How to design graphical applications with Eclipse 3.0

SWT/JFace IN ACTION

Matthew Scarpino
Stephen Holder
Stanford Ng
Laurent Mihalkovic

MANNING

SWT/JFace in Action

Chapter 9

MATTHEW SCARPINO
STEPHEN HOLDER
STANFORD NG
AND LAURENT MIHALKOVIC

MANNING

Greenwich
(74° w. long.)

For online information and ordering of this and other Manning books, please go to www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact:

Special Sales Department
Manning Publications Co.
209 Bruce Park Avenue Fax: (203) 661-9018
Greenwich, CT 06830 email: orders@manning.com

♾ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books they publish printed on acid-free paper, and we exert our best efforts to that end.

Manning Publications Co.
209 Bruce Park Avenue
Greenwich, CT 06830

Copyeditor: Tiffany Taylor
Typesetter: Tony Roberts
Cover designer: Leslie Haimes

ISBN 1-932394-27-3

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – VHG – 08 07 06 05 04

brief contents

9 Tables and menus

This chapter covers

- SWT tables
- JFace tables
- Editing table data
- Creating menus

Just about every time we want to go out to eat, we find ourselves sitting in the car, wracking our brains as we try to think of somewhere to go. We end up naming different styles of food—"Japanese?" "Not bad, but not really what I'm in the mood for." "Italian?" "Not tonight." "Indian?" "That's a good idea, but let's keep thinking." Especially when we're hungry, we have a hard time thinking about what restaurants are nearby and coming up with good options.

Eventually, we came up with a plan: One afternoon, when we weren't hungry and had time to think, we wrote up a list of restaurants in the area, organized by price and type of food. Now, when we decide to go out, we can look at the list and have concrete options to discuss. It doesn't help when we're in the mood for different things, but it makes the process of deciding where to go easier.

In a software application, a menu provides a function similar to our list of restaurants. A finite list of options is presented to users to guide them in deciding what tasks they wish to perform. Just as we sometimes rediscover a favorite place to eat that we haven't visited in a while, users can discover functionality they didn't know existed in your application by seeing it listed in a pull-down or context menu.

We'll cover two tasks in this chapter. First, we'll continue our discussion of the Viewer framework from the previous chapter by covering the last of the basic viewer widgets, the table. The concepts you've already learned are just as applicable to tables as they were to trees and lists, but JFace also provides advanced options in the form of cell editors to make it easy to implement user-editable tables. Once you're familiar with the editing framework, we'll revisit the `Actions` we discussed in chapter 4 and show how to apply them to the creation of menus, so that you can present functions to your users instead of leaving them to guess or remember what your application is capable of. Finally, our example in this chapter shows how to apply a context menu to a table by presenting a small user-editable widget that could be used to edit data in a relational database.

9.1 Tables

To the user, a table looks like a two-dimensional grid composed of many cells. Often this is a convenient way to display items such as the result of a database query—each row of the result set maps nicely to a single row in the table. As you'll see, however, JFace provides advanced facilities for editing table data as well.

9.1.1 Understanding SWT tables

Continuing SWT's trend of intuitive widget names, a table is represented by a class named `Table`. The `Table` class isn't terribly interesting. In general, if you're using

JFace, you'll be better off interacting with a `Table` through the interface provided by a `TableViewer`, which we discuss later in the chapter. However, if you need to manipulate the currently selected table items directly, or you aren't using JFace, you'll need to use the underlying `Table`.

The first thing you'll notice when looking at the methods available on `Table` is that although there are plenty of accessor methods to query its state, there is a distinct lack of setters that would let you customize the `Table`. In fact, rather than adding data or columns directly to the `Table`, you'll pass a `Table` instance to the appropriate dependent class when that dependent is instantiated, similar to the way `Composites` are passed to other widgets rather than the widget being added to the `Composite`. Other than a few setters for straightforward display properties, such as header visibility, the critical methods to be aware of when manipulating a `Table` are summarized in table 9.1.

Table 9.1 Important `Table` methods

Method	Description
`addSelectionListener()`	Notifies you when the table's selection changes
`select()/deselect()`	Overloaded in several ways to let you programmatically add or remove the selection on one or more items
`getSelection()`	Retrieves an array of the currently selected items
`remove()`	Removes items from the table
`showItem()/showSelection()`	Forces the table to scroll until the item or selection is visible

It's also important to remember that `Table` extends `Scrollable` and will therefore automatically come equipped with scrollbars unless you turn them off.

TableItems

To add data to a table, you must use a `TableItem`. Each instance of `TableItem` represents an entire row in the table. Each `TableItem` is responsible for controlling the text and image to display in each column of its row. These values can be set using the `setText()` and `setImage()` methods, each of which takes an integer parameter designating which column to modify.

As we mentioned, `TableItems` are associated with a `Table` in their constructor, as shown here:

```
Table t = ...
//Create a new TableItem with the parent Table
//and a style
```

```
TableItem item = new TableItem(t, SWT.NONE);
item.setText(0, "Hello World!");
...
```

According to the Javadocs, no styles are valid to be set on a `TableItem`, but the constructor accepts a style parameter anyway. This seems rather unnecessary to us, but it's at least consistent with the other widgets we've seen.

