JFace [1,2]

by Lori Watson

Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada

Last revised: February 20, 2004

Overview:

JFace is a UI toolkit built on top of SWT. It uses SWT widgets to implement GUI code that is common among different applications. Using JFace, you can create user interface components with less code than if you had started at the basic SWT widget level. For example, to create a message dialog, as will be done in this document, you can either use the SWT widgets directly or you can use the classes provided by JFace. Using SWT, you would create the shell for the dialog, set its size and location, add and position buttons and labels, and maybe even add an image. With JFace, you could make the same dialog by calling one method with three parameters, the rest of the details are handled by the JFace dialog class. This requires less effort and less code each time you use it in an application. In this document we describe how to get started using JFace. As with the previous documents, the focus is not on developing plugins to extend the Eclipse workbench, but rather on using JFace in a stand-alone application.

Window

Window is one of two JFace base window classes. It is built on the SWT Display and Shell widgets we saw earlier in the *Basic SWT Widgets* section, but they are in a class that does some of the common work for us. To create a basic window using JFace, you need to start by importing the following packages:

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

The Window class is an abstract class, and therefore cannot be instantiated. You need to extend Window and call its constructor in the new window class constructor, as follows:

---

[1] This work was funded by an IBM Eclipse Innovation Grant.

[2] © Lori Watson and David Scuse

```
public class JFaceDemo extends Window
{
        JFaceDemo()
        {
                super(null);
        }
}
```

In the main method, you need to create an instance of the JFaceDemo class and add the following code:

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

The setBlockOnOpen method is set to true so that once the window is opened, it will remain open until the user closes it. The open method is what actually displays the window. Once the user closes the window, the control returns to the next line in the code which gets the current display and disposes of it. It is always good practice to clean up after your code. The following basic window is displayed:
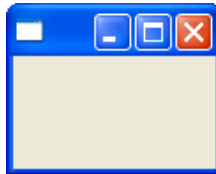
The window's size can be adjusted, as well as other properties, by overriding the createContents method in Window and setting the shell's properties as in the following code:

```
public Control createContents(Composite parent)
{
        parent.getShell().setSize(200,200);
        parent.getShell().setLocation(300,300);
        parent.getShell().setText("Window title");

        return parent;
}
```

The createContents method is called after all widgets have been drawn, but before the window has been displayed. This method is also where we can specify other widgets to be displayed, such as buttons or labels, using the same syntax as in *Basic SWT Widgets*. As an example, add the following code before the return statement of the above method to add a label, text field and button to the window.

```
Composite composite1 = new Composite(parent, SWT.NONE);

Label label = new Label(composite1, SWT.NONE);
label.setText("Read");
label.setLocation(5,5);
label.setSize(50,20);

Text text = new Text(composite1, SWT.BORDER);
text.setText("Write");
text.setLocation(5,25);
text.setSize(50,20);

Button button = new Button(composite1, SWT.PUSH);
button.setText("Click");
button.setLocation(5,50);
button.setSize(50,20);
```

The following window is displayed:
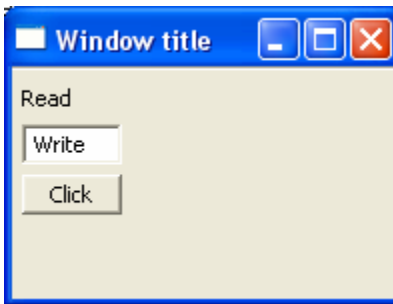
Listeners for the widgets can be added as well, just as was done in the SWT Widget document.

Application Window

The application window is the other base window class in JFace. It is a subclass of the Window class with added functionality which allows us to easily create a standard menu bar, tool bar and status line. To create an application window is the same process as creating a Window, except instead of extending Window, you need to extend ApplicationWindow. If you did this to the above code, you would get an identical looking

window. We will look at how to add a menu bar, tool bar and status line after we introduce JFace's Action.

Action

An action is a command activated by the user either on a menu bar or a tool bar. It allows us to share a behavior between these two widgets, eliminating redundant code. For example, in the Eclipse platform we can save a file by either selecting File>Save from the menu bar, or clicking on the disk icon in the tool bar. Instead of manually adding listeners to handle both events, we can create an Action that we can reuse in both cases.

