Table View Programming Guide for Mac

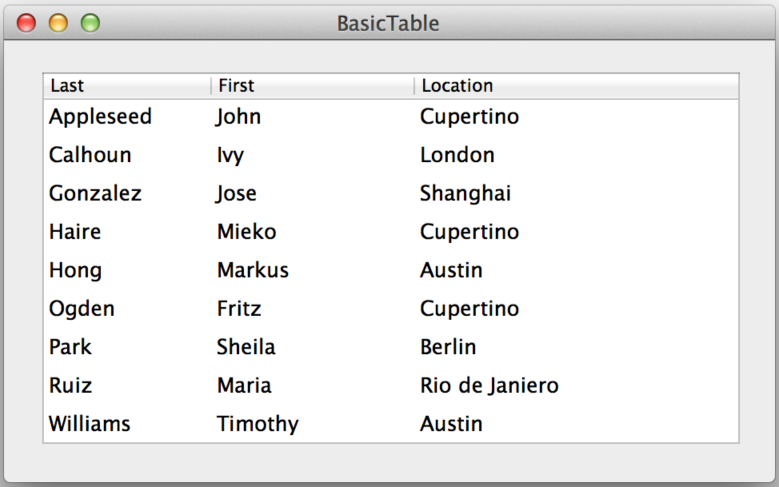
Developer

Contents

Figures and Listings

About Table Views in OS X Applications

A table view displays data for a set of related records, with rows representing individual records and columns representing the attributes of those records. For example, in a table of employee records, each row represents one employee, and the columns might represent employee attributes such as the last name, first name, and office location.



Last	First	Location
Appleseed	John	Cupertino
Calhoun	Ivy	London
Gonzalez	Jose	Shanghai
Haire	Mieko	Cupertino
Hong	Markus	Austin
Ogden	Fritz	Cupertino
Park	Sheila	Berlin
Ruiz	Maria	Rio de Janiero
Williams	Timothy	Austin

A table view can have a single column or multiple columns, and it allows vertical and horizontal scrolling, content selection, and column dragging. Each row in a table view has at least one corresponding cell that represents a field in a data collection.

Note: The generic term **cell** is used to describe the content within a row and column in a table view. When it's necessary to refer to the `NSCell` class and its subclasses, the class name is specified.

At a Glance

Understanding the structure of a table view, and knowing how to build one, lets you create Mac apps that present tabular data in an attractive, functional way.

Tables Use a Collection of Classes to Manage Content

The various components of a table view—including column, row, header, and cell—are each supported by a distinct `NSView` subclass. These classes work together with the `NSTableView` class itself to display content and to enable behaviors such as animation, column rearrangement, sorting, and selection. And, because most tables use `NSView` objects to represent individual cells, it's easy to design custom cell views in Interface Builder and to support animation and column management.

Relevant Chapter: Understanding Table Views (page 9)

Interface Builder Makes It Easy to Create Tables

Using Interface Builder, you add a table view to a window or superview, add and arrange columns, and specify column headers. Then, you typically create cell view prototypes that your app uses to provide the content layout for each table cell. (If you're working with an `NSCell`-based table, you create subclasses of `NSCell` for each table cell.) Many aspects of tables can be set directly in Interface Builder, which means that you can avoid writing additional code.

Relevant Chapter: Constructing a Table View Using Interface Builder (page 18), Working with NSCell-Based Table Views (page 39)

Tables Can Get Data in Two Ways

You must provide data to the table view. You can do this in one of two ways:

- Programmatically, by implementing a data source class
- Using Cocoa bindings

To provide data programmatically, you create a class that conforms to the `NSTableViewDataSource` protocol and implement the method that provides the row and column data as requested.

Use Cocoa bindings to create a relationship between a controller class instance, which manages the interaction between data objects, and the table view. When you use the bindings approach, you don't create a data source class for providing the data or supporting editing.

The techniques you use to create and populate a table differ depending on whether the table is `NSView` based or `NSCell` based.

Relevant Chapters: Populating a Table View Programmatically (page 20), Populating a Table View Using Cocoa Bindings (page 24), Working with NSCell-Based Table Views (page 39)

A Table's Appearance and Behaviors Are Customizable

You can customize various aspects of a table's appearance, including background color, row color, and grid line color. You can also specify how a table should behave when users make selections or sort table data. (The techniques you use to modify a table's appearance and behavior are the same for both `NSView`-based and `NSCell`-based tables.)

Relevant Chapters: Modifying a Table's Visual Attributes (page 27), Enabling Row Selection and User Actions (page 30), Sorting Table View Rows (page 36)

NSCell-Based Tables Are Still Supported

In OS X v10.6 and earlier, each table view cell was required to be a subclass of `NSCell`. This approach caused limitations when designing complex custom cells, often requiring you to write your own `NSCell` subclasses. Providing animation, such as progress views, was also extremely difficult. In this document these types of table views are referred to as `NSCell`-based table views. `NSCell`-based tables continue to be supported in OS X v10.7 and later, but they're typically used only to support legacy code. In general, you should use `NSView`-based tables.

Although you use the same Interface Builder techniques to create both `NSView`-based and `NSCell`-based table views (and to add columns to a table), the code required to provide individual cells, populate the table view, and support programmatic editing differs depending on the table type. In addition, you use different Cocoa bindings techniques depending on whether you're working with an `NSView`-based or `NSCell`-based table.

Relevant Chapter: Working with NSCell-Based Table Views (page 39)

Prerequisites

To develop successfully with the `NSTableView` class, you need a strong grasp of the Model-View-Controller design pattern. To learn more about this fundamental pattern, see Model-View-Controller in Cocoa (OS X).

`NSTableView` instances can be used with Cocoa bindings, both in `NSView`-based and `NSCell`-based tables. However, it's strongly suggested that you thoroughly understand the programmatic interface of the table view before beginning to use the more advanced Cocoa bindings. For a brief overview of bindings, see Cocoa bindings; to learn more, read *Cocoa Bindings Programming Topics* .

To learn about the recommended appearance and behavior of table views in the user interface, see *OS X Human Interface Guidelines* .

See Also

The following sample code projects are instructive when designing your own table view implementations:

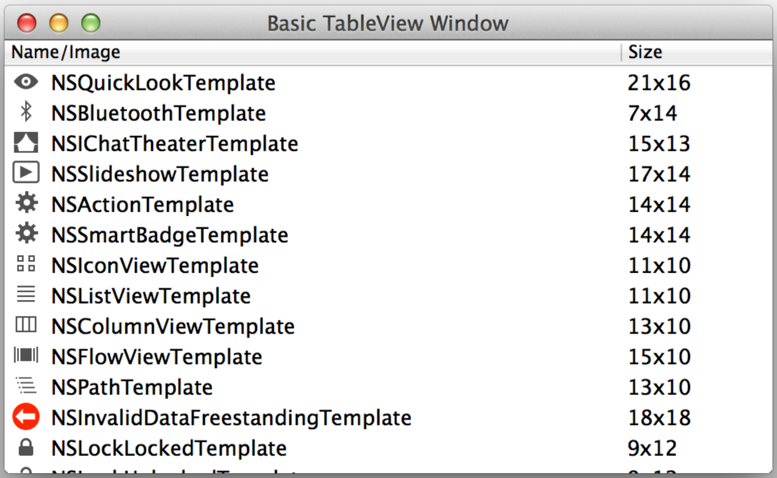
- *TableViewPlayground*
- *AnimatedTableView*
- *iSpend*
- *With and Without Bindings*
- *Cocoa Tips and Tricks*

Understanding Table Views

In the most general terms, a table view is made up of rows and one or more columns that display the content of a data collection. Each row represents a single item within the data model, and each column displays a specific attribute of the model. A cell represents the content at a specific row-column intersection. The user selects rows and the app can perform the appropriate action on those rows.

Tables are made up of versatile user interface elements that can display simple lists of data or complex arrangements that combine data, functionality, and controls to provide a rich user experience. For example, the simple two-column table shown in Figure 1-1 uses one cell that displays an image and text and a second cell that displays only text.