TableColumn

The final class you'll need to work directly with tables is `TableColumn`, which creates an individual column in the table. As with `TableItem`, you must pass a `Table` to the constructor of `TableColumn` in order to associate the two objects.

Each `TableColumn` instance controls one column in the table. It's necessary to instantiate the `TableColumns` you need, or the `Table` will default to having only one column. Several methods are available to control the behavior and appearance of each column, such as the width, alignment of text, and whether the column is resizable. You can add header text by using the `setText()` method. Instead of setting the attributes directly on a column, however, it's usually easier to use a `TableLayout`. By calling `TableLayout`'s `addColumnData()` method, you can easily describe the appearance of each column in the table. The ability to pass `addColumnData()` instances of `ColumnWeightData` is key; doing so lets you specify a relative weight for each column without having to worry about the exact number of pixels required for each one.

The following snippet shows how to create a table using a `TableLayout`. The code creates three columns of equal width and fills two rows with data. The code produces a table that looks similar to figure 9.1.

```
//Set up the table layout
TableLayout layout = new TableLayout();
layout.addColumnData(new ColumnWeightData(33, 75, true));
layout.addColumnData(new ColumnWeightData(33, 75, true));
layout.addColumnData(new ColumnWeightData(33, 75, true));

Table table = new Table(parent, SWT.SINGLE);
table.setLayout(layout);

//Add columns to the table
TableColumn column1 = new TableColumn(table, SWT.CENTER);
TableColumn column2 = new TableColumn(table, SWT.CENTER);
TableColumn column3 = new TableColumn(table, SWT.CENTER);

TableItem item = new TableItem(table, SWT.NONE);
item.setText( new String[] { "column 1",
                             "column 2",
                             "column 3" } );
item = new TableItem(table, SWT.NONE);
item.setText( new String[] { "a", "b", "c" } );
```

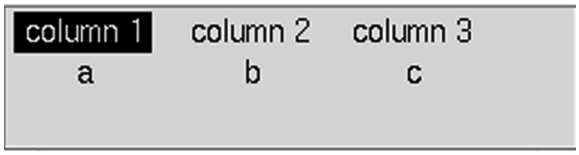

Figure 9.1
A simple three-column table

The first thing to do is set up the structure for this table using a `TableLayout`. Each time you call `addColumnData()`, it adds a new column to the table. We'll have three columns, so we add a `ColumnWeightData` to describe each. The parameters to the constructor that we use here are `weight`, `minimumWidth`, and `resizeable`. `weight` indicates the amount of screen space this column should be allocated, as a percentage of the total space available to the table. `minimumWidth` is, as the name indicates, the minimum width in pixels to use for this column. The `resizeable` flag determines whether the user can resize this column.

After we've set up the table, we need to instantiate three columns so they will be added to the table. It's important to keep in mind that adding columns is a two-step process: create a `TableLayout` that describes how large each column will be, and then create the columns themselves. Because we allow the `TableLayout` to control sizing, we don't need to use the columns after they've been created.

9.1.2 JFace TableViewers

Although it's possible to use a `Table` directly in your code, as you can see, doing so is neither intuitive nor convenient. Similarly to `List`, however, JFace provides a viewer class to make using tables easier. The following snippets demonstrate a basic `TableViewer` that displays data from a database. The same concepts of filters, sorters, and label providers that we discussed in chapter 8 apply here as well. Additionally, we'll use a `ContentProvider` to supply the data to our table, because the same arguments presented in the previous chapter apply here.

First, the table must be set up. This is similar to the process of setting up a `Table`, which you saw in the previous section, using `addColumnData()` for each column that will be created:

```
final TableViewer viewer = new TableViewer(parent,
                           SWT.BORDER | SWT.FULL_SELECTION);

//configure the table for display
TableLayout layout = new TableLayout();
layout.addColumnData(new ColumnWeightData(33, true));
layout.addColumnData(new ColumnWeightData(33, true));
layout.addColumnData(new ColumnWeightData(33, true));

viewer.getTable().setLayout(layout);
```

```
viewer.getTable().setLinesVisible(true);
viewer.getTable().setHeaderVisible(true);
```

Once the table has been configured, we attach the appropriate providers. The most important one in this example is the content provider, which is responsible for retrieving data from the database and passing it back to the viewer. Note that you never return `null` from `getElements()`—instead, return an empty array if there are no more children:

```
viewer.setContentProvider(new IStructuredContentProvider() {
  public Object[] getElements(Object input)
  {
    //Cast input appropriately and perform a database query
    ...
    while( results.next() )
    {
      //read results from database
    }
    if(resultCollection.size() > 0)
    {
      return new DBRow[] { ... };
    }
    else
    {
      return new Object[0];
    }
  }

  //... additional interface methods
});

viewer.setLabelProvider(new ITableLabelProvider() {
  public String getColumnText(Object element, int index) {
      DBRow row = (DBRow)element;
      switch(index)
      {
      //return appropriate attribute for column
      }
  }
  //... additional interface methods
});
```

