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 4

MATTHEW SCARPINO
STEPHEN HOLDER
STANFORD NG
AND LAURENT MIHALKOVIC

MANNING

Greenwich
(74° w. long.)

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

4 Working with events

This chapter covers

- Event processing with SWT
- Typed and untyped listeners
- Mouse and keyboard events
- Event processing with JFace
- Actions and contributions

Without events, the widgets and containers we've looked at are only good for decoration. This chapter focuses on how to configure these components to understand and respond to user actions. In particular, it describes the SWT/JFace framework that acquires these actions and translates them into software constructs called *events*. The process of using a toolset to generate, receive, and respond to these events is the toolset's *event model*. Many books on GUIs leave the event model until later chapters, but we feel the subject's importance demands an early introduction.

The first part of this chapter describes the SWT data structures that enable applications to process events. These include the event classes, which are created when a user carries out actions, and the *listener* interfaces, which receive event objects. By combining these appropriately, an application can provide multiple responses to nearly every form of event that can occur. However, SWT's powerful event-processing mechanisms can make coding more complicated than it needs to be. For this reason, we need to examine how JFace simplifies the process.

This chapter's second part deals with using both SWT and JFace to interface with the user. The JFace library replaces events and listeners with *actions* and *contributions*, which perform the same function as their SWT counterparts but in very different ways. These new classes simplify the process of event programming by separating the event-processing methods from the GUI's appearance. Also, actions and contributions are meant for performing window-oriented interfacing, and this narrowed scope reduces the developer's programming burden.

4.1 Event processing in SWT

The SWT event-processing cycle is depicted in figure 4.1. It begins with the operating system's event queue, which records and lists actions taken by the user. Once an SWT application begins running, its `Display` class sorts through this queue using its `readAndDispatch()` method and `msg` field, which acts as a handle to the underlying OS message queue. If it finds anything relevant, it sends the event to its top-level `Shell` object, which determines which widget should receive the event. The `Shell` then sends the event to the widget that the user acted on, which transfers this information to an associated interface called a *listener*. One of the listener's methods performs the necessary processing or invokes another method to handle the user's action, called an *event handler*.

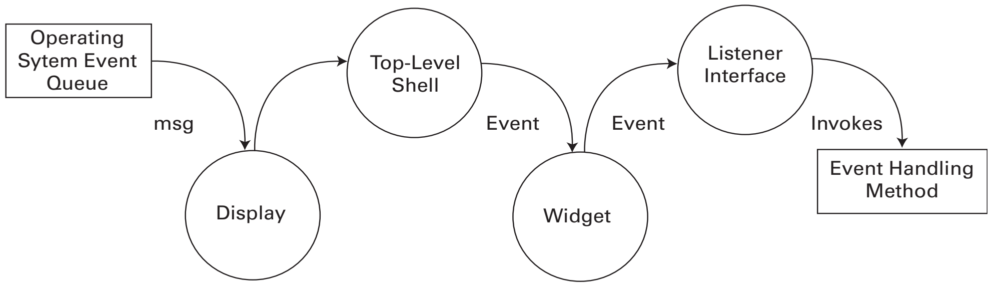


Figure 4.1 Acquiring events from the operating system and processing them in an SWT application

When making a widget responsive to events, the main tasks of the GUI designer are determining which events need to be acted on, creating and associating listeners to sense these events, and then building event handlers to perform the necessary processing. This section will show how to accomplish these tasks using the SWT data structures contained in the `org.eclipse.swt.events` package.

4.1.1 *Using typed listeners and events*

Most of the listener interfaces in SWT only react to a particular set of user actions. They're called *typed listeners* for this reason, and they inherit from the `TypedListener` class. Similarly, the events corresponding to these specific actions are *typed events*, which subclass the `TypedEvent` class. For example, a mouse click or double-click is represented by a `MouseEvent`, which is sent to an appropriate `MouseListener` for processing. Keyboard actions performed by the user are translated into `KeyEvents`, which are picked up by `KeyListeners`. A full list of these typed events and listeners is shown in table 4.1.

In order to function, these listeners must be associated with components of the GUI. For example, a `TreeListener` will only receive `TreeEvents` if it's associated with a `Tree` object. But not every GUI component can use each listener. For example, as shown in the GUI component column of the table, a `Control` component broadcasts many more types of events than a `Tracker` object. There are also listeners, such as `MenuListeners` and `TreeListeners`, that can only be attached to very specific widgets. This attachment is performed by invoking the component's `add...Listener()` method with the typed listener as the argument.

Table 4.1 SWT `Event` classes and their associated listeners

Event	Listener	Listener methods	GUI component
`ArmEvent`	`ArmListener`	`widgetArmed()`	`MenuItem`
`ControlEvent`	`ControlListener`	`controlMoved()` `controlResized()`	`Control, TableColumn, Tracker`
`DisposeEvent`	`DisposeListener`	`widgetDisposed()`	`Widget`
`FocusEvent`	`FocusListener`	`focusGained()` `focusLost()`	`Control`
`HelpEvent`	`HelpListener`	`helpRequested()`	`Control, Menu, MenuItem`
`KeyEvent`	`KeyListener`	`keyPressed()` `keyReleased()`	`Control`
`MenuEvent`	`MenuListener`	`menuHidden()` `menuShown()`	`Menu`
`ModifyEvent`	`ModifyListener`	`modifyText()`	`CCombo, Combo, Text, StyledText`
`MouseEvent`	`MouseListener`	`mouseDoubleClick()` `mouseDown()` `mouseUp()`	`Control`
`MouseMoveEvent`	`MouseMoveListener`	`mouseMove()`	`Control`
`MouseTrackEvent`	`MouseTrackListener`	`mouseEnter()` `mouseExit()` `mouseHover()`	`Control`
`PaintEvent`	`PaintListener`	`paintControl()`	`Control`
`SelectionEvent`	`SelectionListener`	`widgetDefaultSelected()` `widgetSelected()`	`Button, CCombo, Combo, CoolItem, CTabFolder, List, MenuItem, Sash, Scale, ScrollBar, Slider, StyledText, TabFolder, Table, TableCursor, TableColumn, TableTree, Text, ToolItem, Tree`
`ShellEvent`	`ShellListener`	`shellActivated()` `shellClosed()` `shellDeactivated()` `shellDeiconified()` `shellIconified()`	`Shell`

continued on next page

Table 4.1 SWT `Event` classes and their associated listeners *(continued)*

Event	Listener	Listener methods	GUI component
`TraverseEvent`	`TraverseListener`	`keyTraversed()`	`Control`
`TreeEvent`	`TreeListener`	`treeCollapsed()` `treeExpanded()`	`Tree, TableTree`
`VerifyEvent`	`VerifyListener`	`verifyText()`	`Text, StyledText`

Understanding Event classes

The Event column in table 4.1 lists the subclasses of `TypedEvent` that the `Display` and `Shell` objects send to typed listeners. Although programmers generally don't manipulate these classes directly, the classes contain member fields that provide information regarding the event's occurrence. This information can be used in event handlers to obtain information about the environment. These fields, inherited from the `TypedEvent` and `EventObject` classes, are shown in table 4.2.