Figure 1-1 A simple table

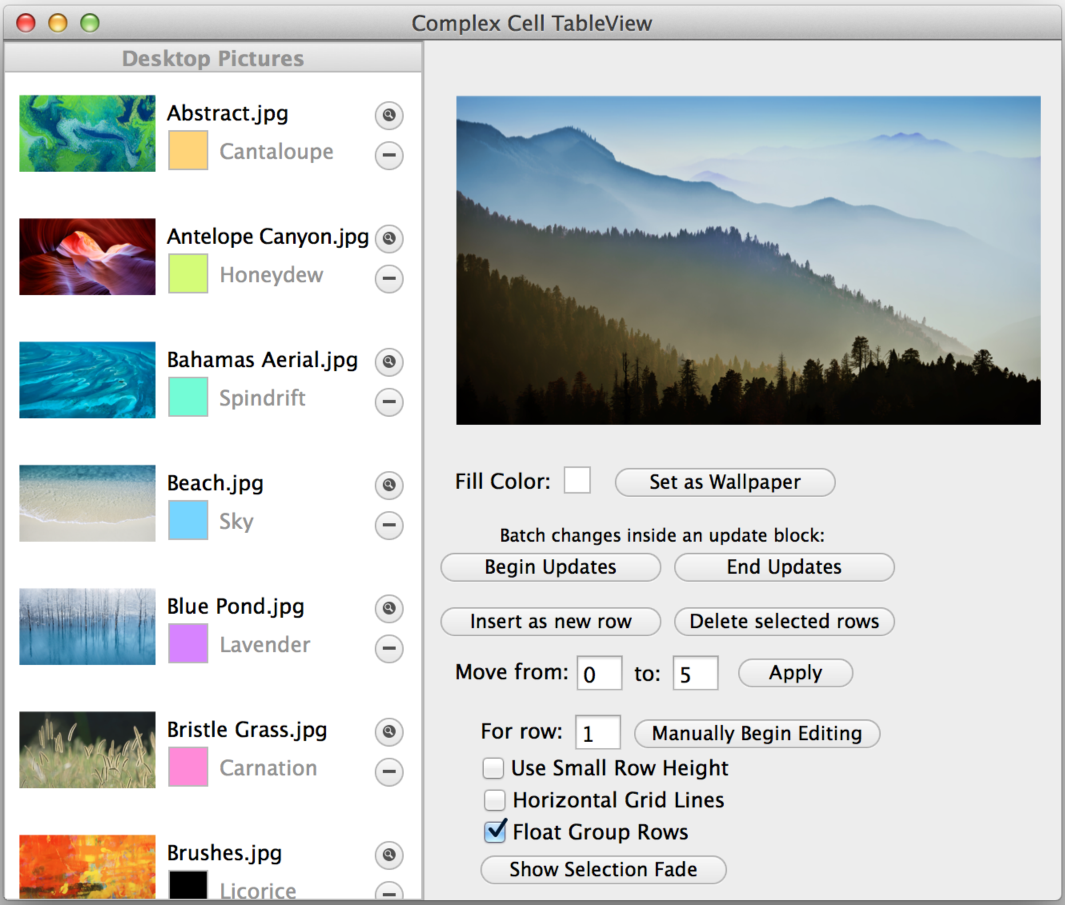


Name/Image	Size
NSQuickLookTemplate	21x16
NSBluetoothTemplate	7x14
NSIChatTheaterTemplate	15x13
NSSlideshowTemplate	17x14
NSActionTemplate	14x14
NSSmartBadgeTemplate	14x14
NSIconViewTemplate	11x10
NSListViewTemplate	11x10
NSColumnViewTemplate	13x10
NSFlowViewTemplate	15x10
NSPathTemplate	13x10
NSInvalidDataFreestandingTemplate	18x18
NSLockLockedTemplate	9x12

Constructing this simple table view required no subclassing to display the content; both cell views are instances of the `NSTableCellView` class. To create a similar table, you simply drag the stock classes from the Interface Builder Object library and set the values of the appropriate subviews. (You can examine the implementation of this table in the *TableViewPlayground* sample project.)

In contrast, the left area of the window shown in Figure 1-2 contains a complex table view in which each row consists of an image, some text, a custom color view, and buttons. Much, if not all, of the complex table view shown here is made using standard AppKit view objects that are arranged in a custom subclass of `NSTableCellView`. (The implementation of this table is also contained in the *TableViewPlayground* sample project.)

Figure 1-2 A complex table



Desktop Pictures	
Abstract.jpg	Cantaloupe
Antelope Canyon.jpg	Honeydew
Bahamas Aerial.jpg	Spindrift
Beach.jpg	Sky
Blue Pond.jpg	Lavender
Bristle Grass.jpg	Carnation
Brushes.jpg	Licorice

Fill Color:

Batch changes inside an update block:

Move from: 0 to: 5

For row: 1

- [ ] Use Small Row Height
- [ ] Horizontal Grid Lines
- [✓] Float Group Rows

Most Tables Are Based on NSView

Most tables are `NSView` based, which means that each cell is provided by an `NSView` subclass, often by `NSTableCellView` (or a subclass). Some tables are `NSCell` based, which means that each table cell is based on a subclass of `NSCell`. For the most part, `NSCell`-based tables are used to support legacy code; if you're creating a new app, you want to use `NSView`-based tables. In this document, a table is assumed to be `NSView` based unless specified otherwise.

Using `NSView` objects as cells allows table views to enable rich design time opportunities. By default, an `NSTableCellView` object includes an image view and a text field. To build a table view in Interface Builder, you create a cell for each column by dragging an `NSTableCellView` instance from the Object library and dropping it into the appropriate table column. You then configure each cell, called a **prototype cell**, with text, images, or other attributes. At runtime, the data source loads a prototype cell for each cell in the row and displays the data according to your configuration. Using Interface Builder, it's easy to modify the cell's subviews, and to move, resize, and hide cells. And when you use `NSTableCellView` instances in a table, VoiceOver automatically speaks the contents of the text field.

You can subclass `NSTableCellView` to add additional subviews and behaviors, and you can use Interface Builder to modify a cell's design and layout. Whether you use the standard `NSTableCellView` class or a custom subclass, an app retrieves a cell view at runtime and populates it with data, either programmatically or using Cocoa bindings. Because tables reuse cell views when possible, cell views can be varied and complex without negatively impacting memory usage.

View classes support animation within their content, so it's straightforward to animate content within `NSTableCellView` instances. For example, the *TableViewPlayground* sample project uses `NSProgressIndicator` instances to display content that's loaded lazily.

Tables support animation of cells as they are moved, inserted, and deleted. Various animation modes are provided and they can be grouped to allow changes to happen in batches as you make changes within the table view and corresponding changes in the model data.

Tables Consist of Several Classes That Work Together

Table views are made up of the following classes:

- `NSTableView`

  This class declares methods that allow you to configure the appearance of the table view—for example, specifying the default height of rows or providing column headers that are used to describe the data within a column. Other methods give you access to the currently selected row as well as to specific rows or cells. You can call other table view methods to manage selections, scroll the table view, and insert or delete rows and sections.

- `NSTableColumn`

  The column class is responsible for managing the horizontal position and the width of the cells of the table. Columns can be configured to allow resizing, re-ordering, and content sorting, with the state (optionally) stored with the app. Each table column has an identifier associated with it, which is key when finding columns and retrieving cells. Identifiers are discussed in more detail in Columns and Cells Have Identifiers That Make It Easy to Find Them (page 15).