Once the providers have been set up, we can add the columns. The text we set on each column will appear as a header for that column when the table is displayed:

```
TableColumn column1 = new TableColumn(viewer.getTable(),
                                      SWT.CENTER);
column1.setText("Primary Key");
TableColumn column2 = new TableColumn(viewer.getTable(),
                                      SWT.CENTER);
column2.setText("Foreign Key");
```

```
TableColumn column3 = new TableColumn(viewer.getTable(),
                                      SWT.CENTER);
column3.setText("Data");
```

Finally, we need to provide input to drive the content provider. The input object (in this case, a `String` describing a query) is set on the viewer, which passes it to the content provider when it's ready to display the table:

```
viewer.setInput(QUERY);
```

This example simulates retrieving multiple rows from a database and displaying the results. However, it suffices to get our point across about content providers. The role of the `IStructuredContentProvider` implementation is straightforward: Given an input element, return all the children elements to be displayed. A table doesn't maintain parent/child relationships, so this method is called only once and is given the current input object. The final issue to be aware of when using a content provider is that it will always execute in the UI thread. This means updates to the interface will be waiting for your methods to complete, so you definitely shouldn't query a database to get your updates. The content provider should traverse a graph of already-loaded domain objects to select the appropriate content to display.

A word about error handling

When you're using JFace—especially the providers that the widgets call internally—it pays to be careful with your error handling. When JFace makes the callback to your class, it typically does so inside a `try/catch` block that catches all exceptions. JFace does some checks to see whether it knows how to handle the exception itself before letting the exception propagate. Unfortunately, these checks rely upon the `Platform` class, which is tightly coupled with Eclipse; it's practically impossible to initialize `Platform` correctly unless you're running Eclipse. This leads to internal assertion failures when JFace tries to use `Platform` outside of Eclipse, and these exceptions end up masking your own errors.

In practical terms, you shouldn't ever let an exception be thrown out of a provider method. If it happens, you're in for strange "The application has not been initialized" messages. If you ever see one of these, check your code carefully—things such as `ClassCastException`s can be hard to spot, and locating them is even more difficult when JFace hides them from you.

Editing table data

Displaying data can be useful on its own, but eventually you'll want to let the user edit it. Often, the most user-friendly way to enable editing is to allow the user to change it directly in the table as it's presented. JFace provides a means to support this editing through `CellEditor`s.

As we mentioned in the chapter overview, `CellEditor`s exist to help decouple the domain model from the editing process. In addition, using these editors can make your UI more user friendly: Users won't be able to enter values your application doesn't understand, thus avoiding confusing error messages further down the line. The framework assumes that each domain object has a number of named properties. Generally, you should follow the JavaBeans conventions, with property `foo` having `getFoo()` and `setFoo()` methods; but doing so isn't strictly necessary as long as you can identify each property given only its name. You begin by attaching an instance of `ICellModifier` to your `TableViewer`. The `ICellModifier` is responsible for retrieving the value of a given property from an object, deciding whether a property can currently be edited, and applying the updated value to the object when the edit has been completed. The actual edit, if allowed, is performed by a `CellEditor`. JFace provides `CellEditor`s for editing via checkbox, combo box, pop-up dialog, or directly typing the new text value. In addition, you can subclass `CellEditor` if you need a new form of editor. After registering `CellEditor`s, you associate each column with a property. When the user clicks on a cell to change its value, JFace does all the magic of matching the proper column with the property to edit and displaying the correct editor, and it notifies your `ICellModifier` when the edit is complete.

We'll show examples of the important parts of the process here. The rest of the snippets in this section are taken from the `Ch9TableEditorComposite`, which is presented in full at the end of the chapter.

The first snippet sets up data that the rest of the code will reference. The array of `String`s in `VALUE_SET` holds the values that will be displayed by our `ComboBoxCellEditor`. We'll need to convert between indices and values several times (see the discussion later in the chapter):

```
private static final Object[] CONTENT = new Object[] {
                new EditableTableItem("item 1", new Integer(0)),
                new EditableTableItem("item 2", new Integer(1))
                };
private static final String[] VALUE_SET = new String[] {
                                          "xxx", "yyy", "zzz"
                                          };
```

```
private static final String NAME_PROPERTY = "name";
private static final String VALUE_PROPERTY = "value";
```

Our class contains several different methods that are each responsible for setting up a different facet of the cell editor. They are called in turn from `buildControls`. The first thing this method does is set up the table and the classes required by the viewer:

```
protected Control buildControls()
{
  final Table table = new Table(parent, SWT.FULL_SELECTION);
  TableViewer viewer = new TableViewer(table);
  ... //set up a two column table
```