Table 4.2 Data fields common to all typed events

`TypedEvent` field	Function
`data`	Information for use in the `Event` handler
`display`	The display in which the `Event` fired
`source`	The component that triggered the `Event`
`time`	The time that the `Event` occurred
`widget`	The widget that fired the `Event`

In addition to these, many event classes have other fields that provide more information about the user's action. For example, the `MouseEvent` class also includes a `button` field, which tells which mouse button was pressed, and `x` and `y`, which specify the widget-relative coordinates of the mouse action. The `ShellEvent` class contains a boolean field called `doit`, which lets you specify whether a given action will result in its intended effect. Finally, the `PaintEvent` class provides additional methods that we'll discuss in chapter 7.

Programming with listeners

There are two main methods of incorporating listeners in code. The first creates an anonymous interface in the component's `add...Listener()` method, which narrows the scope of the listener to the component only. This method is shown in the following code snippet:

```
Button button = new Button(shell, SWT.PUSH | SWT.CENTER);
button.addMouseListener(new MouseListener()
{
  public void mouseDown(MouseEvent e)
  {
    clkdwnEventHandler();
  }

  public void mouseUp(MouseEvent e)
  {
    clkupEventHandler();
  }

  public void mouseDoubleClick(MouseEvent e)
  {
    dblclkEventHandler();
  }
});

static void dblclkEventHandler()
{
  System.out.println("Double click.");
}

static void clkdwnEventHandler()
{
  System.out.println("Click - down.");
}

static void clkupEventHandler()
{
  System.out.println("Click - up.");
}
```

In the first line, a `Button` widget is created and added to the application's `Shell`. Then, the `addMouseListener()` method creates an anonymous `MouseListener` interface and associates it with the button. This interface contains three methods—`mouseDown()`, `mouseUp()`, and `mouseDoubleClick()`—which must be implemented in any instance of a `MouseListener`. If the user presses the mouse button, releases the button, or double-clicks, a `MouseEvent` is sent to one of these methods, which invokes the appropriate event-handling method. These event handlers complete the event processing by sending a message to the console. Although the event-handling routines are simple in this example, they generally demand more effort than any other aspect of event processing.

An anonymous interface can be helpful if you need to access objects (declared with the `final` keyword) in the outer class. However, the listener can't be associated with other components. You can solve this problem by declaring a separate interface that inherits from `MouseListener`. An example is shown here:

```
Button button = new Button(shell, SWT.PUSH | SWT.CENTER);
button.addMouseListener(ExampleMouseListener);

MouseListener ExampleMouseListener = new MouseListener()
{
  public void mouseDoubleClick(MouseEvent e)
  {
    System.out.println("Double click.");
  }

  public void mouseDown(MouseEvent e)
  {
    System.out.println("Click - down.");
  }

  public void mouseUp(MouseEvent e)
  {
    System.out.println("Click - up.");
  }
};
```

The previous code samples declare all three of the `MouseListener`'s member methods. But what if you're only concerned with the double-click event, and you only want to work with the `mouseDoubleClick()` method? If you use the `MouseListener` interface, you have to declare all of its methods, just as in any interface. However, you can eliminate this unnecessary code by using special classes called *adapters*.

4.1.2 Adapters

Adapters are abstract classes that implement `Listener` interfaces and provide default implementations for each of their required methods. This means that when you associate a widget with an adapter instead of a listener, you only need to write code for the method(s) you're interested in. Although this may seem like a minor convenience, it can save you a great deal of programming time when you're working with complex GUIs.

NOTE The *adapters* mentioned in this section are very different from the *model-based adapters* provided by the JFace library, first mentioned in chapter 2. Here, adapters reduce the amount of code necessary to create listener interfaces. Although model-based adapters can simplify event processing, as you'll see in section 4.2, they also help with many other aspects of GUI programming.

Adapters are only available for events whose listeners have more than one member method. The full list of these classes is shown in table 4.3, along with their associated `Listener` classes.

Table 4.3 SWT adapter classes and their corresponding listener interfaces

Adapter	Listener
`ControlAdapter`	`ControlListener`
`FocusAdapter`	`FocusListener`
`KeyAdapter`	`KeyListener`
`MenuAdapter`	`MenuListener`
`MouseAdapter`	`MouseListener`
`MouseTrackAdapter`	`MouseTrackListener`
`SelectionAdapter`	`SelectionListener`
`ShellAdapter`	`ShellListener`
`TreeAdapter`	`TreeListener`

Adapter objects are easy to code and are created with the same `add...Listener()` methods. Two examples are shown here:

```
button.addMouseListener(new MouseAdapter()
{
  public void mouseDoubleClick(MouseEvent e)
  {
    dblclkEventHandler();
  }
)};

static void dblclkEventHandler()
{
  System.out.println("Double click.");
}
```

As shown, using the `MouseAdapter` class allows you to disregard the other methods associated with the `MouseListener` interface and concentrate on handling the double-click event. Similar to listener interfaces, adapters can be coded as anonymous classes or local classes.

4.1.3 *Keyboard events*

Although most of the events in table 4.1 are straightforward to understand and use, the keyboard event classes require further explanation. Specifically, these events include the KeyEvent class, which is created any time a key is pressed, and its two subclasses, TraverseEvent and VerifyEvent. A TraverseEvent results when the user presses an arrow key or the Tab key in order to focus on the next widget. A VerifyEvent fires when the user enters text that the program needs to check before taking further action.

In addition to the fields inherited from the TypedEvent and EventObject classes, the KeyEvent class has three member fields that provide information concerning the key that triggered the event:

- *character*—Provides a char value representing the pressed key.
- *stateMask*—Returns an integer representing the state of the keyboard modifier keys. By examining this integer, a program can determine whether any of the Alt, Ctrl, Shift, and Command keys are currently pressed.
- *keyCode*—Provides the SWT public constant corresponding to the typed key, called the *key code*. These public constants are presented in table 4.4.

The following code snippet shows how to use a KeyListener to receive and process a KeyEvent. It also uses the fields (character, stateMask, and keyCode) to acquire information about the pressed key:

```
Button button = new Button(shell, SWT.CENTER);
button.addKeyListener(new KeyAdapter()
{
  public void keyPressed(KeyEvent e)
  {
    String string = "";
    if ((e.stateMask & SWT.ALT) != 0) string += "ALT-";
    if ((e.stateMask & SWT.CTRL) != 0) string += "CTRL-";
    if ((e.stateMask & SWT.COMMAND) != 0) string += "COMMAND-";
    if ((e.stateMask & SWT.SHIFT) != 0) string += "SHIFT-";
    switch (e.keyCode)
    {
      case SWT.BS:  string += "BACKSPACE"; break;
      case SWT.CR:  string += "CARRIAGE RETURN"; break;
      case SWT.DEL: string += "DELETE"; break;
      case SWT.ESC: string += "ESCAPE"; break;
      case SWT.LF:  string += "LINE FEED"; break;
      case SWT.TAB: string += "TAB"; break;
      default:      string += e.character; break;
    }
    System.out.println (string);
  }
});
```

