Customizing Eclipse RCP applications

Techniques to use with SWT and JFace

Skill Level: Intermediate

Scott Delap (scott@clientjava.com)
Desktop/Enterprise Java Consultant

Annas Andy Maleh (andy@obtiva.com)
Consultant
MichaelDKelly.com

27 Feb 2007

Most developers think that an Eclipse Rich Client Platform (RCP) application must look similar in nature to the Eclipse integrated development environment (IDE). This isn't the case, however. This tutorial will explain a number of simple techniques you can use with the Standard Widget Toolkit (SWT) and JFace to create applications that have much more personality than the Eclipse IDE.

Section 1. Before you start

About this tutorial

This tutorial will explain a number of UI elements that can be changed in Eclipse RCP, JFace, and SWT. Along the way, you will learn about basic changes you can make, such as fonts and colors. You will also learn advanced techniques, including how to create custom wizards and section headers. Using these in conjunction should provide you the ability to go from a typical-looking Eclipse RCP application to a distinctive but visually appealing one.

© Copyright IBM Corporation 2007. All rights reserved.

Prerequisites

You should have a basic familiarity with SWT, JFace, and Eclipse RCP.

System requirements

To run the examples, you need a computer capable of adequately running Eclipse V3.2 and 50 MB of free disk space.

Section 2. Heavyweight and lightweight widgets

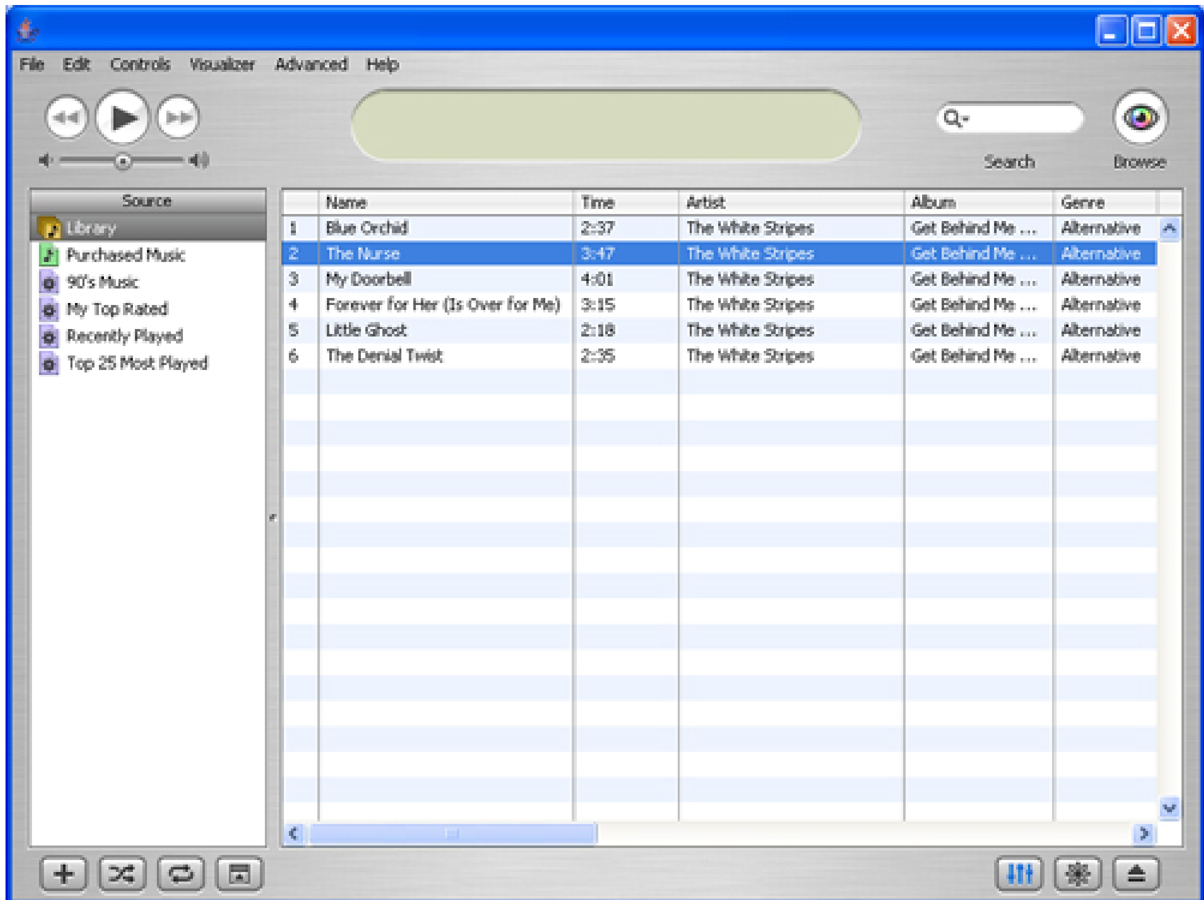
Before diving into techniques that can be used to modify SWT, JFace, and Eclipse RCP in general, it's important to cover the fundamental characteristics of SWT and how they apply to the appearance of the widget set. SWT is a wrapper around native code objects, such as GTK+ objects. Because of this, SWT widgets are often referred to as *heavyweight*. In cases where native platform GUI libraries don't support the functionality required for SWT, SWT implements its own GUI code in the Java™ programming language. This is similar to another popular Java widget toolkit: Swing. In essence, SWT is something of a compromise between the performance of native widgets, the look and feel of a toolkit like Abstract Windows Toolkit (AWT), and Swing's high-level ease of use. The AWT is also a heavyweight toolkit.

In contrast, Swing is a *lightweight toolkit*, meaning that Java technology handles all the painting activities related to the painting and visual appearance of each widget.

This difference in implementation has resulted in the stereotype that SWT applications have to rigidly appear as native applications. Although this is the goal of the SWT API in general, there is always room for interpretation and enhancement. In the past, developers used Swing to construct nonnative-looking UIs, such as this reimplementation (see Resources) of the iTunes theme shown below.

Figure 1. The iTunes interface recreated in Swing

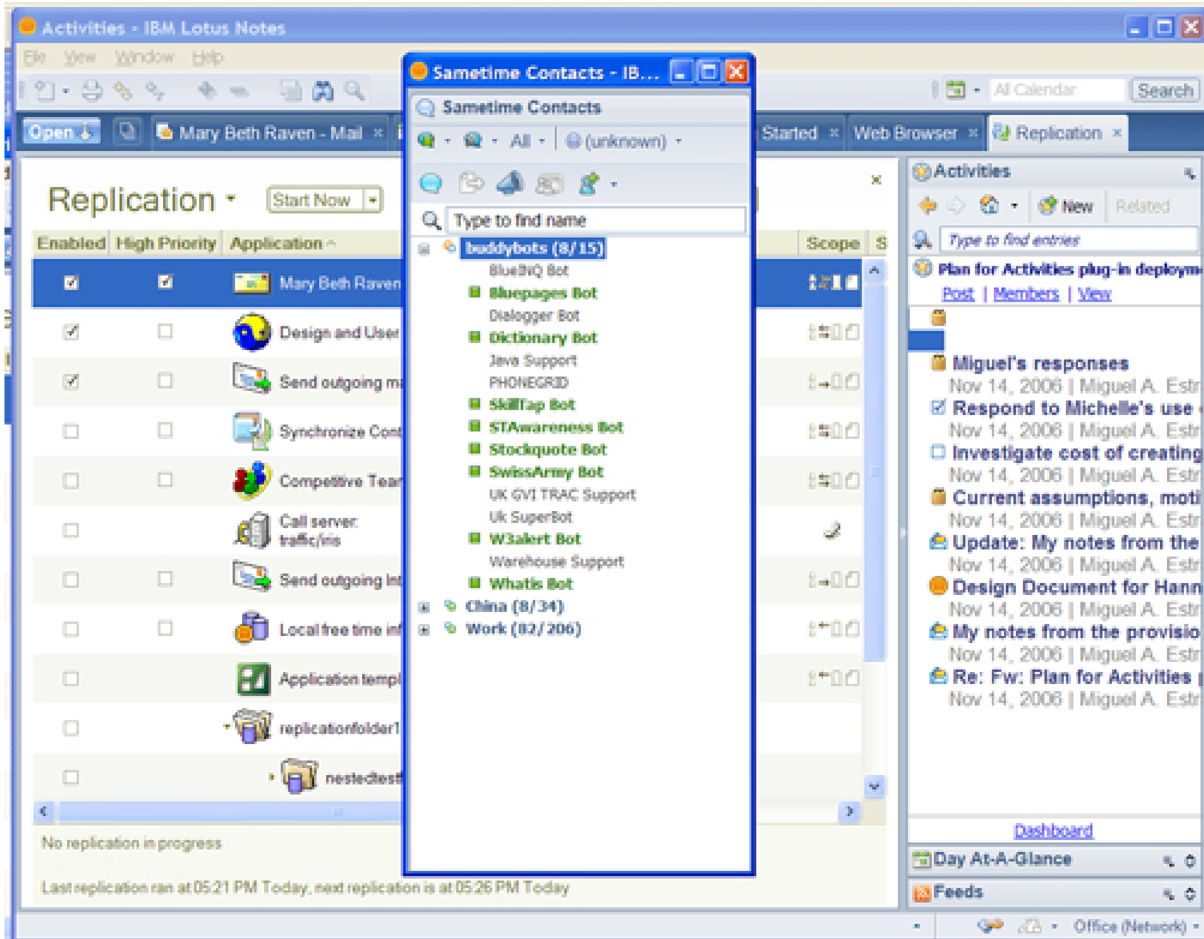
© Copyright IBM Corporation 2007. All rights reserved.



