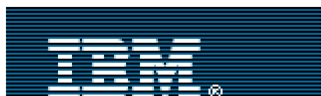


Search for: within All of dW Search
Use + - ( ) " " Search help

IBM home | Products & services | Support & downloads | My account

IBM developerWorks > Open source projects | Java technology developerWorks

PDF e-mail it!

Using the Eclipse GUI outside the Eclipse Workbench, Part 1: Using JFace and SWT in stand-alone mode

Building a simple file explorer application

Level: Intermediate

Adrian Van Emmenis (van@vanemmenis.com)
Independent consultant
January 23, 2003

Although the Eclipse GUI components (JFace and SWT) are often used inside the Eclipse Workbench, they were designed as self-contained frameworks in their own right. Even outside the Eclipse Workbench, JFace's pluggable design still allows you to develop sophisticated GUIs with surprisingly little code. In this series of three articles, A. O. Van Emmenis shows how to build just such a stand-alone application. Part 1 starts with a "Hello, World" example and builds, step by step, into a (very) simple file explorer. He introduces some of the major JFace classes (and a few SWT widgets), along with some tips, tricks, and design issues.

Contents:

Introduction
Hello, World
JFace application windows
Using a TreeViewer
Implementing a TreeContentProvider
Implementing the top-level Explorer class
Implementing a label provider
Using a table viewer
Implementing the file table viewer content provider
Listening for events
Implementing a file table label provider
Conclusion
Resources
About the author
Rate this article

Related content:

Part 2 of this series
Getting started with the Eclipse Platform
Developing Eclipse plug-ins
Plug a Swing-based development tool into Eclipse
Subscribe to the developerWorks newsletter
developerWorks Toolbox subscription

Also in the Open source projects zone:

Tutorials
Projects
Code and components
Articles

Also in the Java zone:

Tutorials
Tools and products
Code and components
Articles

Introduction

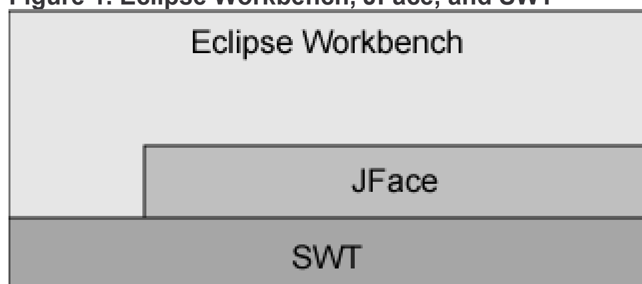
The open source Eclipse project is one of the most interesting recent developments in the Java world. Eclipse describes itself as "a kind of universal tool platform -- an open extensible IDE for anything and nothing in particular". For an introduction to Eclipse, see the *developerWorks* article "Getting started with the Eclipse Platform".

Two of its major components are a graphical library called SWT and a matching utility framework called JFace. I'll concentrate on these components in this article. The Eclipse Technical Overview on the Eclipse Web site (see Resources later in this article) describes them like this:

- **SWT** is a widget set and graphics library integrated with the native window system but with an OS-independent API.
- **JFace** is a UI toolkit implemented using SWT that simplifies common UI programming tasks. JFace is window-system-independent in both its API and implementation, and is designed to work with SWT without hiding it.

Figure 1 shows the relationships between Eclipse, JFace, and SWT.

Figure 1. Eclipse Workbench, JFace, and SWT



Most of the articles published about JFace and SWT (so far) have discussed them from the point of view of using them in the context of the larger Eclipse framework. In this article I am going to take a different approach. I will show how you can use

JFace and SWT in a standalone Java program.

The example I have chosen is a file explorer. We won't actually implement very much real functionality, but we will visit enough of the GUI for you to see how a fully featured program might be built.

Installation notes

You can download the source code for the examples in this article, but take my system setup into account:

- Windows 2000
- Eclipse, stable build M3 (November 15, 2002)
- Eclipse installed in C:\eclipse-2.1.0

I will leave you to do any swizzling of names and file separators in what follows, so that the programs work correctly on your system.

Build/run instructions

You need these jar files on your class path:

```
C:\eclipse-2.1.0\plugins\org.eclipse.jface_2.1.0\jface.jar
C:\eclipse-2.1.0\plugins\org.eclipse.runtime_2.1.0\runtime.jar
C:\eclipse-2.1.0\plugins\org.eclipse.swt.win32_2.1.0\ws\win32\swt.jar
C:\eclipse-2.1.0\plugins\org.eclipse.ui.workbench_2.1.0\workbench.jar
C:\eclipse-2.1.0\plugins\org.eclipse.core.runtime_2.1.0\runtime.jar
```

Ensure that the Java VM picks up the correct shared libraries for the GUI you are using at runtime by running it with the following argument:

```
-Djava.library.path=C:\eclipse-2.1.0\plugins\org.eclipse.swt.win32_2.1.0\os\win32\x86\
```

Finally, so that the examples can find the gif files containing the icons, run the programs from the folder that contains the **icons** folder.

Hello, World

Let's start with the simplest JFace program I can think of and build it up to the ubiquitous "Hello, World" program.