Once the table has been initialized, we continue by adding an instance of `ITableLabelProvider` to our viewer. The idea is similar to the label providers we discussed in chapter 8. However, because each row of a table has many columns, the signature of our methods must change slightly. In addition to the element, each method now takes the integer index of the column that is being requested. The label provider must therefore contain the logic to map column indices to properties of the domain objects. The next snippet shows how this is done:

```
  viewer.setLabelProvider(new ITableLabelProvider() {
    public String getColumnText(Object element,
                                int columnIndex) {
      switch(columnIndex)
      {
        case 0:
          return ((EditableTableItem)element).name;
        case 1:
          Number index = ((EditableTableItem)element).value;
          return VALUE_SET[index.intValue()];
        default:
          return "Invalid column: " + columnIndex;
      }
    }

  });

  attachCellEditors(viewer, table);
  return table;
}
```

The `attachCellEditors()` method is where we set up our `ICellModifier`, which is responsible for translating a property name into data to be displayed, deciding whether a given property can be edited, and then applying whatever changes the user makes. When the user double-clicks a cell to edit it, `canModify()` is called to determine whether the edit should be allowed. If it's allowed, `getValue()` is called next to retrieve the current value of the property being edited. Once the edit is

complete, `modify()` is called; it's `modify()`'s responsibility to apply the changes the user made back to the original domain object. While in `getValue()` and `canModify()`, it's safe to cast parameters directly to the domain objects; this doesn't work in `modify()`. `modify()` receives the `TableItem` that's displaying the row. This `TableItem` has had the domain object set as its data, so we must retrieve it using `getData()` before we can update it:

```
private void attachCellEditors(final TableViewer viewer,
                               Composite parent)
{
  viewer.setCellModifier(new ICellModifier() {
    public boolean canModify(Object element,
                             String property) {
      return true;
    }

    public Object getValue(Object element, String property) {
      if( NAME_PROPERTY.equals(property))
        return ((EditableTableItem)element).name;
      else
        return ((EditableTableItem)element).value;
    }
    //method continues below...
```

When `modify()` is finished updating the domain object, we must let the viewer know to update the display. The viewer's `refresh()` method is used for this purpose. Calling `refresh()` with the domain object that changed causes the viewer to redraw the given row. If we skip this step, users will never see their changes once the edited cell loses focus:

```
    public void modify(Object element,
                       String property, Object value) {
      TableItem tableItem = (TableItem)element;
      EditableTableItem data =
                  (EditableTableItem)tableItem.getData();
      if( NAME_PROPERTY.equals( property ) )
        data.name = value.toString();
      else
        data.value = (Integer)value;

      viewer.refresh(data);
    }
  });
```

The items given in the `CellEditor` array here are matched in order with the columns of the underlying table:

```
  viewer.setCellEditors(new CellEditor[] {
            new TextCellEditor(parent),
```

```
                new ComboBoxCellEditor(parent, VALUE_SET )
              });
```

Next, the strings in `setColumnProperties()` are the names of the editable properties on our domain objects. They're also matched in order with the table's columns, so that in our example clicking column 0 will try to edit the `name` property, and column 1 will edit the `value` property:

```
      viewer.setColumnProperties(new String[] {
                NAME_PROPERTY, VALUE_PROPERTY
              });
    }
  }

  class EditableTableItem
  {
    ... //name and value properties
  }
```

Using a `ComboBoxCellEditor` as we do here is tricky. The editor's constructor takes an array of `String`s that are the values presented for the user to choose from. However, the editor expects `Integer`s from `getValue()` and returns an `Integer` to `modify()` when the edit is complete. These values should correspond to the index of the selected value in the array of `String`s passed to the `ComboBoxCellEditor` constructor. In this simple example we save the `Integer` directly in the value field, but in a real application you'll probably need utilities to easily convert back and forth between indices and values.

Again, using `CellEditor`s is an area where it's smart to pay attention to your casting and error handling. Especially when different methods require you to cast to different objects, as in the `ICellModifier`, it's easy to make a mistake the compiler can't catch for you. Due to JFace's exception handling, as we discussed earlier, these issues show up as cryptic "Application not initialized" runtime errors that can be hard to track down if you don't know what you should be looking for.

9.2 Creating menus

Every graphical application uses a menu of some sort. You'll often find File, Edit, and so on across the top of your application's window. These menus fill an important role, because they provide a place for users to browse through the functionality offered by your application.

We'll first discuss creating menus using SWT. We'll then revisit the JFace `Action` classes that we mentioned in chapter 4, to discuss an alternate way to create menus that allows for easy sharing of common code.

9.2.1 Accelerator keys

Before we get too deep into the specifics of menus, let's discuss how SWT handles accelerator keys. *Accelerator keys* are keyboard shortcuts that activate a widget without the user having to click it with the mouse. The best example is the ubiquitous Ctrl-C (or Open Apple-C if you're using a Mac) to copy text to the clipboard, the same as if you selected Copy from the Edit menu that's present in most applications. Offering accelerator keys for common tasks can greatly increase advanced users' productivity, because their hands don't have to continually switch between the keyboard and mouse. The accelerator keystroke for an item customarily appears next to the item's name in drop-down menus for the application, making it easier for users to learn the keystrokes as they use the application.