The IBM® Lotus Notes® team has done a good job in dispelling the myth (see Resources) of what a SWT UI can look like.

Figure 2. The Lotus Notes Eclipse RCP-based application

© Copyright IBM Corporation 2007. All rights reserved.



This tutorial covers a number of techniques you can use to provide UI variations outside of the normally thought-of appearance. Modifications to heavyweight UI widgets such as labels and tables are provided along with options for SWT-implemented UI elements such as FormToolkit and CTabs.

Section 3. Operating system-level theming

The most basic way to modify Eclipse is to take advantage of the built-in theming of the operating system SWT is running on. These changes can be applied without the need to write custom code.

Windows

In versions of Eclipse prior to 3.2, a file called java.exe.manifest (see Resources) was needed in the jre\bin directory of the Java Virtual Machine (JVM) running

© Copyright IBM Corporation 2007. All rights reserved.

Eclipse to allow it to use of the Microsoft® Windows® XP theme. This is no longer necessary for Eclipse V3.2 and later. The moral of the story: upgrade.

GTK

On GTK, the .gtkrc or .gtkrc-2.0 file in your home directory can be used to modify default colors and fonts (see Resources).

Listing 1. Example GTK properties

```
style "eclipse" {
  font_name = "Sans 12"
  bg[NORMAL] = "#d6d3ce"
  bg[ACTIVE] = "#c7c2bc"
  bg[INSENSITIVE] = "#828282"
  bg[PRELIGHT] = "#3a6ea5"
  fg[NORMAL] = "#000000"
  fg[ACTIVE] = "#000000"
  fg[INSENSITIVE] = "#d4d0c8"
  fg[PRELIGHT] = "#ffffff"
}
class "GtkWidget" style "eclipse"
```

Motif

Motif similarly uses the .Xdefaults file to specify the colors and fonts Eclipse uses.

Listing 2. Example Motif properties

```
Eclipse*spacing:0
Eclipse*XmForm.background:#e8e7e3
Eclipse*XmList.background:#e8e7e3
Eclipse*XmTextField.background:#e8e7e3
Eclipse*background:#d6d3ce
Eclipse*fontList:-misc-fixed-medium-r-normal-*-10-100-75-75-c-60-iso8859-1
```

Section 4. Workspace setup

Before going through the many code examples in this tutorial, you need to set up a sandbox workspace. You can do this two ways:

- Select a new workspace location on Eclipse IDE startup.
- If your IDE defaults to the same workspace every time at startup, select **File > Switch Workspace**.

© Copyright IBM Corporation 2007. All rights reserved.

After you establish a workspace, a few support projects are needed. Select **File > Import**. Select **plug-ins and Fragments** from the **plug-in Development** category, then click **Next**. Click **Next** on the following screen, then select the following projects for import and add them to the list on the right:

- org.eclipse.swt
- org.eclipse.swt.win32.win32.x86 (or platform-specific SWT plug-in)
- org.eclipse.equinox.common
- org.eclipse.core.commands

Click **Finish** to import. Next, download the archive referenced in the Resources section. Right-click in the Package Explorer and select **Import**. This time, select **Existing Projects into Workspace** from the **General** category, then click **Next**. Select the **Archive File** option and browse to find the archive project file. Click **Finish** to import.

Confirm that all the code in the tutorial project compiles without errors. If it doesn't, you may need to modify the project's required libraries. To do so, right-click the project, and select **Properties**. Next, go to the Java Build path. Under the **Projects** tab, remove all projects. Finally, click **Add** to read the projects that were imported as dependencies. Upon a clean compile, all errors should disappear. Now you're ready to begin widget customizations.

Section 5. Change fonts and colors

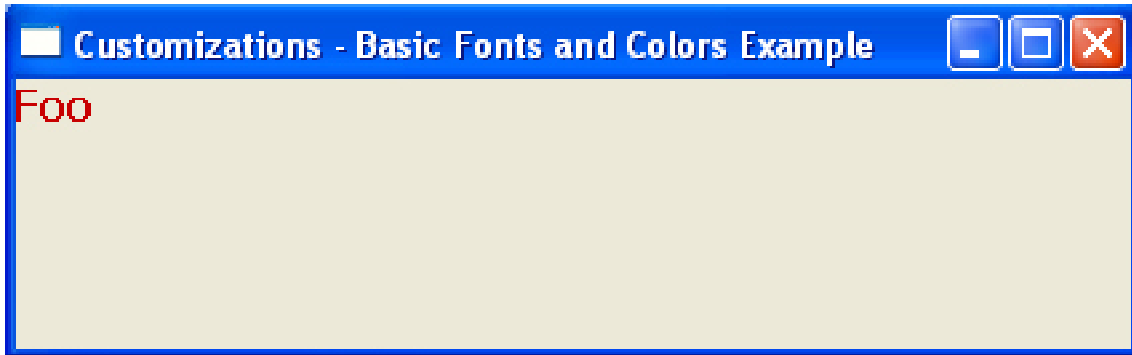
When most developers think of customization, they think of special widgets and custom painting. However, some of the most effective customizations in the appearance of an application are the simplest. You can easily change the fonts and colors of individual widgets to enhance usability. The example snippet in Listing 3 changes the font type and size of a label to make it more suitable as a section heading. Colors work in a similar manner, as shown in Listing 4 and Figure 3.

Listing 3. Change fonts and colors

```
Label label = new Label(parent, SWT. NONE);
label.setText("Foo");

label.setFont(JFaceResources.getFont\
Registry().get(JFaceResources.HEADER_FONT));
Color color = new
Color(Display.getCurrent(), 200, 0, 0);
label.setForeground(color);
```


© Copyright IBM Corporation 2007. All rights reserved.

Figure 3. Label with a custom color



This may seem trivial at first, but it's quite powerful. Consider label colors with respect to validation. Listing 4 shows an example SWT form that contains a few text controls/labels.

Listing 4. A basic set of label text pairs

```
parent.setLayout(new GridLayout(2, true));
Label labelFirst = new Label(parent, SWT. NONE);
labelFirst.setText("First:");

Text textFirst = new Text(parent, SWT.BORDER);

Label labelLast = new Label(parent, SWT. NONE);
labelLast.setText("Last:");

Text textLast = new Text(parent, SWT.BORDER);
```

As currently written, it's difficult for the user to determine which fields are invalid at what time. Adding color can make all the difference. Modify the method as shown below.

