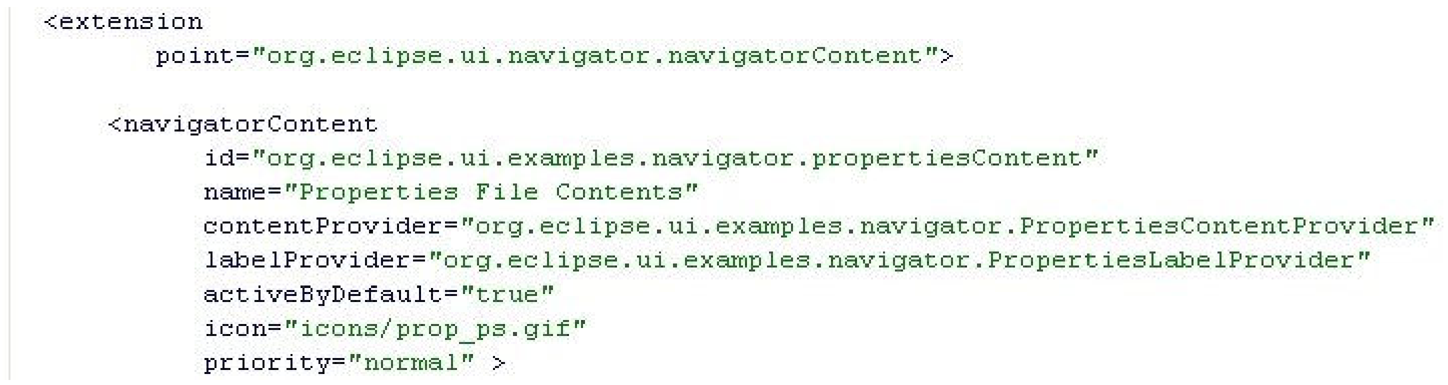
Building a Common Navigator Framework (CNF) Viewer Part II: Adding Content

CREATIVITY IS A DRUG I CANNOT LIVE WITHOUT.
-- CECIL B. DEMILLE

In this post, we're going to walk through adding a simple content extension to the viewer we defined in the last example. To avoid getting bogged down in an overly complex content provider and label provider, we will focus on a file structure with a very simple model, in this case a plain old *.properties file. When finished, our content extension will allow us to expand any *.properties file in the Example viewer and render the data in the file right in the viewer.

You may also download the full example as it stands at the end of this posting.

First, let's take a look at what the content extension will look like in the plugin.xml file. If you have read the last post, you'll know that you can build an extension like the following example using the *Extensions* tab of the *Plug-in Manifest Editor*. First by selecting *Add...* on the *Extensions* page, choosing **org.eclipse.ui.navigator.navigatorContent**, then using the right-click menu, and following the *New > navigatorContent* menu path. Alternatively, you can create this extension in the *plugin.xml* tab of the same editor. Either approach is fine, but the first approach drives the *New* menu from the Extension Point Schema, so you will see other options that are available in the raw *plugin.xml* tab.



```
<extension
       point="org.eclipse.ui.navigator.navigatorContent">

    <navigatorContent
          id="org.eclipse.ui.examples.navigator.propertiesContent"
          name="Properties File Contents"
          contentProvider="org.eclipse.ui.examples.navigator.PropertiesContentProvider"
          labelProvider="org.eclipse.ui.examples.navigator.PropertiesLabelProvider"
          activeByDefault="true"
          icon="icons/prop_ps.gif"
          priority="normal" >
```

The extension declares a content extension with the id "**org.eclipse.ui.examples.navigator.propertiesContent**" with the display name "Properties File Contents". The name is the string used in the "Available Extensions" tab of the Filters dialog.* We will get into the code for the content and label providers shortly, but for now just note that each are specified; you cannot specify just one or the other.