Listing 1. Hello (version 1)

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

public class Hello
{
  public static void main(String[] args)
  {
    ApplicationWindow w = new ApplicationWindow(null);
    w.setBlockOnOpen(true);
    w.open();
    Display.getCurrent().dispose();
  }
}
```

Here we have a class called `Hello` where the main method merely creates an ApplicationWindow, and then opens it. The `setBlockOnOpen()` makes the `open()` block until the window is closed.

After the window has closed, we get the current Display and dispose of it. This releases the resources used in the operating system (I'll discuss why it's always good practice to do this later).

When you run this program, you see a window that looks like Figure 2:

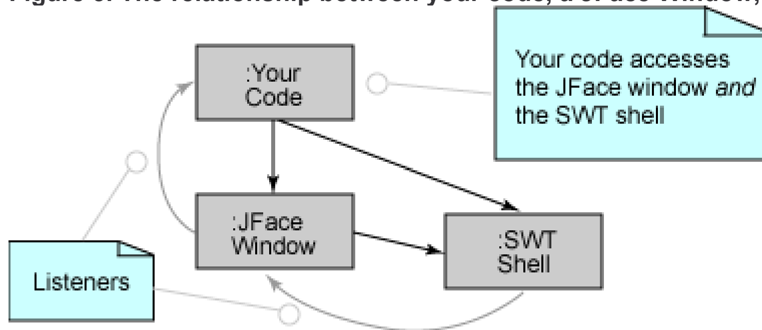
Figure 2. Hello (version 2)

And that's it. It doesn't even say "Hello, World". Before we fix that, let's digress into JFace windows.

JFace application windows

A window is the `JFace` class for a top level window -- in other words, one that is managed by the OS window manager. A JFace window is not actually the GUI object for a top level window (SWT already provides one, called a *Shell*). Instead, a JFace window is a helper object that knows about a corresponding SWT Shell object and provides code to help create/edit it, listen to its events, etc. Figure 3 shows the relationship between your code, JFace, and SWT.

Figure 3. The relationship between your code, a JFace Window, and an SWT Shell

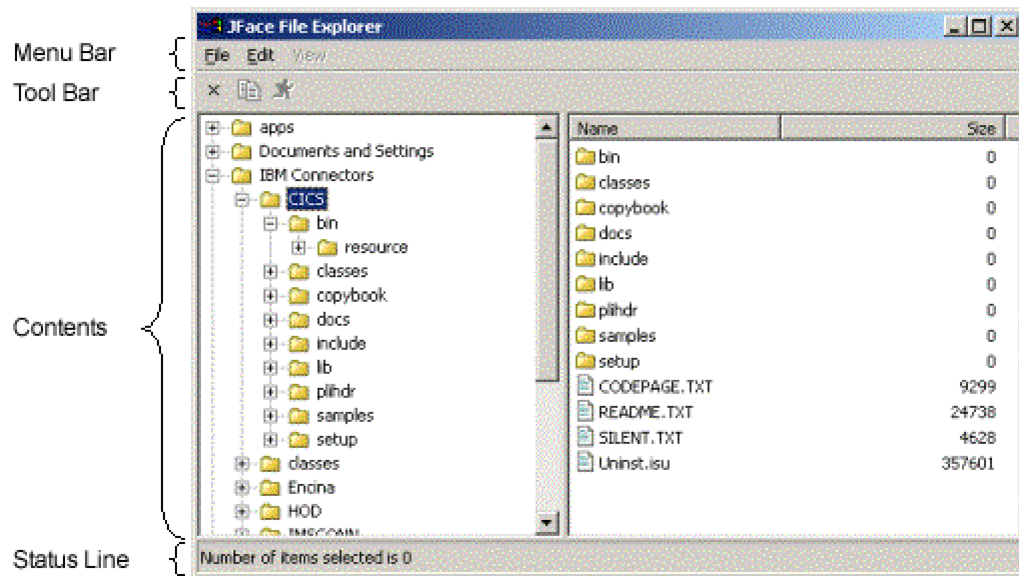


In fact, this model is the key to understanding how JFace works. It is not really a layer on top of SWT, and it doesn't try to hide SWT from you. Instead, JFace recognizes that there are several common patterns of use for SWT, and it provides utility code to help you program these patterns more easily.

To do this, JFace either provides an object that you use or a class that you can subclass (and sometimes it provides both).

Although we just used an `ApplicationWindow` directly, they are actually designed for you to subclass and fill in your specific behavior. They come ready-made with a menu bar, a tool bar, an area for you to insert your application-specific contents, and a status line -- all optional. Figure 4 shows these areas on the JFace File Explorer example itself.

Figure 4. The parts of an application window



Let's refine Hello to make it a subclass of `ApplicationWindow`. The changed lines are highlighted in Listing 2.

Listing 2. Hello (version 2)

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

public class Hello extends ApplicationWindow
{
  public Hello()
  {
    super(null);
  }

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

The constructor that you write must invoke the superclass constructor (as usual). Let's ignore the argument to that constructor for now.

Running this doesn't give us anything different from the previous program. The default is not to give us any decorations.