- `NSTableHeaderView`

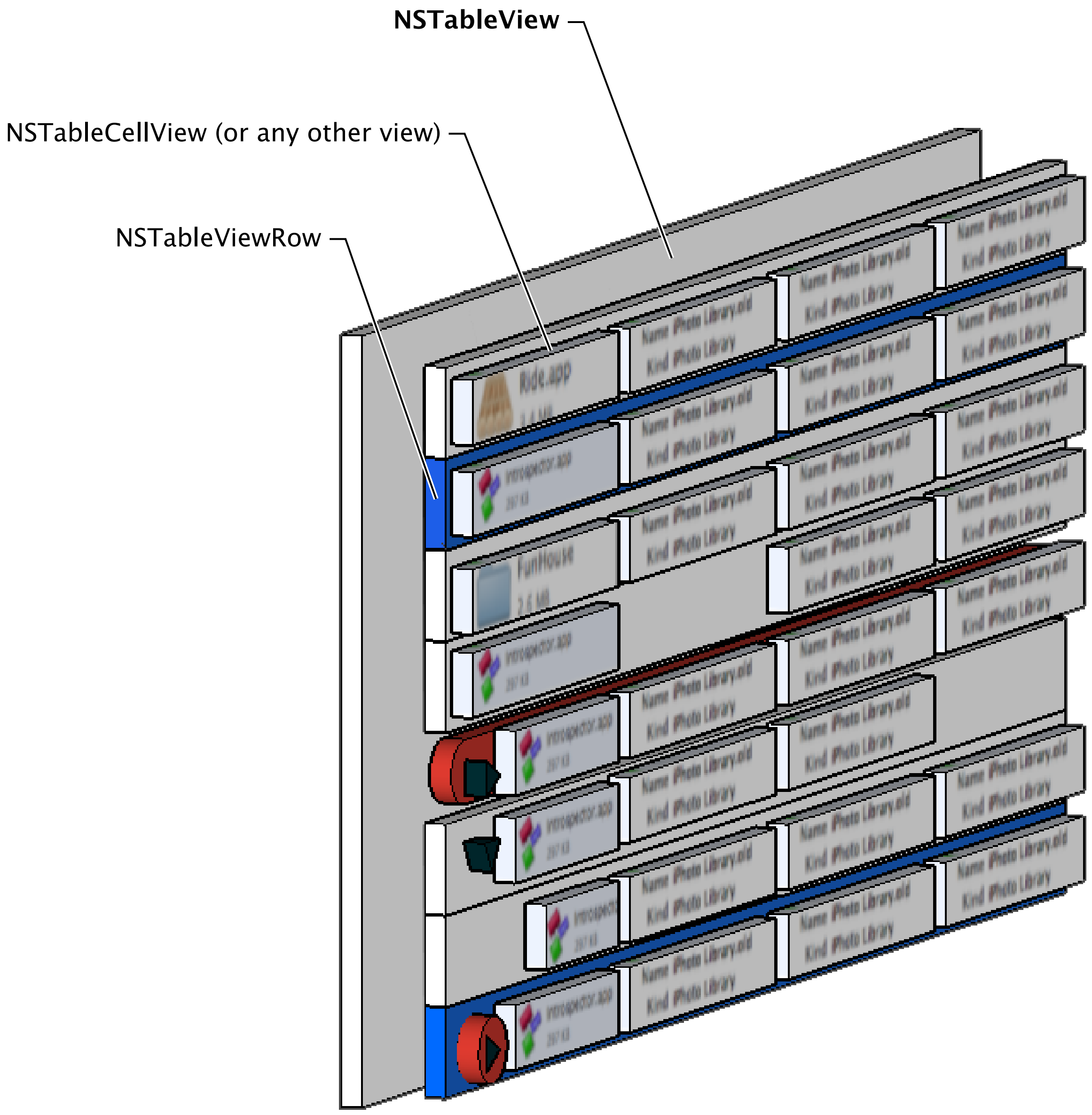
  The header view class is responsible for the cell that does its actual work, the `NSTableHeaderCell`. The header cell is responsible for drawing the column name—if it's visible—as well as the header content itself, optional sort indicators, drawing highlighting, and more.

- `NSScrollView` and `NSClipView`

  These two classes aren't part of the table view—nor are they required—but virtually all table views are displayed using the classes that make up the scroll view mechanism.

`NSView`-based table views rely heavily on two additional classes (`NSTableRowView` and `NSTableCellView`) and their subclasses. Figure 1-3 shows how these components come together to create a table view.

Figure 1-3 Breakdown of an `NSView`-based table view



The blue rows shown in Figure 1-3 are highlighted as if selected by the user. The red rows are rows that have been subclassed to draw in their background in a custom manner. Empty or missing cells are shown in the table and are allowed within table views.

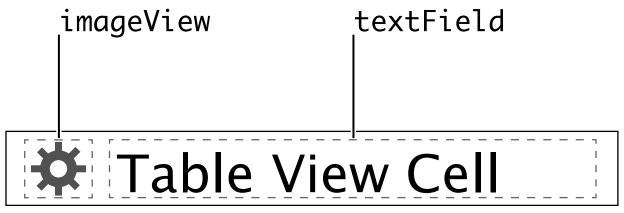
A table view consists of a collection of multiple `NSTableRowView` instances, one for each visible row. The frame rectangle of the row view is the full width of the table view and the height of the row, taking into account the `intercellSpacing`. Row views are responsible for drawing selection, drag and drop feedback, highlighting, column dividers, and any additional custom indicators that may be required, including a custom background. Each `NSTableRowView` has a collection of subviews that contain each table column cell.

You can use the delegate method `tableView:rowViewForRow:` to customize row views. You typically use Interface Builder to design and lay out `NSTableRowView` prototype rows within the table. As with prototype cells, prototype rows are retrieved programmatically at runtime. Implementation of `NSTableRowView` subclasses is entirely optional.

Each `NSTableCellView` instance is inserted as a subview of an `NSTableRowView` instance, which represents the entire row. The default `NSTableCellView` class has Interface Builder outlets for a text field and an image view. VoiceOver automatically speaks the contents of the text field, giving your app basic accessibility capabilities without effort on your part.

Apps typically create subclasses of `NSTableCellView` to add additional properties; optionally, you can use custom `NSView` instances to create new cells. Figure 1-4 shows the components of the default `NSTableCellView` object.

Figure 1-4 `NSTableCellView` components



A Table View Needs a Data Source and Should Have a Delegate

Following the Model-View-Controller design pattern, an `NSTableView` object must have a data source. To control the display of data, a table view should have a delegate.

The data source class (`NSTableViewDataSource`) mediates between the table view and the table view's model data. The data source is responsible for implementing the support that provides the model data in a pasteboard format that allows both copying of data and dragging of rows. (Note that to support the drag and drop of single rows to the Finder, your model object must be compliant with the `NSPasteboardWriting` protocol.) The data source is also responsible for determining whether incoming drag actions are valid and how they should be handled.

The delegate class (`NSTableViewDelegate Protocol`) allows you to customize a table view's behavior without requiring you to subclass the table view. It supports table column management and type-to-select functionality, and lets you specify whether specific rows should allow selection, among other behaviors.

The data source and the delegate are often (but not necessarily) the same object.

The data source object must adopt the `NSTableViewDataSource` protocol, and the delegate object must adopt the `NSTableViewDelegate Protocol`. When programmatically populating a table view, there are methods that must be implemented in both the data source and the delegate (to learn more about this, see Populating a Table View Programmatically (page 20)).

To enable editing of table view cells, use the target-action methodology in order to edit the content.

The responsibilities of both the delegate and data source classes differ when Cocoa bindings are used for populating tables—for more information, see Populating a Table View Using Cocoa Bindings (page 24). (If you're using an `NSCell`-based table, the responsibilities of the data source and delegate objects are somewhat different; to learn more, see Working with NSCell-Based Table Views (page 39).)

Columns and Cells Have Identifiers That Make It Easy to Find Them

Every column in a table view has an associated identifier string that is set in the Identity inspector of the Xcode Interface Builder editor. This string is a convenient and versatile way of referring to and retrieving individual columns in a table view.

Apps typically use context-based names for the identifier of a table column, such as Artist or Name, so that the columns can be easily retrieved and identified by other aspects of the app. For example, if your app allows the showing and hiding of table columns, it can easily identify the column that must be shown or hidden by using the identifier and the `NSTableView` class's `tableColumnWithIdentifier:` method. Using column identifiers is important because if the columns get reordered, a column's index in the `tableColumns` array also changes.

Column identifiers relate a table column to a view instance that the column displays. When the `NSTableView` delegate method `tableView:viewForTableColumn:row:` attempts to locate the cell for the table column, it uses the column's identifier to locate the cell. If you set identifiers manually, you must ensure that the column and cell view identifiers stay in sync if you change either one. If the identifier values get out of sync, a table that relies on bindings for its data won't work. If you plan to use bindings, it's recommended that you use the Automatic setting for the identifier values.

If your table view doesn't require complex table column management, such as retrieving individual table columns or showing and hiding columns, you can take advantage of the Automatic function of the Identifier. By default, the Identifier field is set to Automatic. When you accept the Automatic setting, Interface Builder creates a unique identifier for the table column and ensures that the cell view instance within that table column has the same identifier. Further, it keeps these identifiers in sync, relieving you of that responsibility.