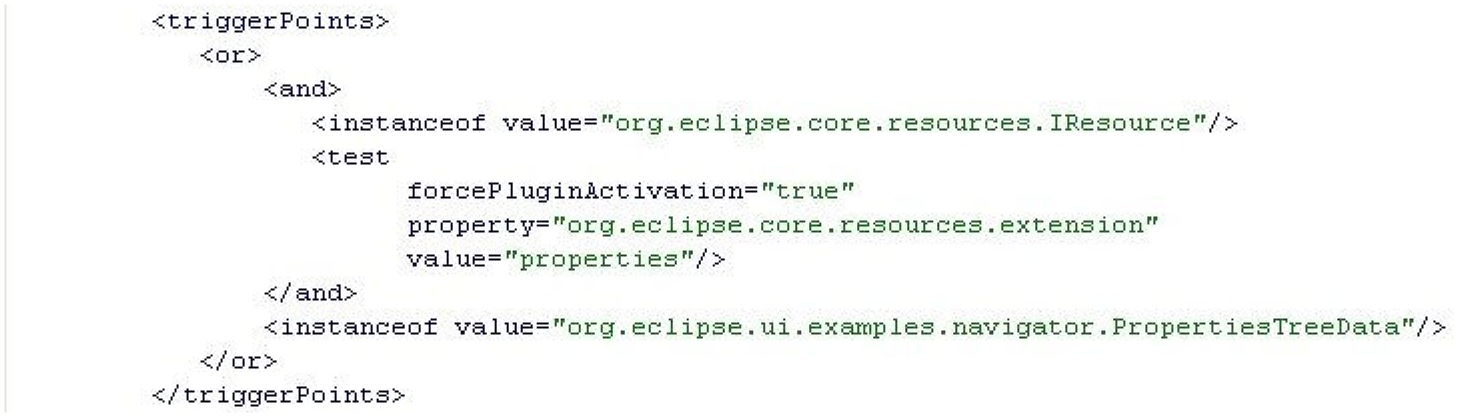
Finally we set some attributes to tell the framework how we would like our extension rendered in the viewer.

- *activeByDefault* determines whether the extension should be *active* in the default configuration (e.g. new workspace).
- *icon* determines what icon should be used when referring to the extension in the User Interface
- *priority* is used in a couple of different ways. The most prominent is to determine the relative ordering of items in the viewer (highest priority items towards the top of the viewer down to lowest priority items towards the bottom of the viewer). In general, either "normal" or "high" should be sufficient for most extensions, indicating that they should be mixed in with the resources extension (when priorities match, the label is used to sort items alphabetically) or placed just above resources under projects.

Within each **<navigatorContent />** element, we can specify many different types of extensions, but before we grow the example, we must describe to the framework when our extension should be invoked. We need to describe when we can provide children, parents, or labels and icons for nodes in the tree. We do this using Eclipse Core Expressions. For now, I'll refer you to the documentation for **org.eclipse.core.expressions** for more detail on what is possible with these, or you can use the *New > ...* menu in the *Extensions* tab to create these easily.

For content extensions, there are two important expressions: **<triggerPoints />** and **<possibleChildren />**.

The **<triggerPoints />** expression indiciates which types of nodes in the tree might be interesting to our extension. When the framework finds a node that matches the **<triggerPoints />** expression, our extension will be invoked to provide elements or children for that node. Our extension may not be the only one to be given the opportunity to provide children, as the framework will aggregate all contributed children under each node.



```
<triggerPoints>
   <or>
       <and>
          <instanceof value="org.eclipse.core.resources.IResource"/>
          <test
                forcePluginActivation="true"
                property="org.eclipse.core.resources.extension"
                value="properties"/>
       </and>
       <instanceof value="org.eclipse.ui.examples.navigator.PropertiesTreeData"/>
   </or>
</triggerPoints>
```

The **<possibleChildren />** expression indiciates which types of nodes in the tree our extension may be able to provide a label or parent for. For your scenarios that must support link with editor, or setSelection() on the viewer, the **<possibleChildren />** expression must be accurate and complete.

```
        <possibleChildren>
           <or>
              <instanceof value="org.eclipse.ui.examples.navigator.PropertiesTreeData"/>
           </or>
        </possibleChildren>
```

Once we have an extension defined, we must *bind* it to the viewers we want it associated with. Specifying a **<viewerContentBinding />** indicates that any extension which matches an included pattern is *visible* to any viewer with the *viewerId* specified in the **<viewerContentBinding />** element. Recall that we saw these in the last posting.