Listing 5. Modify label color on text change

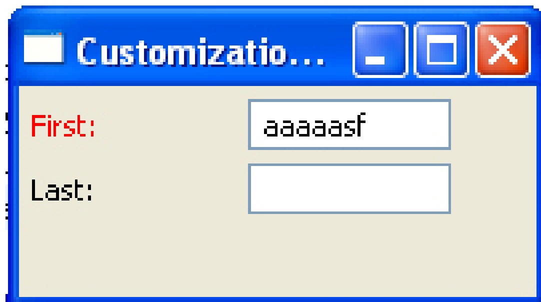
```
parent.setLayout(new GridLayout(2, true));
final Label labelFirst = new Label(parent, SWT. NONE);
labelFirst.setText("First:");

originalColor = labelFirst.getForeground();

final Text textFirst = new Text(parent, SWT.BORDER);

textFirst.addModifyListener(new ModifyListener(){

        public void modifyText(ModifyEvent e) {
                if (textFirst.getText().length() > 5) {
                        labelFirst.setForeground(validationColor);
                } else {
                        labelFirst.setForeground(originalColor);
                }
            textFirst.redraw();
        }});
```


© Copyright IBM Corporation 2007. All rights reserved.

```
Label labelLast = new Label(parent, SWT. NONE);
labelLast.setText("Last:");

Text textLast = new Text(parent, SWT.BORDER);
```

Next, right-click the class and select **Run as an SWT application**. Type more than five letters into the first text widget and watch the change from black to red when the text is invalid, as shown in Figure 4. This form is now far more user-friendly.

Figure 4. Label color change on text change



Operating system resources

A discussion of colors and fonts in SWT isn't complete without mentioning how the API handles them as resources. SWT is designed with its fonts and colors allocating operating system resources. As a result, these resources must be disposed when the application is finished using them to avoid application leaks. If you create a resource, dispose of it. If you use an existing resource, let the owner of that resource handle disposing it. In the previous example, you can add a dispose listener to the overall parent container, which disposes of any color resources.

Comparatively, however, this would not be correct because each label might have a font supplied internally instead of by the developer after creation. Disposing of such a resource would leave the control in an incorrect state.

Section 6. Buttons

Buttons are also a common UI element and are frequently customized. The most basic way to customize a button is via font and color properties similar to what you did with labels in the previous section. Create a class in your example Eclipse project using the snippet below.

```
Button button = new Button(parent, SWT.NONE);
button.setText("Press Me");
```

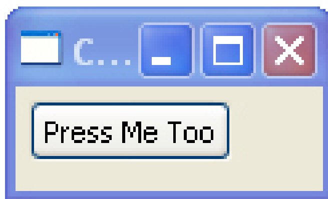

© Copyright IBM Corporation 2007. All rights reserved.

Next, add the font-style adjustments from Listing 6. Right-click the class and choose **Run as SWT application** to see the changes reflected as shown below.

Listing 6. A basic button with font changes

```
Button button = new Button(parent, SWT.NONE);
button.setText("Press Me Too");

Color blueColor = new Color(Display.getCurrent(), 0, 0, 255);
FontData[] fontData = button.getFont().getFontData();
fontData[0].setStyle(SWT.BOLD);
button.redraw();
```

Figure 5. The button with font changes



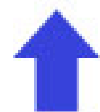
Although these simple changes suffice for many cases, UI designs often call for nontraditional but button-like elements. For example, an **Up** button can be more easily represented with an image. Change the `addChildControls` method in the example to instead include the snippet below.

Listing 7. A new Up button

```
FormToolkit toolkit = new FormToolkit(Display.getCurrent());
ImageHyperlink imageHyperlink = toolkit.createImageHyperlink(
                parent, SWT.NONE);
Image image = new Image(Display.getCurrent(),
            ImageHyperLinkExample.class.getResourceAsStream(
              "up.jpg"));
imageHyperlink.setImage(image);
```

The result of these changes is shown below.

Figure 6. The new Up button

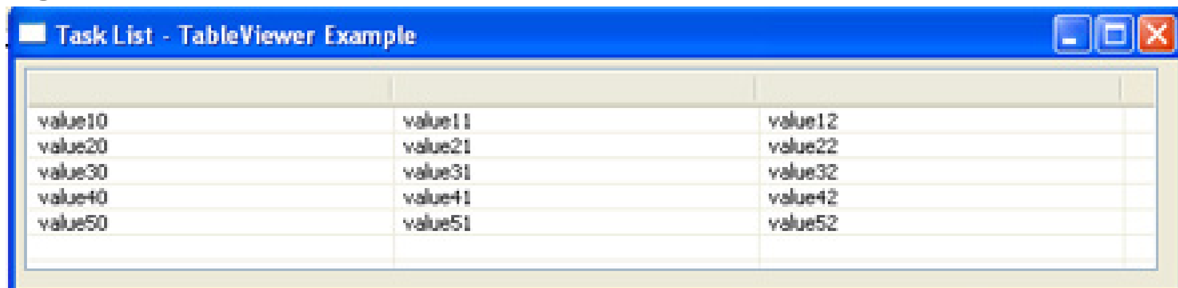


Section 7. Tables

© Copyright IBM Corporation 2007. All rights reserved.

Tables are used extensively for the presentation of collections of data in Eclipse RCP applications. Two of the most effective modifications you can make involve images and background colors. Run the TableViewerExample included in the project provided with this tutorial. You'll see an example table similar to the one below.

Figure 7. A basic table



This table is relatively plain. The first way to make it more visually appealing is to add images to the cells. The JFace TableViewer API uses LabelProviders to determine the text and images for each row. Changing the `getColumnImage` method causes images to be displayed. To try this, modify the BeginningLabelProvider as shown below.

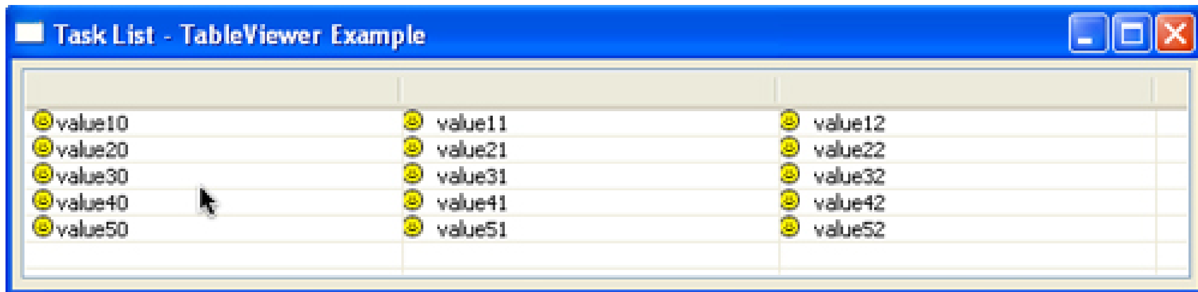
Listing 8. An image label provider

```
public class BeginningLabelProvider
        extends LabelProvider
        implements ITableLabelProvider {

        private static ImageRegistry imageRegistry = new ImageRegistry();

        static {
                imageRegistry.put("smileImage", ImageDescriptor.createFromFile(
                                TableViewerExample.class,
                                "smile.jpg"));
        }
        public String getColumnText(Object element, int columnIndex) {
                return ((String) element) + columnIndex;
        }

        public Image getColumnImage(Object element, int columnIndex) {
                return imageRegistry.get("smileImage");
        }
}
```

Right-click the TableViewerExample and select **Run as SWT application**. You should see a window similar to the one shown below.

Figure 8. Table viewer with images

© Copyright IBM Corporation 2007. All rights reserved.



Although images are a step in the right direction, it would be nice to also have alternating row colors. You can modify row colors by implementing the IColorProvider interface. To do this in the BeginningLabelProvider, first change the `implements` method to include this interface. Next, choose the **Override and Implement** from the **Source** menu. After clicking **OK**, you should have the new methods `getBackground` and `getForeground`. Next, modify the constructor to include a reference to the TableViewer. Finally, modify the `getBackground` method as shown below.