Table Views Reuse Rows to Increase Speed and Efficiency

Table views create and maintain a queue of used cell views that allow for the efficient reuse of previously created cells. A cell view is considered to be *in use* if it:

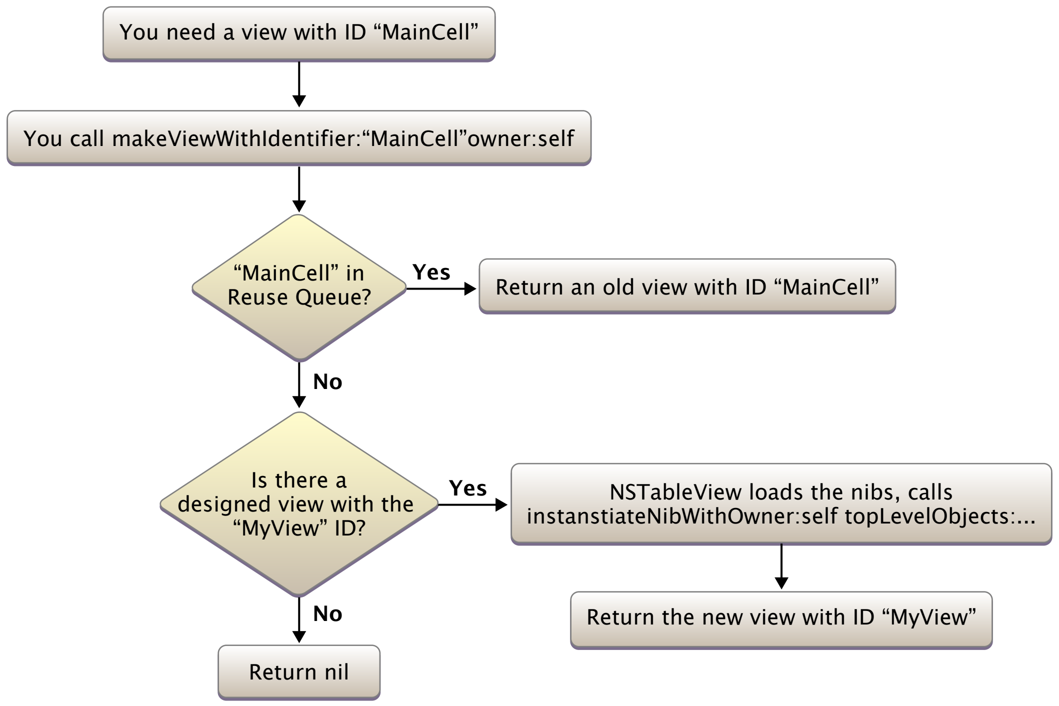
- Is currently visible
- Is the currently selected cell (even if it isn't currently visible)
- Has editing in progress, for example, a text field the user is editing

Views that don't fit the in-use criteria are inserted into the reuse queue.

Because table view cells are instantiated from within the `NSTableView` instance in the Interface Builder editor, each cell is treated specially, as if it were its own nib. Each cell view has an owner. By default, a cell view's owner is the table view's default owner, which is usually the table's delegate.

When a new view for a specific ID is required, the table uses the process shown in Figure 1-5 to find and return the requested view.

Figure 1-5 View reuse logic


You need a view with ID "MainCell"
You call makeViewWithIdentifier:"MainCell"owner:self
"MainCell" in Reuse Queue?
Yes
Return an old view with ID "MainCell"
No
Is there a designed view with the "MyView" ID?
Yes
NSTableView loads the nibs, calls instanstiateNibWithOwner:self topLevelObjects:...
Return the new view with ID "MyView"
No
Return nil

By using the reuse queue, the table view is able to use memory effectively as well as increase the speed with which cell views are retrieved.

Note: The reuse queue doesn't reset any of the properties of the cell views. It's your responsibility to set all the properties of a cell view when the view is returned.

Constructing a Table View Using Interface Builder

Table views consist of many different views and objects, and as a result, they are best constructed within Interface Builder. It's also easier to create individual cells using Interface Builder rather than manually.

Create a Table View

The following steps describe how to create a table view in Interface Builder. (Use these steps whether you're creating an `NSView`-based table or an `NSCell`-based table.)

1. In Xcode, open the nib file to which you want to add the table view.
2. Drag an `NSTableView` object from the object library to the appropriate window or view on the canvas.
3. Position and configure the table view as appropriate.
4. In the Size inspector, set the table's resizing behavior with respect to the window and any other related objects.
5. In the Attributes inspector, add or remove columns.
6. Connect an `IBOutlet` in the appropriate class to the table view to allow for manipulating the table view and its contents.
7. Set the table view's delegate and data source to the appropriate objects (alternatively, you can use `setDelegate:` and `setDataSource:` to programmatically make these settings).

   If you don't associate the table view with its delegate, actions sent by objects in the table's cell views won't work.

Create Cells for Your Table

After you add a table view to your window, you need to create cells for it. Although you can create individual table view cells manually, it's far easier to create them in Interface Builder. The following steps provide the basics for creating cells within a table. (If you need to create cells for an `NSCell`-based table, see Working with NSCell-Based Table Views (page 39).)

1. Drag an `NSTableCellView` object (or a custom view) from the object library to the appropriate column in the table view.

Interface Builder provides two types of `NSTableCellView`: Image & Text Table Cell View and Text Table Cell View.

If you choose to use a custom view instead of one of the provided cell views, set its view class in the Attributes inspector. Typically, the view class is a subclass of `NSTableCellView`.

2. Repeat step 1 as many times as necessary to provide cells for every column.
3. (Optional) Set a custom identifier in the Identity area of the Identity inspector for each column.

   If your table is simple and you don't need to do much column management—or if you're using bindings—let Interface Builder automatically assign a unique identifier to each column. Using an automatic assignment can be convenient because it ensures that the column's cells have the same identifier and that these identifiers are in always in sync. For more information, see Columns and Cells Have Identifiers That Make It Easy to Find Them (page 15).

Populating a Table View Programmatically

To populate a table view programmatically, you must implement the `NSTableViewDataSource` and the `NSTableViewDelegate` protocols. Both protocols contain methods that are essential to providing the content and creating the cells for the table view. Specifically, you must implement:

- The `NSTableViewDataSource` protocol method `numberOfRowsInTableView:`, which tells the table how many rows it needs to display.
- The `NSTableViewDelegate Protocol` method `tableView:viewForTableColumn:row:`, which provides the table with the view to display in the cell at a specific column and row. It also populates that cell with the appropriate data.

Implementing the Data Source

The data source method is required for the table to work. Fortunately, the method's implementation is very straightforward.

Note: The following implementation assumes that the delegate and data source are implemented in the same class (which conforms to both protocols) and that this class stores the content in the `nameArray` property, which is an array of name strings.

Listing 3-1 shows a simple implementation of the data source method `numberOfRowsInTableView:`, which simply returns the number of items in the array of name strings.

Listing 3-1 A simple implementation of `numberOfRowsInTableView`

```
- (NSInteger)numberOfRowsInTableView:(NSTableView *)tableView {
   return nameArray.count;
}
```

Implementing the Delegate

The `NSTableViewDelegate` is responsible for providing the cell that's displayed in the table view, along with implementing any target-action or notification code so that it can interact with the delegate. What follows are two simple implementations of `tableView:viewForTableColumn:row:` that can be used to display the content of the `namesArray` model object. The first implementation uses a text field object for the cell of the table view; the second uses a custom table cell view that's been defined in Interface Builder.

In both of these simple cases no action is required by the cell. But if the cell view contained buttons, allowed editing, or was required to interact with the delegate, it would have to use target-action or notifications to work. For an example of implementing some of these actions, see the "Complex TableView" example in the *TableViewPlayground* sample project.

Creating and Configuring an NSTextField Cell

The `NSTableViewDelegate` method `tableView:viewForTableColumn:row:` creates and configures the cell that's displayed in the table view. The pseudocode in Listing 3-2 creates an `NSTextField` object as the cell and populates it with the appropriate name for the row. (In this example, the table has only one column.)