For the example, we will *bind* our properties content extension (with id "**org.eclipse.ui.examples.navigator.propertiesContent**" to our Example View (with id "**org.eclipse.ui.examples.navigator.view**").

```
<!-- Bind the Properties content extension to the viewer -->
<extension
      point="org.eclipse.ui.navigator.viewer">
   <viewerContentBinding viewerId="org.eclipse.ui.examples.navigator.view">
      <includes>
         <contentExtension pattern="org.eclipse.ui.examples.navigator.propertiesContent"/>
      </includes>
   </viewerContentBinding>
</extension>
```

Now that we have setup our extension, let's take a look at the code that will actually do the heavy lifting.

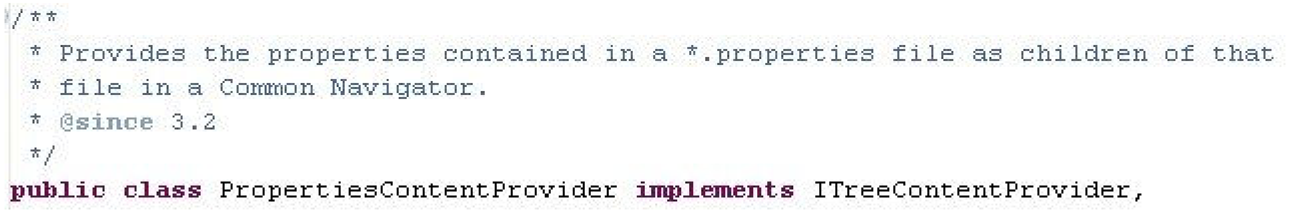
First, we need a model. A properties file has an extremely simple structure, and we will model this using just one type of model object, called **PropertiesTreeData**, which will have three fields: *name* (the name of the property), *value* (the value of the property), and *container* (the file that contains the properties model. One of these model elements will be populated for each property in the *.properties file.

In our example, the properties model will only be loaded when requested through the content provider. The content provider is used by the framework to determine the children of each element in the tree, or for any given element, determine its parent (or parents as there could be potentially more than one).

Our example content provider implements **org.eclipse.jface.viewers.ITreeContentProvider** to provide information about the tree structure. The Common Navigator framework also supports implementations of the new **org.eclipse.jface.viewers.ITreePathContentProvider**, but that is out of scope for this example.

The **PropertiesContentProvider** also handles some of the other functions required by our extension, such as listening for resource changes and updating the model (and viewer) accordingly. We will not go over these in this posting, but you can see how it works in the full source.

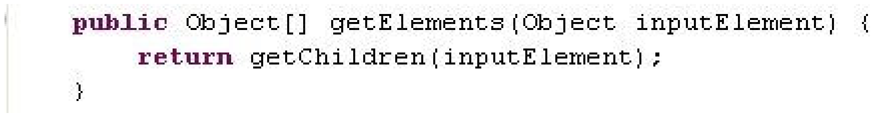
For now, we will focus on the viewer integration methods as defined by **ITreeContentProvider**.



```
/**
 * Provides the properties contained in a *.properties file as children of that
 * file in a Common Navigator.
 * @since 3.2
 */
public class PropertiesContentProvider implements ITreeContentProvider,
```

An **ITreeContentProvider** must implement **getElements(Object input)**, **getChildren(Object parent)**, **hasChildren(Object element)** and **getParent(Object element)**.