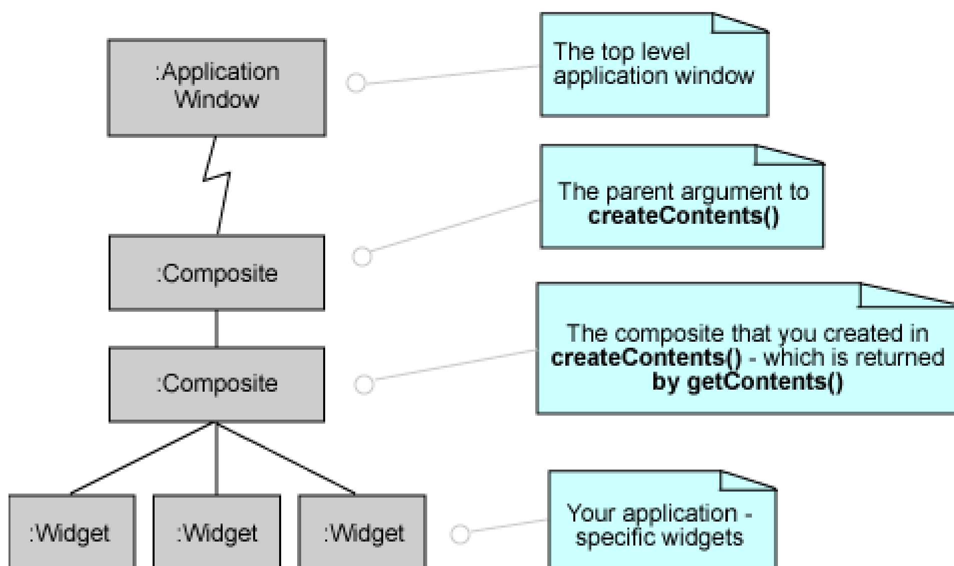
Our program is going to create a button with the text "Hello, World". This button is going to appear in the contents area. To do this, we must implement the `Control createContents(Composite parent)` method.

`ApplicationWindow` will call this after all the other widgets have been created but before the window is displayed on the screen.

The argument parent is the composite widget that represents the contents area.

The idea is that you create a composite widget, add it to parent, then add your widgets, and return the composite widget that you created. Figure 5 shows the instance hierarchy.

Figure 5. The instance hierarchy of an Application Window



Our contents are going to be fairly simple for now: a single button under parent, as shown in Listing 3.

Listing 3. Hello (version 3)

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

public class Hello extends ApplicationWindow
{
  public Hello()
  {
    super(null);
  }

  protected Control createContents(Composite parent)
  {
    Button b = new Button(parent, SWT.PUSH);
    b.setText("Hello World");
    return b;
  }

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

The result is Figure 6.

Figure 6. Hello (version 3)

And there we are. Our first "Hello, World" program in JFace: a window containing a single button.

Now let's move on to the file explorer. First, we'll create the tree viewer that will display the folder hierarchy.

Using a TreeViewer

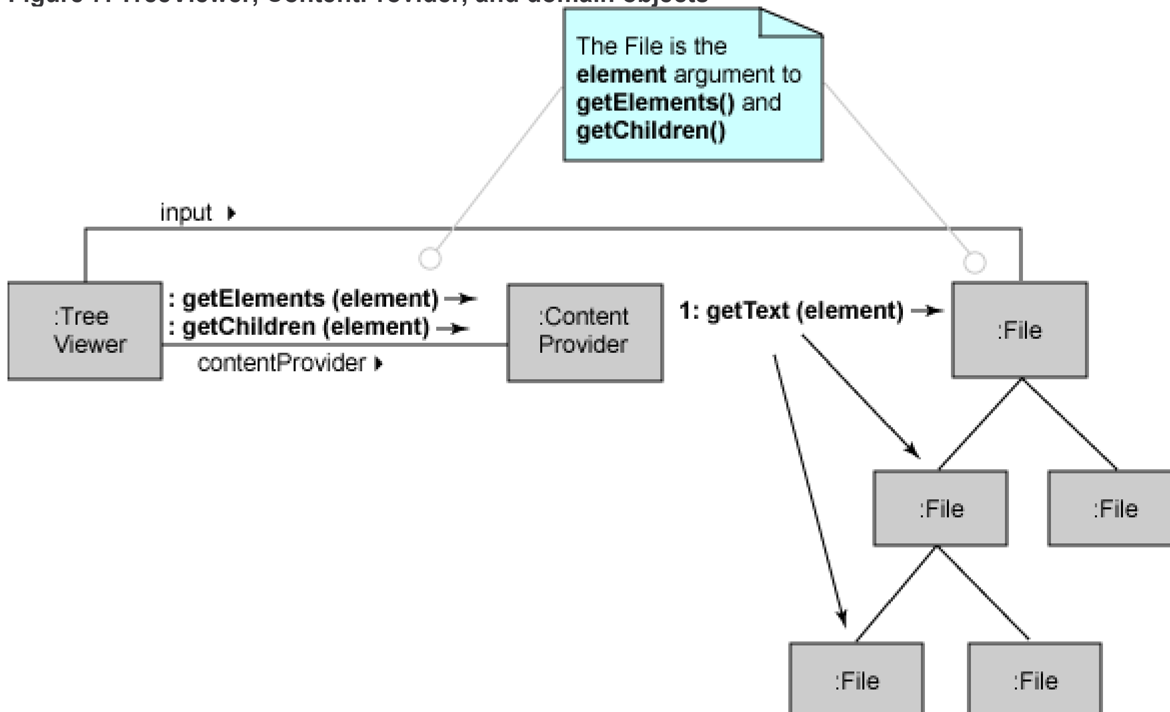
Just as with the `ApplicationWindow`, a `TreeViewer` is not the actual SWT widget, nor does it try to hide the SWT widget from you. It uses an SWT tree widget to display the items, and it uses a number of other objects to help it, too.

Unlike the `ApplicationWindow`, JFace `TreeViewer` is not intended to be subclassed.

The idea is that a `TreeViewer` knows about the root element of the tree that it is going to display. You have to tell it what that object is, of course: `TreeViewer: void setInput(Object rootElement)`

To get started, it asks that root element for its children and then displays them. Then, as the user expands one of the children, the tree viewer asks that node for its children and so on. Actually, that's not quite true. The `TreeViewer` doesn't talk to the domain objects directly -- instead it uses another object called a `ContentProvider`, and this object uses your domain objects as in Figure 7.

Figure 7. TreeViewer, ContentProvider, and domain objects



You have to implement the `ContentProvider`, of course. For a `TreeViewer`, your class must implement the `ITreeContentProvider` interface.

Implementing a TreeContentProvider

There are six methods to implement. We can actually get away with only implementing three of them, so, in the spirit of instant gratification, let's just consider those for now.

This is how the tree viewer asks the content provider for the top-level elements directly beneath the root element:

```
ITreeContentProvider: public Object[] getElements(Object element)
```

and then, whenever it needs the children of a particular element, it uses this:

```
ITreeContentProvider: public Object[] getChildren(Object element)
```

In order to figure out if a node has any children (and then put that little plus sign next to it), the tree viewer could just ask for the children of the node, and then it could ask how many there are. Just in case your code knows of a quicker way to do this, there is another method that you must implement:

```
public boolean hasChildren(Object element)
```

As you can see, the content provider doesn't hold a reference to any domain objects. It is the tree viewer that holds on to them itself and passes them as arguments to each method in the content provider.

In our case, a node is a *File* object. To get the children, we use `listFiles()`. We must remember to check for `listFiles()` returning null and turn that into an empty array.

To get the top-level elements, just underneath the root element, we can just reuse the `getChildren()` method.

The `getParent()` method is used to implement the `reveal(Object element)` method, which makes the tree viewer scroll its SWT tree widget in order to display a particular node in the tree. The question is: if that node is not actually being displayed at the moment, where should it be displayed? JFace looks at its parent and then the parent's parent, etc. until it reaches a node that is displayed and it then tracks down again until the target node is displayed.

The `hasChildren()` method just does the obvious (unoptimized) thing, and we end up with the code in Listing 4.

Listing 4. FileTreeContentProvider (version 1)

```
import java.io.*;
import java.util.*;
import org.eclipse.jface.viewers.*;

public class FileTreeContentProvider implements ITreeContentProvider
{
  public Object[] getChildren(Object element)
  {
    Object[] kids = ((File) element).listFiles();
    return kids == null ? new Object[0] : kids;
  }

  public Object[] getElements(Object element)
  {
    return getChildren(element);
  }

  public boolean hasChildren(Object element)
  {
    return getChildren(element).length > 0;
  }

  public Object getParent(Object element)
  {
    return ((File)element).getParent();
  }

  public void dispose()
  {
  }

  public void inputChanged(Viewer viewer, Object old_input, Object new_input)
  {
  }
}
```

Implementing the top-level Explorer class

We'll take our Hello, World program, change its name, and then, in the `createContents()` method, instead of creating a button, we create a `TreeViewer`, set its content provider to our file tree content provider, and then set the input to a folder. In this case, the folder I have chosen is the top-level folder in the C: drive.

Note that we are required to return the SWT widget from `createContents()`. As we mentioned above, the JFace Tree Viewer is not the SWT widget, so we can't return that. We need to get the actual widget from the tree viewer. We do this using `getTree()`.

Our main window class should now look like this:

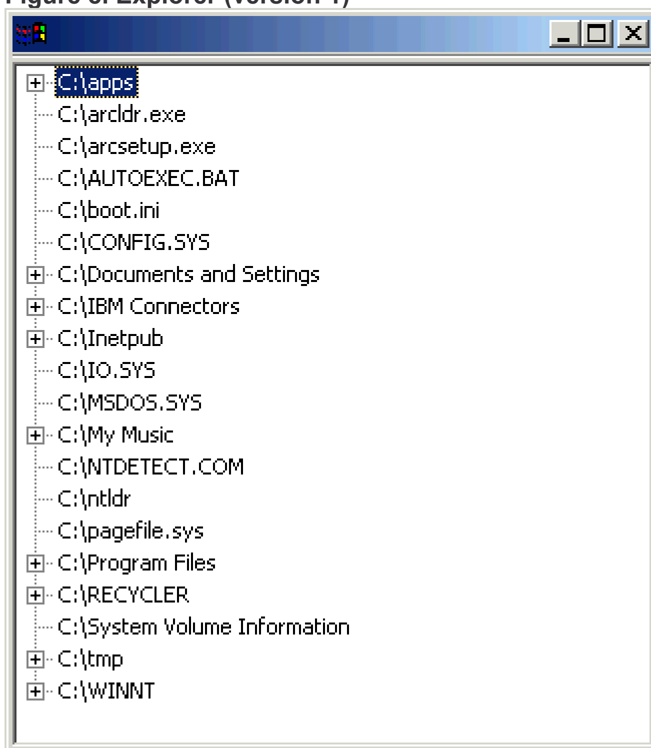
Listing 5. Explorer (version 1)

```
import java.io.*;

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

public class Explorer extends ApplicationWindow
{
  public Explorer()
  {
```

```
    super(null);
  }

  protected Control createContents(Composite parent)
  {
    TreeViewer tv = new TreeViewer(parent);
   tv.setContentProvider(new FileTreeContentProvider());
   tv.setInput(new File("C:\\"));
   return tv.getTree();
  }

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

Run this program, and you will see something like Figure 8.