Listing 3-2 A simple implementation of `tableView:viewForTableColumn:row:` that returns a text field

```
- (NSView *)tableView:(NSTableView *)tableView
   viewForTableColumn:(NSTableColumn *)tableColumn
                  row:(NSInteger)row {

    // Get an existing cell with the MyView identifier if it exists
    NSTextField *result = [tableView makeViewWithIdentifier:@"MyView" owner:self];

    // There is no existing cell to reuse so create a new one
    if (result == nil) {

         // Create the new NSTextField with a frame of the {0,0} with the width
of the table.
         // Note that the height of the frame is not really relevant, because the
 row height will modify the height.
         result = [[NSTextField alloc] initWithFrame:...];

         // The identifier of the NSTextField instance is set to MyView.
         // This allows the cell to be reused.
```

```
            result.identifier = @"MyView";
        }

        // result is now guaranteed to be valid, either as a reused cell
        // or as a new cell, so set the stringValue of the cell to the
        // nameArray value at row
        result.stringValue = [self.nameArray objectAtIndex:row];

        // Return the result
        return result;

}
```

The code in Listing 3-2 first calls `makeViewWithIdentifier:owner:`, passing the identifier `@"MyView"` to determine whether there is a cell view available in the pool of resuable cells. If a resuable cell is available, the code assigns the specified row's `nameArray` value to the text field's `stringValue` property and returns the result.

> **Note:** Calling `makeViewWithIdentifier:owner:` causes `awakeFromNib` to be called multiple times in your app. This is because `makeViewWithIdentifier:owner:` loads a NIB with the passed-in owner, and the owner also receives an `awakeFromNib` call, even though it's already awake.

If no cell with the identifier `@”MyView”` is available, a new text field is created. The code sets the new text field's identifier to `@”MyView”` so that it can be reused when the opportunity arises. Finally, as in the first case, the `stringValue` of the text field is set to the correct `nameArray` value and the result is returned.

Getting a Table Cell View from Interface Builder

The most likely situation is that you'll have designed a cell in Interface Builder and will want to fetch it and then populate the values in that cell. The implementation for this situation is shown in Listing 3-3. This example assumes that there is already a table created in Interface Builder with a column identifier of `@”MyView”` and a cell view that also has the identifier `@”MyView”`.

Listing 3-3 A simple implementation of `tableView:viewForTableColumn:row:` that returns a table cell view

```
- (NSView *)tableView:(NSTableView *)tableView
```

```
   viewForTableColumn:(NSTableColumn *)tableColumn
                  row:(NSInteger)row {

      // Retrieve to get the @"MyView" from the pool or,
     // if no version is available in the pool, load the Interface Builder version
      NSTableCellView *result = [tableView makeViewWithIdentifier:@"MyView"
owner:self];

      // Set the stringValue of the cell's text field to the nameArray value at
row
      result.textField.stringValue = [self.nameArray objectAtIndex:row];

      // Return the result
      return result;
```

This implementation attempts to retrieve a cell view with the `@”MyView”` identifier from the pool of reusable cells. If there is no reusable view, the code looks in Interface Builder for the table column and cell with the `@”MyView”` identifier. Once found, the `makeViewWithIdentifier:owner:` method returns the cell view and the rest of the code sets the cell’s the string value.

Populating a Table View Using Cocoa Bindings

Populating a table view using Cocoa bindings is considered an advanced topic. Although using bindings requires significantly less code—in some cases no code at all—the bindings are hard to see and debug if you aren't familiar with the interface. It's strongly suggested that you become comfortable with the techniques for managing table views programmatically before you decide to use Cocoa bindings.

> **Note:** The information below is an illustration of the steps required for using the Interface Builder Bindings inspector to create bindings—it is not a tutorial. To learn more about using bindings in general, see *Cocoa Bindings Programming Topics* .

The following example assumes that your app's delegate has an array property called `peopleArray`, populated with instances of the `Person` class, which serves as the model object. The `Person` class is a subclass of `NSObject` and declares two properties, `name` and `image`. The table view is a single column table that displays an image adjacent to the name.

It's assumed that the delegate class, and a NIB with a single column table view, have already been created. The identifier for the column is Automatic, which is strongly recommended.

> **Note:** In an `NSView`-based table, you bind the table view's content to the array controller's `arrangedObjects`. This differs from the technique you use if you're working with an `NSCell`-based table. To learn more about using bindings with an `NSCell`-based table, see Creating Bindings for an NSCell-Based Table View (page 44).

Relate the Array Controller to the Table View

The content binding is all that's required to relate the controller to the table view. Regardless of how many columns the table has, or how complex the views are, if the model class has the fields to populate it, this binding makes the most important connection to the table view.

To create bindings for the person table view:

1. Select the MainMenu nib file in the File Manager view.

   This displays the Interface Builder editor, which already contains the table view and the `NSTableCellView`.

2. Drag an array controller from the Object library to the Objects dock.
3. Select the array controller in the Objects dock, and bind it to the model object array. This binding sets the `peopleArray` of the app delegate as the content array of the array controller.

Binding field	Value
Bind to:	Simple Bindings App Delegate ("Simple Bindings" is the example app name)
Model key path	`peopleArray`

4. Select the table view, and then select the Bindings inspector.
5. Configure the bindings for the table view's `Content` binding.

Binding field	Value
Bind to:	Names Array Controller
Controller key	`arrangedObjects`

Create Bindings for the Table's Subviews

When you make the connection to the table view's `Content` binding, the `objectValue` property of the `NSTableViewCell` used as the cell for the table column is configured such that for each item in the model array, the table updates the `objectValue` property to the current row's `Person` object.

1. Select the text field in the table column cell and create the binding.

   All bindings of subviews in the cell are made to the `objectValue`.

   Configure the `textField` instance's `value` binding.

Binding field	Value
Bind to:	Table Cell View
Model key path	`objectValue.name`

2. Select the image view in the table column cell, and create the binding to the `Value` binding.

   All bindings of subviews in the cell are made to the `objectValue`.

   Configure the `imageView` instance's `value` binding.

Binding field	Value
Bind to:	Table Cell View

Binding field	Value
Model key path	`objectValue.image`

Modifying a Table's Visual Attributes

A table's appearance is highly customizable. You can customize the color of the background, rows, and grid lines, the selection highlight style, and the row height. In most cases, you can use either table view methods or Interface Builder settings to customize a table's appearance.

Background Color

The background color specifies the color OS X uses to draw the background of the table view. To set the background color of a table view in Interface Builder, use the Table View Attributes inspector. To programmatically set the background color, use the `setBackgroundColor:` and `backgroundColor` methods.

If your app needs to display a transparent table view, set the background color of the table view to be clear and set the enclosing scroll view to not draw its background. The following code snippet shows one way to display a transparent table:

```
[theTableView setBackgroundColor:[NSColor clearColor]];
[[theTableView enclosingScrollView] setDrawsBackground:NO];
```

By default, a table view uses the `NSCompositeSourceOver` style when drawing the background color. The default background color is returned by the `NSColor` method `controlBackgroundColor`. Because this color is defined by the system, any view or control that needs to draw its background will do so in a consistent manner.

Alternating Row Colors

Using alternating colors can make it easier for users to visually match the data in one column with the data in another column. The Alternating Rows checkbox in the Table View Attributes inspector lets you specify whether all table rows use the same background color, or if alternating rows use the color returned by the `NSColor` method `controlAlternatingRowBackgroundColors`. To set alternating background colors programmatically, use `setUsesAlternatingRowBackgroundColors:` and `usesAlternatingRowBackgroundColors`.

Selection Highlight Style

The Highlight pop-up menu in the Table View Attributes inspector lets you choose the highlighting style used when users select row items. Setting the Highlight attribute provides a different background color for the table view and a corresponding highlight color. You can choose None, Regular, or Source List (the Finder sidebar and the iTunes playlist view both use a source list).

To achieve the same behavior programmatically, implement the following code fragment:

```
[theTableView setSelectionHighlightStyle:
NSTableViewSelectionHighlightStyleSourceList]
```

This code fragment sets the background color and highlight colors that are appropriate for the source list style. To specify the regular highlighting style, use the same method, but pass `NSTableViewSelectionHighlightStyleRegular` as the parameter.

Grid Color and Style

Use the Horizontal and Vertical Grid pop-up menus in the Table View Attributes inspector to enable or disable the drawing of grid lines between columns and rows. To set the grid line color, use the Grid Color pop-up menu.

To set grid lines programmatically, use the `setGridStyleMask:` method with the constants `NSTableViewSolidHorizontalGridLineMask`, `NSTableViewSolidVerticalGridLineMask` and `NSTableViewDashedHorizontalGridLineMask`. To display both horizontal and vertical lines, combine the constants using the bitwise C operator. To disable grid line drawing entirely, pass `NSTableViewGridNone` as the parameter.