In both SWT and JFace, accelerator keys are expressed by using constants from the `SWT` class. The concept is the same as for styles: All the constants are bitwise ORed together to determine the final key combination. Additionally, `char`s are used to represent letters or numbers on the keyboard. Because a Java `char` can be automatically converted to an `int`, `char`s can be used just like the `SWT` style constants to build a bitmask. This bitmask is passed to the `setAccelerator()` method on a `Menu` to register the combination of keys that will activate that menu item. For example, a `MenuItem` whose accelerator is set to `SWT.CONTROL | SWT.SHIFT | 't'` will activate when the Ctrl, Shift, and T keys are pressed simultaneously.

9.2.2 Creating menus in SWT

When you're creating menus using SWT, you'll use only two classes: `Menu` and `MenuItem`. Although the classes themselves aren't complicated, several areas of complexity arise once you begin to use them.

`Menu` acts as a container for `MenuItem`s. `Menu` extends `Widget` and contains methods for adding `MenuItem`s and controlling the visibility and location of the menu. `Menu` also broadcasts events to implementors of the `MenuListener` interface, which receives notification when the menu is shown or hidden.

`Menu` supports three different styles, which go beyond controlling the visual appearance to determine the type of menu created:

- *`SWT.POP_UP`*—Creates a free-floating pop-up menu of the type that typically appears when you right-click in an application.
- *`SWT.BAR`*—Creates the menu bar at the top of an application window. A menu bar doesn't typically have selectable menu items; instead, it acts as a container for menu items that contain menus of type `SWT.DROP_DOWN`.

- *SWT.DROP_DOWN*—Creates the File, Edit, and other drop-down menus that we're all familiar with. These menus may contain a mix of MenuItems and submenus of their own.

A MenuItem is a widget that either can be selected by the end user or can display another menu. A MenuItem is always created as a child of a Menu. A variety of styles are available for MenuItems:

- *SWT.PUSH*—Creates a standard menu item with no frills.
- *SWT.CHECK, SWT.RADIO*—Add either a checkbox or radio button, as appropriate, which flips between on and off each time the item is selected.
- *SWT.SEPARATOR*—Visually separates groups of menu items. It displays the standard separator for your platform (usually a thin line) and may not be selected by the user.
- *SWT.CASCADE*—Creates a submenu. When a cascading menu item has a menu assigned to it, highlighting that item results in the submenu being displayed.

All MenuItems except separators broadcast SelectionEvents that can be listened for. Figure 9.2 shows the different menu styles.

Creating Menus is straightforward. Classes are instantiated and configured, and then assigned to the widgets on which they should be displayed. The following snippet shows how to create a File menu attached to the main window of your application:

```
Composite parent = ... //get parent
Menu menuBar = new Menu(parent.getShell(), SWT.BAR);

MenuItem fileItem = new MenuItem(menuBar, SWT.CASCADE);
fileItem.setText("&File");

Menu fileMenu = new Menu(fileItem);
fileItem.setMenu(fileMenu);

parent.getShell().setMenuBar(menuBar);
```

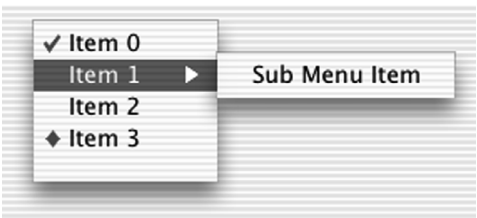

Figure 9.2
Menu types. From top to bottom, SWT.CHECK, SWT.CASCADE, SWT.PUSH, and SWT.RADIO.

Notice that you must first create the root menu bar and then add a menu item to hold each drop-down menu that will appear on it. At this point, we have a menu bar that displays File but is empty. Our next task is to populate this menu:

```
MenuItem open = new MenuItem(fileMenu, SWT.PUSH);
open.setText("Open...");
open.setAccelerator(SWT.CONTROL | 'o');
open.addSelectionListener(new SelectionListener() {
  public void widgetSelected(SelectionEvent event) {
    ... //handle selection
  }
};
```

Clicking File will now reveal a drop-down menu with an Open option. If Open is selected, the selection listener we've defined is invoked to display an Open File dialog or do whatever other action is appropriate to the application. We've also set the keyboard accelerator for this option to Ctrl-O by calling `setAccelerator()` with a bitmask of the keys we wish to assign. The result is that pressing Ctrl-O invokes the selection listener just as if it was selected with the mouse.

Creating a pop-up menu is similar to what we've done here, but there is a slight wrinkle. We don't need a menu bar, so we can start with the pop-up:

```
Composite parent = ... //get composite
final Menu popupMenu = new Menu(parent.getShell(), SWT.POP_UP);
```

Notice that we declare the `Menu` instance to be `final`. This is important, because we'll need to reference it in a listener later.