Figure 8. Explorer (version 1)



Leaving aside the boilerplate code, we had to add only 9 lines of code to our Hello, World program to achieve this.

As you might guess, the files are being displayed using the `toString()` method on *File*, which is not really what we want. To change this, we need to provide a label provider.

Implementing a label provider

Just as there is a content provider object that gets the children of the tree nodes, when it comes to actually displaying the nodes, the tree viewer has another helper object: the label provider. As before, we need to set it:

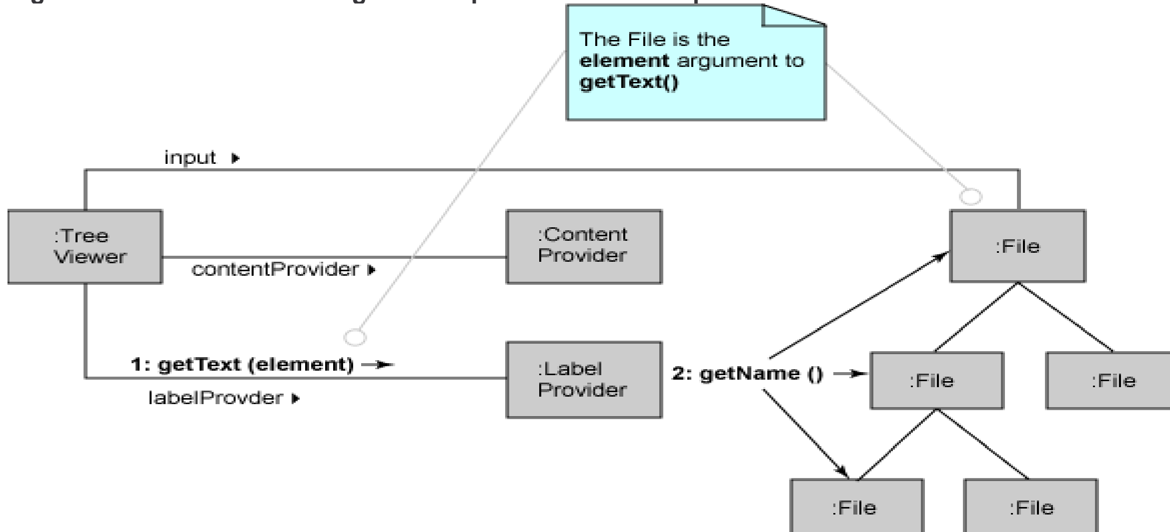
```
public void setLabelProvider(IBaseLabelProvider labelProvider)
```

and we need to implement the method to return the text to display for each element:

```
public String getText(Object element)
```

If we add the label provider to the diagram for the tree viewer, we get Figure 9.

Figure 9. Tree Viewer showing content provider and label provider



There is an interface we could implement, `ILabelProvider`, but it's easier to subclass the default implementation, `LabelProvider`. (This class is the one used if we don't set the label provider explicitly).

What we want to do in `getText()` is to return the last part of the file name - the relative file name rather than the absolute file name that `toString()` uses by default. Listing 6 shows the code.

Listing 6. FileTreeLabelProvider (version 1)

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

public class FileTreeLabelProvider extends LabelProvider
{
  public String getText(Object element)
  {
    return ((File) element).getName();
  }
}
```

And we have to remember to make the tree viewer use this label provider, as shown in Listing 7.

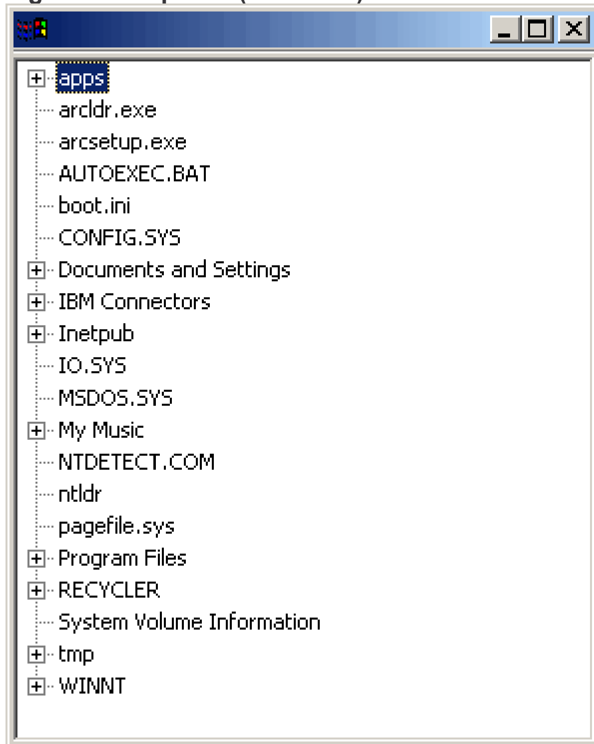
Listing 7. Explorer (version 2)

```
import java.io.*;