Table 4.4 Keyboard entries and their SWT code constants

Key	Key code
Alt	`SWT.ALT`
Arrow (down)	`SWT.ARROW_DOWN`
Arrow (left)	`SWT.ARROW_LEFT`
Arrow (right)	`SWT.ARROW_RIGHT`
Arrow (up)	`SWT.ARROW_UP`
Backspace	`SWT.BS`
Mouse button 1	`SWT.BUTTON1`
Mouse button 2	`SWT.BUTTON2`
Mouse button 3	`SWT.BUTTON3`
Carriage return	`SWT.CR`
Ctrl	`SWT.CTRL`
End	`SWT.END`
Esc	`SWT.ESC`
F1–F12	`SWT.F1-SWT.F12`
Home	`SWT.HOME`
Insert	`SWT.INSERT`
Line feed	`SWT.LF`
Mod1–Mod4	`SWT.MOD1-SWT.MOD4`
Page Down	`SWT.PAGE_DOWN`
Page Up	`SWT.PAGE_UP`
Shift	`SWT.SHIFT`
Tab	`SWT.TAB`

This code uses the `KeyEvent` fields and the public constants to create a `String` that displays the name of the pressed key and any associated modifier keys. The first step in the event handler's operation involves checking the event's `stateMask` field to see whether the Alt, Ctrl, Shift, and Command keys are pressed. If so, the name of the modifier key is added to the `String`. The method continues by checking whether the event's `keyCode` corresponds to an alphanumeric character or one of the support keys. In either case, the name of the key is appended to the String, which is sent to the console.

The TraverseEvent fires when the user presses a key to progress from one component to another, such as in a group of buttons or checkboxes. The two fields contained in this class let you control whether the traversal action will change the focus to another control, or whether the focus will remain on the widget that fired the event. The simplest field, doit, is a boolean value that allows (TRUE) or disallows (FALSE) traversal for the given widget. The second field of the TraverseEvent class, detail, is more complicated. It's an integer that represents the identity of the key that caused the event. For example, if the user presses the Tab key to switch to a new component, the detail field will contain the SWT constant TRAVERSE_TAB_NEXT.

Each type of control has a different default behavior for a given traversal key. For example, a TraverseEvent that results from a TRAVERSE_TAB_NEXT action will, by default, cause a traversal if the component is a radio button, but not if it's a Canvas object. Therefore, by setting the doit field to TRUE, you override the default setting and allow the user to traverse. Setting the field to FALSE keeps the focus on the component.

The use of the VerifyEvent is similar to that of the TraverseEvent. The goal is to determine beforehand whether the user's action should result in the usual or default behavior. In this case, you can check the user's text to determine whether it should be updated or deleted in the application. Two of the class fields, start and end, specify the range of the input, and the text field contains the input String under examination. Having looked at the user's text, you set the boolean doit field to allow (TRUE) or disallow (FALSE) the action.

4.1.4 Customizing event processing with untyped events

Typed events and listeners enable event processing with classes and interfaces expressly suited to their tasks. Further, typed listeners provide specific methods to receive and handle these events. By narrowing the scope of listeners and events to handle only particular actions, the use of typed components reduces the possibility of committing coding errors.

However, if you prefer coding flexibility over safety, SWT provides untyped events and listeners. When an untyped listener, represented by the Listener class, is associated with a GUI component, it receives every class of event that the component is capable of sending. Therefore, you have to manipulate the catch-all event, represented by the Event class, to determine which action the user performed. The proper event-handling method can then be invoked.

It's important to note that Eclipse.org recommends against using untyped events and listeners. In fact, it mentions that they are "not intended to be used by applications." These mechanisms also aren't included with their typed counterparts in the `org.eclipse.swt.events` package. Instead, both the untyped `Listener` interface and the `Event` class are located in the `org.eclipse.swt.widgets` package.

Despite this, the SWT code snippets provided by the Eclipse website use untyped listeners and events exclusively. This makes coding convenient, since you can create a customized listener that reacts to a specified set of events. An example is shown here:

```
Listener listener = new Listener ()
{
  public void handleEvent (Event event)
  {
    switch (event.type)
    {
      case SWT.KeyDown:
        if (event.character == 'b')
          System.out.println("Key"+event.character);
        break;
      case SWT.MouseDown:
        if (event.button == 3)
          System.out.println("Right click");
      break;
      case SWT.MouseDoubleClick:
        System.out.println("Double click");
      break;
    }
  }
};
Button button = new Button(shell, SWT.CENTER);
button.addListener(SWT.KeyDown, listener);
button.addListener(SWT.MouseDown, listener);
button.addListener(SWT.MouseDoubleClick, listener);
```

In this code, the `Listener` object sends any `Event` instance to its single method, `handleEvent()`. Then, the `Event`'s `type` field determines what processing needs to be done. If the event has type `SWT.Keydown` and the character is the letter *b*, then a statement is sent to the console. If the type is `SWT.MouseDown` and the third mouse button was pressed (that is, the user right-clicked), then the statement *Right click* is shown. If an `SWT.MouseDoubleClick` event fires, then *Double click* is displayed.

You can obtain this capability using typed listeners and events, but the process is more involved. The button needs to add both a `MouseListener` and `KeyListener`, with corresponding adapters. Then, you need to place the event-handling routines in the appropriate listener method. Clearly, untyped event processing is not only more convenient in this case, but also reduces the number of classes necessary to handle the event.

In order to take the place of typed events, the `Event` class contains all the fields in each typed event. It has the same character field as a `KeyEvent` and the same button field as a `MouseEvent`. As shown in the previous code, it also has a field called `type`, which refers to the nature of the event. A listing of these types is presented in table 4.5.

Table 4.5 SWT `type` values for the `Event` class

Values for `type` field			
SWT.Activate	SWT.FocusIn	SWT.KeyUp	SWT.Move
SWT.Arm	SWT.FocusOut	SWT.MenuDetect	SWT.None
SWT.Close	SWT.Expand	SWT.Modify	SWT.Paint
SWT.Collapse	SWT.HardKeyDown	SWT.MouseDoubleClick	SWT.Resize
SWT.Deactivate	SWT.HardKeyUp	SWT.MouseEnter	SWT.Selection
SWT.DefaultSelection	SWT.Help	SWT.MouseExit	SWT.Show
SWT.Deiconify	SWT.Hide	SWT.MouseHover	SWT.Traverse
SWT.Dispose	SWT.Iconify	SWT.MouseMove	SWT.Verify
SWT.DragDetect	SWT.KeyDown	SWT.MouseUp	

4.1.5 *An SWT listener/event application*

Before we discuss the JFace event model, we'll present an SWT `Composite` that integrates and summarizes the material covered. This class, shown in listing 4.1, contains two buttons, a label, and the necessary event processing. We recommend creating a `com.swtjface.Ch4` package to your project and adding this class to it.