To programmatically set and retrieve the color of the grid lines, use the `setGridColor:` and `gridColor` methods.

Row Height

The row height, as expected, specifies the height of a table view's rows. This value can be set using the method `setRowHeight:` and retrieved using the method `rowHeight`.

When `setRowHeight:` is called, the table view invokes the `tile` method. The `tile` method properly resizes the rows and the header view, and it marks the table view as needing display. Another effect of setting the row height is that it modifies the line scroll amount in the enclosing scroll view so that scrolling by line continues to provide the expected results.

The `NSTableViewDelegate` method `tableView:heightOfRow:` allows you to customize the height of rows on a row-by-row basis.

Enabling Row Selection and User Actions

Displaying a list of data in a table view is of little use if the user can't select the rows. Using the methods of the `NSTableView` class, you let users select one or more rows, drag to change a selection, and type to select. In addition, you can enable programmatic selection and display a contextual menu that's associated with the table.

Getting the Current Row Selection

The `NSTableView` class provides several methods for getting information about a table view's currently selected rows: `selectedRowIndexes`, `selectedRow`, `numberOfSelectedRows`, and `isRowSelected:`.

The `selectedRowIndexes` method provides the full and correct selection whether a single item is selected or multiple items are selected. The method returns an `NSIndexSet` object containing the indexes of the selected rows.

The `selectedRow` method returns the index of only the last selected row, or -1 if no row is selected. Using this method is the easiest way to get the current selection when multiple selection is disabled.

The `numberOfSelectedRows` method, which returns the number of selected rows, can be used as a shortcut to determine whether any objects are selected when empty selection is permitted. Its value can also be used to determine whether user interface items should be enabled or disabled if they depend upon multiple items being selected in the table view. For example, if the `numberOfSelectedRows` is greater than 1, you can disable the portions of the user interface that are relevant only when a single item is selected.

The `isRowSelected:` method allows an app to find out whether a row at a specific index is selected.

Apps often need to iterate over the selected rows. You can implement this by iterating over the contents of the selection returned by `selectedRowIndexes` or by using the `NSIndexSet` block enumeration method `enumerateIndexesUsingBlock:`. For more information about using these techniques, see Iterating Through Index Sets in *Collections Programming Topics* .

Responding to Changes in Row Selection

Users can select multiple rows using the Shift key (for continuous selections) or the Command key (for non-contiguous selections). For more information about user selection actions, see Make User Input Convenient in *OS X Human Interface Guidelines* .

As the user selects rows by dragging (using a trackpad or mouse), delegate methods can be informed of the changes in row selection. The delegate receives the following messages as changes in the selection occurs by the user. (Delegates for table views that are managed by Cocoa bindings also receive these messages and can act on them as required.) An app can receive these messages on a continuous basis, so it's important to create efficient implementations.

- `selectionShouldChangeInTableView:`

  This method is usually implemented when it's necessary to perform validation on editing of a row before allowing the row selection to change.

  For example, if the user is editing a row in the table view and the content of that row doesn't meet the required criteria, the delegate should return `NO` from this method to prevent the user from changing the selection. If the action was denied, this method is responsible for telling users why the selection change was disallowed and what action they can take to rectify the situation. By default, this method returns `YES`, which allows the selection to be changed.

- `tableViewSelectionIsChanging:`

  This method informs the delegate that the row selection is about to change due to interaction with the mouse or trackpad. Keyboard selection does not invoke this method.

- `tableViewSelectionDidChange:`

  This method informs the delegate that the row selection has changed and that, effectively, the table view selection action is completed. The method is sent to the delegate when the user releases the mouse button, whether selection is limited to a single row or multiple selection is enabled.

The `tableViewSelectionIsChanging:` and `tableViewSelectionDidChange:` messages are notifications. Each is passed an `NSNotification` object. Sending the notification instance an `object` message returns the table view relevant to the selection change.

A table view delegate can also implement the `tableView:selectionIndexesForProposedSelection:` delegate method to allow or disallow the selection of a specific set of rows in a table view. The method is passed the indexes of the rows that are proposed to be selected and it returns the rows that will actually be selected. Implementing this method allows the app to selectively allow and disallow row selection as appropriate. For example, if the user clicks in an area of the table row that shouldn't trigger selection, this method can be used to prevent that selection from taking place.

Allowing Multiple and Empty Selection

A table view can be configure selection in three ways:

- Allow a single row to be selected at once
- Allow multiple rows to be selected simultaneously
- Attempt to prevent there from being an empty selection (that is, at least one row is always selected)

These attributes can be configured in Interface Builder or programmatically.

Programmatically, the table view methods for enabling and disabling these options are set using the following methods: `setAllowsMultipleSelection:` and `setAllowsEmptySelection:`.

Selecting and Deselecting Rows Programmatically

To select rows programmatically, the `NSTableView` class provides the `selectRowIndexes:byExtendingSelection:` instance method.

The `selectRowIndexes:byExtendingSelection:` method expects an `NSIndexSet` containing the indexes (zero-based) of the rows to be selected, and a parameter that specifies whether the current selection should be extended. If the extending selection parameter is `YES`, the specified row indexes are selected in addition to any previously selected rows; if it's `NO`, the selection is changed to the newly specified rows. When this method is called, the delegate receives only the `tableViewSelectionDidChange:` notification.

To deselect a row, pass the index of the row to deselect to `deselectRow:`. When this method is called the delegate receives only the `tableViewSelectionDidChange:` notification.

There are also two convenience methods that allow the selection and deselection of all the items in the table view. In general, these methods are connected to the user interface so that users can select or deselect all items in a table. The `deselectAll:` method deselects all the selected rows, but only if `allowsEmptySelection` returns `YES`. Similarly, `selectAll:` selects all the table view rows, but only if `allowsMultipleSelection` returns `YES`. Unlike the other programmatic methods, these two methods call `selectionShouldChangeInTableView:` on the delegate object, if implemented, followed by `tableViewSelectionDidChange:`. If either `deselectAll:` or `selectAll:` is called without the proper `allows...` setting, it is ignored.

Note: Your app is responsible for ensuring the validity of the contents of the currently selected row if it is being edited. The reason is that when the row is changed programmatically, the `selectionShouldChangeInTableView:` message is not sent to the delegate.

Using Type Selection to Select Rows

To simplify navigation in tables or to allow a user to select items using the keyboard, a table view can support type selection. Using type selection, a user types the first few letters of an entry and the table view content is searched for a matching row. You can use the `setAllowsTypeSelect:` method to enable or disable type selection (by default, type selection is enabled).

Type selection uses the delegate method `tableView:typeSelectStringForTableColumn:row:` to figure out which rows to match against. This delegate method returns a string that type selection uses for matching.

When the `tableView:typeSelectStringForTableColumn:row:` method is not implemented by the delegate, the behavior is to call `preparedCellAtColumn:row:`, passing the column index and row and then returning the `stringValue`. This allows type selection to work on the values in any column and row in the table view. To restrict type selection to specific parts of the table, return `nil` for columns that should be excluded. For example, the following method implementation allows type selection to match values only in the "name" column.

```
- (NSString *)tableView:(NSTableView *)tableView
typeSelectStringForTableColumn:(NSTableColumn *)tableColumn


row:(NSInteger)row
{
    if ([[tableColumn identifier] isEqualToString:@"name"])
    {
        NSUInteger tableColumnIndex=[[tableView tableColumns]
indexOfObject:tableColumn];
        return [[tableView preparedCellAtColumn:tableColumnIndex
                                            row:row] stringValue];
    }
    return nil;
}
```

Implementing the delegate method `tableView:nextTypeSelectMatchFromRow:toRow:forString:` allows the delegate to further customize type selection to match only a specific range of rows. This method is passed the current type selection string, and it compares the appropriate values within the selected rows, returning the row to select, or -1 if no row matches.

Finally, the `tableView:shouldTypeSelectForEvent:withCurrentSearchString:` method gives the delegate the option to allow or disallow type selection for a particular keyboard event. You can implement this method to prevent certain characters from causing type selection. Note that returning `NO` from this method prevents the specified event from being used for type selection; it doesn't prevent standard event processing. Don't use `tableView:shouldTypeSelectForEvent:withCurrentSearchString:` to handle your own keyboard shortcuts. If you need to provide custom keyboard shortcut handling, override `keyDown:` instead.