Creating the `MenuItem`s is the same as for a drop-down menu. For variety, we'll show how to create a menu item that reveals a submenu when highlighted. The important point to notice in this process is that after the submenu is created, it must be assigned to its parent menu item using `setMenu()`, just as we did with the menu bar in our earlier example:

```
MenuItem menuItem = new MenuItem(popupMenu, SWT.CASCADE);
menuItem.setText("More options");

Menu subMenu = new Menu(menuItem);
menuItem.setMenu(subMenu);
MenuItem subItem = new MenuItem(subMenu, SWT.PUSH);
subItem.setText("Option 1");
subItem.addSelectionListener( ... );
```

Unlike a menu bar, a pop-up menu isn't displayed by default—you must decide when to display it. Typically this is done in response to a mouse right-click, so we'll use a `MouseListener` on the parent `Composite`. This is where we need the pop-up menu instance to be final, so we can reference it within our anonymous inner class:

```
parent.addMouseListener(new MouseListener() {
  public void mouseDown(MouseEvent event) {
    if(event.button == 2)
    {
      popupMenu.setVisible(true);
    }
  }
  ... //other MouseListener methods
});
```

`MouseEvent` contains information about the button that was clicked. The buttons are numbered: 1 is the left mouse button, and 2 is the right button. If this button was clicked, we make the pop-up menu visible; it's displayed at the location that was clicked. Pressing Esc or clicking anywhere other than on the menu automatically causes the pop-up to be hidden.

Now that you've seen how SWT handles menus, we'll turn our attention to the menu options offered by JFace.

9.2.3 Using JFace actions to add to menus

We've already discussed the design of JFace's `Action` classes in chapter 4. To review briefly, an action encapsulates the response to a single application level event, such as "Open a file" or "Update the status bar." This action can then be reused and triggered in different contexts, such as a toolbar button or a menu item. We'll discuss this last case here. By using actions to create your menus, instead of doing it by hand, you can simplify the design of your application and reuse common logic.

Using actions in a menu is similar to using them anywhere else. Remember that an `IContributionManager` is responsible for assembling individual `Actions` and transforming them into a form that can be displayed to the user. For menus, we'll use the `MenuManager` implementation of `IContributionManager`. After adding whatever actions are needed to the `MenuManager`, we can tell it to create a new menu or to add the actions to another menu. The code looks something like this:

```
Shell shell = ... //obtain a reference to the Shell
MenuManager fileMenuManager = new MenuManager("File");

IAction openAction = new OpenAction(...);
... //create other actions as appropriate

fileMenuManager.add(openAction);
... //add other actions

Menu menuBar = new Menu(shell, SWT.BAR);
fileMenuManager.fill(menuBar, -1);
shell.setMenuBar(menuBar);
```

Although we've still created the menu bar manually, we can add actions to the manager and let it worry about how the menu should be built. In this case, we end up with a File menu on the window's menu bar, because that is the name we gave the `MenuManager` when we instantiated it. The advantage of doing it this way instead of building menus by hand is that the action classes can be easily reused elsewhere. For example, if we have a toolbar that includes a button to let users open files, we can use the same `OpenAction` class there.

You must keep one caveat in mind when you're using menu managers: Once `fill()` or `createXXX()` has been called on a given instance, `Menu` and `MenuItem` instances are created and cached internally. This is necessary so that the manager can be used to update the menu. However, it also means that you shouldn't make further calls to `fill()` or `create()`, especially for a different type of menu. For example, suppose that after the previous code we called `createContextMenu()` on `fileMenuManager`. We would get exceptions when we tried to add the menu to a composite, because the menu would be the cached instance with type `SWT.CASCADE` instead of type `SWT.POP_UP` (which is required by context menus).

9.3 Updating WidgetWindow

Our pane for this chapter combines a table viewer, cell editors, and a context menu. We'll expand the snippets of a database editor that we discussed earlier and add a right-click menu that lets the user insert a new row. The final product looks like figure 9.3.

Listing 9.1 is longer than the code for most of our chapter panes, so we'll point out the most interesting bits before you begin reading it. The first thing to notice is the inner class `NewRowAction`. This class holds the logic to insert a new row into the table; it's added to the `MenuManager` we create in `createPane()`.

Next is the `createPane()` method, which is the entry point into the class. After delegating to methods to lay out the table and attach a label provider, content provider, and cell editor, we instantiate a `MenuManager` and use it to build a context

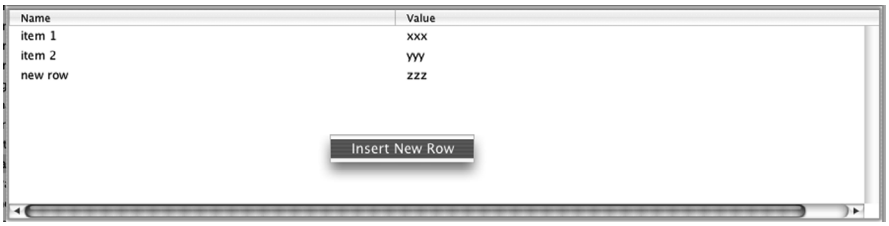

Figure 9.3 Our database table editor

menu that we then attach to the newly created `Table`. Finally, we pass the initial content to the viewer.

After `createPane()` come the private utility methods. The most important for our purposes is `attachCellEditors()`, which contains the logic to allow editing of individual table cells. Note that these modifications are performed directly on the domain objects.

At the end of the listing is the `EditableTableItem` class, which serves as a domain object for this example and is included in the same file for convenience.