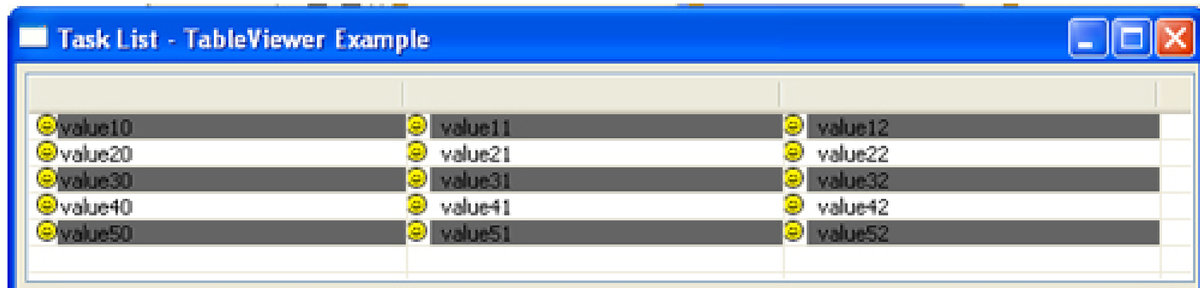
Listing 9. Creating a color provider for alternating colors

```
public class ImageAndColorProvider
        extends LabelProvider
        implements ITableLabelProvider, IColorProvider{

        private static ImageRegistry imageRegistry = new ImageRegistry();
        private TableViewer tableViewer;
        private Color gray = new Color(Display.getCurrent(), 100, 100, 100);

        static {
                imageRegistry.put("smileImage", ImageDescriptor.createFromFile(
                                TableViewerExample.class,
                                "smile.jpg"));
        }

        public ImageAndColorProvider(TableViewer tableViewer) {
                this.tableViewer = tableViewer;
        }
        public String getColumnText(Object element, int columnIndex) {
                return ((String) element) + columnIndex;
        }

        public Image getColumnImage(Object element, int columnIndex) {
                return imageRegistry.get("smileImage");
        }

        public Color getBackground(Object element) {
                ArrayList list = (ArrayList) this.tableViewer.getInput();
                int index = list.indexOf(element);
                if ((index % 2) == 0) {
                        return gray;
                } else {
                        return null;
                }
        }

        public Color getForeground(Object element) {
                return null;
        }
 }
```


© Copyright IBM Corporation 2007. All rights reserved.

Right-click the TableViewerExample and select **Run as SWT application**. You should see a window similar to the one shown below.

Figure 9. Table with alternating row colors



Note that this example has a basic implementation in the `getBackground` method to determine the elements index for color purposes. In large tables, a more efficient algorithm is recommended.

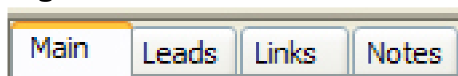
Section 8. Tabs

Tabs are where the mixed heavyweight and lightweight nature of SWT begins to appear. The previous widgets were all heavyweight. Tabs, on the other hand, feature both heavy and lightweight implementations.

TabItems and TabFolders

You can insert tabs into Eclipse RCP UIs using the TabFolder and TabItem widgets. The TabFolder widget displays tabs and allows the user to select a particular tab. TabItem holds the contents of a single tab. Figure 10 illustrates the look of basic tabs.

Figure 10. Basic tabs



TabFolders can be customized in two ways. You can modify the orientation to change whether tabs appear on the top or bottom by passing the style bit SWT.TOP or SWT.BOTTOM to the TabFolder constructor. You can change the font of the tab text using `setFont`, similar to the technique discussed for buttons and labels.

To see these changes in action, run the TabFolderExample in the tutorial project.

© Copyright IBM Corporation 2007. All rights reserved.

After you see the initial appearance, replace the TabFolder constructor with the lines below.

Listing 10. Change the tab font

```
TabFolder tabFolder = new TabFolder(parentComposite, SWT.BOTTOM);
FontData[] fontData = tabFolder.getFont().getFontData();
            for (int i = 0; i < fontData.length; i++) {
                FontData fontDatum = fontData[i];
                fontDatum.setHeight(9);
           fontDatum.setStyle(SWT.BOLD);
            }
            FontRegistry fontRegistry = JFaceResources.getFontRegistry();
            fontRegistry.put("tab.font", fontData);
            tabFolder.setFont(fontRegistry.get("tab.font"));
```

Right-click the TabFolderExample and select **Run as SWT application**. You should see a series of tabs similar to those below.

Figure 11. Tabs with a bigger font

TabItem can also be customized in two ways. You can modify the width by padding the tab text with extra white space to the left and right when using the `setText` method. You can also customize the basic look by setting an image using the `setImage` method.

To demonstrate, add a few padding spaces to the text set on the TabItem in the `buildTabItem` method.

```
tabItem.setText("   " + text + "   ");
```

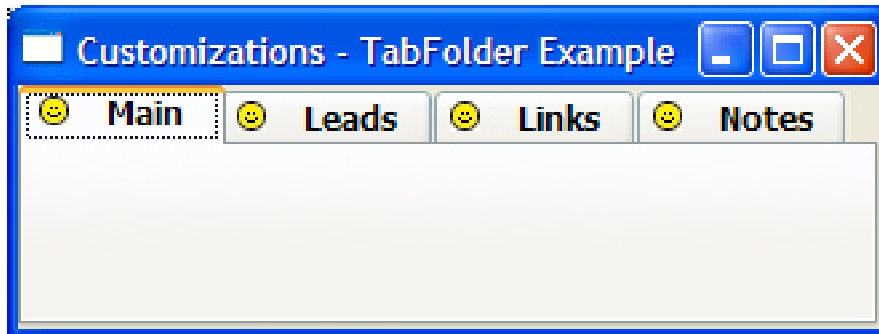
Next, modify the tabs to include an image, as shown below.

```
Image image = imageRegistry.get("smileImage");
tabItem.setImage(image);
```

After these changes, run the example again. You should see a set of tabs similar to those below.

Figure 12. Bigger tabs with an image

© Copyright IBM Corporation 2007. All rights reserved.



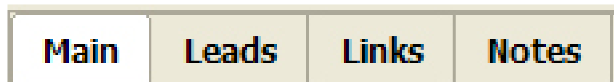
Section 9. CTabFolder and CTabItem

Companies sometimes require further customizations, such as changing the tab colors to match the company's standard color theme. Because TabFolder and TabItem are native widgets, they aren't very customizable and may not satisfy all business customization needs. On the other hand, CTabFolder and CTabItem are custom-drawn counterparts that are customizable. CTabFolder includes all the customizations in TabFolder, along with a number of additions.

Tab height

You can make tabs stand out more by increasing their height. Alternatively, you can save space by decreasing their height. Both tasks can be accomplished using the `setTabHeight` method.

Figure 13. Simple CTabs




```
tabFolder.setTabHeight(25);
```

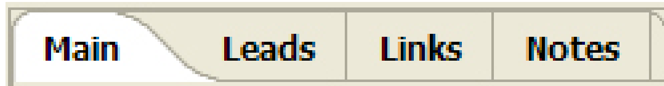
This technique can be used in conjunction with larger font sizes to make the tab titles really stand out.

Fancy look

You can give your tabs a fancy look that has round edges, just like the tabs used for editors and views in the Eclipse IDE by default. To do so, use the `setSimple` method and pass it a `false` Boolean value.

Figure 14. Fancy CTabs

© Copyright IBM Corporation 2007. All rights reserved.

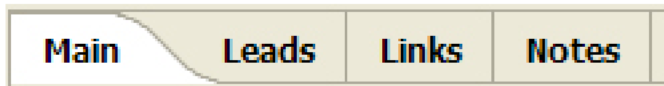



```
tabFolder.setSimple(false);
```

Border visibility

The rounded-tab border can be toggled off using the `setBorderVisible` method.

Figure 15. Border turned off




```
tabFolder.setBorderVisible(false);
```

Run the CTabFolderExample to see these properties in action.