Displaying a Contextual Menu in a Table

You can use a contextual menu to offer users a convenient way to access a small set of commands that act on one or more items in a table. For example, when users Command-click a song listed in iTunes, a contextual menu appears that makes it easy to (among other things) play, copy, or delete the song.

An easy way to add a contextual menu to a table is to set the `menu` outlet of the table to an `NSMenu` object. And if you want to customize the contextual menu, you set an appropriate object as the menu's delegate and implement the `menuWillOpen:` method to customize the menu before it appears.

In the action method for a menu item, determine whether the clicked table row is in the set of indexes returned by `selectedRowIndexes`. If it is, apply the action to all indexes in the set; otherwise, apply the action only to the clicked row. Here's a convenience method that checks the selected row indexes and returns the set of indexes the action method should process:

```
- (NSIndexSet *)_indexesToProcessForContextMenu {
    NSIndexSet *selectedIndexes = [_tableViewMain selectedRowIndexes];
    // If the clicked row is in selectedIndexes, then process all selectedIndexes.
 Otherwise, process only clickedRow.
    if ([_tableViewMain clickedRow] != -1 && ![selectedIndexes
containsIndex:[_tableViewMain clickedRow]]) {
        selectedIndexes = [NSIndexSet indexSetWithIndex:[_tableViewMain clickedRow]];
    }
    return selectedIndexes;
}
```

To see an example that uses the `_indexesToProcessForContextMenu` method, download the *TableViewPlayground* sample project and look at the `mnuRevealInFinderSelected:` method in `ATComplexTableViewController.m`.

Specifying How Subviews Should Respond to Events

Views or controls in a table sometimes need to respond to incoming events. To determine whether a particular subview should receive the current mouse event, a table view calls `validateProposedFirstResponder:forEvent:` in its implementation of `hitTest`. If you create a table view subclass, you can override `validateProposedFirstResponder:forEvent:` to specify which views can become the first responder. In this way, you receive mouse events.

The default `NSTableView` implementation of `validateProposedFirstResponder:forEvent:` uses the following logic:

1. Return `YES` for all proposed first responder views unless they are instances or subclasses of `NSControl`.
2. Determine whether the proposed first responder is an `NSControl` instance or subclass.
   - If the control is an `NSButton` object, return `YES`.
   - If the control is not an `NSButton`, call the control's `hitTestForEvent:inRect:ofView:` to see whether the hit area is trackable (that is, `NSCellHitTrackableArea`) or is an editable text area (that is, `NSCellHitEditableTextArea`), and return the appropriate value. Note that if a text area is hit, `NSTableView` also delays the first responder action.

Sorting Table View Rows

Table views can be configured so that users can click on a column's header to toggle scrolling from none to ascending or descending order. You use sort descriptors to enable this behavior.

Sorting a Table View Programmatically

Your app can specify a default sort descriptor to use when displaying the table view data. It does this using the `NSTableViewsetSortDescriptors:` method. The sort descriptors are an array of `NSSortDescriptors` instances. For example, the following code sorts the content based on the last name and first name:

```
    [theTableView setSortDescriptors:[NSArray arrayWithObjects:
                                     [NSSortDescriptor
sortDescriptorWithKey:@"lastName" ascending:YES selector:@selector(compare:)],
                                     [NSSortDescriptor
sortDescriptorWithKey:@"firstName" ascending:YES selector:@selector(compare:)],
                                     nil]];
```

This code snippet sorts the content indirectly through the delegate callback (see Responding to a Sort Request (page 38)) but won't allow a user to click in a table header to change the sort order for that column. To give users this capability, you need to configure sorting on a per-column basis. In a table configured for per-column sorting, the user's click on a table header modifies the table view's sort descriptor to reflect the current sort state of the table view (you can use the `sortDescriptors` method to access the sort descriptor).

Configuring Sorting for Individual Table Columns

You can sort individual table columns programmatically by specifying an `NSSortDescriptor` prototype instance for the appropriate `NSTableColumn` instance, or by using Interface Builder.

Configuring Per-Column Sorting in Code

You can set a sort method for a column programmatically by creating a sort descriptor prototype. You create the prototype by using the method `sortDescriptorWithKey:ascending:selector:` to create an `NSSortDescriptor` instance. The resulting sort descriptor instance is then set as the table column's sorting prototype using the method `setSortDescriptorPrototype:`. When a user clicks the header, the table view's sort descriptors are modified and the table column header reflects the sort direction.

The following code creates a sort descriptor prototype for a `lastName` column using localized sorting:

```
NSTableColumn *tableColumn = [theTableView tableColumnWithIdentifier:@"lastName"];
NSSortDescriptor *lastNameSortDescriptor = [NSSortDescriptor
sortDescriptorWithKey:@"lastName"

ascending:YES

selector:@selector(compare:)];
[tableColumn setSortDescriptorPrototype:lastNameSortDescriptor];
```

Note: Because OS X is a multilingual operating system, you should also consider using `localizedCompare:` or `localizedCaseInsensitiveCompare:` for sorting strings. These methods compare strings using the methodology appropriate to the user's language selection and are preferred to using the simple `compare...:` methods.

Configuring Per-Column Sorting in Interface Builder

To use Interface Builder to configure per-column sorting, select a table column and open the Table Column Attributes inspector. The inspector contains three controls that help you configure sorting behavior:

- The Sort Key field specifies the key in the table view data model that the column sorts on.
- The Selector field contains the selector that the sort uses. Tabbing to the field causes the field to fill with the `compare:` selector.

  This comparison selector expects a single parameter—the value that needs to be sorted—and returns an `NSComparisonResult`. You can substitute any method selector that adheres to that pattern, including custom methods of your own.

  The `compare:` method works with `NSString`, `NSDate`, and `NSNumber` objects. If your table column contains only strings, you may want to consider using the `caseInsensitiveCompare:` method if case sensitivity is unimportant. However, consider replacing these method signatures with the `localizedCompare:` or `localizedCaseInsensitiveCompare:` methods to take into the account the user's language requirements.

- The Order pop-up menu allows you to specify the initial order of the sort, as ascending (default) or descending.

Leaving the Sort Key and Selector fields blank disables sorting of the column using the settings within Interface Builder. However, it does not disable sorting of the column using other techniques.

Responding to a Sort Request

When a table view is sorted through user action or by setting the table view's sort descriptors, the data source receives a `tableView:sortDescriptorsDidChange:` message. The data source must implement this method to support sorting.

The `tableView:sortDescriptorsDidChange:` method applies the table view's current sort descriptors to the model collection, and then reloads the table view data. A typical implementation of this delegate method is shown in Listing 7-1.

Listing 7-1 Sample implementation of the delegate method `tableView:sortDescriptorsDidChange:`

```
- (void)tableView:(NSTableView *)tableView sortDescriptorsDidChange:(NSArray
*)oldDescriptors
{
    [peopleArray sortUsingDescriptors: [tableView sortDescriptors]];
    [tableView reloadData];
}
```

The implementation shown in Listing 7-1 sorts the current model content using the table view sort descriptors, but it also loses the initial order of the data. If the initial order is important to your app, you'll want to take steps to cache the original order so that it can be restored.

Important: To sort a table—either on a per-column basis or for the entire table view using `setSortDescriptors:`—the table view data source must implement the `tableView:sortDescriptorsDidChange:` method.

Working with NSCell-Based Table Views

Although most tables use `NSView` subclasses for cells, you might need to work with a table that instead uses `NSCell` subclasses. Working with an `NSCell`-based table is similar to working with an `NSView`-based table, but there are differences in how you add cells to the table and how you use Cocoa bindings.

Creating an NSCell-Based Table in Interface Builder

The steps you take to create an `NSCell`-based table in Interface Builder are identical to the steps you take to create an `NSView`-based table. For specifics, see Constructing a Table View Using Interface Builder (page 18).

After you add an `NSCell`-based table to your user interface, you can add individual cells either manually or using Interface Builder. To use Interface Builder to add cells:

1. Drag the correct cell type from the object library to the appropriate column within the table view.

   Only subclasses of `NSCell` can be inserted into `NSCell`-based table views.
2. Repeat step 1 as often as necessary to provide cells for every column.
3. Consider setting an identifier for each column. Set the identifier in the Identity area of the Identity inspector.

   Setting a column identifier makes populating and retrieving columns substantially easier.