We will create a subclass of Action called ExitAction to use in our discussion of the menu bar and the tool bar. This Action will be used to close the window. To begin, you need to import the following packages:

```
import org.eclipse.jface.action.*;
import org.eclipse.jface.window.*;
```

Next you need to create the class ExitAction, extending the Action class. It will accept the application window as an argument to the class constructor. The only method you must override is the run method, which defines the steps to be taken with this action, in this case, close the window. So far, this is done in the following code:

```
class ExitAction extends Action
{
        ApplicationWindow window;

        public ExitAction(ApplicationWindow w)
        {
                window = w;
        }
        public void run()
        {
                window.close();
        }
}
```

In the constructor, you can set some properties of the Action. The following code sets the text to be displayed for the action.

```
setText("E&xit@Ctrl+W");
```

The '&' character in front of a letter indicates that the letter is a mnemonic, and the '@' character indicates which accelerator or keyboard shortcut is used. You are now ready to

use this action in a menu bar. We will discuss more properties for Actions when we implement the Tool Bar.

Menu Bar

You have already seen menu bars in the *Basic SWT Widgets* document. Here we will look at how JFace simplifies menu bar creation. To start, you import the following package so that you may use the ExitAction class created above:

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

Next, you create a basic application window as done previously. In this window you declare an ExitAction variable, in this case call it ourAction, and then create an instance of the ExitAction in the constructor as follows:

```
ourAction = new ExitAction(this);
```

To add a standard menu bar to an application window, you add the following before the call to the application window's open method:

```
demo.addMenuBar();
```

You then override the application window's createMenuManager method which returns a menu manager used to create the menu bar. Create an instance of the MenuManager with the following code:

```
MenuManager menuBar = new MenuManager();
```

From here you can add other menu managers to menuBar, creating the skeleton of the menu, but each time passing a string value to MenuManager, which represents the name to be displayed. Notice the mnemonic indicator is again being used in the following code:

```
MenuManager fileMenu = new MenuManager("&File");
menuBar.add(fileMenu);
```

Finally, to create an option for the user to select, you need to add an Action to the menu bar skeleton.

```
fileMenu.add(exitAction);
```

You can also create and add a separator to the skeleton for visual grouping using the following code:

```
Separator separator = new Separator();
```

```
        fileMenu.add(separator);
```

The following code creates a menu bar, the skeleton, and adds an option to exit the window using the ExitAction Action:

```
protected MenuManager createMenuManager()
{
        MenuManager menuBar = new MenuManager();
        MenuManager fileMenu = new MenuManager("&File");
        MenuManager newMenu = new MenuManager("&New");
        MenuManager helpMenu = new MenuManager("&Help");
        Separator separator = new Separator();

        fileMenu.add(newMenu);
        fileMenu.add(separator);
        fileMenu.add(exitAction);
        menuBar.add(fileMenu);
        menuBar.add(helpMenu);
        return menuBar;
}
```

The following window is displayed when selecting the File menu:

Notice that ExitAction can be used to close this window in three ways: by clicking on "File" then "Exit" with the mouse, by simultaneously holding down "Alt" and pressing "f" then "x", or by holding down the Control button and "w" at the same time.

Tool Bar

As with the menu bar, to create a tool bar you need to add the following before the call to the open method:

```
demo.addToolBar(SWT.FLAT | SWT.WRAP);
```

The styles that can be used for a tool bar are FLAT, WRAP, and NONE. You need to override application window's createToolBarManager method, which returns a toolbar manager. In this method you will create an instance of ToolBarManager and then add ExitAction to it, as follows:

```
protected ToolBarManager createToolBarManager(int style)
{
      ToolBarManager toolBarManager = new ToolBarManager(style);
      toolBarManager.add(exitAction);
      return toolBarManager;
}
```

As with the menu bar, we can also use separators in the tool bar to group the buttons. If you were to run this code as it is, the tool bar button would have a small red square on it as a default image. Instead you want to specify in the ExitAction class the image to be displayed. First, you need to import the following into ExitAction:

```
import org.eclipse.jface.resource.*;
import java.net.*;
```