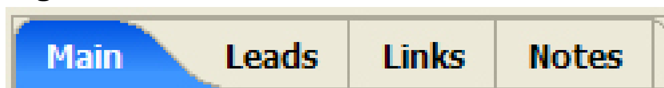
Tabs can also have a number of their color properties modified. You can customize the background color of selected tabs to be a gradient formed from multiple colors. In addition, you can customize the foreground color of a selected tab to fit with the background color using the method `setSelectionForeground`. To demonstrate, open the CTabFolderExample in the Java editor. Add the code below after the construction of the CTabFolder.

Listing 11. A gradient background

```
Display display = Display.getCurrent();
int colorCount = 3;
Color[] colors = new Color[colorCount];
colors[0] = display.getSystemColor(SWT.COLOR_TITLE_BACKGROUND);
colors[1] = display.getSystemColor(SWT.COLOR_TITLE_BACKGROUND_GRADIENT);
colors[2] = colors[0];
int[] percents = new int[colorCount - 1];
percents[0] = 4;
percents[1] = 60;
tabFolder.setSelectionBackground(colors, percents, true);
tabFolder.setSelectionForeground(display.getSystemColor(SWT.COLOR_TITLE_FOREGROUND));
```

Running the example results in a change in appearance similar to below.

Figure 16. 2-D tabs



`setSelectionBackground` takes three parameters:

- `color[]` array containing different colors involved in the gradient

© Copyright IBM Corporation 2007. All rights reserved.

- `int[]` array containing the percentage that each color must occupy in the gradient; the last color percentage isn't specified explicitly, but is implied
- `boolean` indicating whether the direction of the gradient is vertical (top to bottom) or horizontal (left to right)

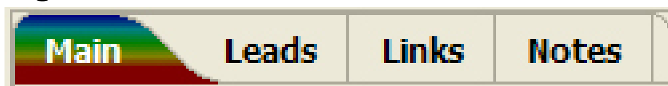
`setSelectionForeground` takes one parameter representing the foreground color (text color) of the selected tab.

Note the use of system colors SWT.COLOR_TITLE_BACKGROUND, SWT.COLOR_TITLE_BACKGROUND_GRADIENT, and SWT.COLOR_TITLE_FOREGROUND. These are the operating system theme colors. Using them ensures that the application looks and feels like the platform it's running on. As a more extreme example, modify the colors array shown below. Running the example shows a tab similar to that below.

Listing 12. A rainbow gradient

```
            Display display = Display.getCurrent();
int colorCount = 5;
Color[] colors = new Color[colorCount];
colors[0] = display.getSystemColor(SWT.COLOR_DARK_BLUE);
colors[1] = display.getSystemColor(SWT.COLOR_DARK_CYAN);
colors[2] = display.getSystemColor(SWT.COLOR_DARK_GREEN);
colors[3] = display.getSystemColor(SWT.COLOR_DARK_YELLOW);
colors[4] = display.getSystemColor(SWT.COLOR_DARK_RED);
int[] percents = new int[colorCount - 1];
percents[0] = 20;
percents[1] = 20;
percents[2] = 20;
percents[3] = 20;
tabFolder.setSelectionBackground(colors, percents, true);
tabFolder.setSelectionForeground\
(display.getSystemColor(SWT.COLOR_TITLE_FOREGROUND));
```

Figure 17. Rainbow tabs



Single tabs

Sometimes it's more visually appealing to show a single tab if the tab titles are long. This makes only one tab appear at a time; a drop-down lets users switch to other tabs. Modify the CTabFolderExample with the line of code below.

```
tabFolder.setSingle(true);
```

You should now see tabs appear similar to those shown in figures 19 and 20.

© Copyright IBM Corporation 2007. All rights reserved.

Figure 18. Single tab

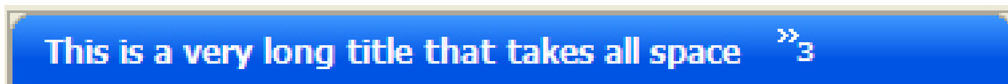
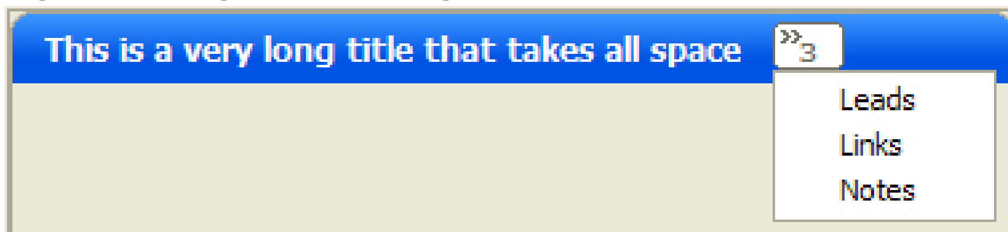


Figure 19. Single tab showing other tabs



Section 10. Forms API customization

Sections

The contents of a form may be split into multiple sections, each containing a cohesive set of functionality. You can do this using the Section widget included in the Forms API. Figure 20 shows how sections look.

Figure 20. Basic sections

© Copyright IBM Corporation 2007. All rights reserved.
Trademarks

Name

First Name Last Name

Address

Address 1

Address 2

City State

Zip

Contact Information

Phone Email

These sections are basic, created using the FormToolkit's `createSection` method with the style SWT.NONE. Open the SectionExample in the Java editor and run it to see an example.

You can perform a number of customizations on sections, as explained next.

Collapsibility

You can make a section collapsible by constructing it with the Section.TWISTIE style bit. Doing so adds a twistie (a triangle-shaped button) to the left of a section. The twistie triangle initially points to the right, indicating that the section's contents are collapsed. When you click the twistie, the triangle points down to indicate that the section has been expanded to display its contents. The benefit of collapsible sections is that they take less real estate on the screen. Furthermore, they help users focus on one thing at a time, decreasing the complexity of use when an application offers a lot of functions. Figure 21 shows collapsed sections.

Figure 21. Collapsed twisties

© Copyright IBM Corporation 2007. All rights reserved.
Trademarks

Name

Address

Contact Information

When you click the Name section's twistie, the section expand, revealing its contents.

Figure 22. Name section expanded

Name

First Name Last Name

Address

Contact Information

Note that sections can be set to initially expand instead of collapse, by passing the style bit Section.EXPANDED to the section constructor. An alternative to twistie collapsible sections is tree-node collapsible sections. They behave the same way. However, they look different because the twistie is replaced with the tree node expand/collapse control. To create such a section, pass Section.TREE_NODE to the section constructor. Figure 23 shows a tree-node collapsible section.

Figure 23. Address expanded

Name

First Name Last Name

Address

Contact Information

Description text

You can add a description to the bottom of a section to give users an idea of what the section is about or provide helpful instructions. Passing the style bit Section.DESCRIPTION to the section constructor makes the description composite appear at the bottom of the section. To set description text, use the `setDescription(String description)` method. In the `buildSection` method inside SectionExample, replace the section-creation code with the lines from Listing 13 and run the example to see a section with a description.

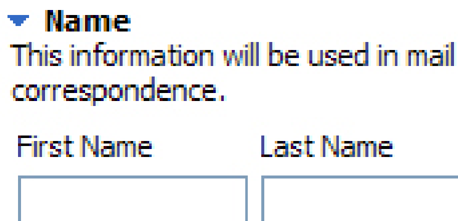
Listing 13. Include a description

© Copyright IBM Corporation 2007. All rights reserved.

```
private Section buildSection(final Composite parent) {
    Section section = toolkit.createSection(parent, Section.TWISTIE
                | Section.EXPANDED | Section.DESCRIPTION);
    section.setDescription("This information will be used in mail correspondence.");
    return section;
}
```

Figure 24 shows the customization.

Figure 24. Sections with descriptions



Section descriptions can be customized further by setting a custom description control. For example, you can set a hyperlink control as the section description control, instructing users to click the link for more information. To demonstrate, replace the code of the `buildSection` method in SectionExample with the code below.