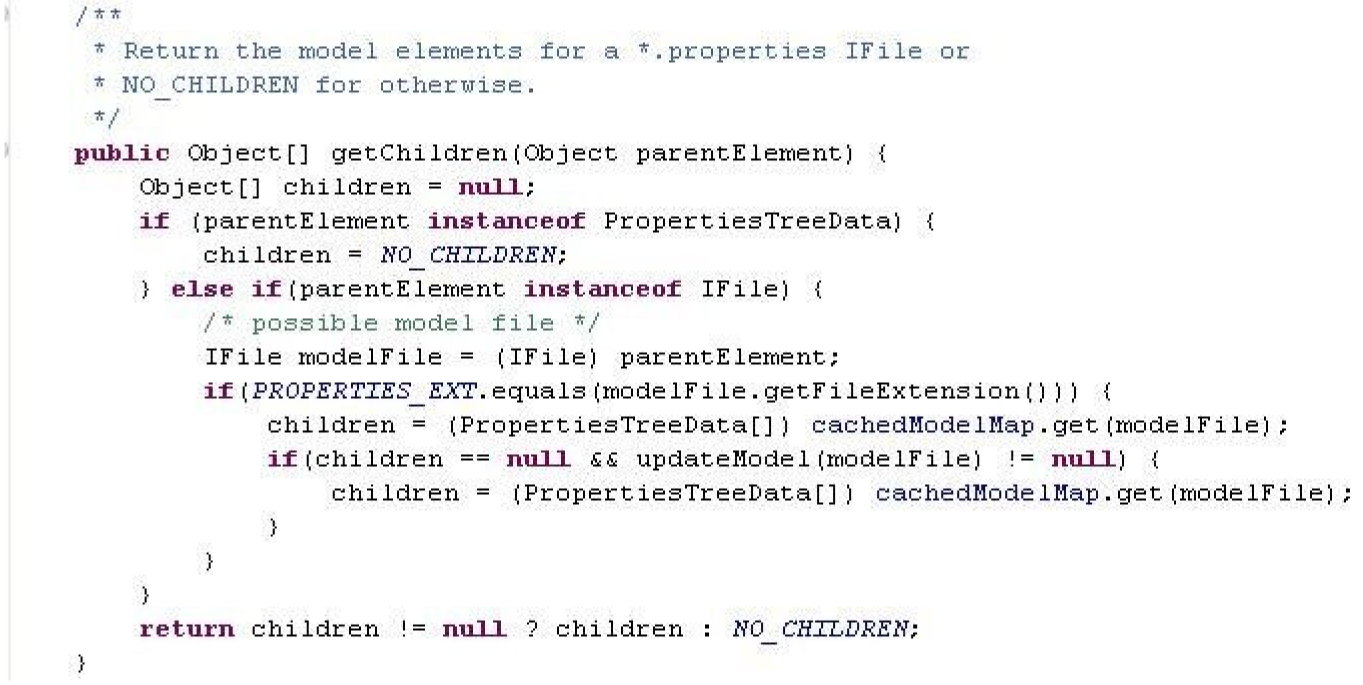
The **getElements()** method is queried for elements at the root of the viewer. Many implementations will just forward this call to **getChildren()**, and that's just what we'll do.



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

The **getChildren()** accepts an object (in our case either a *.properties **IFile** or an instance of our **PropertiesTreeData** model object because we described these in our **<triggerPoints />** expression.