Listing 4.1 Ch4_MouseKey.java

```
package com.swtjface.Ch4;
import org.eclipse.swt.events.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.*;

public class Ch4_MouseKey extends Composite
{
  Label output;

  Ch4_MouseKey(Composite parent)
  {
    super(parent, SWT.NULL);
```

```
      Button typed = new Button(this, SWT.PUSH);
      typed.setText("Typed");
      typed.setLocation(2,10);
      typed.pack();

      typed.addKeyListener(new KeyAdapter()
      {
        public void keyPressed(KeyEvent e)
        {
          keyHandler();
        }
      });

      Button untyped = new Button(this, SWT.PUSH);
      untyped.setText("Untyped");
      untyped.setLocation(80,10);
      untyped.pack();
      untyped.addListener(SWT.MouseEnter, UntypedListener);
      untyped.addListener(SWT.MouseExit, UntypedListener);

      output = new Label(this, SWT.SHADOW_OUT);
      output.setBounds(40,70,90,40);
      output.setText("No Event");

      pack();
    }

    Listener UntypedListener = new Listener()
    {
      public void handleEvent(Event event)
      {
        switch (event.type)
        {
          case SWT.MouseEnter:
            output.setText("Mouse Enter");
            break;
          case SWT.MouseExit:
            output.setText("Mouse Exit");
            break;
        }
      }
    };

    void keyHandler()
    {
      output.setText("Key Event");
    }
  }
```

The first button is associated with an anonymous typed listener that receives keyboard events when selected. An `untypedListener` interface is added to the second

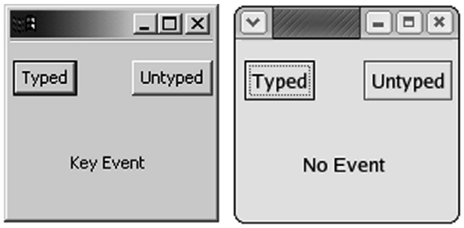


Figure 4.2
The `Ch4_MouseKey Composite`. This example combines many types of SWT classes and interfaces used for event handling.

button, which catches events that occur when the mouse pointer enters and exits the button. Whenever either button fires an event, a `String` is sent to the label.

By integrating this `Composite` in the `CompViewer` application from the previous chapter, the displayed `Shell` should resemble figure 4.2.

The SWT structure of this code allows a widget to receive many types of events and provides for many different responses. But in the majority of GUIs, this isn't necessary. In these cases, SWT's broad capabilities only increase the complexity of coding event processing. Those willing to trade power for simplicity will find the JFace event model very helpful.

4.2 Event processing in JFace

A listener interface can provide the same event handling for different controls, but its usage depends on the component that launched the event. Listeners that receive `MouseEvents` can't be used for menu bar selections. Even untyped `Events` are only useful after the program determines which type of control triggered the event.

But when you're dealing with complex user interfaces, it's helpful to separate the event-handling capability from the GUI components that generated the event. This allows one group to work on a GUI's event handling independently from the group designing its appearance. Also, if a listener's capability can be attached to any component, then its code can be reused more often. Finally, if one section of a program deals strictly with the GUI's view and another is concerned only with event processing, then the code is easier to develop and understand.

JFace provides this separation with its `Action` and `ActionContributionItem` classes. Put simply, an `ActionContributionItem` combines the function of a GUI widget and its attached listener class. Whenever the user interfaces with it, it triggers its associated `Action` class, which takes care of handling the event. Although this may seem similar to SWT's listener/event model, these classes are more abstract, simpler to use, and narrower in scope.

Because these classes are more abstract than their SWT counterparts, it may take time to appreciate their merits. However, once you understand them, we feel certain that you'll use them regularly when handling repetitive event processing. This can be best proven through coding examples. But first, a technical introduction is in order.

4.2.1 Understanding actions and contributions

Although it's interesting to know that you can handle `TraverseEvents` and `ArmEvents` if they occur, few applications use them. Also, it may be fascinating to attach multiple listeners and event handlers to a widget, but GUI components usually perform only a single function in response to a single input type. Because SWT's structure provides for every conceivable component and combination of events, even the simplest listener/event code requires complexity.

It would make event programming easier if a toolset concentrated on only those few widgets and events that are used most often and made their usage as simple as possible. JFace's event-processing structure does exactly this: Its goal is to make event processing more straightforward, allowing programmers to receive and use common events with fewer lines of code. In reaching this goal, JFace makes three assumptions:

- The user's actions will involve buttons, toolbars, and menus.
- Each component will have only one associated event.
- Each event will have only one event handler.

By taking these assumptions into account, JFace simplifies event processing considerably. The first assumption means that contributions only need to take one of three forms. The second assumption provides the separation of contributions from their associated actions; that is, if each contributing component triggers only one event, then it doesn't matter what action is triggered or which component fired the event. The third assumption means that each action needs only one event-handling routine. This simplified event model for SWT/JFace is shown in figure 4.3.

Like the SWT event model, the interface process begins with the `Display` class keeping track of the operating system's event queue. This time, though, it passes information to the `ApplicationWindow`, which contains the `Display`'s `Shell` object. The `ApplicationWindow` creates an `Actionb` class and sends it to the contribution that generated the original event. The contribution then invokes the `run()` method of the `Action` class as the single event handler.

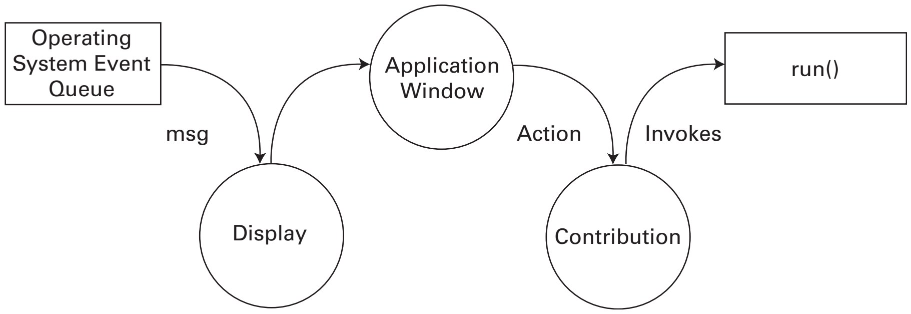


Figure 4.3 By combining listeners and widgets into contributions, this event model is much easier to code.

The *Action* class behaves similarly to SWT's *Event* class, but the contribution capability is more complicated. The two main contribution classes are the *ContributionItem* class and the *ContributionManager* class. The *ContributionItem* class provides individual GUI components that trigger actions, and the *ContributionManager* class produces objects capable of containing *ContributionItem*s. Because these are both abstract classes, event handling is performed with their subclasses. Figure 4.4 shows these inheritance relationships.

Although the `ActionContributionItem` class is one of many concrete subclasses of `ContributionItem`, it's the most important. This class is created and implemented in an `ApplicationWindow` to connect an action to the GUI. It has no set appearance, but instead takes the form of a button, menu bar item, or toolbar item, depending on your use of the `fill()` method.