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

public class Explorer extends ApplicationWindow
{
  public Explorer()
  {
    super(null);
  }

  protected Control createContents(Composite parent)
  {
    TreeViewer tv = new TreeViewer(parent);
    tv.setContentProvider(new FileTreeContentProvider());
    tv.setLabelProvider(new FileTreeLabelProvider());
    tv.setInput(new File("C:\\"));
    return tv.getTree();
  }

  public static void main(String[] args)
  {
    Explorer w = new Explorer();
```

```
      w.setBlockOnOpen(true);
      w.open();
      Display.getCurrent().dispose();
   }
}
```

When we run the program this time, we get a cleaner looking result, as shown in Figure 10.

Figure 10. Explorer (version 2)

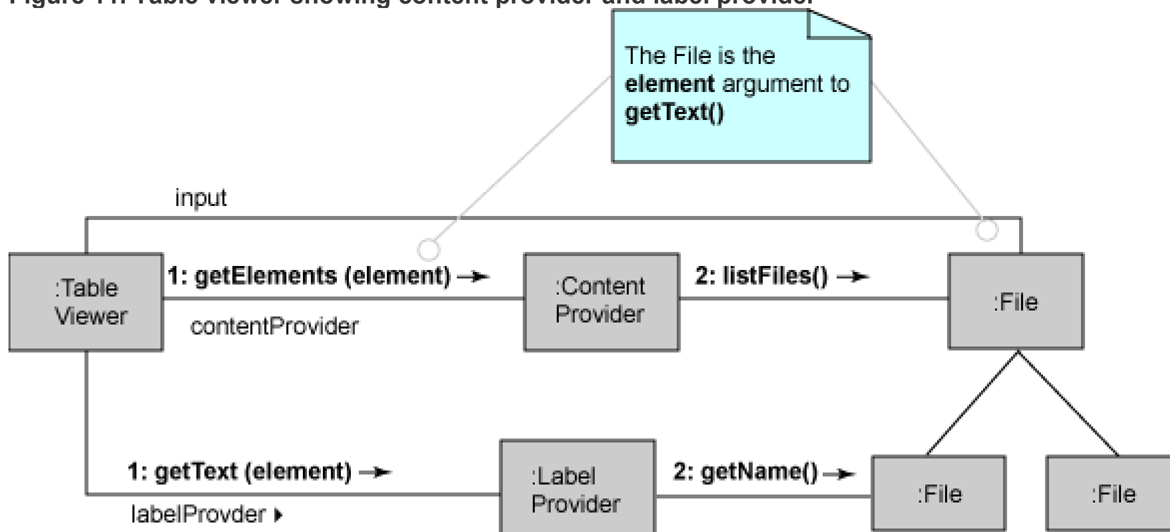


What we want to do now is to move the tree viewer to the left and put a table view on the right showing a list of the files in the folder that we have selected in the tree viewer.

Using a table viewer

To handle tables, JFace has a `TableViewer`. Just like the `TreeViewer`, it has an input (a root object), a content provider, and a label provider. It's simpler than the tree viewer since it doesn't need to deal with trees. Figure 11 shows the content provider and label provider.

Figure 11. Table viewer showing content provider and label provider



The method to set the input object is the same as before:

```
TableViewer: void setInput(Object rootElement)
```

Implementing the file table viewer content provider

Let's look at the content provider. This time, the root element is simpler than the tree viewer root element. The table viewer merely expects that the root object has a number of children, so the only interesting method to implement is the one that gets the children:

```
public Object[] getElements(Object rootElement)
```

The interface we need to implement is `IStructuredContentProvider`.

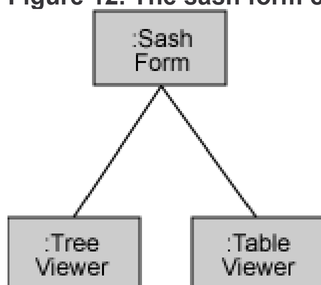
The root object will be a folder; its children will be the files/folders that it contains. So our file table content provider class looks like Listing 8.

Listing 8. FileTableContentProvider (version 1)

```
import java.io.*;

import org.eclipse.jface.viewers.*;

public class FileTableContentProvider implements IStructuredContentProvider
{
  public Object[] getElements(Object element)
  {
    Object[] kids = null;
    kids = ((File) element).listFiles();
    return kids == null ? new Object[0] : kids;
  }

  public void dispose()
  {
  }

  public void inputChanged(Viewer viewer, Object old_object, Object new_object)
  {
  }
}
```

So now we have two viewers: the tree viewer and the table viewer. To arrange them next to each other, we create an SWT SashForm widget. This widget separates its children with a border that the user can adjust. We then add the tree and the table to the sash form (Figure 12).

Figure 12. The sash form contains the tree viewer and the table viewer



The next thing we need to do is to make the table viewer look at each folder that the user selects in the tree viewer. To do this we must listen for events.

Listening for events

When the user selects an item in the tree viewer, the tree viewer fires an event, the `SelectionChangedEvent`. We need to listen for that event, and when it fires, we need to set the input for the table to be the currently selected file in the tree viewer.

To listen for the selection changed events that come from the tree viewer, we use this:

```
public void addSelectionChangedListener(ISelectionChangedListener listener)
```

When the user selects/deselects a node in the tree viewer, the selection changed listener gets called with this:

```
public void selectionChanged(SelectionChangedEvent event)
```

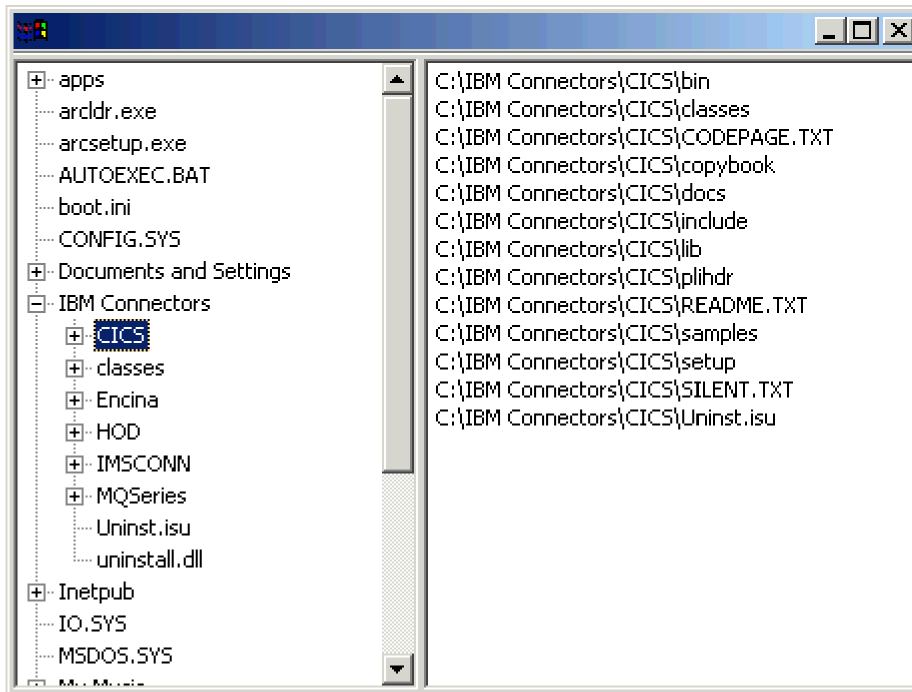
To implement this listener class, we'll code an anonymous class in the main Explorer Window. In the `selectionChanged()` method, we will need to get hold of the object that has just been selected and make this the input for the table viewer. Putting it all together, we get Listing 9.

Listing 9. Explorer (version 3)

```
import java.io.*;
import org.eclipse.jface.viewers.*;
import org.eclipse.jface.window.*;
import org.eclipse.swt.*;
import org.eclipse.swt.custom.*;
import org.eclipse.swt.widgets.*;

public class Explorer extends ApplicationWindow
{
  public Explorer()
  {
    super(null);
  }

  protected Control createContents(Composite parent)
  {
    SashForm sash_form = new SashForm(parent, SWT.HORIZONTAL | SWT.NULL);

    TreeViewer tv = new TreeViewer(sash_form);
    tv.setContentProvider(new FileTreeContentProvider());
    tv.setLabelProvider(new FileTreeLabelProvider());
    tv.setInput(new File("C:\\"));

    final TableViewer tbv = new TableViewer(sash_form, SWT.BORDER);
    tbv.setContentProvider(new FileTableContentProvider());

    tv.addSelectionChangedListener(new ISelectionChangedListener()
    {
      public void selectionChanged(SelectionChangedEvent event)
      {
        IStructuredSelection selection =
          (IStructuredSelection) event.getSelection();

        Object selected_file = selection.getFirstElement();
        tbv.setInput(selected_file);
      }
    });

    return sash_form;
  }

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

If we run this program, we get something like Figure 13.

Figure 13. Explorer (version 3)



Just as with the tree viewer, if you don't explicitly set the label provider on the table viewer, it uses the default label provider. This is what has happened here -- and if you remember, the default behavior is to display the string that is returned by the element's `toString()` method, which happens to be the absolute file name.

Let's implement our own table label provider.

Implementing a file table label provider

Only one method to worry about for now:

```
public String getColumnText(Object element, int column)
```

There are two arguments: the element to get the label for, and the column index (starting at 0).

The implementation for this is simple enough -- if we ignore the column index argument, as shown in Listing 10.

Listing 10. FileTableLabelProvider (version 1)

```
import java.io.*;
import org.eclipse.jface.viewers.*;
import org.eclipse.swt.graphics.*;

public class FileTableLabelProvider implements ITableLabelProvider
{
  public String getColumnText(Object obj, int i)
  {
    return ((File) obj).getName();
  }

  public void addListener(ILabelProviderListener ilabelproviderlistener)
  {
  }

  public void dispose()
  {
  }

  public boolean isLabelProperty(Object obj, String s)
  {
    return false;
  }

  public void removeListener(ILabelProviderListener ilabelproviderlistener)
  {
```

```
  }

  public Image getColumnImage(Object arg0, int arg1)
  {
    return null;
  }
}
```