Listing 9.1 Ch9TableEditorComposite.java

```
package com.swtjface.Ch9;

import org.eclipse.jface.action.*;
import org.eclipse.jface.viewers.*;
import org.eclipse.swt.SWT;
import org.eclipse.swt.graphics.Image;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.widgets.*;

public class Ch9TableEditorComposite extends Composite
{

  private static final Object[] CONTENT = new Object[] {   ❶ Initial content
          new EditableTableItem("item 1", new Integer(0)),
          new EditableTableItem("item 2", new Integer(1))
          };

  private static final String[] VALUE_SET = new String[] {
                                            "xxx", "yyy", "zzz"
                                            };
  private static final String NAME_PROPERTY = "name";
  private static final String VALUE_PROPERTY = "value";

  private TableViewer viewer;

  public Ch9TableEditorComposite(Composite parent)
  {
    super(parent, SWT.NULL);
    buildControls();
  }

  private class NewRowAction extends Action    ❷ NewRowAction class
  {
    public NewRowAction()
    {
      super("Insert New Row");
    }

    public void run()   ❸ run() method
    {
```

```
    EditableTableItem newItem =
          new EditableTableItem("new row", new Integer(2));
    viewer.add(newItem);
  }
}

protected void buildControls()
{
  FillLayout compositeLayout = new FillLayout();
  setLayout(compositeLayout);

  final Table table = new Table(this, SWT.FULL_SELECTION);
  viewer = buildAndLayoutTable(table);

  attachContentProvider(viewer);
  attachLabelProvider(viewer);
  attachCellEditors(viewer, table);

  MenuManager popupMenu = new MenuManager();
  IAction newRowAction = new NewRowAction();
  popupMenu.add(newRowAction);
  Menu menu = popupMenu.createContextMenu(table);
  table.setMenu(menu);

  viewer.setInput(CONTENT);
}

private void attachLabelProvider(TableViewer viewer)
{
  viewer.setLabelProvider(new ITableLabelProvider() {
    public Image getColumnImage(Object element,
                                int columnIndex) {
      return null;
    }

    public String getColumnText(Object element,
                                int columnIndex) {
      switch(columnIndex)
      {
        case 0:
          return ((EditableTableItem)element).name;
        case 1:
          Number index = ((EditableTableItem)element).value;
          return VALUE_SET[index.intValue()];
        default:
          return "Invalid column: " + columnIndex;
      }
    }

    public void addListener(ILabelProviderListener listener) {
    }

    public void dispose(){
    }
```

❹ **Build menu**

❺ **getColumnText() method**

```
    public boolean isLabelProperty(Object element,
                                   String property){
      return false;
    }

    public void removeListener(ILabelProviderListener lpl) {
    }
  });
}

private void attachContentProvider(TableViewer viewer)
{
  viewer.setContentProvider(new IStructuredContentProvider() {
    public Object[] getElements(Object inputElement) {
      return (Object[])inputElement;
    }

    public void dispose() {
    }

    public void inputChanged(Viewer viewer,
                             Object oldInput,
                             Object newInput) {
    }
  });
}

private TableViewer buildAndLayoutTable(final Table table)
{
  TableViewer tableViewer = new TableViewer(table);

  TableLayout layout = new TableLayout();
  layout.addColumnData(new ColumnWeightData(50, 75, true));
  layout.addColumnData(new ColumnWeightData(50, 75, true));
  table.setLayout(layout);

  TableColumn nameColumn = new TableColumn(table, SWT.CENTER);
  nameColumn.setText("Name");
  TableColumn valColumn = new TableColumn(table, SWT.CENTER);
  valColumn.setText("Value");
  table.setHeaderVisible(true);
  return tableViewer;
}

private void attachCellEditors(final TableViewer viewer,
                               Composite parent)
{
  viewer.setCellModifier(new ICellModifier() {
    public boolean canModify(Object element, String property){
      return true;
    }

    public Object getValue(Object element, String property) {
```

❻ **getElements() method**

❼ **buildAndLayoutTable() method**

```
        if( NAME_PROPERTY.equals(property))
          return ((EditableTableItem)element).name;
        else
          return ((EditableTableItem)element).value;
      }

      public void modify(Object element,
                         String property,
                         Object value) {      ❽ modify() method
        TableItem tableItem = (TableItem)element;
        EditableTableItem data =
                    (EditableTableItem)tableItem.getData();
        if( NAME_PROPERTY.equals( property ) )
          data.name = value.toString();
        else
          data.value = (Integer)value;

        viewer.refresh(data);
      }
    });

    viewer.setCellEditors(new CellEditor[] {
              new TextCellEditor(parent),
              new ComboBoxCellEditor(parent, VALUE_SET )
            });

    viewer.setColumnProperties(new String[] {
              NAME_PROPERTY, VALUE_PROPERTY
            });
  }

}

class EditableTableItem      ❾ EditableTableItem class
{
  public String name;
  public Integer value;

  public EditableTableItem( String n, Integer v)
  {
    name = n;
    value = v;
  }
}
```

❶ These constants hold the data we'll use for our initial content. In a real application, this data would likely be read from a database or other external source.