We use Action's setImageDescriptor method which requires an image descriptor for an argument. The following code creates an image descriptor and then sets it for the action:

```
try
{
      ImageDescriptor icon = ImageDescriptor.createFromURL(new
            URL("file:images/close_view.gif"));
      setImageDescriptor(icon);
}catch(MalformedURLException e)
{
      System.err.println(e.getMessage());
}
```
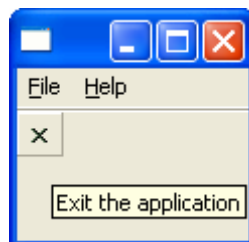
You can also set the tool tip that will be displayed when the mouse hovers over the image with the following code:

```
setToolTipText("Exit the application");
```

The following window is displayed:

Adding the image to the action may also display it on the menu bar, depending on the platform used. The tool bar image will display on all platforms, but not all platforms will display the image on the menu bar as well. In the following window, the image displays

in the menu bar but the line under the 'x' for the mnemonic disappears. The window can still be closed using the mnemonic.
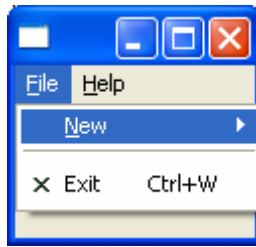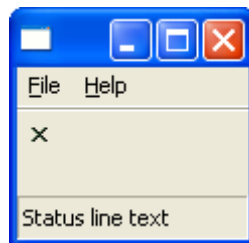

Status Line

The status line is the bar at the bottom of an application that displays information to the user. The following code must be added before the window's open method to create a status line:

```
demo.addStatusLine();
```

There are two ways to set the status line text in the window's createContents method. One way is to call the method setStatus() passing a string representing the text to be displayed. The following code also sets the text in the status line:

```
getStatusLineManager().setMessage("Status line text");
```
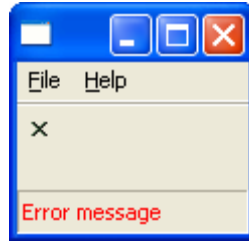
The following window is displayed:

We can also display an error message in the status line with the following code:

```
getStatusLineManager().setErrorMessage("Error message");
```

This displays the status line text in red as follows:

Dialogs:

Dialogs are used to communicate information either to or from the user. To use a JFace dialog in an application you must import the following package:

```
org.eclipse.jface.dialogs.*;
```

Message Dialog

The message dialog displays information to the user. There are five "convenience" methods to quickly create the more commonly used message dialogs. To quickly create a message dialog use the following code, where shell is the display shell:

```
MessageDialog.openWarning(shell, "Title", "Warning message");
```

This produces the following dialog:
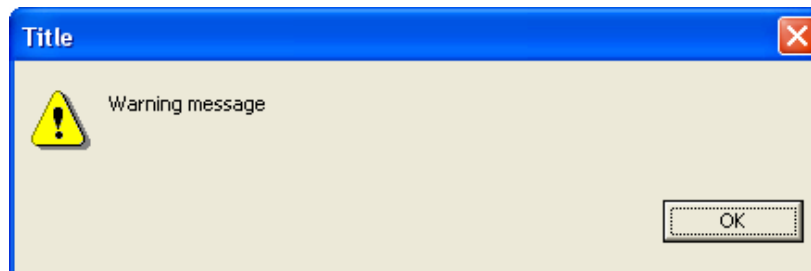
The other convenience methods are openConfirm, openError, openInformation, and openQuestion, all accepting the same parameters. The openWarning, openInformation, and openError message dialogs display only OK buttons as shown above. The openConfirm message dialog displays two buttons: OK and Cancel. This method call will return true if the user selects the OK button and false if the user selects the Cancel button. It is similar for the openQuestion message dialog. It displays both a Yes and a No button. If the user selects the Yes button, the method returns true; if the user selects the No button, the method returns false.