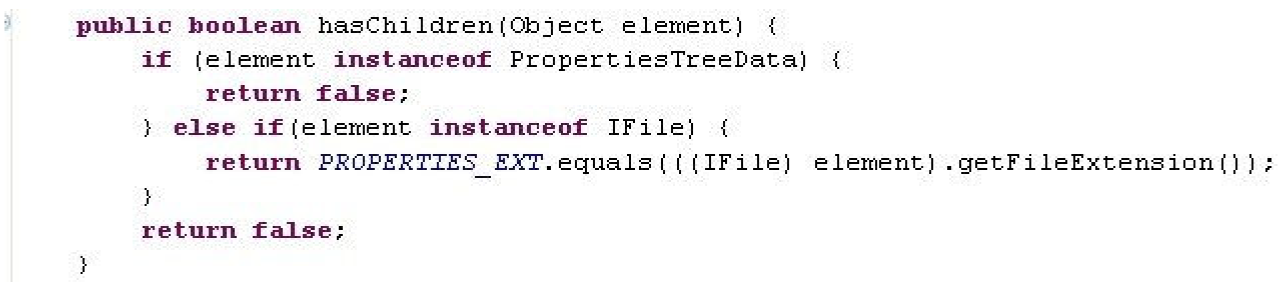
In the following implementation, we check to see if the incoming element is an instance of **org.eclipse.core.resources.IFile** and if the file extension ends in *.properties. If the incoming element meets these criteria, we check our cache of loaded models or attempt to load the model if not cached. The method **updateModel()** will create a **PropertiesTreeData** object for each property, and cache what it finds in the *cachedModelMap*. Check out the full source to see how **updateModel()** is implemented.



```
    /**
     * Return the model elements for a *.properties IFile or
     * NO_CHILDREN for otherwise.
     */
    public Object[] getChildren(Object parentElement) {
        Object[] children = null;
        if (parentElement instanceof PropertiesTreeData) {
            children = NO_CHILDREN;
        } else if(parentElement instanceof IFile) {
            /* possible model file */
            IFile modelFile = (IFile) parentElement;
            if(PROPERTIES_EXT.equals(modelFile.getFileExtension())) {
                children = (PropertiesTreeData[]) cachedModelMap.get(modelFile);
                if(children == null && updateModel(modelFile) != null) {
                    children = (PropertiesTreeData[]) cachedModelMap.get(modelFile);
                }
            }
        }
        return children != null ? children : NO_CHILDREN;
    }
```

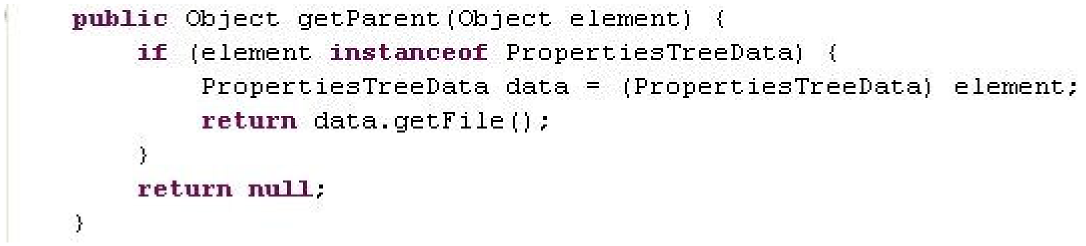
The **hasChildren()** method is optimized to return true whenever it is called with an **IFile** with the *.properties extension. The other alternative is to eagerly load the contents of the file to perform the calculation, but that approach comes with an added performance penality.

If called with a **PropertiesTreeData** model element, **hasChildren()** will return false, since none of our model elements may have children (a property has no children, but other models clearly may).



```
public boolean hasChildren(Object element) {
    if (element instanceof PropertiesTreeData) {
        return false;
    } else if(element instanceof IFile) {
        return PROPERTIES_EXT.equals(((IFile) element).getFileExtension());
    }
    return false;
}
```

The **getParent()** method will return the **IFile** that contains the **PropertiesTreeData** item or **null** if called with any other object. The Common Navigator framework will continue querying other extensions until a non-**null** parent is found or all relevant extensions have been queried. Recall that an extension may be asked to provide a parent for an element if that element matches its **<possibleChildren />** expression.



```
public Object getParent(Object element) {
    if (element instanceof PropertiesTreeData) {
        PropertiesTreeData data = (PropertiesTreeData) element;
        return data.getFile();
    }
    return null;
}
```