The second way to incorporate contributions in an application involves the use of a `ContributionManager` subclass. These subclasses serve as containers for

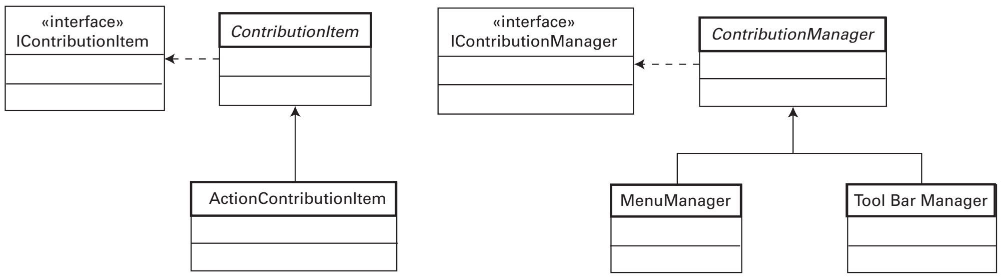


Figure 4.4 The classes and interfaces that provide contribution capability in the SWT/JFace model

ContributionItems, combining them to improve GUI organization and simplify programming. The MenuManager class combines ContributionItems in a window's top-level menu, and the ToolBarManager class places these objects in a toolbar located just under the menu.

4.2.2 Creating Action classes

Listing 4.2 creates a subclass of the abstract Action class called Ch4_StatusAction. This class functions by sending a String to an ApplicationWindow's status line whenever it triggers. We recommend that you add this class to your project directory.

Because this class will be implemented in a toolbar, it needs an associated image. The simplest way to do this is to enter the $ECLIPSE_HOME/plugins/org.eclipse.platform_x.y.z directory, copy the eclipse.gif file, and paste it into the current project folder.

Listing 4.2 Ch4_StatusAction.java

```
package com.swtjface.Ch4;

import org.eclipse.jface.action.*;
import org.eclipse.jface.resource.*;

public class Ch4_StatusAction extends Action
{
  StatusLineManager statman;
  short triggercount = 0;

  public Ch4_StatusAction(StatusLineManager sm)
  {
    super("&Trigger@Ctrl+T", AS_PUSH_BUTTON);
    statman = sm;
    setToolTipText("Trigger the Action");
    setImageDescriptor(ImageDescriptor.createFromFile
      (this.getClass(),"eclipse.gif"));
  }

  public void run()
  {
    triggercount++;
    statman.setMessage("The status action has fired. Count: " +
      triggercount);
  }
}
```

The first thing to observe in this class is what *isn't* present. Although the constructor receives a StatusLineManager object to display output, the Ch4_StatusAction class has no idea what components are firing its action. Therefore, any control that can generate actions can have an associated `Ch4_StatusAction` without additional code. Also, there is only one event-handling routine, `run()`, as opposed to the multiple handlers associated with SWT events.

The `run()` method handles the event processing, but the main work in this class is performed in the constructor. First, it invokes the constructor of its superclass, `Action`, and initializes its `TEXT` and `STYLE` fields. This way, if the `Ch4_StatusAction` is incorporated in a menu, the item label will read Trigger. The *&* before the *T* means that this letter will serve as the accelerator key for the action. The *Ctrl+T* in the `TEXT` field ensures that the action will fire if the user presses the Ctrl and T keys simultaneously.

Beneath the `Action` constructor, further methods are invoked to configure its appearance in the GUI. If it's implemented in a `Composite`, the `Ch4_StatusAction` class will take its form according to the `AS_PUSH_BUTTON` style, as opposed to the `AS_RADIO_BUTTON` or `AS_CHECK_BOX` style. Next, the `setToolTipText()` method initializes the `TOOL_TIP_TEXT` field of the class, creating the `String` that will appear when a mouse pointer hovers over the toolbar item. Finally, the constructor associates an image with the `Ch4_StatusAction` class, which will appear on the toolbar item and button.

Every time the `Ch4_StatusAction` is generated, the `run()` method is invoked. In this case, the `triggercount` accumulator is updated, and a message is sent to the `StatusLineManager` object. In most applications, however, this method will be much more involved in order to serve your event-processing needs.

4.2.3 Implementing contributions in an ApplicationWindow

Because actions and contributions can only be associated with buttons, toolbar items, and menu items, any application demonstrating their capability must rely on these components. So, although a formal introduction to these widgets will have to wait until later chapters, we must include them here for that purpose.

Listing 4.3 shows how `ContributionItem` and `ContributionManager` classes are added to a window. Three contributor classes, `ActionContributionItem`, `MenuManager`, and `ToolBarManager`, all trigger the `Ch4_StatusAction` when acted on. This action sends a message to the status line at the bottom of the window.

We recommend that you create the `Ch4_Contributions` class in `com.swtjface.Ch4` and run the executable with the `Ch4_StatusAction` class in the same directory.

NOTE On many platforms, the `Contribution` operation can't take place unless the OSGi library is added. For this reason, we recommend that you create an `OSGI_LIB` variable and match it to the osgi.jar file located at $ECLIPSE/plugins/osgi_x.y.z/. The full process for adding classpath variables is described in appendix A.

OSGi refers to the Open Services Gateway Initiative, which was formed to enable networking for smart devices in consumer electronics, cars, and homes. Although its widespread adoption seems uncertain at the time of this writing, it's certain that IBM wants it to succeed very badly.

Listing 4.3 Ch4_Contributions.java

```
package com.swtjface.Ch4;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.jface.window.*;
import org.eclipse.jface.action.*;

public class Ch4_Contributions extends ApplicationWindow
{
  StatusLineManager slm = new StatusLineManager();
  Ch4_StatusAction status_action = new Ch4_StatusAction(slm);
  ActionContributionItem aci = new
    ActionContributionItem(status_action);   ❶ Assign status_action
                                                contribution
  public Ch4_Contributions()   ❷ Add resources to
  {                               ApplicationWindow
    super(null);
    addStatusLine();
    addMenuBar();
    addToolBar(SWT.FLAT | SWT.WRAP);
  }

  protected Control createContents(Composite parent)
  {
    getShell().setText("Action/Contribution Example");
    parent.setSize(290,150);
    aci.fill(parent);   ❸ Create button
    return parent;         within window
  }

  public static void main(String[] args)
  {
    Ch4_Contributions swin = new Ch4_Contributions();
    swin.setBlockOnOpen(true);
    swin.open();
    Display.getCurrent().dispose();
  }

  protected MenuManager createMenuManager()
  {
      MenuManager main_menu = new MenuManager(null);
      MenuManager action_menu = new MenuManager("Menu");
      main_menu.add(action_menu);
      action_menu.add(status_action);   ❶ Assign status_action
      return main_menu;                   contribution
    }

    protected ToolBarManager createToolBarManager(int style)
    {
      ToolBarManager tool_bar_manager = new ToolBarManager(style);
      tool_bar_manager.add(status_action);   ❶
      return tool_bar_manager;
    }

    protected StatusLineManager createStatusLineManager()
    {
      return slm;
    }
  }
```

The only difference between this JFace application and those in prior chapters is the introduction of actions and contributions.

❶ Beneath the class declaration, the program constructs an instance of the `Ch4_StatusAction` with a `StatusLineManager` object as its argument. Then, it creates an `ActionContributionItem` object and identifies it with the `Ch4_StatusAction` instance. This contribution has no form yet, but is simply a high-level means of connecting an action to the user interface.