Listing 14. A custom description control

```
private Section buildSection(final Composite parent) {
    Section section = toolkit.createSection(parent,
                        Section.TWISTIE |
                        Section.EXPANDED);
  Hyperlink link = toolkit.createHyperlink(section,
                        "Click here for more information",
                        SWT.NONE);
    link.addHyperlinkListener(new HyperlinkAdapter() {
        public void linkActivated(HyperlinkEvent e) {
            openAddressHelpDialog(parent.getShell());
        }
    });
    section.setDescriptionControl(link);
    return section;
}

private void openAddressHelpDialog(Shell shell) {
    MessageBox box = new MessageBox(shell);
    box.setText("Help");
    box.setMessage("This is a help message.");
    box.open();
}
```

Figure 25 illustrates a section with a custom description control.

Figure 25. Section with custom description control

© Copyright IBM Corporation 2007. All rights reserved.

Address
Click here for more information
Address 1
Address 2
City State
Zip

Indentation

You can align a section widget with its title to make the relationship more obvious and improve the look of the UI. To do so, pass the Section.CLIENT_INDENT style bit to the section constructor. Figure 26 shows an example.

Figure 26. Indented sections

© Copyright IBM Corporation 2007. All rights reserved.

Name

First Name Last Name

Address

Address 1

Address 2

City State

Zip

Contact Information

Phone Email

Title bar

To make sections stand out more, you can paint a title bar decoration behind the title text. To do so, pass the Section.TITLE_BAR style bit to the section constructor. Figure 27 shows how the UI looks more rich and colorful with section title bars.

Figure 27. Sections with title bars

© Copyright IBM Corporation 2007. All rights reserved.

Note that the colors are taken from the operating system color theme. This helps ensure that the user interface looks consistent with the operating system it's running on.

Another version uses short title bars. These can be displayed by passing the Section.SHORT_TITLE_BAR style bit to the Section constructor.

Figure 28. Short title bars

© Copyright IBM Corporation 2007. All rights reserved.
Trademarks

Colors

You can customize section colors in many ways, which is helpful when you need to match a particular color theme. The methods involved with customizing section colors are `setBackground`, `setForeground`, `setTitleBarBackground`, `setTitleBarBorderColor`, `setTitleBarForeround`, and `setTitleBarGradientBackground`.

To demonstrate basic sections with a green background and white foreground, open SectionExample and replace the `buildSection` method with the code below.

Listing 15. Green section titles

```
private Section buildSection(final Composite parent) {
    Section section = toolkit.createSection(parent, Section.CLIENT_INDENT);
    Display display = Display.getCurrent();
    section.setBackground(display.getSystemColor(SWT.COLOR_DARK_GREEN));
    section.setForeground(display.getSystemColor(SWT.COLOR_WHITE));
    return section;
}
```

Figure 29 illustrates.

Figure 29. Basic colored title bars

© Copyright IBM Corporation 2007. All rights reserved.

Name

First Name Last Name

Address

Address 1

Address 2

City State

Zip

Contact Information

Phone Email

To produce twistie sections with colors borrowed from the operating system color theme, replace the `buildSection` method in SectionExample with the code from the code below.

Listing 16. Custom twistie title bars

```
private Section buildSection(final Composite parent) {
  Section section = toolkit.createSection(parent,
                                Section.TWISTIE |
                                Section.CLIENT_INDENT |
                                Section.TITLE_BAR |
                                Section.EXPANDED);
    Display display = Display.getCurrent();
    section.setTitleBarForeground(display.getSystemColor(SWT.COLOR_TITLE_FOREGROUND));
    section.setTitleBarBackground(display.getSystemColor(SWT.COLOR_TITLE_BACKGROUND));
    section.setTitleBarGradientBackground(
                display.getSystemColor
                        (SWT.COLOR_TITLE_BACKGROUND_GRADIENT));
    return section;
}
```

Figure 30 shows the result.

Figure 30. Gradient twisties

© Copyright IBM Corporation 2007. All rights reserved.

The final example is about basic sections that have a 3-D look. Replace the `buildSection` method in SectionExample with the code below.

Listing 17. 3-D gradient titles

```
private Section buildSection(final Composite parent) {
Section section = \
toolkit.createSection(parent, Section.CLIENT_INDENT | Section.TITLE_BAR);
Display display = Display.getCurrent();
section.setTitleBarForeground(
        display.getSystemColor(SWT.COLOR_TITLE_FOREGROUND));
section.setTitleBarBackground(
        display.getSystemColor(SWT.COLOR_TITLE_BACKGROUND_GRADIENT));
section.setTitleBarGradientBackground(
        display.getSystemColor(SWT.COLOR_TITLE_BACKGROUND));
return section;
}
```

Figure 31 illustrates.

Figure 31. Gradient title bars

© Copyright IBM Corporation 2007. All rights reserved.

Name

First Name Last Name

Address

Address 1

Address 2

City State

Zip

Contact Information

Phone Email

Section 11. FormToolkit

FormToolkit is a widget factory you can use to create widgets with a Web page-style flat look and feel. Normally, widgets created with FormToolkit have a white background and a border. However, the FormToolkit used can be customized to change the background color and remove the border from around each widget you create. To do so, you use the methods `setBackground` and `setBorderStyle`, respectively.

To demonstrate, open the FormToolkitExample in the Java editor. To customize the FormToolkit background color, add the code below after the construction of FormToolkit.

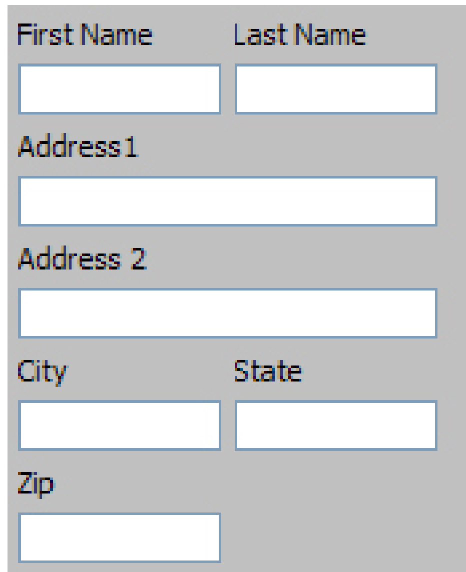
```
Display display = Display.getCurrent();
```


© Copyright IBM Corporation 2007. All rights reserved.

```
toolkit.setBackground(display.getSystemColor(SWT.COLOR_GRAY));
```

This causes the form to render with a gray background. Such functionality is useful when you're required to match a company color theme. Figure 32 illustrates the effect.

Figure 32. FormToolkit with a gray background



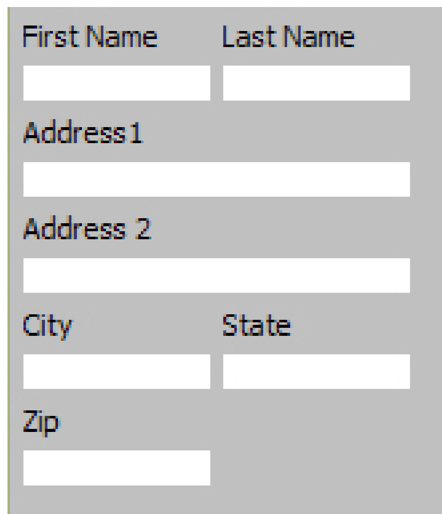
To remove the border from the form widgets, add the line below to your code after the construction of the FormToolkit.

```
toolkit.setBorderStyle(SWT.NULL);
```

Figure 33 shows the form with a gray background and no border around the widgets.

Figure 33. FormToolkit with no borders

© Copyright IBM Corporation 2007. All rights reserved.



Section 12. Form title area

Forms created with the Forms API support the display of a form title in a big bold font at the top of the form's contents. You can achieve this by using the `setText` method on the Form instance object. To demonstrate, open the FormTitleAreaExample in the Java editor. To add a title to the form, add the line of code below after the construction of the Form.

```
form.setText("Name and Address");
```

You can customize the form title text color by calling the `setForeground` method. Add the code below after the previous snippet to change the title text color to match the operating system theme's title foreground color.

```
Display display = Display.getCurrent();
Color titleForegroundColor = \
display.getSystemColor(SWT.COLOR_TITLE_FOREGROUND);
form.setForeground(titleForegroundColor);
```

The form title area's background colors can be customized using the `setTextBackground` method call, which renders a gradient formed from multiple colors. You need several parameters: the array of colors, the percentage area each color can have and whether the gradient is vertical or horizonal.