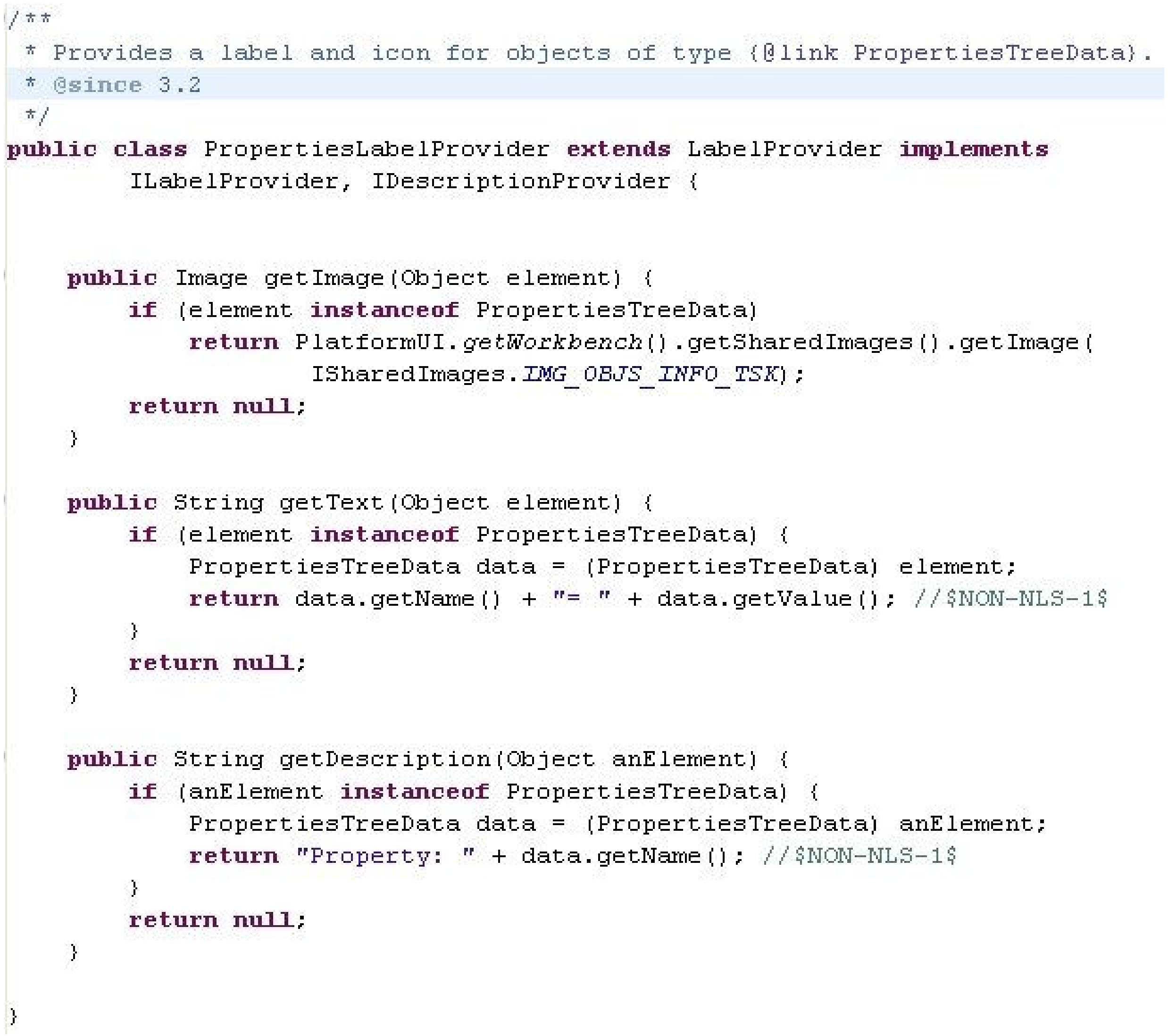
Finally, the label provider of our content extension tells the viewer how to render the icon and label for our model elements (**PropertiesTreeData**). Since we're only conerned with these elements, we do not need to worry about providing labels or icons for any other elements. Other extensions will render the labels and icons for other elements in the tree.

The **PropertiesLabelProvider** implements **org.eclipse.jface.viewers.ILabelProvider** and also **org.eclipse.ui.navigator.IDescriptionProvider**.

ILabelProvider is the default interface required for proving labels and icons by JFace.

IDescriptionProvider is specific to the Common Navigator framework and is used to provide text for the status bar in the lower left hand corner of the Eclipse window.