❷ The constructor method creates an `ApplicationWindow` object and adds a menu, toolbar, and status line.

❸ The `createContents()` method sets the title and size of the window and then invokes `aci.fill()`. This method is important since it places the `ActionContributionItem` object in the GUI. In this case, because the `fill()` argument is a `Composite` object, the contributor takes the form of a button that triggers a `StatusEvent` whenever it's pressed.

The last three methods in `Ch4_Contributions` are also straightforward. The `main()` method takes care of creating and opening the window and then disposing of the GUI resources. Then, the `createMenuManager()` method creates a menu instance at the top of the window. Because it's a subclass of `ContributionManager`, an `Action` object can be associated with it, and the `status_action` object is added with the `add()` method. This method is also used in the `createToolBarManager()` method to associate the action instance. In both cases, an `ActionContributionItem` is

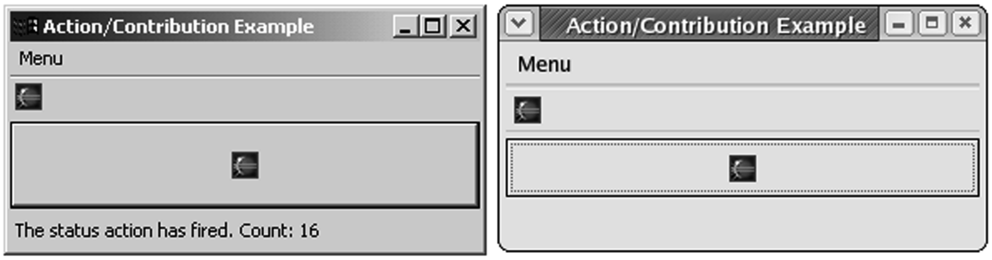

Action/Contribution Example
Menu
The status action has fired. Count: 16
Action/Contribution Example
Menu

Figure 4.5 `Ch4_Contributions`**. This application shows the three ways a** `ContributionItem` **can be incorporated in a window.**

implicitly created and added to the menu in the form of a menu item and to the toolbar as a toolbar item.

Figure 4.5 shows the user interface of `Ch4_Contributions`. The status line at the bottom keeps a running count of the number of `Ch4_StatusActions` that trigger.

4.2.4 Interfacing with contributions

There are two main ways of incorporating an `ActionContributionItem` in a GUI. The first method is to use the `add()` method of a `ContributionManager` subclass, as performed by the `MenuManager` and `ToolBarManager` in the `Ch4_Contributions` application. The second is to use the `fill()` method associated with the `Action-ContributionItem` class and add an SWT widget as its argument. If the argument is a `Composite`, as in `Ch4_Contributions`, then the contributor will appear as determined by the `STYLE` property of the action. If the argument is an SWT `Menu` object, then the contributor will take the form of a menu item. Finally, if the argument is an SWT `ToolBar` object, then the contributor will appear as an item in a toolbar. The characteristics of the `fill()` method are shown in table 4.6.

Table 4.6 Overloaded fill() methods of the `ActionContributionItem` **and their associated appearances**

`fill()` method	GUI implementation (appearance)
`fill(Composite)`	`According to Action's STYLE property`
`fill(Menu, index)`	`MenuItem` with index position
`fill(ToolBar, index)`	`ToolBarItem` with index position

An interesting characteristic of the `ContributionManager` class is that its `add()` method is overloaded to accept arguments of both `Action` and `ActionContributionItem` classes. So, you can associate a `ContributionItem` with a `ContributionManager` implicitly (with the `Action`) or explicitly (with the `ActionContributionItem`). But there's a fundamental difference: You can perform implicit contribution association repeatedly with the same `Action` object, as shown in the `Ch4_Contributions` class. Explicit contribution association can be performed only once.

4.2.5 Exploring the Action class

Although `Ch4_StatusAction` was simple to code and understand, you need to keep in mind many more aspects of the `Action` class. The `Action` class contains a large number of methods to enhance the capability of your user interface. These have been divided into categories and listed in the tables that follow.

The first set of methods, shown in table 4.7, is important in any implementation of the `Action` class. The first and most important method is `run()`. As we mentioned earlier, this is the single event-handling routine in an `Action` class, and it's invoked every time the action is triggered. The next method in the table serves as the default constructor. In addition, constructor methods initialize the member fields associated with the `Action` class, which we'll fully describe shortly.

Table 4.7 Important methods of the `Action` class

`Action` method	Function
`run()`	Performs event processing associated with the `Action`
`Action()`	Default constructor
`Action(String)`	Constructor that initializes the `TEXT` field
`Action(String, ImageDescriptor)`	Constructor that initializes the `TEXT` field and associates an image with the `Action`
`Action(String, int)`	Constructor that sets the `TEXT` and `STYLE` fields

As shown in the `Ch4_StatusAction` code sample, an instance of the `Action` class contains a number of fields that provide information about displaying the `Action` in a GUI. You can access and manipulate these fields using the methods listed in table 4.8. The `TEXT` field, set and accessed by the first two methods, contains a `String` that displays a title or menu item description in a contributor. The next two deal with the `DESCRIPTION` field, which is generally written to a status line to provide additional help. When the user rests the pointer on a contributor, the

String in the TOOL_TIP_TEXT field is shown. The last two methods in this table set and access the IMAGE property of the Action class, which contains a String representing an object of the ImageDescriptor class. As we'll further explain in chapter 7, an ImageDescriptor isn't an image, but an object that holds information needed to create one.

Table 4.8 Property methods for the Action class

Action property method	Function
setText(String)	Sets the TEXT field
getText()	Returns the TEXT field
setDescription(String)	Sets the DESCRIPTION field
getDescription()	Returns the DESCRIPTION field
setToolTipText(String)	Sets the TOOL_TIP_TEXT field
getToolTipText()	Returns the TOOL_TIP_TEXT field
setImageDescriptor(ImageDescriptor)	Sets the IMAGE field
getImageDescriptor()	Returns the IMAGE field

The final field contained in the Action class is the STYLE. This integer value is set by a constructor and accessed through the getStyle() method listed at the top of table 4.9. The next two methods, setEnabled() and getEnabled(), determine whether the component(s) associated with the Action object can be acted on by the user. If not, they are grayed out by default. The final methods, setChecked() and isChecked(), are useful if the Action is associated with a radio button or checkbox. They're used to set the default state of the button or determine whether the user has checked it.

Table 4.9 Style methods for the Action class

Action style method	Function
getStyle()	Returns the STYLE field
setEnabled(boolean)	Sets the ENABLED field
getEnabled()	Returns the ENABLED field
setChecked(boolean)	Sets the CHECKED field
isChecked(void)	Returns the CHECKED field

Table 4.10 shows the methods that deal with accelerator keys and keyboard conversion. *Accelerator keys* are keyboard shortcuts that accomplish the same function as a mouse click. As mentioned in section 4.1.4, pressed keys are represented in SWT with integer key codes, which include all alphanumeric keys and modifier keys (Alt, Ctrl, Shift, Command). The first method creates an accelerator key for the `Action` object and associates it with an SWT key code. The next method provides the key code for the `Action`'s accelerator key. The next two methods convert back and forth between an accelerator key's key code and its `String` representation. The `removeAcceleratorKey()` method parses text and deletes occurrences of the `Action`'s accelerator key. The last four methods in the table provide conversion between `String`s representing keyboard characters and modifier keys, and their SWT code representations.