Listing 18. Change the background

© Copyright IBM Corporation 2007. All rights reserved.

```
Color[] colors = new Color[] {
    display.getSystemColor(SWT.COLOR_TITLE_BACKGROUND),
    display.getSystemColor(SWT.COLOR_TITLE_BACKGROUND_GRADIENT)
};
int[] percents = new int[] {100};
form.setTextBackground(colors, percents, true);
```

The form title area's background can be made solid by using one color. Replace the previous snippet with the code below to achieve that effect.

Listing 19. Make the background solid

```
Color[] colors = new Color[] {
    display.getSystemColor(SWT.COLOR_TITLE_BACKGROUND)
};
int[] percents = new int[] {};
form.setTextBackground(colors, percents, true);
```

You can customize the form title area even further by inserting a custom control. For example, you can insert form-summary information, calculation totals, or data-validation status in the title area next to the text representing the form header. To do so, you use the `setHeadClient` method, which takes as a parameter a control constructed with the form head as its parent. The form head can be obtained through the `getHead` method. Add the code below after the construction of the `LastName` text widget to insert a control that echoes the first and last name entered in the form.

Listing 20. Custom controls

```
final Label lblNameEcho = new Label(form.getHead(), SWT.NONE);
form.setHeadClient(lblNameEcho);
ModifyListener nameEchoModifyListener = new ModifyListener() {
    public void modifyText(ModifyEvent e) {
        String firstName = txtFirstName.getText();
        String lastName = txtLastName.getText();
        lblNameEcho.setText(" " + firstName + " " + lastName);
    }
};
txtFirstName.addModifyListener(nameEchoModifyListener);
txtLastName.addModifyListener(nameEchoModifyListener);
```

When you enter the first and last name into the form, the form header echoes the full name within the custom control.

Section 13. Use the Forms API's look and feel with wizards

© Copyright IBM Corporation 2007. All rights reserved.

Wizards are a variation on dialogs that let users accomplish a particular task by going through a series of pages. Using wizards provides a number of benefits. Long, arduous tasks are broken into a series of easy-to-follow steps, providing a predictable workflow instead of requiring the user to figure out the workflow by trial and error.

Wizards in RCP have a particular style and follow specific UI standards. An example is the Eclipse wizard for creating an RCP plug-in project. However, if an application's views and editors were all built using the style of the Forms API, it may be desirable to have wizards follow the same style, for the sake of consistency.

To obtain the flat look and feel in a wizard, use the FormToolkit when building widgets inside the `createControl` method of a WizardPage subclass. Listing 21 shows the code for a wizard and a wizard page that achieves this result.

Listing 21. A wizard with a flat look and feel

```
public class ContactWizardPage extends WizardPage {

        public ContactWizardPage() {
                super("New Contact Wizard", "New Contact Wizard", null);
                setMessage("Please enter contact info.");
        }

        public void createControl(Composite parent) {
                FormToolkit toolkit = new FormToolkit(Display.getCurrent());

                // snippet 1
//          parent.setBackground(toolkit.getColors().getBackground());
//          parent.getParent().setBackground(toolkit.getColors().getBackground());
//          parent.setLayout(new GridLayout());

                Form form = toolkit.createForm(parent);

                Composite composite = form.getBody();
                composite.setLayout(new GridLayout(2, true));

                Label lblFirstName = toolkit.createLabel(composite, "First Name");

                Label lblLastName = toolkit.createLabel(composite, "Last Name");

                Text txtFirstName = toolkit.createText(composite, "");

                Text txtLastName = toolkit.createText(composite, "");

                Label lblEmail = toolkit.createLabel(composite, "Email");
                GridDataFactory.swtDefaults().span(2, 1).align(
                        SWT.FILL,
                        SWT.BEGINNING).applyTo(lblEmail);

                Text txtEmail = toolkit.createText(composite, "");
            GridDataFactory.swtDefaults().span(2, 1).align(
                        SWT.FILL,
                        SWT.BEGINNING).applyTo(txtEmail);

                setControl(composite);

                parent.getShell().setSize(240, 320);
        }
```


© Copyright IBM Corporation 2007. All rights reserved.

```
}
```

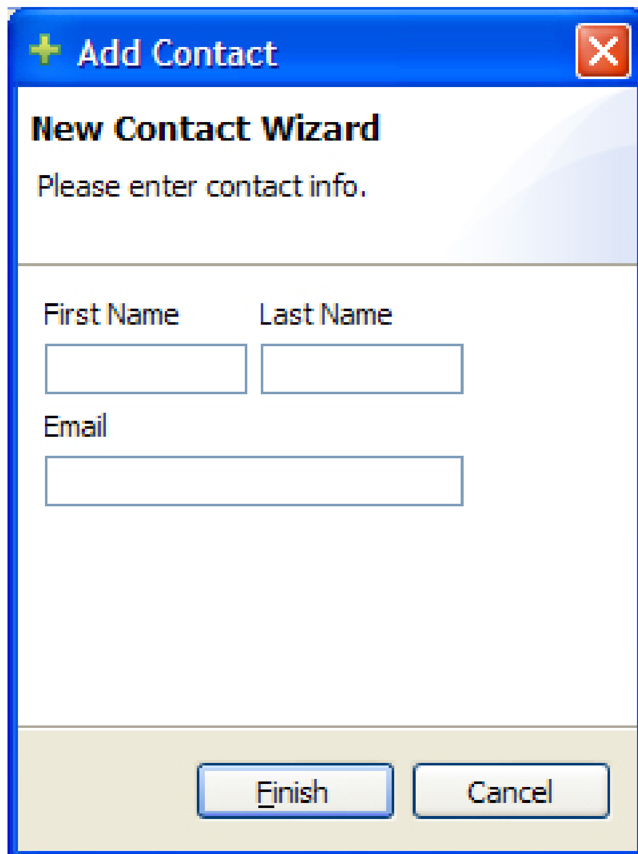
The wizard page contents have a white background due to the use of FormToolkit, but note how the border area that surrounds the wizard page still has a gray background. To make that white, you need to make further customizations to the `Wizard` subclass. Listing 22 shows the code for a customized `Wizard` subclass compatible with the flat look and feel. The finished wizard is shown below.

Listing 22. The wizard subclass

```
public class ContactWizard extends Wizard implements INewWizard {

        public ContactWizard() {
                setTitleBarColor(new RGB(2,43,43));
                setWindowTitle("Add Contact");
        }

        @Override
        public boolean performFinish() {
                return false;
        }

        public void init(IWorkbench workbench, IStructuredSelection selection) {
        }
}
```

Figure 34. A flat-look wizard

© Copyright IBM Corporation 2007. All rights reserved.



Section 14. Custom-view toolbar

RCP has the notion of contributions, which allow plug-ins to contribute actions to a view toolbar that are rendered as buttons in the UI. There is no way to contribute actions rendered as controls other than buttons, such as text search filter boxes or informative labels. However, this can be accomplished by building a custom-view toolbar.

Open the `ContactListView` class to see an example of a custom toolbar in the `buildCustomToolBar` method. This custom toolbar has a text search filter box that enables quick searching of the view table row's contents.

A horizontal RowLayout is used to lay out the controls in the custom toolbar. Although the toolbar has only a text box, you can also insert other widgets or actions. Figure 35 shows this view.

Figure 35. A custom-view toolbar

© Copyright IBM Corporation 2007. All rights reserved.

First Name	Last Name	Email
Kathy	Maine	k@maine.com
Erl	Shields	erl@sh.com
Joey	Maid	jm@m.com
Bob	Star	star2@b.com
Jenny	Martin	jj@m.com

Here are the steps to create this custom-view toolbar:

1. In the `createPartControl()` method, change the parent to use GridLayout.
2. Add a composite to the parent composite in `createPartControl()` that will be used as a custom toolbar. Set its layout to RowLayout.
3. Add widgets that you want in the custom-view toolbar.
4. Add a second composite to the parent composite in `createPartControl()` that will be used to hold the main contents of the view.
5. Add any widgets you want in the view, such as a table or a tree.