The message dialog can also be easily customized to your application's needs. To create your own buttons, you put the button text in a String array which will be used when

creating the message dialog. The order of the button text will determine the order that the buttons appear in the message dialog.

```
String[] buttonText = new String[]{"Button1", "Button2",
      "Button3"};
```

Then you create the message dialog with the following code:

```
MessageDialog messageBox;
messageBox = new MessageDialog(shell, "Title", null, "Message",
      MessageDialog.ERROR, buttonText, 1);
messageBox.open();
```

The first parameter is for the display's shell. The second parameter is the message dialog title (or null if none). The third parameter is for the message dialog title image (or null if none). The fourth parameter is the message to be displayed. The fifth parameter is the dialog image type. The image types that can be used are: MessageDialog.NONE, MessageDialog.ERROR, MessageDialog.INFORMATION, MessageDialog.QUESTION, and MessageDialog.WARNING. The sixth parameter is a String array of the message dialog button text. The last parameter is the message dialog button text array index which sets the default button focus. After creating the MessageDialog, the MessageDialog's open method must be called. The following message dialog is displayed:
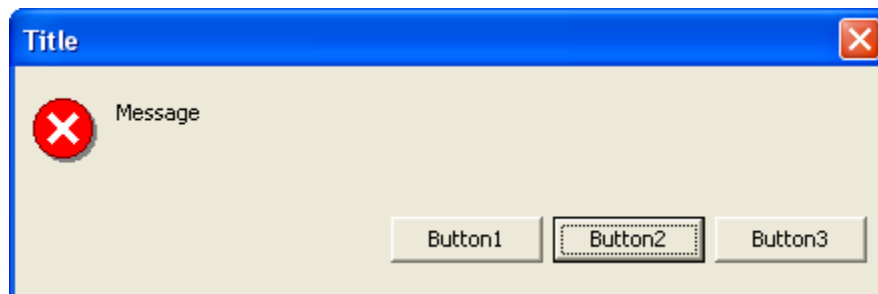
To determine which button was selected, use the following code:

```
messageBox.getReturnCode();
```

This returns an integer which is the array index of the button text. For example, if the user selects Button1, it returns 0, for Button2 it returns 1, etc...

Input Dialog

This dialog accepts an input string from the user and gives either an OK or a Cancel button. The following code creates an input dialog:

```
InputDialog inputBox;
```

```
inputBox = new InputDialog(shell, "Title", "Message",
        "Initial Value", null);
inputBox.open();
```

The first parameter is for the display's shell. The second parameter is the dialog title (or null if none). The third parameter is the message to be displayed to prompt the user (or null if none). The fourth parameter is the initial value to put in the text box (or null if none). The final parameter is for an input validator (or null if none). This will not be expanded on here. To display the input dialog, its open method is called. The following input dialog is displayed:
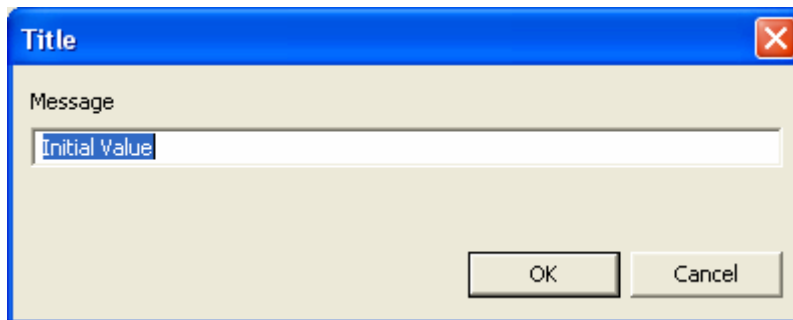
The following code is used to determine the button selected by the user and the input entered in the text box, respectively:

```
inputBox.getReturnCode();
inputBox.getValue();
```

The return code is 0 if the OK button was pressed and 1 if the Cancel button was pressed. If the Cancel button was pressed, the value (from the textbox) will automatically be null. Otherwise, the getValue method will return the String value of the text in the text box.

Error Dialog

This dialog displays one or more errors to the user. It is recommended that the message dialog openError be used instead unless the error object being displayed involves child items.

JFace Viewers

Viewers display an object using different SWT widgets. The concrete viewer types available for quick use are CheckboxTableViewer, CheckboxTreeViewer, ListViewer, TableTreeViewer, TableViewer, and TreeViewer. These concrete viewers have built-in support for filtering and sorting

In general, there are four steps involved in using a viewer. First, we must create an instance of the viewer. Then, we must create and set the viewer's content provider. The content provider works in between the data and the viewer, doing such tasks as retrieving the data elements to be displayed and keeping the viewer's data up to date. The minimum three methods that must be implemented in the content provider are getElements(), dispose(), and inputChanged(). The content provider works with our input, which is the next step; we must set the viewer's input to the object we want to display. Finally, we must create and set the viewer's label provider. The label provider specifies what text or image is to be displayed in a specific label in the viewer. These are the basic steps; depending on the type of viewer being used, there may be more particular steps involved.