Table 4.10 Accelerator key / keyboard methods for the `Action` class

Keyboard method	Function
`setAccelerator(int)`	Set the key code as the `Action`'s accelerator key
`getAccelerator()`	Returns the key code for the `Action`'s accelerator key
`convertAccelerator(int)`	Converts the accelerator key to a `String`
`convertAccelerator(String)`	Converts the `String` to an accelerator key
`removeAcceleratorText(String)`	Removes the accelerator keys from a given `String`
`findKeyCode(String)`	Converts the key name to an SWT key code
`findKeyString(int)`	Converts the key code to a key name
`findModifier(String)`	Converts the modifier name to a modifier key code
`findModifierString(int)`	Converts the modifier key code to a modifier name

Although JFace uses actions to replace the SWT listener/event mechanism, the `Action` class can still incorporate listeners for special-purpose event handling. These methods are shown in table 4.11; they mainly concern the `IPropertyChangeListener` interface. This interface pays attention to user-customized `PropertyChangeEvents`, which fire whenever a given `Object` changes into a different `Object` in a manner you describe. Although dealing with property changes may seem complicated, they let you create custom listener/event relationships instead of being limited to those provided by SWT.

The first two methods in table 4.11 take care of associating and disassociating `PropertyChangeListeners`. You can use the next two methods to test these listeners by triggering property changes, based on a precreated event class or a specified change in a given `Object`. The final methods in this table relate to `HelpListeners`, which deal with the user's attempt to obtain information concerning a given component.

Table 4.11 Listener methods for the `Action` class

`Action` listener method	Function
`addPropertyChangeListener (IPropertyChangeListener)`	Associates a property change listener with the `Action`
`removePropertyChangeListener (IPropertyChangeListener)`	Removes a property change listener from the `Action`
`firePropertyChange(Event)`	Changes a property according to an event
`firePropertyChange (String, Object, Object)`	Changes a property according to old and new objects
`setHelpListener(HelpListener)`	Associates a help listener with the `Action`
`getHelpListener()`	Returns a help listener associated with the `Action`

Table 4.12 lists a group of diverse methods contained in the `Action` class. The first four are used to obtain and access identifiers for both the `Action` class and its definition. The next two, `setMenuCreator()` and `getMenuCreator()`, work with `IMenuCreator` interfaces that can be associated with an `Action` object. This interface provides a simple way of creating a drop-down or pop-up menu when a particular action triggers. The last four methods concern more images that can be linked to an action. When an `Action`'s `ENABLED` field is set to `FALSE`, you can specify which image will represent the action by using the `setDisabledImageDescriptor()` method and retrieve the image with the `getDisabledImageDescriptor()` method. Also, if you want to change an image while a pointer hovers above it, the `setHoverImageDescriptor()` method will set this property.

Table 4.12 Miscellaneous methods of the `Action` class

Method	Description
`setID(String)`	Sets an `Action` identifier
`getID()`	Returns an `Action` identifier
`setActionDefinitionID(String)`	Sets an `Action` definition identifier
`getActionDefinitionID()`	Returns an `Action` definition identifier

continued on next page

Table 4.12 Miscellaneous methods of the `Action` class *(continued)*

Method	Description
`setMenuCreator(IMenuCreator)`	Sets a menu creator for the `Action`
`getMenuCreator()`	Returns a menu creator for the `Action`
`setDisabledImageDescriptor(ImageDescriptor)`	Sets the disabled `Action` image
`getDisabledImageDescriptor()`	Returns the disabled `Action` image
`setHoverImageDescriptor(ImageDescriptor)`	Sets the mouse-hovering image
`getHoverImageDescriptor()`	Returns the mouse hovering image

With these methods, the JFace toolset broadens the functionality of the `Action` class far beyond the simple `Ch4_StatusAction` class. Although you may not need all of them, it's important to know how they function and how they can be used in applications.

4.3 Updating the WidgetWindow

To continue populating the `WidgetWindow` application, this chapter provides a `Composite` subclass containing widgets that receive and respond to user actions. This will incorporate code presented earlier in the chapter.

4.3.1 Building the chapter 4 Composite

Listing 4.4 presents the `Ch4_Composite` class, which subclasses the `Ch4_MouseKey` class from section 4.1 and launches the `Ch4_Contributions` class developed in section 4.2. We recommend that you add this class to the `com.swtjface.Ch4` package.

Listing 4.4 Ch4_Composite.java

```
package com.swtjface.Ch4;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;

public class Ch4_Composite extends Ch4_MouseKey
{
  public Ch4_Composite(Composite parent)
  {
    super(parent);
    Button launch = new Button(this, SWT.PUSH);
    launch.setText("Launch");
    launch.setLocation(40,120);
    launch.pack();
```

```
      launch.addMouseListener(new MouseAdapter()
      {
        public void mouseDown(MouseEvent e)
        {
          Ch4_Contributions sw = new Ch4_Contributions();
          sw.open();
        }
      });
    }
  }
```

The operation of `Ch4_Composite` is simple to understand. By extending the `Ch4_MouseKey` class, it incorporates the typed and untyped SWT listeners associated with that `Composite`. It also adds a third button labeled Launch. When clicked, this button creates an instance of the JFace window that uses actions and contributors to perform event processing.

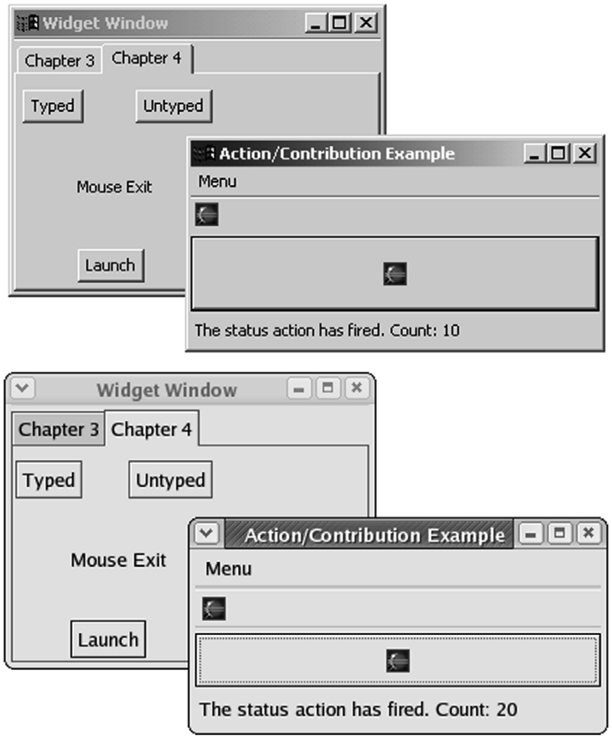
4.3.2 *Adding Ch4_Composite to the WidgetWindow*

Next a tab is added to the `WidgetWindow` `Tabfolder` that comprises the `Composite` created in this chapter. The code for the main `WidgetWindow` application is shown in listing 4.5, with the lines added in this chapter in boldface.