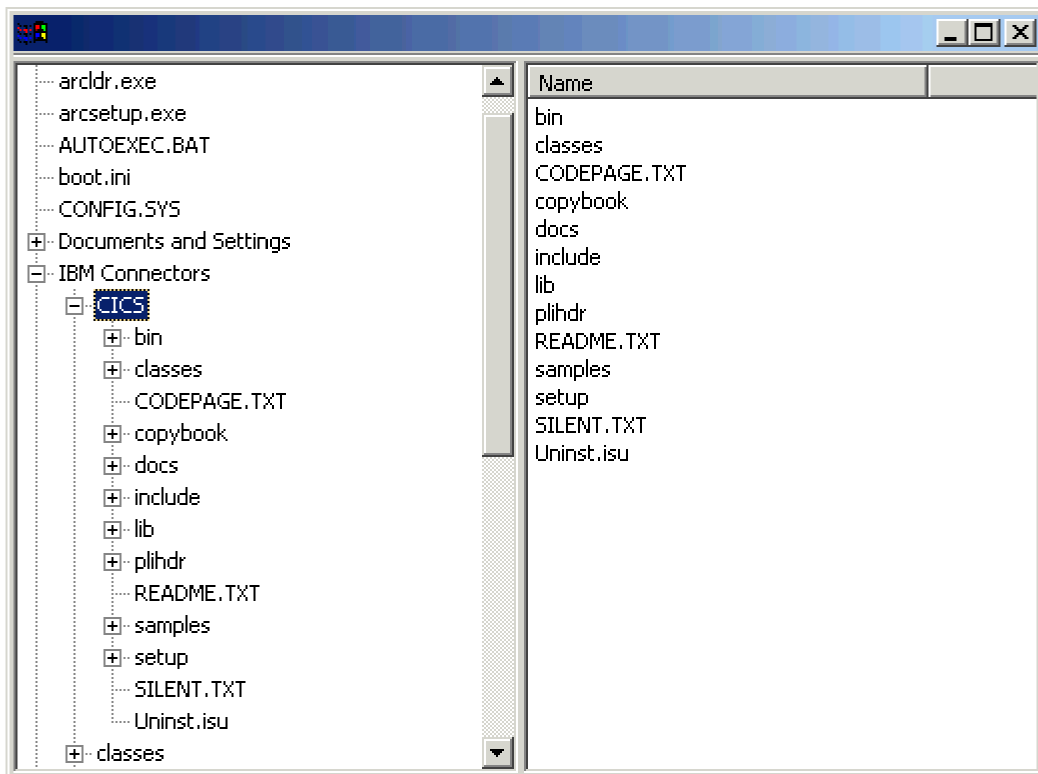
To configure the table to have one column with a header labeled "Name", we must extract the table widget from the table viewer, create a table column widget as a child of the table, and set some properties for it, as shown in Listing 11.

Listing 11. Explorer (version 4)

```
import java.io.*;
import org.eclipse.jface.viewers.*;
import org.eclipse.jface.window.*;
import org.eclipse.swt.*;
import org.eclipse.swt.custom.*;
import org.eclipse.swt.widgets.*;

public class Explorer extends ApplicationWindow
{
  public Explorer()
  {
    super(null);
  }

  protected Control createContents(Composite parent)
  {
    SashForm sash_form = new SashForm(parent, SWT.HORIZONTAL | SWT.NULL);

    TreeViewer tv = new TreeViewer(sash_form);
    tv.setContentProvider(new FileTreeContentProvider());
    tv.setLabelProvider(new FileTreeLabelProvider());
    tv.setInput(new File("C:\\"));

    final TableViewer tbv = new TableViewer(sash_form, SWT.BORDER);
    tbv.setContentProvider(new FileTableContentProvider());
    tbv.setLabelProvider(new FileTableLabelProvider());

    TableColumn column = new TableColumn(tbv.getTable(), SWT.LEFT);
    column.setText("Name");
    column.setWidth(200);
    tbv.getTable().setHeaderVisible(true);

    tv.addSelectionChangedListener(new ISelectionChangedListener()
    {
      public void selectionChanged(SelectionChangedEvent event)
      {
        IStructuredSelection selection =
          (IStructuredSelection) event.getSelection();

        Object selected_file = selection.getFirstElement();
        tbv.setInput(selected_file);
      }
    });
    return sash_form;
  }

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

Having done this, we should get a result like Figure 14.

Figure 14. Explorer (version 4)



Conclusion

We have worked through quite a lot of JFace in a very short time. We have used an application window and two viewers (Tree and Table) and implemented their content and label providers. We have used SWT widgets: Button, SashForm, Table, TableColumn, and implemented an event listener.

There are few ragged edges here, though. We ignored some of the methods in the content/label providers, the tree viewer is displaying files as well as folders, there are no icons being displayed, and there is no sign of a menu bar, tool bar, status line, or any pop-up menus.

In the next article, we'll tidy up the content/label providers and use sorting and filtering on the viewers. We'll add a status line to the window, add icons to both viewers, and learn about the JFace image registry.

Resources

- Read Part 2 in this series.
- Download the source code for the examples in this article.
- See the main Eclipse Web site for downloads, documentation, mail archives, and articles.
- For a description of using a tree viewer inside the Eclipse Workbench, see How to use the JFace Tree Viewer at the Eclipse Web site.
- For project development plans, a FAQ, and a list of handy SWT code snippets, check out the SWT component development resources.
- Learn more about Eclipse in these *developerWorks* articles:
- Find more Open source resources in the *developerWorks* Open source projects zone.

About the author

A. O. Van Emmenis is an independent consultant, specializing in Java/J2EE training and consulting, based in Cambridge, UK. Van has worked in the software industry for 20 years or so. He first started working with objects using Smalltalk in the CAD industry and now works mainly in Java. He is particularly interested in Agile methods and GUI design. You can contact Van at van@vanemmenis.com.

What do you think of this document?

Killer! (5) Good stuff (4) So-so; not bad (3) Needs work (2) Lame! (1)

Comments?

Submit feedback

IBM developerWorks > Open source projects | Java technology

developerWorks

About IBM | Privacy | Terms of use | Contact