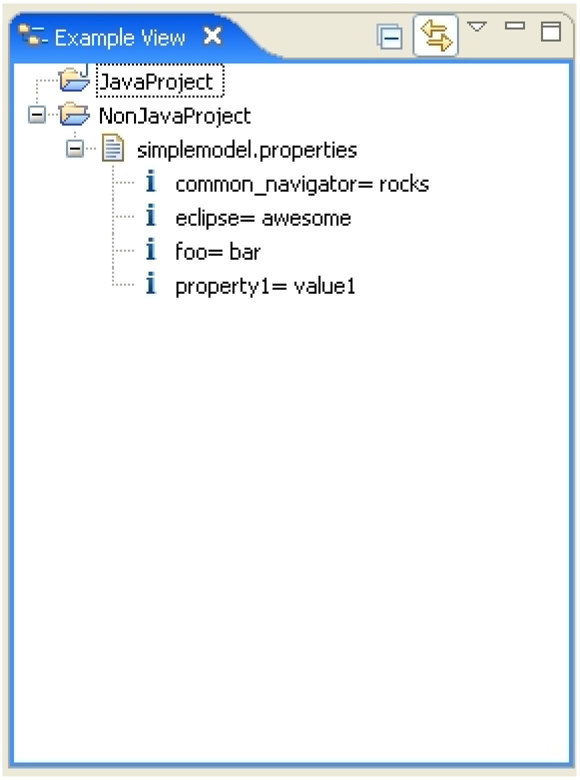
The methods required by **ILabelProvider** are **getText()** and **getImage()**. We will render the label for our model elements as the "*name= value*" string for the property (without the quotes). For the icon, we just use one of the shared icons provided by Platform/UI.



```java
/**
 * Provides a label and icon for objects of type {@link PropertiesTreeData}.
 * @since 3.2
 */
public class PropertiesLabelProvider extends LabelProvider implements
        ILabelProvider, IDescriptionProvider {


    public Image getImage(Object element) {
        if (element instanceof PropertiesTreeData)
            return PlatformUI.getWorkbench().getSharedImages().getImage(
                    ISharedImages.IMG_OBJS_INFO_TSK);
        return null;
    }

    public String getText(Object element) {
        if (element instanceof PropertiesTreeData) {
            PropertiesTreeData data = (PropertiesTreeData) element;
            return data.getName() + "= " + data.getValue(); //$NON-NLS-1$
        }
        return null;
    }

    public String getDescription(Object anElement) {
        if (anElement instanceof PropertiesTreeData) {
            PropertiesTreeData data = (PropertiesTreeData) anElement;
            return "Property: " + data.getName(); //$NON-NLS-1$
        }
        return null;
    }

}
```

The label provider will be queried for labels, icons, or descriptions for any element contributed directly by the extension or for any element that matches the <possibleChildren /> expression for that extension. If **null** is returned for the icon or the label, the framework will continuing other applicable extensions based on their <possibleChildren /> expressions and viewer **bindings**. If your extensions wishes to pass on its opportunity to provide a label or icon, always return **null**.

And that's it. The final view renders properties files right in our *Example View*.



```
JavaProject
NonJavaProject
    simplemodel.properties
        common_navigator= rocks
        eclipse= awesome
        foo= bar
        property1= value1
```

*-In general, you should externalize your strings, but we will not worry about that for these examples.

ABOUT THE AUTHOR

MICHAEL ELDER

RESEARCH TRIANGLE PARK, NORTH CAROLINA, UNITED STATES

Michael has been a Java developer since 1999 and currently contributes to Eclipse in open source and commercial venues. Most recently, Michael has contributed the Common Navigator Framework (CNF) to Eclipse 3.2. Prior to that, he helped design several of the frameworks and API in the Web Tools Platform (WTP), and continues to contribute fixes to WTP through the open source process. Michael is currently part of the Services Oriented Architectures (SOA) Tools Team for IBM Rational® based in Research Triangle Park, NC.