❷ This class contains the logic to insert new rows into the data set. It extends `Action` so it can be used by a `MenuManager`.

3. To perform the necessary logic, we override the `run()` method defined in `Action`. The action framework ensures that this method is invoked at the appropriate time. Our implementation creates a new domain object and calls `add()` on the table viewer. Most real applications will need additional logic here to manage the collection of domain objects.
4. We build a simple context menu by creating a new `MenuManager` and adding the actions we want to use. In this case, we add the menu directly to the `Table`. If the tab contained more controls than just this table, then the menu would appear only when the user right-clicked on the table. If we wanted it to appear when the user clicked anywhere on the tab, we would need to add the menu to the parent `Composite`.
5. This is a standard `LabelProvider` implementation, similar to ones you've seen earlier. It returns the value of whichever property matches the requested column.
6. Our content provider assumes that whatever input it's given is an array of `Objects`. It performs the appropriate cast and returns the result.
7. Here we construct the table. We add two columns and set the header text.
8. The `modify()` method is the most important part of our `CellModifier` implementation. The `element` parameter contains the `TableItem` for the cell that was just changed. The domain object associated with this item is retrieved with the `getData()` method. We then check the `propertyName` parameter to determine what property was modified; we update the matching property on the domain object using the `value` parameter, which contains the date entered by the user.
9. This small class serves as the domain objects for our example.

Run this example by adding the following lines to `WidgetWindow`:

```
TabItem chap9TableEditor = new TabItem(tf, SWT.NONE);
chap9TableEditor.setText("Chapter 9");
chap9TableEditor.setControl(new Ch9TableEditorComposite(tf));
```

When you run this example, the initial window contains two rows with sample data. Right-clicking brings up a context menu that lets you insert a new row into the table. Double-clicking a cell allows you to edit the data, either by typing or by choosing from a drop-down menu.

9.4 Summary

Most of what you've seen with `Table`s and `TableViewer`s should be familiar from chapter 8. The basic concepts of viewers and providers are identical to those we discussed earlier. Because tables impose a two-dimensional structure on data, they require more configuration than some of other widgets we've examined. The `TableLayout` and `TableColumn` classes create this structure for each table and control the details of how the table appears to the user.

After working through these two chapters, you should be well equipped to handle any requirement that calls for the use of one of these viewers, or any of the more esoteric classes such as `TableTreeViewer` that are included in JFace.

`CellEditor`s, however, are a useful feature unique to `TableViewer`s. `CellEditor`s provide a framework for handling updates to specific cells in a table, and the predefined `CellEditor` classes provide an easy way to provide discrete options for the user to choose from.

Just about any application will need to provide a menu bar, and it's common to provide context menus that show only options that are relevant to what the user is currently doing. For example, right-clicking in a word processor typically brings up options related to formatting text. SWT makes creating these menus easy, and JFace adds the action framework to facilitate reusing logic easily regardless of the context from which it was invoked. We discussed the theory behind actions in chapter 4, and the examples we've shown here should give you a good feel for how they're used in practice.

JAVA

SWT/JFace IN ACTION

Scarpino • Holder • Ng • Mihalkovic

SWT and JFace—Eclipse's graphical libraries—enable you to build nimble and powerful Java GUIs. But this is only the beginning. With Draw2D and the Graphical Editing Framework, you can go beyond static applications and create full-featured editors. And with the Rich Client Platform, you can build customized workbenches whose capabilities far exceed those of traditional interfaces.

FSWT/JFace in Action covers the territory, from simple widgets to complex graphics. It guides you through the process of developing Eclipse-based GUIs and shows how to build applications with features your users will love. The authors share with you their intimate knowledge of the subject in a helpful and readable style.

This book encourages you to learn through action. Many code samples show you how SWT/JFace works in practical applications. Not only do these examples help you understand, they are working programs you can reuse in your own interfaces.

What's Inside

- Understanding SWT/JFace design
- Creating workbenches with the Rich Client Platform
- Building editors with Draw2D and the Graphical Editing Framework
- Integrating SWT with Microsoft's COM
- And much more

Matthew Scarpino, **Stephen Holder**, **Stanford Ng**, and **Laurent Mihalkovic** together have a rich and varied background from work on applications for reconfigurable computing, financial management, and enterprise development.

"An excellent work! It is timely, comprehensive, and interestingly presented."

—Phil Hanna
SAS Institute Inc.
author of *JSP: The Complete Reference*

"I recommend this book to anyone getting into development with the Eclipse libraries."

—Steve Gutz
Senior Software Developer, IBM
author of *Up to Speed with Swing*

"I really enjoyed the authors' style. It was easy to read, and the information stayed with me."

—Carl Hume
Software Architect

"… a good and useful treatment. There is no other book like it in the market."

—Robert D. McGovern
co-author of *Eclipse in Action*

AUTHOR ONLINE
Ask the Authors

Ebook edition

www.manning.com/scarpino

MANNING $44.95 US/$62.95 Canada

9 781932 394276 54495
ISBN 1-932394-27-3