]];
[self saveDocItems];

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

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

```

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

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

5 Save the data to internal storage.

Implement **saveDocItems**:

```
- (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.

6 Archive and unarchive the document's data.

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

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

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

Implement **awakeFromNib** as shown below.

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

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

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

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

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

Implement **activateDoc** as shown below.

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

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