Note: Make sure that your custom cells properly implement the `NSCopying` protocol because a cell is copied when tracking begins. (When tracking stops, the cell is released.)

The Role of a Table View Data Source Object

`NSCell`-based table views are similar to `NSView`-based table views: They have a data source that adopts the `NSTableViewDataSource` protocol, and an optional delegate that adopts the `NSTableViewDelegate Protocol` to allow custome table view behavior.

To tell the table view how many rows it has to display, `NSCell`-based table views must implement the `numberOfRowsInTableView:NSTableViewDataSource` method. It must also implement the `tableView:objectValueForTableColumn:row:` method that returns the data displayed in each column cell. If the table view data is editable, then a method to allow the updating the model values can be implemented. Note that when using Cocoa bindings for populating an `NSCell`-based table view, the data source methods required for programmatic populating are unnecessary.

A table view class that conforms to the data source protocol is responsible for:

- Providing the number of rows in the table view
- Providing the content for each item within the table view
- Responding to edit requests if the table view supports editing
- Receiving notifications that the information used when sorting the table view has changed
- Providing pasteboard data for supporting drag and drop within your own app as well as within other apps

Although the `NSTableViewDelegate Protocol` declares all the data-providing methods to be optional, all data sources that programmatically populate `NSCell`-based table views must implement those methods. The methods that provide the number of rows and the content for each item within the table are optional when using Cocoa bindings.

Note: Sometimes an app uses Cocoa bindings to populate most of the table view's columns, but not all. In that situation you become responsible for implementing the data source methods to provide and edit content for the columns that aren't managed by Cocoa bindings. For more information, see Populating a Table View Programmatically (page 20).

If your app uses a data source, the data source class must adopt the `NSTableViewDataSource` protocol and implement some or all of the methods defined by the protocol. Even though all the methods in the data source are optional, the `numberOfRowsInTableView:` method is not optional if you are not using Cocoa bindings.

Providing Data to a Table View Programmatically

To populate a table view programmatically, use this process:

- Create a data source class that adopts the `NSTableViewDataSource` protocol. Note that you don't have to create a separate data source class; instead, you can integrate this functionality into a another class, such as the class that also implements the delegate methods.
- Implement the `numberOfRowsInTableView:` and `tableView:objectValueForTableColumn:row:` methods that provide the data.
- If the table view allows the user to edit the data, implement the `tableView:setObjectValue:forTableColumn:row:` method.

- Your app must also set the object as the data source of the table view. (Do this in Interface Builder or in code, using the `NSTableView` instance method `setDataSource:`.)

> **Note:** Forgetting to set the data source is a step that is often missed. If your table view doesn't populate after sending it the `reloadData` message, double-check that `dataSource` is set in your code or in Interface Builder.

Populating a Sample Table

The following example populates a simple table view and enables editing. The table consists of a single column called "Names".

The table column has had its column identifier set to `name`. This can be done in Interface Builder by selecting the table column, opening the Table Column Attributes inspector, and setting the identifier string. It can also be done programmatically by invoking `setIdentifier:` on a table column, passing the column name. The table column identifier is a great convenience when using `NSCell`-based table views with multiple columns (and it's an absolute necessity when using `NSView`-based table views). To access a table column instance, simply ask the table view for the column by using the identifier string. In this way you eliminate the need to cache the table columns for comparison. Setting the column identifier is a simple practice that you should always perform.

The code fragment in Listing 8-1 shows the implementation of the `numberOfRowsInTableView:` data source method. For this simple code snippet, it's assumed that the app's delegate is acting as the data source object of the table view and that the sample data is a simple array of strings stored in a property declared as `namesArray`.

Listing 8-1 Example `numberOfRowsInTableView:` implementation

```
- (NSInteger)numberOfRowsInTableView:(NSTableView *)tableView
{
    // This is a defensive move
    // There are cases where the nib containing the NSTableView can be loaded
before the data is populated
    // by ensuring the count value is 0 and checking to see if the namesArray is
not nil, the app
    // is protecting itself agains that situation
    NSInteger count=0;
    if (self.namesArray)
```

```
        count=[self.namesArray count];
    return count;
}
```

The `numberOfRowsInTableView:` implementation returns the number of objects in `namesArray`. First it does some defensive programming. It declares a `count` variable and sets it to `0` and then ensures that `self.namesArray` is not `nil`. This is a good practice because sometimes a NIB containing a table view may be loaded and sent a `reload` message before the data itself has been initialized. Because this method returns a scalar, an exception may occur. By performing this simple test, your app can ensure that the data has been loaded and is available before any attempt is made to display the content.

Having implemented the method that returns the number of items in the table view, you must now implement the method that populates those rows. The `tableView:objectValueForTableColumn:row:` method returns the appropriate value to display for the requested row and column of the specified `NSTableView`. You can see in Listing 8-2 an example of why the table column identifier is so useful: The identity of the column that requires population is sent the `identifier` message and the name returned is compared to determine the data to provide.

Listing 8-2 Example `tableView:objectValueForTableColumn:row:` sample implementation

```
- (id)tableView:(NSTableView *)aTableView objectValueForTableColumn:(NSTableColumn
 *)aTableColumn row:(NSInteger)rowIndex
{
    // The return value is typed as (id) because it will return a string in most
cases.
    id returnValue=nil;

    // The column identifier string is the easiest way to identify a table column.
    NSString *columnIdentifer = [aTableColumn identifier];

    // Get the name at the specified row in namesArray
    NSString *theName = [namesArray objectAtIndex:rowIndex];



    // Compare each column identifier and set the return value to
    // the Person field value appropriate for the column.
    if ([columnIdentifer isEqualToString:@"name"]) {
        returnValue = theName;
    }



    return returnValue;
}
```

In Listing 8-2, `returnValue` is a local variable for storing the value returned by the method. The return value is of type `NSString` because all the column contents are strings. If the table had a combination of `NSTextFieldCell`, `NSImageCell`, and so on, the `returnValue` would typically be typed more generically, as `id`.

Next, the object assigns `anObject` to the object at the `rowIndex` offset in the `namesArray` property. Because there is a one-to-one correlation between a row and the array index for the corresponding value, accessing the object for a row is straightforward.

Finally, the `returnValue` variable is set to the value that's displayed in the column. To eliminate code and simplify returning the data, it takes advantage of key-value coding.

Note: Because the `numberOfRowsInTableView:` and `tableView:objectForTableColumn:row:` methods are called often, they must be efficient.

Programmatically Editing Data in an NSCell-Based Table

For your app to edit the content of an `NSCell`-based table view, you implement the `tableView:setObjectValue:forTableColumn:row:` data source protocol method. This method is similar to `tableView:objectValueForTableColumn:row:`, which provides the data for the table view, but instead of requesting that you return a value for the specified row and column, it provides the new value for that row and cell.

Listing 8-3 shows an implementation of the `tableView:setObjectValue:forTableColumn:row:` method.

Listing 8-3 Example `tableView:setObjectValue:forTableColumn:row:` implementation

```
- (void)tableView:(NSTableView *)aTableView setObjectValue:(id)anObject
forTableColumn:(NSTableColumn *)aTableColumn row:(NSInteger)rowIndex
{
    // The column identifier string is the easiest way to identify a table column.
    NSString *columnIdentifer = [aTableColumn identifier];

    if ([columnIdentifer isEqualToString:@"name"]) {
        [namesArray replaceObjectAtIndex:rowIndex withObject:anObject];
    }

}
```

The example implementation in Listing 8-3 examines the column `identifier` method result to ensure that it's the "Names" table column that's being edited. This is a good practice regardless of the number of columns in a table. After verifying the column's identity, the code replaces the object in the `namesArray` with the new value.

Creating Bindings for an NSCell-Based Table View

The Cocoa bindings technique used in `NSView`-based tables differs significantly from that used in `NSCell`-based tables. In an `NSCell`-based table you bind the table column's content binding to the array controller's `arrangedObjects`, and then configure the column's cell's bindings. You never bind directly to the table view's content binding.

Document Revision History

This table describes the changes to *Table View Programming Guide for Mac* .

Date	Notes
2014-07-15	Updated the Responding to a Sort Request section.
2013-12-16	Added information about displaying a contextual menu in a table, and reorganized content to emphasize NSView-based tables.
2013-08-08	Corrected typos and linked unlinked classes.