Listing 23 shows the complete source code.

Listing 23. A custom-view toolbar

```
public class ContactListView extends ViewPart {
        public final static String ID = "com.dw.tutorial.views.ContactListView";

        private TableViewer tableViewer;
        private ViewerFilter tableFilter;

        private ContactList contacts;

        public void createPartControl(Composite parent) {
                GridLayout gridLayout = new GridLayout();
                gridLayout.marginHeight = 0;
                gridLayout.marginWidth = 0;
                gridLayout.verticalSpacing = 0;
                parent.setLayout(gridLayout);

                buildCustomToolBar(parent);

                buildTableViewer(parent);
                GridDataFactory.fillDefaults().grab(true, true).\
applyTo(tableViewer.getTable());

                contacts = new ContactList();
                contacts.addPropertyChangeListener(new PropertyChangeListener() {
```


© Copyright IBM Corporation 2007. All rights reserved.

```
                public void propertyChange(PropertyChangeEvent evt) {
                        tableViewer.setInput(contacts);
                }
        });

}

private void buildCustomToolBar(Composite parent) {
        Composite customToolBar = new Composite(parent, SWT.NONE);
        customToolBar.setLayout(new RowLayout(SWT.HORIZONTAL));
        buildTextFilter(customToolBar);
}

private void buildTextFilter(Composite customToolBar) {
        final Text text = new Text(customToolBar, SWT.BORDER);
        text.addModifyListener(new ModifyListener() {
                public void modifyText(ModifyEvent e) {
                        if (tableFilter != null) {
                                tableViewer.removeFilter(tableFilter);
                        }
                        tableFilter = new ViewerFilter() {
                                @Override
                                public boolean select(Viewer viewer, \
                                Object parentElement, Object element) {
                                        if (text.getText().trim().equals("")) {
                                                return true;
                                        }
                                        String[] row = (String[])element;
                                        for (String column : row) {
                                                if (column.toUpperCase().\
                                                contains(text.getText().\
                                                toUpperCase())) {
                                                        return true;
                                                }
                                        }
                                        return false;
                                }
                        };
                        tableViewer.addFilter(tableFilter);
                        tableViewer.refresh();
                }
        });
}

private void buildTableViewer(Composite parent) {
        tableViewer = new TableViewer(buildTable(parent));
        tableViewer.setColumnProperties(createColumnNames());
        tableViewer.setContentProvider(createContentProvider());
        tableViewer.setLabelProvider(createLabelProvider());
        tableViewer.setInput(createInput());
}

private Table buildTable(Composite parent) {
        Table table = new Table(parent, SWT.FULL_SELECTION);
        table.setHeaderVisible(true);
        table.setLinesVisible(true);
        buildTableColumns(table);
        return table;
}

private void buildTableColumns(Table table) {
        String[] columnNames = createColumnNames();
        Integer[] columnWidths = getColumnWidths();
        for (int i = 0; i < columnNames.length; i++) {
                TableColumn column = new TableColumn(table, SWT.LEFT, i);
                column.setText(columnNames[i]);
                column.setWidth(columnWidths[i]);
        }
}
```


© Copyright IBM Corporation 2007. All rights reserved.

```
	private TableLabelProvider createLabelProvider() {
		return new TableLabelProvider();
	}

	private TableContentProvider createContentProvider() {
		return new TableContentProvider();
	}

	private Integer[] getColumnWidths() {
		return new Integer[] {70, 70, 150};
	}

	private String[] createColumnNames() {
		return new String[] {"First Name", "Last Name", "Email"};
	}

	protected Object createInput() {
		return new Object[] {new String[] {"Kathy","Maine","k@maine.com"},
				new String[] {"Erl","Shields","erl@sh.com"},
				new String[] {"Joey","Maid","jm@m.com"},
				new String[] {"Bob","Star","star2@b.com"},
				new String[] {"Jenny","Martin","jj@m.com"}};
	}

	public void setFocus() {
		// NOOP
	}

	private final class TableContentProvider implements IStructuredContentProvider {
		public Object[] getElements(Object inputElement) {
			return (Object[])inputElement;
		}

		public void dispose() {
			// NOOP
		}

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

	private final class TableLabelProvider implements ITableLabelProvider {
		public Image getColumnImage(Object element, int columnIndex) {
			return null;
		}

		public String getColumnText(Object element, int columnIndex) {
			String[] row = (String[])element;
			return row[columnIndex];
		}

		public void addListener(ILabelProviderListener listener) {
			// NOOP
		}

		public void dispose() {
			// NOOP
		}

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

		public void removeListener(ILabelProviderListener listener) {
			// NOOP
		}
	}
```


© Copyright IBM Corporation 2007. All rights reserved.

```
        public ContactList getContacts() {
                return contacts;
        }
}
```

Section 15. Conclusion

This tutorial has explained a number of UI elements that can be changed in Eclipse RCP, JFace, and SWT. Along the way, you've learned about basic changes you can make, such as fonts and colors. You've also learned advanced techniques, such as how to create custom wizards and section headers. Using these should provide you the ability to go from a typical-looking Eclipse RCP application to a distinctive but visually appealing one.

© Copyright IBM Corporation 2007. All rights reserved.

Downloads

Description	Name	Size	Download method
Source code	os-eclipse-rcp1.customizing-final.zip	185KB	HTTP

Information about download methods

© Copyright IBM Corporation 2007. All rights reserved.

Resources

Learn

- Learn about authoring with Eclipse from an Eclipse Foundation article title "Authoring with Eclipse."
- Learn about Eclipse User Assistance by attending EclipseCon.
- Stay up to date on what's happening in the Eclipse community by visiting Planet Eclipse.
- For an excellent introduction to the Eclipse platform, see "Getting started with the Eclipse Platform."
- Check out the "Recommended Eclipse reading list."
- Browse all the Eclipse content on developerWorks.
- Expand your Eclipse skills by checking out IBM developerWorks' Eclipse project resources.
- To listen to interesting interviews and discussions for software developers, check out developerWorks' podcasts.
- Stay current with developerWorks' technical events and webcasts.
- Check out upcoming conferences, trade shows, webcasts, and other Events around the world that are of interest to IBM open source developers.
- Visit the developerWorks Open source zone for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.

Get products and technologies

- Check out the Swing iTunes look and feel.
- Learn more about the Lotus Notes look and feel.
- Learn more about the java.exe.manifest file.
- Learn how to customize SWT GTK look and feel.
- Download Eclipse Platform and get started with Eclipse now.
- Check out the latest Eclipse technology downloads at IBM alphaWorks.
- Innovate your next open source development project with IBM trial software, available for download or on DVD.

Discuss

- The Eclipse Platform newsgroups should be your first stop to discuss questions

© Copyright IBM Corporation 2007. All rights reserved.

regarding Eclipse. (Selecting this will launch your default Usenet news reader application and open eclipse.platform.)

- The Eclipse newsgroups has many resources for people interested in using and extending Eclipse.
- Participate in developerWorks blogs and get involved in the developerWorks community.

About the authors

Scott Delap

Scott Delap is president of Rich Client Solutions Inc., a software consulting firm focusing on technologies such as Swing, Eclipse RCP, GWT, Flex, and Open Laszlo. He is actively involved in the Java community, speaking at events such as NFJS, QCon and JavaOne. He is also the Java editor of InfoQ.com and runs ClientJava.com, a portal focused on desktop Java development.

Annas Andy Maleh

Annas "Andy" Maleh is a consultant at Obtiva Corp., a firm that specializes in Eclipse RCP development, Ruby on Rails development and training, and helping teams transition to Agile methodologies. He is currently involved with an Eclipse RCP project to build a custom CRM application for an international corporation. He works on a team that follows eXtreme Programming practices, programs professionally in Java and Ruby, and participates in work relating to UI design enhancement. At EclipseWorld 2006, he gave two presentations relating to Eclipse RCP development. He is a Sun Certified Java Programmer who holds a bachelor's degree in computer science from McGill University.

© Copyright IBM Corporation 2007. All rights reserved.