Listing 4.5 The updated WidgetWindow

```
package com.swtjface.Ch2;

import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.jface.window.*;

import com.swtjface.Ch3.*;
import com.swtjface.Ch4.*;

public class WidgetWindow extends Window {

  public WidgetWindow() {
    super(null);
  }

  protected Control createContents(Composite parent) {
    TabFolder tf = new TabFolder(parent, SWT.NONE);

    TabItem chap3 = new TabItem(tf,SWT.NONE);
    chap3.setText("Chapter 3");
    chap3.setControl(new Ch3Comp(tf));

    TabItem chap4 = new TabItem(tf,SWT.NONE);
    chap4.setText("Chapter 4");
    chap4.setControl(new Ch4_Composite(tf));
    getShell().setText("Widget Window");
    return parent;
  }

  public static void main(String[] args) {
    WidgetWindow wwin = new WidgetWindow();
    wwin.setBlockOnOpen(true);
    wwin.open();
    Display.getCurrent().dispose();
  }
}
```

Once updated, the `WidgetWindow` should appear similar to the GUI shown in figure 4.6. `Ch4_Contributions` appears when the Launch button is clicked.



The status action has fired. Count: 10

The status action has fired. Count: 20

Figure 4.6
The updated `WidgetWindow`

4.4 Summary

Event handling is simple in theory but complicated in practice. It's obvious that when a user clicks a button or enters text, a software routine should respond. But the process of keeping track of which widget fired the event, what type of event occurred, and which software routine should execute isn't obvious and requires effort. To an extent, the degree of effort depends on the toolset. If the toolset provides processing of as many events as possible, for as many widgets as possible, then you'll pay for this vast scope by having to comply with a complicated code structure.

This is the situation with SWT's event model. Because there are so many different types of events, you need tables 4.1 and 4.5 in order to write responsive code. So many methods are available for responding to events that a separate adapter class becomes necessary. This event processing demands a fair amount of understanding, but when you need to keep track of right-click events and whether the user can traverse a widget, SWT is the best toolset available.

The developers of JFace, on the other hand, used the Pareto Rule in designing the toolset. This rule, applied to GUI programming, states that 80% of the code needed for event processing will deal with only 20% of the available events. Similarly, the majority of these events will be fired by a small set of widgets. By following these rules, the developers of JFace concluded that there is no need for listeners, adapters, or widgets. Instead, JFace performs event processing with actions, which are triggered when a user interfaces the GUI, and contributors, which can take multiple forms but trigger a single action.

Clearly, a user interface of any complexity must incorporate both event-processing methods. Although JFace will provide rapid coding for menus, toolbars, and buttons, SWT is needed to process keyboard actions as well as events related to widgets like `Shells` and tables. Also, JFace's classes won't help you when you need to distinguish between a left click and a right click. Therefore, a GUI developer seeking to provide a maximum of capability with a minimum of code should be familiar with both toolsets.

As shown by the tables in this chapter, effective event programming depends on keeping track of a myriad of rules, classes, and details. Because of this complexity, we thought long and hard about where to present this material in this book. We first planned to present the SWT/JFace event model in the later chapters, but then all of the preceding code would be static. So, to ensure that future code examples will be more helpful to readers, we decided to introduce this convoluted subject early on.

Let's start building dynamic GUIs!

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

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

MANNING $44.95 US/$62.95 Canada

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

Manning Publications Co.   Copyeditor: Tiffany Taylor
209 Bruce Park Avenue   Typesetter: Tony Roberts
Greenwich, CT 06830   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" } );
```

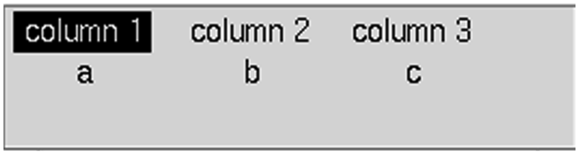


Figure 9.1
A simple three-column table

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 `ClassCastExceptions` 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 `CellEditors`.

As we mentioned in the chapter overview, `CellEditors` 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 `CellEditors` 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 `CellEditors`, 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 `Strings` in `VALUE_SET` holds the values that will be displayed by our `ComboBox-CellEditor`. 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 `Strings` that are the values presented for the user to choose from. However, the editor expects `Integers` 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 `Strings` 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 `CellEditors` 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, `chars` are used to represent letters or numbers on the keyboard. Because a Java `char` can be automatically converted to an `int`, `chars` 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 `MenuItems`. `Menu` extends `Widget` and contains methods for adding `MenuItems` 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);
```

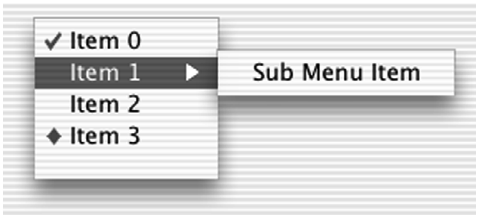


Figure 9.2
Menu types. From top to bottom, `SWT.CHECK`, `SWT.CASCADE`, `SWT.PUSH`, and `SWT.RADIO`.

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.

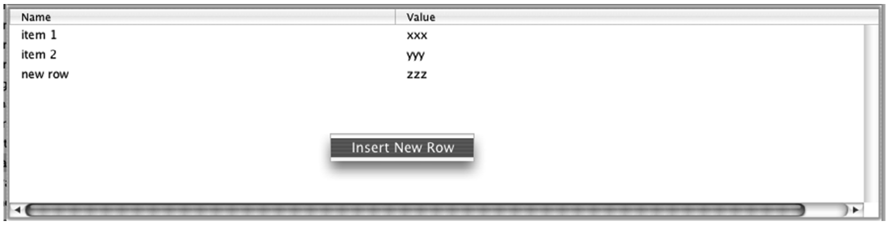
You must keep one caveat in mind when you're using menu managers: Once `fill()` or `createXXX()` has been called on a given instance, `Menu` and `MenuItem` instances are created and cached internally. This is necessary so that the manager can be used to update the menu. However, it also means that you 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



Name	Value
item 1	xxx
item 2	yyy
new row	zzz

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();     ❹ Build menu
  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) {     ❺ getColumnText()
      switch(columnIndex)                                 method
      {
        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(){
    }
    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) {      ❻ getElements()
      return (Object[])inputElement;                           method
    }

    public void dispose() {
    }

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

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) {
        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`.

❸ 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.

❹ 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`.

❺ This is a standard `LabelProvider` implementation, similar to ones you've seen earlier. It returns the value of whichever property matches the requested column.

❻ Our content provider assumes that whatever input it's given is an array of `Objects`. It performs the appropriate cast and returns the result.

❼ Here we construct the table. We add two columns and set the header text.

❽ 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.

❾ 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 Tables and TableViewers 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.

CellEditors, however, are a useful feature unique to TableViewers. CellEditors 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

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

MANNING $44.95 US/$62.95 Canada