List Viewer Example

The list viewer uses the SWT List widget to display the object. For an example using the list viewer, we create a simple, small application that displays a book's author and quantity when the title is selected from the list of titles in the list viewer. First, we need data. To keep this example simple, we create our data in the BookList class, which creates an array of six Books with the title, author, and quantity variables and an accessor for this array. The Book class has a constructor that accepts all three variables as arguments, as well as an accessor method for each of these variables that returns the variable's value as a String. This is done in the following code:

```
public class Book
{
        private String title;
        private String author;
        private int quantity;
        Book(String theTitle, String theAuthor, int theQuantity)
        {
                title = theTitle;
                author = theAuthor;
                quantity = theQuantity;
        }
        public String getTitleAsString()
        {
```

```
                    return title;
            }
            public String getAuthorAsString()
            {
                    return author;
            }
            public String getQuantityAsString()
            {
                    return Integer.toString(quantity);
            }
    }
```

The BookList class looks as follows:

```
    public class BookList
    {
            private Book books[] = new Book[6];

            BookList(){ initializeBookList();}

            private void initializeBookList()
            {
                    books[0] = new Book("Iliad", "Homer", 10);
                    books[1] = new Book("The Odyssey", "Homer", 9);
                    books[2] = new Book("Pride and Prejudice", "Jane
                            Austen", 8);
                    books[3] = new Book("Sense and Sensibility", "Jane
                            Austen", 7);
                    books[4] = new Book("The Europeans", "Henry James",
                            6);
                    books[5] = new Book("Jane Eyre", "Charlotte Bronte",
                            5);
            }
            public Book[] getBookList()
            {
                    return books;
            }
    }
```

Now that we have data, we will create the content provider class which implements the IStructuredContentProvider interface. We need to import the following:

```
    import org.eclipse.jface.viewers.*;
```

As mentioned above, three methods must be implemented, but the only one that we will be using is the getElements method. Here, given the input object, we return an array of

that object to be displayed in the viewer; in this case it will be the BookList. The following code does this:

```
public class BookListContentProvider implements
                                    IStructuredContentProvider
{
        public Object[] getElements(Object inputElement)
        {
                return ((BookList)inputElement).getBookList();
        }

        public void dispose() {}

        public void inputChanged(Viewer v, Object o, Object n){}
}
```

Next we will create the label provider class which extends LabelProvider. This class, as with the content provider, also needs to import the JFace viewers package. Here we specify how the text for the viewer is to be obtained, as in the following code:

```
public class BookListLabelProvider extends LabelProvider
{
        public String getText(Object element)
        {
                return ((Book)element).getTitleAsString();
        }

}
```

Now we can create our main application window and create an instance of the list viewer. We create the list viewer in the application window's createContents method with the following code:

```
ListViewer listViewer = new ListViewer(composite, SWT.H_SCROLL);
```

The first argument is for the composite to which the viewer is added and the second parameter is the style. The available style options are MULTI, H_SCROLL, V_SCROLL and BORDER and can be combined using the '|' character. All four are used as the default style when the style argument is not specified. If MULTI is not specified in the given style then the list will only be able to select single items. The next three lines set the content provider, the input and the label provider, respectively:

```
listViewer.setContentProvider(new BookListContentProvider());
listViewer.setInput(new BookList());
listViewer.setLabelProvider(new BookListLabelProvider());
```

In addition to this, we add two text fields, txtAuthor and txtQuantity, which will display the author and quantity of the selected book title in the list viewer. For this we will add a listener that will listen for a selection made in the list viewer and display the corresponding information as in the following code:

```
listViewer.addSelectionChangedListener(new
                                ISelectionChangedListener()
{
     public void selectionChanged(SelectionChangedEvent event)
     {
          IStructuredSelection selection =
               (IStructuredSelection)event.getSelection();
          Book selectedBook =
               (Book)selection.getFirstElement();
          txtAuthor.setText(selectedBook.getAuthorAsString());
          txtQuantity.setText(
               selectedBook.getQuantityAsString());
     }
});
```

An example of this application is shown in the following window: