Extending JBossIDE

An introduction to extending eclipse and JBossIDE

1.0

Table of Contents

1

Introduction

1.1. Document Purpose

This document will give an introduction to extending eclipse and the JBoss IDE plug-ins with plug-ins of your own. It will give an introduction into the main libraries the JBoss IDE plug-ins make use of (webtools, JDT, etc), and a general overview of the structure of the JBoss IDE projects. All plug-ins to eclipse and JBoss IDE are written in Java. It will detail how to use common extension points in eclipse.

This document will not teach you how to use eclipse, JBoss IDE, or program in Java. A familiarity with eclipse and java in general will be assumed. For those who are not familiar with eclipse, a good guide on the software is available at http://www.eclipse.org/documentation/pdf/org.eclipse.platform.doc.user_3.0.1.pdf. Terminology from this document (view, context menu, editor, perspective, etc) will be used throughout the document. If you've never used eclipse to program in Java, the Java Development User Guide is available at http://www.eclipse.org/documentation/pdf/org.eclipse.jdt.doc.user_3.0.1.pdf. Other resources are listed at the end of the article.

1.2. What is Eclipse?

Aside from being an IDE, what exactly is eclipse? Eclipse in a programatic sense is simply a plug-in loader. It has a small core codebase which is in charge of loading plug-ins, libraries, and other resources. Each plug-in is represented by an OSGi bundle when running.

Part of eclipse's success has been due to its extensible nature and the ease with which plugins can extend and enhance each other. Any plug-in can define their own extension points through which others may extend them. Any plug-in, for example, can add a top level menu to the main menu bar of the application, because the plug-in in charge of that menu has defined extension points through which to do so. If a plug-in defines no extension points for others to use, then the plug-in is not extendable.

1.3. What can I extend in JBoss IDE?

JBoss IDE is a term that refers to a handfull of projects, specifically: JBoss Aop IDE, EJB3 tools, jBPM tools, Hibernate Tools, and what is called the IDE Core. Each one of them may consist of multiple plug-ins themselves. Aop IDE, for example, consists of two plugins: aop.core and aop.ui. Other projects consist of several plugin-ins as well.

JBoss is currently growing, and is always taking on and integrating new projects. The IDE team, likewise, has also taken on new contributers, and a refactoring may be necessary in the future. Historically the IDE has not had many re-usable features separated, and hopefully this will change. Because of this, JBoss IDE currently does not define

any extension points, but we encourage you to contact us with requests or ideas as to which views, context menus, or other eclipse elements in our plug-ins you'd like to see extensible. Also, if you see some functionality (a dialog box for example) that you think could be separated and put in some IDE-commons project for others to use, that would also be helpful.

If you'd like to add additional functionality to the AOP AspectManager view's context menu, for example, and would like us to define an extension point there so that your plug-in can do so, don't hesitate to ask. If you desired, and your added features proved helpful to others, you could contribute your code back to JBoss IDE for integration. Because JBoss IDE currently does not define any extension points, this document will focus primarily on using eclipse's extension points.

1.4. Technologies We Use

You will not need to understand all of these technologies to extend eclipse, however each one has a purpose and if your plugin intends to provide similar functionality, they should make use of these libraries whenever possible. These libraries are currently used by some of our plug-ins, and so there will be example code to look at.

1.4.1. JDT

Java Development Tooling "allows users to write, compile, test, debug, and edit programs written in the Java programming language." Through APIs they can create java source files, add methods, fields, or inner types. These APIs are used throughout the JBoss IDE plug-ins. One example would be in the ejb3 plugin, adding a business or home method via the j2ee context menu. Another example is reflection and using the JDT to check if a user's java source file in an editor has a specific method. These are, of course, just two examples of many.

Eclipse provides a tutorial on JDT here: http://www.eclipse.org/documentation/pdf/org.eclipse.jdt.doc.isv_3.0.1.pdf

1.4.2. GEF

Gef is the Graphical Editing Framework. It "allows developers to create a rich graphical editor from an existing application model." This library allows you to use graphical shapes, arrows, etc, to create flow charts, uml diagrams, or other things. JBoss IDE primarily uses this library in JBoss jBPM's Graphical Process Designer.

Further information on GEF can be found at: http://www.eclipse.org/gef/

1.4.3. EMF

"EMF is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. Models can be specified using annotated Java, XML documents, or modeling tools like Rational Rose, then imported into EMF. Most important of all, EMF provides the foundation for interoperability with other EMF-based tools and applications. "

More information can be found at: http://www.eclipse.org/emf/

1.4.4. Webtools

"The WTP project includes the following tools: source editors for HTML, Javascript, CSS, JSP, SQL, XML, DTD, XSD, and WSDL; graphical editors for XSD and WSDL; J2EE project natures, builders, and models and a J2EE navigator; a Web service wizard and explorer, and WS-I Test Tools; and database access and query tools and models."

More information can be found at http://www.eclipse.org/webtools/

1.4.5. SWT

SWT: Swt is a graphics widget toolkit that competes with Swing. It is used entirely throughout eclipse and is a nececssity for any and all gui plug-ins. JFace is a layer above SWT, and is also used extensively. SWT is implemented specifically for each hardware platform, unlike Swing which is coded platform independent. This should not affect your coding, however, as SWT contains a consistant API across platforms. In eclipse, all SWT elements run as one thread.

More information can be found at http://www.eclipse.org/swt/

1.5. Current JBoss IDE Projects

Following is a description of the current plugins in the IDE, as well as some information about their plug-in structure and what other parts of eclipse they extend. This information can be used later when looking for a concrete example of some type of extension.

1.5.1. JBoss AOP IDE

Homepage: http://jboss.com/products/jbosside

This project is an IDE layer on top of the JBoss AOP libraries, used to make development of AOP projects easier. (AOP IDE: , AOP: http://www.jboss.org/products/aop) It makes use of JDT throughout the application. It contains two main views, the Aspect Manager and Advised Members views. It comprises two main plugins, a core and a ui. The core has a model that allows the ui to graphically represent pointcuts, expressions, typedefs, advisors, and other information graphically through the views.

Each view has a context menu that changes depending upon what is currently selected. This plug-in also adds to the other views' context menus via the org.eclipse.ui.popupMenus extension point. Other features include resource listeners that update the model every time a file is changed, markers on the sides of editors to more easily show information, and MarkerResolutionGenerators (ctrl+1) which pops up advice to the user based on what markers are present on the selected element. It also marginally extends the default editor.

1.5.2. EJB3 Tools

Homepage: http://jboss.com/products/jbosside

This a project centered around the simplification of development of EJB3 elements such as session beans, message driven beans, and entity beans.

1.5.3. jBPM Designer

Homepage: http://jbpm.org/gpd/

A graphical designer for jBPM. Automatically works on PARs as a mutli-tabbed editor with a graphical process design editor and XML descriptor editor.

1.5.4. Hibernate Tools

Homepage: http://tools.hibernate.org

A fully featured hibernate toolset that includes reverse engineering (with ant integration and ejb3 source support), context-sensitive auto complete on hibernate descriptors, HQL query editor and result set view, live entity property viewing and editing, and more.

1.5.5. Others

Your Plug-in Here

2

Example 1 - Getting Started

This lab will help you set up the PDE (Plug-in Development Environment). It will also take you through the process of generating the Hello World template, and explain the pieces of the code where applicable.

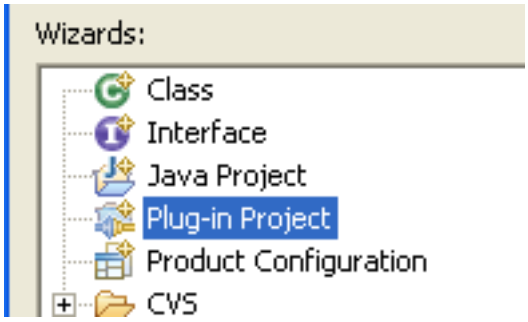
2.1. Configuring your PDE

Select the "Target Platform" page under Plug-in Developmeznt in the Window->Preferences dialog and verify the location of your target platform. This step sets the run-time workbench instance path, which is where your plug-in will be run when testing it. You don't need to change any values here, but look around to see what is available to be changed should you need it.

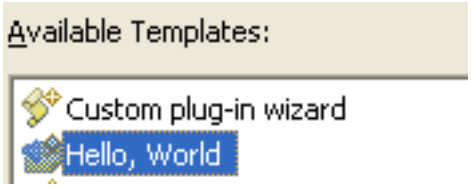
2.2. Creating a New Plug-in Project

The first thing you should do is switch to the plug-in development perspective. To do this, select `Window->Open Perspective->Other...` and choose Plug-in Development.

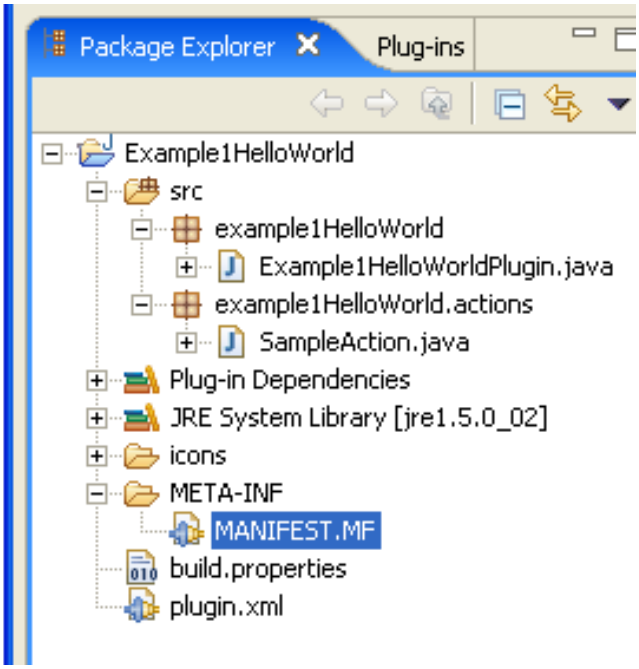
From here, the next step is to create a new plug-in project. `Select File->New->Project...` and choose Plug-in Project in the list of wizards shown at right.



You should call your plug-in `Example1HelloWorld`. You can click next twice without making any changes. From here, select the Hello World template.



After expanding your project from the package manager, it should look something like this.



So what has been created for us?

First we see `Example1HelloWorldPlugin.java`, which is the actual plugin class first loaded by the eclipse platform. We also see a `SampleAction.java` as well. Finally, there is the `MANIFEST.MF` and `plugin.xml` files.

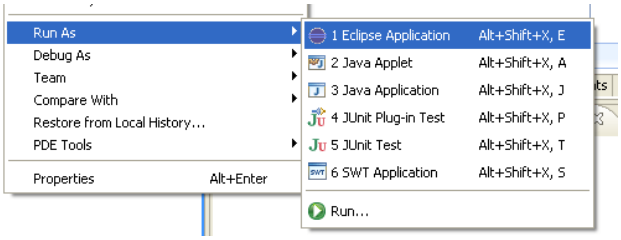
If we look at the `Example1HelloWorldPlugin` class, we notice that it has a singleton instance which is used by the platform and can be used by you as well. There's no reference anywhere to the `SampleAction`. Most of the methods completed just complete behaviors expected by the eclipse platform.

The `SampleAction` just pops up a dialog that says, predictably, Hello World.

The `MANIFEST.MF` and `plugin.xml` files can both be modified using the same graphical editor, accessed by double-clicking on either file in the package explorer view. The plugin.xml file contains information about your plugin, how to load it, and what extensions it plans on using. You can also see the raw text for each file by clicking on their respective tabs inside their editor.

You can declare your intention to use extension points through the gui interface, which will then automatically update the `MANIFEST.MF` and `plugin.xml` files.

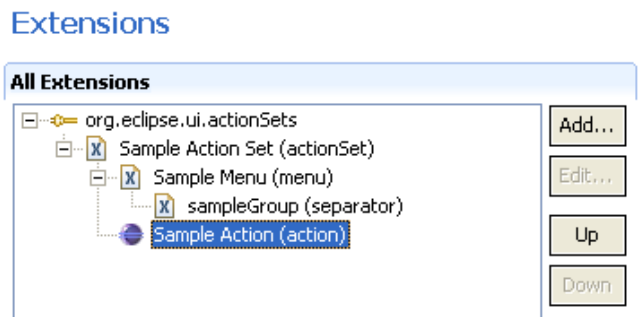
To run this plug-in, right click on its project, and select the `run as->Eclipse Application` menu item. This will invoke the runtime environment used to test your plug-in. When the new eclipse instance is loaded, you'll notice a `Sample Menu` menu with a `Sample Action` sub-menu.



2.3. Exploring the Code

The code that links your plug-in with the top-level menu is the extension points your plug-in are using. Looking at your plug-in's `plugin.xml` file's `Extensions` tab shows you an extension to `org.eclipse.ui.actionSets`, which directs execution to the `example1HelloWorld.actions.SampleAction` action.

To the right is what we see when browsing the extensions tab of the plugin. What it shows is that we have one extension to the org.eclipse.ui.actionSets extension point. That extension defines a new menu, and an action to be added to that menu. The action element contains a class name field to help facilitate lazy initialization.



The xml code for this is as follows:

```
<extension
         point="org.eclipse.ui.actionSets">
      <actionSet
            label="Sample Action Set"
            visible="true"
            id="Example1HelloWorld.actionSet">
         <menu
               label="Sample &Menu"
               id="sampleMenu">
            <separator
                  name="sampleGroup">
            </separator>
         </menu>
         <action
               label="&Sample Action"
               icon="icons/sample.gif"
               class="example1HelloWorld.actions.SampleAction"
               tooltip="Hello, Eclipse world"
               menubarPath="sampleMenu/sampleGroup"
               toolbarPath="sampleGroup"
               id="example1HelloWorld.actions.SampleAction">
         </action>
      </actionSet>
   </extension>
```

3

Adding Context Menus to Other Views

3.1. Goal

Context menus in a view are generated from an array of places. The first place is the view itself, and what menu actions it wants to provide to its user. The second and third places are object contributions and viewer contributions, which are how other plug-ins can contribute to a view's context menu.

When looking at the package explorer view, the view is displaying some underlying model. For this specific view, it is trying to keep track of resources, and using JDT to provide further data other than that this is simply a file. The view chooses how to present and display its data through content providers and label providers, and we'll look into those more in later chapters. This is why if you expand the tree for a .java file, you'll see its constructors, its methods, and its inner classes. You won't see things such as that in the Resource View.

When you right click on an item in the Package Explorer view, you are generating the context menu for some object that is being modeled by the view, exactly how it is modeled by the view. You may select a .java file and expect it to give you an IFile back, but this assumption would be incorrect. It is entirely dependent on how that view models its data, and for the package explorer view, it would try to give you that object as a JDT element, such as ICompilationUnit. Selecting a package would not give you a directory object, but rather an IPackageFragment object. It is completely dependent on the view as to how they would like to present, model, and display their data.

The goal for this example is to add two different types of context menus. One menu will be added to a specific view via a viewerContribution. The other will be added when certain types of objects inside a view are selected, via an objectContribution.

3.2. Create the Plugin

Select `File-> new->other...` and Plug-in Project. Name this project `Example2ContextMenus`. We won't use any template for this example, though the basic plugin class will still be generated for you.

Next, create a new package by right-clicking on the `src` directory and selecting `New->Package`. Name this package `example2ContextMenus.actions`.

Create a new class by right-clicking on that package and selecting `New->Class`. Name this class `ContextAction1.java` and complete it as follows:

```
package example2ContextMenus.actions;

/* assorted import statements removed for brevity */

public class ContextAction1 implements IViewActionDelegate {

        public void run(IAction action) {
```

```
                MessageDialog.openInformation(
                        new Shell(),
                        "Example2 Action 1",
                        "Hello, Eclipse world Action 1");
        }

        public void selectionChanged(IAction action, ISelection selection) {
        }

        public void dispose() {
        }

        public void init(IViewPart view) {
                // Used for viewerContributions, not for objectContributions
        }
}
```

The `IActionDelegate` interface provides a method `selectionChanged`, which allows the action to know what is selected, or, in other words, what object this action is acting upon. Since we are just popping up a dialog box, the selection does not matter at all, however if you were implementing an action that would delete some item from an underlying model, you would need to keep a reference to that selection.

3.3. Creating a popup with an ObjectContribution

3.3.1. Creating the Extension

The next step is to add this information to a context menu somewhere, so that the action can actually be invoked. This is done through the plugin.xml file or its gui editor. We will want our new menu to appear when right-clicking on a file in any view, such as the resource view or the package explorer view. The class for this is IFile, and is located in the org.eclipse.core.resources plug-in.

So, first go to the `extensions` tab for the gui editor to the `plugin.xml` file. Next, click add, and select `org.eclipse.ui.popupMenus` from the list. We will NOT use template provided. Click `finish`.

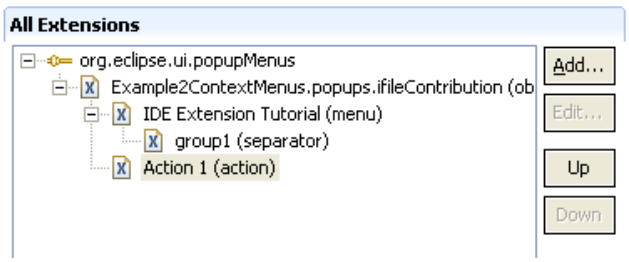
In the gui editor, right click on the new extension and select `new->objectContribution`. On the right, set the id field to `Example2ContextMenus.popups.ifileContribution` and the objectClass field to `org.eclipse.core.resources.IFile`.

Under that, we'll make a menu and an action. To create the menu, right click on our objectContribution and select `new->menu` set the id to Example2ContextMenus.menu1 and the label to IDE Extension Tutorial. Right click on the new menu and select `new->separator` to create a separator in our menu. Set the name of this separator to group1.

Set the values as follows: id = Example2ContextMenus.action1, label = Action 1, menubarPath = Example2ContextMenus.menu1/group1, class = example2ContextMenus.actions.ContextAction1

As you can see, the class is the same one we created earlier, and the menubarPath matches the separator we made just a few seconds ago.

When done, save the file. The extension tree should now look like this.

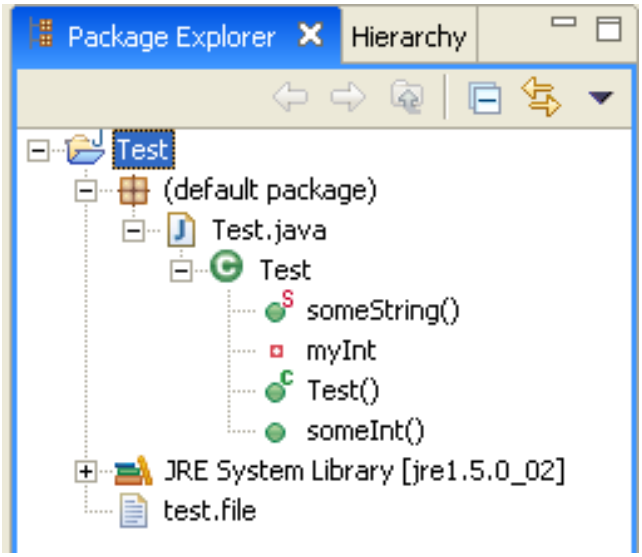


3.3.2. Testing

The first thing is to run the application much as you did in the last example. This menu won't appear until you right-click on a file inside the runtime environment, though, so once the workbench is running, we'll need to create a project and some files. First I created a simple text file (`file-<new-<File`). I also created a Test.java file and added a few dummy methods via the following code:

```
public class Test {
        private int myInt;
        public Test() {
                this.myInt = 1;
        }
        public int someInt() {
                return myInt;
        }
        public static String someString() {
                return "HelloWorld";
        }
}
```

Once this code is added, your expanded package explorer view in the runtime environment should look like the one to the right.



3.3.3. Results

Now, if we right click on Test.java, on someString(), myInt,Test(), or someInt() in the package explorer, our menu will not be present. If we wanted our menus to appear there, our objectContribution would have needed to have an objectClass of IResource for the Test.java element, IMember for a method or field, IField for just a field, IMethod for just a method, etc. If you HAD used one of these class types for our object contribution, our menu element would appear when right-clicking on an element of that type in ANY view. So if we had used IField, it would appear both in the package explorer and the outline views, when a member field is selected and its context-menu requested.. If we instead right-click on test.file in the package explorer view, our menu addition DOES appear.

If we then load up the Navigator view (`window->show view->Navigator`), clicking on either test.file OR test.java will result in our menu being shown. The reason for this is how that view keeps track of the objects in their model. Navigator represents everything as IFiles, whereas the package explorer and the outline views represent elements as members of JDT when possible (IMethod, IField, IType), and IFiles when JDT simply does not apply.

3.4. Creating a popup with a ViewerContribution

3.4.1. Creation

Right-click on our popupMenus extension and select `new->viewerContribution`. Set the id to `Example2ContextMenus.viewerContribution2` and the targetID to `org.eclipse.ui.views.TaskList`
.

This targetID designates the id of the extension point that our extension matches. For this example, our menu will appear in the task list, another view which keeps track of all of the TODO's in your java code.

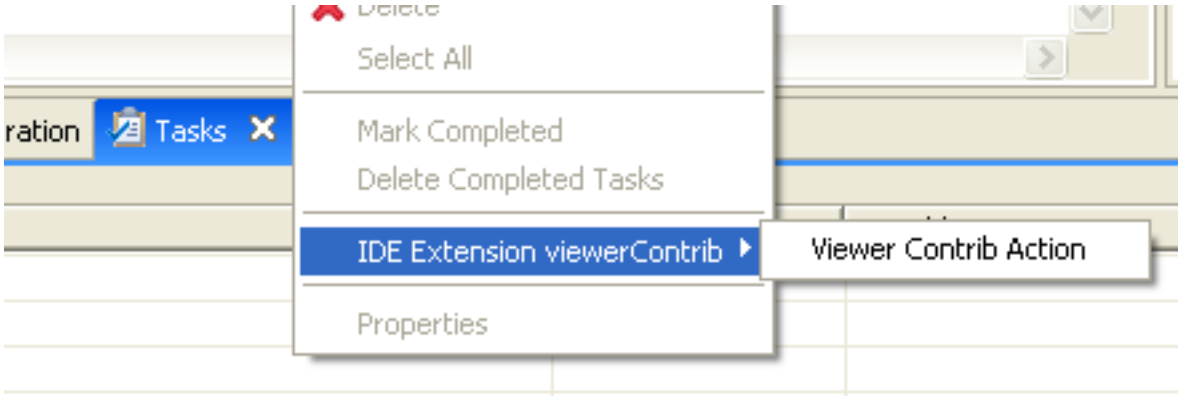
To the viewerContribution, add a menu with an id of `Example2ContextMenus.menu2`, and a label of `IDE Extension viewerContrib`. Give the menu, add a separator named `group2`.

To our viewer contribution, add an action. Give the action an id of `Example2ContextMenus.action3`, a label of `Viewer Contrib Action`, a class of `example2ContextMenus.actions.ContextAction1` (which is the same action as in the objectContribution), and a menubarPath of `Example2ContextMenus.menu2/group2`.

Then save the file.

3.4.2. Testing this Extension

Run the program the same way as before, by right-clicking on the project and selecting `Run As->Eclipse Application`. Once the runtime environment has loaded, open the Task List view by selecting `Window->Show View->Tasks`. Then try right-clicking inside the view. The menu option should appear, as shown below.



3.5. Example Conclusions

Object contributions to popup menus are easy to implement, but they require knowing how the plugin you're adding a menu item to stores its data. As we saw earlier, the Navigator view stores a file as an IFile, whereas the Package Explorer will try to convert a java file into a JDT-style model.

Viewer contributions, on the other hand, require you to know the id name of the viewer where you want your menu to appear. This could involve digging through the code of that plug-in to discover the id of the viewer. You can also find this id in the target plug-in's `plugin.xml`

One thing we did not do with the menus was visibility, which can be added to either an objectcontribution or a viewercontribution. Visibility allows you to use some basic parameters (such as object class, object state, plugin state, or system property) and some simple boolean conjunctions, to more finely tune which elements will show your menu and which will not. I encourage you to play around with sub-elements to the popupMenu extension point.

4

Dependencies

4.1. Introduction

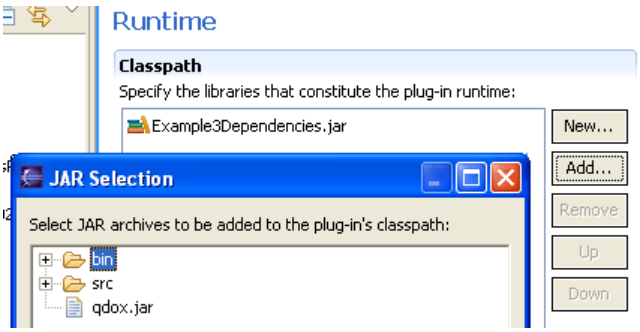
Your plug-in project can have two types of dependencies. One is to depend on another project. The other is to depend on a .jar file or library.

4.2. Jar Dependencies

There are a few things to remember when depending on a library jar file. The first is that these .jar files must be in your plug-in's directory structure, so that they can be included in the OSGi bundle of your plugin, and then accessed by your classes. They must then be added to your classpath, and designated as jars to search through.

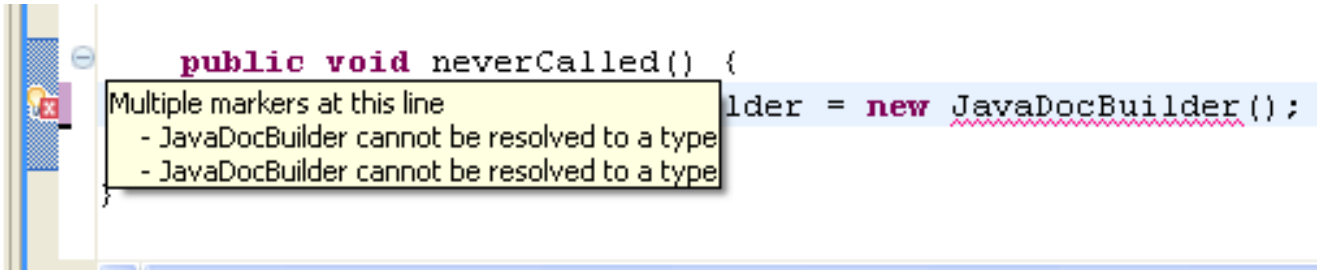
This is all done very quickly and easily. So to begin, we first make a new plugin and name it Example3Dependencies. We then put our .jar file into the top level directory of our plugin. (Just copy it in or paste it in.)

The next step is to add the jar to our runtime tab in our manifest editor. So we open the editor, click to the runtime tab and look at the classpath box. Here, we click "add", and select the jars we want to include. Then we save the editor.



If we add the following code to the plugin class file, we'll notice it didn't work.

```
        public void neverCalled() {
                JavaDocBuilder builder = new JavaDocBuilder();
        }
```



Our classpath was NOT updated. It added the jar to the manifest file, or the plugin.xml, but it did not update the classpath. To do that, right click on your project and select `PDE Tools>Update Classpath`. This usually fixes most problems regarding packaging jars with your plugin.

And in this case, it did just that. (If your error didn't go away right away, try focussing on the editor of your plugin and pressing `ctrl+shift+o`, which organizes your import statements and should import this item. Then save the file) We won't run this application, as it doesn't actually do anything. This exercise was just demonstrative.

4.3. Project Dependencies

Most .ui plug-ins depend on a corresponding .core plugin. How do we fascilitate this?

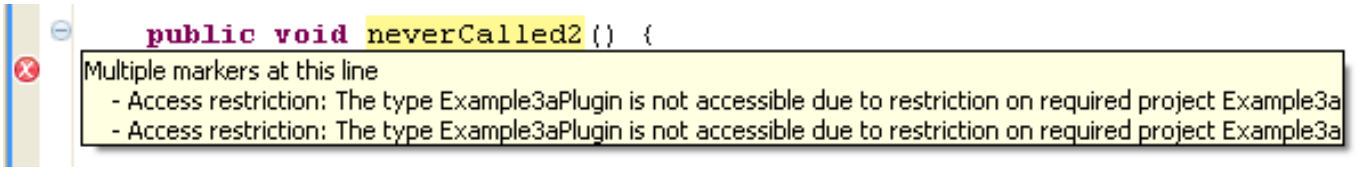
First, make a new plug-in project, and call it Example3Z.

Next, let's try to get a reference to that plug-in's singleton object in a new method. Put the following method in the `example3Dependencies.Example3Dependencies.java` .java file, right after our `neverCalled()` method.

```
public void neverCalled2() {
                Example3ZPlugin plugin = Example3ZPlugin.getDefault();
}
```

Next, we have to add it to our dependencies, in the manifest gui editor for the `Example3Dependencies` project, on the `Dependencies` tab. Click `add` and select `Example3Z`. Then save the file.

If we go back and look at our plugin .java file, `example3Dependencies.Example3Dependencies.java`, we'll see a new error.



What we must do now to complete the dependency is have `Example3Z` open up its packages for extending or referencing plug-ins to have access. To do this, open up the `Example3Z`'s manifest gui editor, select the `runtime` tab, and click `Add...` next to the Exported Packages list. Finally, select the `example3Z` package and save the file.

4.4. Conclusion

And that's how you depend on jars and other plug-in projects in eclipse plug-in development.

5

A plug-in with views

5.1. Adding a view

Adding a view to a plug-in is as simple as using the `org.eclipse.ui.views` extension point, declaring a category for it to be listed in, and then declaring your view with some parameters. If you're feeling really ambitious, you can designate where this view should appear when loaded relative to other views, or even what perspective you want it to appear in. You can do this through the gui editor for the plugin.xml file, or you can do it manually.

In the included example, `Example4Views`, I've declared a plug-in with two views. The main plug-in class contains only the default information and nothing more. It contains only the boiler-plate code needed to allow the plug-in to run.

Looking at the plug-in.xml, however, we'll notice exactly what is detailed above. Whether you use the gui or look at the raw xml, what we have is an extension to `org.eclipse.ui.views` as well as one to `org.eclipse.ui.perspectiveExtensions`. For completeness, the xml for one view is listed below.

```
<extension point="org.eclipse.ui.views">
   <category name="Sample Category" id="Example4Views">
   </category>
   <view name="Table / List View"
                  icon="icons/sample.gif" category="Example4Views"
         class="example4Views.views.TableView"
         id="example4Views.views.TableView">
   </view>
</extension>
<extension
      point="org.eclipse.ui.perspectiveExtensions">
   <perspectiveExtension
         targetID="org.eclipse.ui.resourcePerspective">
      <view
            ratio="0.5"
            relative="org.eclipse.ui.views.TaskList"
            relationship="right"
            id="example4Views.views.TableView">
      </view>
   </perspectiveExtension>
</extension>
```

5.2. Creating the control

Creating the control for the view involves any amount of SWT layout coding that you find desirable. For our first two examples, `TableView` and `TreeView` the SWT is minimal. All we do there is create our JFace Viewer, which helps us organize and display our data. For those two classes, the actual SWT code is as follows:

```
                                // TreeView.java
                                viewer = new TreeViewer(parent, SWT.MULTI | SWT.H_SCROLL | SWT.V_SCROLL);

                                //TableView.java
                                viewer = new TableViewer(parent, SWT.MULTI | SWT.H_SCROLL | SWT.V_SCROLL)
```

The `SWTView.java>` has a much more SWT-oriented `createPartControl` method, and it may provide as a slight jumping off point for investigating SWT code, but it is really just a few simple examples.

NOTE: Having Views with ample SWT controls is not customary, and is often frowned upon in view implementations. However, it is possible to do it and so it was included as an example.

5.3. Content Providers

5.3.1. Introduction

The content provider for a viewer helps organize the data from some underlying model and figure out how to display it in the viewer. The content provider is a JFace element and so rests a level above the actual list, table, or tree lying underneath. It provides APIs that will quickly allow you to change the SWT element's data without dealing with the annoyances of those underlying elements.

5.3.2. TableViewer's Content Provider

The TableViewer only requires an `IStructuredContentProvider`, which only requires three methods. I think the code is rather self-explanatory here, and there's not much more I can say on it.

`inputChanged` is used to change the top level of the model, to designate a new model, etc. In the two examples mentioned above, we use the viewSite as our input, but that's completely arbitrary amd just serves as an example. For a real model (as opposed to the strings example), the root element might be something like an xml Document, or whatever model your view intends to display data from.

For Completeness:

```
      class ViewContentProvider implements IStructuredContentProvider {
              public void inputChanged(Viewer v, Object oldInput, Object newInput) {
              }
              public void dispose() {
              }
              public Object[] getElements(Object parent) {
                      ArrayList list = new ArrayList();
                      list.addAll(Arrays.asList(new String[] {"One", "Two", "Three"}));
                      list.add(new Integer(5));
                      return list.toArray();
              }
      }
```

5.3.3. TreeViewer's ContentProvider

The TreeViewer requires an ITreeContentProvider, and its associated methods. This implementation also uses the viewSite as its root, however any calls requesting the children of the root are deferred to the proxy element, the `invisibleroot`. Then, from the children returned, further children can be found through the appropriate APIs.

The required methods for a complete `ITreeContentProvider` are listed below:

```
public void inputChanged(Viewer v, Object oldInput, Object newInput);
public void dispose();
public Object[] getElements(Object parent);
public Object getParent(Object child);
public Object [] getChildren(Object parent);
public boolean hasChildren(Object parent);
```

5.4. Label Providers

5.4.1. The Generic LabelProvider

The label provider is used to show the text and the image associated with a given object element. Because of how simple the idea is, and also the implementation, it only requires two methods: `getText` and `getImage`, both taking an object to evaluate.

NOTE: If you use a shared image in the eclipse platform, you do not need to create or destroy it. If you use your own image, you are in charge of creating it AND destroying it. One good practice is to create shared images for your entire plug-in during it's initialization, and then destroy those images when your plug-in shuts down. This way you aren't creating and destroying images needlessly.

5.4.2. ITableLabelProvider

This label provider is different only in that the table gives to each element an entire row, and so given some object, it can have multiple columns, each of which can have a text representation and an associated image.

5.5. Context Menus

Context menus for your own view are a lot easier to add, and can be added or removed, set enabled or disabled, over a much wider range of criteria then those through the `plugin.xml` file.

5.5.1. Creating a (mostly) static Context Menu

If we look at the `SWTView`, we see that we register our menu during when our control is created in the `createPartControl` method call. We declare our actions right there, and register them with our menu manager. The menuManager can still be changed programatically, but that usually means the menu intends to stay relatively constant regardless of what items are selected.

The menu manager is actually pretty robust. You can mark it as dirty, add or remove actions from it, as well as other things. In the SWTView example, we've specifically noted not to clear itself before showing each time. To clear

before showing every time would be characteristic of a menu that is very sensitive to what elements are selected, as we'll see in the other views. For the SWTView, we've created an unchanging menu that is not context sensitive at all.

```
// creating the Menu and registering it
MenuManager menuMgr = new MenuManager("#PopupMenu");
menuMgr.setRemoveAllWhenShown(false);
Menu menu = menuMgr.createContextMenu(viewer.getControl());
viewer.getControl().setMenu(menu);
getSite().registerContextMenu(menuMgr, viewer);

// create our action
Action act1 = new Action() {
        public void run() {
                showMessage("Action1");
        }
} ;
act1.setText("SWT Action 1");

// add it
menuMgr.add(act1) ;

// required, for extensions
menuMgr.add(new Separator(IWorkbenchActionConstants.MB_ADDITIONS));
```

5.5.2. A dynamic Context Menu

Looking at our `TableView` implementation, the first thing we do to start off the menu process is create our actions. We give them titles, descriptions, tooltips, and run methods. The next step is to create the menu manager and hook the actions into it.

The differing code is seen below:

```
menuMgr.setRemoveAllWhenShown(true);
menuMgr.addMenuListener(new IMenuListener() {
        public void menuAboutToShow(IMenuManager manager) {
                TableView.this.fillContextMenu(manager);
        }
});
```

Here, we declare that we want to clear out the menu every time it's about to be shown. Then we add a listener, where we can add actions to the menu in a much more fine-tuned manner.

If we go down to the `fillContextMenu` method, we get the selection from the viewer. I know it will return an `IStructuredSelection`, but I've decided to add no actions if more than one element is selected. I add one action regardless of what element is selected, but if a String is selected, I add the second action as well.

So when we run this example and open the List / Table View, if we right click on the integer element, we should only see one action, and if we right-click on one of the other three, we should see two. Whether the object is a String or an Integer goes back to the Content Providers, and what they have given us as their list of children. If we look at the `TableView`'s `ViewContentProvider` inner class and its `getElements` method, the code returns a list with 3 Strings and one Integer.

```
public Object[] getElements(Object parent) {
        ArrayList list = new ArrayList();
        list.addAll(Arrays.asList(new String[] {"One", "Two", "Three"}));
        list.add(new Integer(5));
        return list.toArray();
}
```

5.5.3. Drilldown Adapters

Drilldown Adapters are a feature available for TreeViewers to help navigate through the data, instead of having to deal with ever expanding trees. In this way you can essentially zoom in to view just one part of the tree, i.e., drill down. Only two lines of code were added throughout the initialization of the viewer to invoke this feature in the context menu. Only one line was needed to add them to the view's toolbar. They are shown below

```
// Create the Drill Down Adapter
drillDownAdapter = new DrillDownAdapter(viewer);

// Add the options to the context menu (IMenuManager)
drillDownAdapter.addNavigationActions(manager);

// Add the options to the view's toolbar (IToolBarManager)
drillDownAdapter.addNavigationActions(manager);
```

Without drilldown adapters, your tree's content provider could probably get away without completing the `getParent` method. If you do use drilldown adapters, however, this method will need to be completed so that the model can navigate both up and down the trees.

5.6. Conclusion

Other features are able to be added to views as well. Viewers can have doubleclick actions, and I didn't get into the specifics of adding the toolbar icons, but the code is all very short, concise, and in the examples. The ease with which one can make a view should help make adding them to your plug-in as painless as possible.

One thing of note to anyone using a view to add, remove, or change data to some underlying model. By convention, any changes made to data in a view (such as right clicking on an element and selecting a delete action) should be saved to the model right away, and to its underlying file resource if one exists. Editors, on the other hand, follow the open, save, store, model, where changes must be specifically saved by the user. Views should not behave in this manner. All changes should be live as soon as they are executed.

Finally, there will be many other code snippets for views in future examples, and occasional tips about specific quirks found when using them.

6

Editors

6.1. Introduction

Making your own editor involves extending current editor classes that are available to you, like `TextEditor`. The project named `Example5aCustomEditor` is the eclipse template generated for you, with absolutely no modifications at all by me to mark up or elaborate on the topic. It primarily parses and then colors xml documents.

`Example5bMultiPageEditor` is a two-page editor that has a text-style editor on one page, and a view on the other to represent that data graphically. While crude in its implementation, it serves pretty well to demonstrate what goes into making a multi-page editor.

6.2. Custom Editor

The basic TextEditor is a large text field centered inside your workbench, and provides columns for line numbers and markers on the sides. Because of this, most editors you produce should probably inherit from TextEditor. The steps to creating your editor are: declaring your dependencies, declaring your extension, and finally, making the editor.

In order to provide a custom editor, you will need to depend on `org.eclipse.jface.text`, `org.eclipse.ui.editors`, and `org.eclipse.ui.workbench.texteditor`. Once you have these dependencies, we're ready to declare our extension point.

To create our extension, we add `org.eclipse.ui.editors` as an extension point, right-click on it once added and select `new->editor`. Our new editor needs an id, a name, preferably some extension denoting what types of files we can be used to edit, a class for the editor, and a contributor class, which manages the context menus for editors. (You are, of course, welcome to create your own contributer after studying some of the generalized ones, but that is an exercise left up to the reader, should they care to do so. My example has used a common contributor class.)

The astute reader who is actively browsing the project code now will notice that the actual editor class itself is rather tiny. For those who aren't reading along, the editor class looks as follows:

```
public class XMLEditor extends TextEditor {

	private ColorManager colorManager;

	public XMLEditor() {
		super();
		colorManager = new ColorManager();
		setSourceViewerConfiguration(new XMLConfiguration(colorManager));
		setDocumentProvider(new XMLDocumentProvider());
	}
	public void dispose() {
		colorManager.dispose();
```

```
                super.dispose();
        }

}
```

So apparently all this editor does is change our source view configuration and our document provider. Unfortunately, I don't know much about either of these classes, so our only clues as to what they do are in the javadocs for their respective classes.

By pressing `ctrl` and mousing over these classes, then clicking, we can browse to them and read their javadocs or explore their code. For Document Providers, we find is that:

```
/**
 * A document provider maps between domain elements and documents. A document provider has the
 * following responsibilities:
 * <ul>
 * <li>create an annotation model of a domain model element
 * <li>create and manage a textual representation, i.e., a document, of a domain model element
 * <li>create and save the content of domain model elements based on given documents
 * <li>update the documents this document provider manages for domain model elements to changes
 * directly applied to those domain model elements
 * <li>notify all element state listeners about changes directly applied to domain model
 * elements this document provider manages a document for, i.e. the document provider must
 * know which changes of a domain model element are to be interpreted as element moves, deletes, etc.
 * </ul>
 * Text editors use document providers to bridge the gap between their input elements and the
 * documents they work on. A single document provider may be shared between multiple editors;
 * the methods take the editors' input elements as a parameter.
 * <p>
 * This interface may be implemented by clients; or subclass the standard abstract base class
 * <code>AbstractDocumentProvider</code>.
```

For Source Viewer Configurations, we find:

```
/**
 * This class bundles the configuration space of a source viewer. Instances of
 * this class are passed to the configure method of
 * <code>ISourceViewer</code>.
 * Each method in this class get as argument the source viewer for which it
 * should provide a particular configuration setting such as a presentation
 * reconciler. Based on its specific knowledge about the returned object, the
 * configuration might share such objects or compute them according to some
 * rules.
```

To test how this editor looks, run the plug-in, and in the runtime workspace, create a new file (`file->new->file`) with a `.xml` extension and open it. It should open by default into an editor of our example type, and as you type in some xml, it should color it appropriately.

Unfortunately, that's all the information I can really provide on overriding a simple editor. As we just saw, it mostly consists of declaring your dependencies, declaring your extension, and following through by creating an editor that overrides some default functionality. The rest really involves using the classes provided as basepoints from which to branch off and do your own thing as necessary.

6.3. Multi-Page Editor

Multi-page editors are most often represented as some group of models that represent or display data from a raw or source text file. In this example, I've simply used our simple editor from the last example as a base, and added a second page that displays all top level elements under the <xml> tag.

Editors follow the open, save, close methodology, so any changes you make to some editor data via its gui interface should change it's associated text editor immediately, but should NOT save the changes to the underlying resource until the user requests it to do so.

A multi-page editor should extend the `MultiPageEditorPart` superclass. Our example will have two pages, one with our xml editor, the other with a viewer. I've declared two private members for these elements.

The only method we MUST implement from the abstract superclass is the `createPages` method. Everything else is done for us, though we're always welcome to override what we need to. In this example, we don't override much at all.

We begin by creating each of our two pages, and then changing the title for the editor, which is simply a cosmetic change. Because our editor page is the simplest to look at, I'll use it to demonstrate how to create a new page.

```
editor = new XMLEditor();
int index = addPage(editor, getEditorInput());
setPageText(index, "Source");
```

That's about all that's involved. If your editor is going to use a simple regular text editor, you can set editor to a new `TextEditor` object instead. The next step is to then create our other page, the viewer page. The code for our viewer page could actually be just as short:

```
viewer = new TableViewer(getContainer(), SWT.NONE);
int index = addPage(viewer.getControl());
setPageText(index, "Viewer");
```

The next thing we need to do here is set up the data model for the viewer. Remember, we want the viewer to match the text document at every conceivable point, and since users can only look at one page in an editor at a time, a good time to update the viewer is when the user switches tabs.

```
protected void pageChange(int newPage) {
        if( newPage == 0 ) {
                // make it refresh this
                viewer.setInput(editor);
        }
}
```

Next, our viewer needs a content provider and a label provider. How do we want our view to display the xml data at hand? In this example, I just made it display the top level elements. The only really interesting code here that you haven't seen is how to get the text from your editor.

```
```

```
                String editorText = editor.getDocumentProvider().
                getDocument(editor.getEditorInput()).get();
```

Now this should work well to display the editor's data to the user, but there's very little point to having a multi-paged editor with a gui interface unless you're going to provide actions for the user to make it easier to modify the file. Because of this we've added our menu managers and a context menu with a `REMOVE` action. And, of course, we need to change the text in the editor when we switch back to it.

```
                editor.getDocumentProvider().getDocument(editor.getEditorInput()).set(newXML);
                viewer.refresh();
```

Good luck making your tabbed text-editor

7

Tracking Changes to Resources and Java Elements

Tracking changes in resoures is easy to implement but could be indespensible for your plug-in. The code that I need to relate is short, but I figured it should be included here to be a more complete guide to common tasks for eclipse plug-ins.

I've implemented a view who's input is a model that listens for changes in resources as well as changes in java elements, such as classes, compilation units, packages, methods, fields, etc. It then adds these change events to its model, alerts the view, and displays them in the view. This example provides another example of a content provider for a view, and a label provider.

7.1. Dependencies

As you can see by browsing the `MANIFEST.MF`, I've included the following plug-in dependencies:

```
Require-Bundle: org.eclipse.ui,
 org.eclipse.core.runtime,
 org.eclipse.core.resources,
 org.eclipse.jdt,
 org.eclipse.jdt.core,
 org.eclipse.jdt.ui,
 org.eclipse.ui.intro
```

The JDT packages are required to do any sort of processing regarding methods, fields, compilation units, and others. The resource package is required to become a listener for changes in resources.

7.2. Project Structure

For this example, I've made a model class, `Example6Model`. It has an inner type `ModelRoot` which keeps track of two arraylists, one containing change events for resources, the other containing change events for elements. I've set the `Example6Model` up as a listener for both types of events.

I also added an interface called `MyChangeListener`, which the model will fire events to after receiving either type of event. The view has been registered as a listener to the model.

So the general flow of execution is that a resource or java element will be changed and fire events to our model who is a listener for them. Our model will update itself by adding this new event, then alert the view that the model has changed and the viewer should refresh itself.

7.3. Resource Changes

Resources are only marked as changed, and thus the events fired, when the resource is saved or resynced with the underlying file representation. Simply typing in the editor will NOT triger a resource changed event to fire.

The following code is necessary:

```
/*
 * To register as an IResourceChangeListener:
 */
IWorkspace workspace = ResourcesPlugin.getWorkspace();
workspace.addResourceChangeListener(this);


/*
 * The interface must be implemented
 */
public void resourceChanged(IResourceChangeEvent event) {
        root.addResourceChange(event);

        /*
         * Tell the view, and other listeners,
         * the model has changed
         */
        for( Iterator i = listeners.iterator(); i.hasNext();) {
                ((MyChangeListener)i.next()).fireMyChangeEvent();
        }
}
```

The IResourceChangeEvent object and it's change delta are only valid for the duration of the `resourceChanged` call. This call is made to a valid and registered listener when the platform notices changed resources.

Because in my implementation, the model is the listener, when the model receives an event, it adds it the event to one of it's arraylists and alerts the view to refresh itself because the model has changed.

7.4. Element Changes

Element changes are fired as soon as a delta between the current file buffer and the most recent file buffer can be calculated. This is usually within a few seconds of when you stop typing. As soon as the delta is calculated, it is fired off to all available listeners.

JBoss AOP IDE makes use of these in calculating typedef matching, advisors, and other things that are dependent upon the exact text of a java source file. Changing a method name from "someMethod" to "newMethod", will fire off one event with two changed elements burried inside, one denoting the removal of "someMethod", and one denoting the addition of "newMethod".

The program code necessary to become an elementChangeListener is as follows:

```
/* Register yourself as a listener */
JavaCore.addElementChangedListener(this);
```

```
		/* fulfill the IElementChangedListener interface */
		public void elementChanged(ElementChangedEvent event) {

			root.addElementChange(event);

			/*
			 * Tell the view, and other listeners,
			 * the model has changed
			 */
			for( Iterator i = listeners.iterator(); i.hasNext();) {
				System.out.println("firing");
				((MyChangeListener)i.next()).fireMyChangeEvent();
			}
		}
```

From here, the tree viewer should do all of the work.

7.5. View Notes

During the `ElementChanged` method call, the view and the viewer will not be updated, even if you specifically call `viewer.refresh()` inside the view's class. The reason for this is that in eclipse, SWT is modeled as one single thread, and these events must be fired while other threads are modifying the gui.

In order to force the viewer to update, you must do it via an `asyncExec` call on your display, which is a member of the SWT thread. Seen below:

```
	// Called from our model when *IT* receives a real event
	public void fireMyChangeEvent() {
		// event fired
		System.out.println("fired");
		final Example6Model model = Example6ResourceTrackingPlugin.getDefault().getModel();

	if (!viewer.getControl().isDisposed())
	{
		Display display = viewer.getControl().getDisplay();

		if (!display.isDisposed()) {
		    display.asyncExec(new Runnable() {
			public void run() {
			    //make sure the tree still exists
			    if (viewer != null && viewer.getControl().isDisposed())
				return;
				    viewer.refresh();
			}
		    });
		}
		}

	}
```

The asyncExec call makes sure that your viewer's refresh occurs as soon as possible without any side effects. There are no events fired after such a gui update is complete to alert you that the viewer's update is complete.

7.6. Some JDT examples

The ViewContentProvider internal class for our SampleView manages how to display the content, what objects have children, and other things for our model. Because the entry point is our model, it's first children would be the input object, the model. From there, we declare that it's 2 children are strings that represent which collection we plan to display.

So the next time it tries to check for children, we return which ever collection it seeks, a collection of either `ElementChangedEvents` or `ResourceChangeEvents`, which contain deltas and then eventually elements that have been changed.

The `ElementChangedEvents` have `IJavaElementDeltas`, which correspond to a change for one java element (method, field, package fragment, type, etc).

Because this example focuses on deltas, we don't get into a lot of the JDT possibilities, but it works just as any heirarchy you'd expect for java classes. Given some IType (class type), you can get a list of member variables, methods, inner types, its package, and other things.

One last thing I'd like to note in our example is the use of the `JavaUILabelProvider` to assist in providing images and text if we had wanted it to. This is what has allowed members, types, folders, and others, to show icons in the SampleView's tree. This class will be very helpful if you want to display icons and information about some JDT elements that you'll be modeling, and it's listed here so that you may do so.

8

Preferences

Every plug-in has a preference-store. It can be used practically for anything, however it does not take objects that are not serialized in some way. The preference store can only hold low-level data types, such as numbers or Strings, and not generic objects.

In this example, I've created a simple action that just tests to see if some preference is set or is null. I also create a preference page that can be accessed via the `Window->Preferences->` menu items. The preference page uses any number of SWT controlls you desire, so long as you keep control of saving then as preferences when the user is done and clicks `apply` or `ok`.

8.1. Preference Stores

Preference Stores demand Strings as keys for all preferences. They can store a default value for any preference, as well as a current value. The values can be of type boolean, double, float, int, long, or String. This data is usually held in the .metadata directory of the current workspace.

According to the javadoc:

```
 * A property change event is reported whenever a preferences current
 * value actually changes (whether through setValue,
 * setToDefault, or other unspecified means). Note, however,
 * that manipulating default values (with setDefault)
 * does not cause such events to be reported.
```

8.1.1. API

The IPreferenceStore interface contains the following groups of methods. The character sequence `TYPE` will denote any of the words `boolean, int, double, float, long, String`.

1. contains(String key)
2. getDefaultTYPE(String key)
3. getTYPE(String key)
4. setDefault(String key, TYPE value)
5. setValue(String key, TYPE value)
6. setToDefault(String key)

7. isDefault(String key

8. putValue(String key, String value) [note: Does not fire a property change event.]

9. [add|remove]PropertyChangeListener(IPropertyChangeListener)

8.2. Our Action

We'll start just by looking at our action. It's an extremely simple action, and the only completed method is the following one:

```
public void run(IAction action) {
        // Here we'll just get our preference store and see if it
        // contains a preference by that name.
        IPreferenceStore store = Example7PreferenceStorePlugin
                .getDefault().getPreferenceStore();
        boolean contains = store.contains(PreferencePage1.TARGET_PREFIX + "0");
        System.out.println("Does store contain one target? " + contains );
        if( contains ) {
                String s = store.getString(PreferencePage1.TARGET_PREFIX + "0");
                PreferencePage1.NameUriPair pair = new PreferencePage1.NameUriPair(s);
                System.out.println("name: " + pair.getName() + ", uri: " + pair.getUri());
        }
}
```

Basically, we just get the string from the preferencestore, if it exists, and make a NameUriPair out of it. Then just display the pair's parts. If the preference has been set in that workspace and is currently in the preference store, it will display it.

8.3. The Extension Point

The extension point to extend is `org.eclipse.ui.preferencePages` and only takes a `page` element underneath it. The `page` element only requires a `name, id,` and `class`. So it's a pretty simple extension point to use. From there, the rest of the preference page is handled by your preference page class file and it's interaction with the plug-in's preference store.

```
   <extension
         id="Example7PreferenceStore.preferencePage"
         name="Example7PreferenceStore.preferencePage"
         point="org.eclipse.ui.preferencePages">
      <page
            class="example7PreferenceStore.preferences.PreferencePage1"
            id="Example7PreferenceStore.page1"
            name="Example7PreferenceStore.page1"/>
   </extension>
```

8.4. The PreferencePage

The preference page that I created extends the `PreferencePage` class, which an abstract class containing only one

abstract method: `abstract Control createContents(Composite parent);` You will also need to override the `performOk` method, which is called from the superclass whenever either the `apply` or the `ok` buttons are pressed. This is the method that should save the preferences selected by the user in the preference store.

Our `createContents` method should return a composite that we want to go in the main content area of the preference page. Ideally, you should create any widgets using `parent` as the parent control, and then at the end of the method, return your main composite to the superclass.

The superclass uses a grid-layout, but if you return your new composite, it will make sure it fills the entire area in the middle of the preference page, both horizontally and vertically. If you don't return your composite, any controls you add to the parent control will be added in the superclass's gridlayout fashion. In this example, I've created one form layout composite, with sub-elements, and returned it to the superclass.

For loading the preferences, I just take each preference beginning with a prefix and ending with an integer, until I find a null element. Then I insert each of those into their respective tables in the gui. For saving them, I do the same but serialize them.

The file chooser list (top list) stores everything as a String, so no serialization is necessary to put it in the preference store. The Target list (bottom) stores elements as a NameUriPair object, and they must be serialized with a newline as the separator to be inserted into the preference store.

8.5. Testing

To test the example, run the plug-in in a runtime environment. From there, right click on any file in any prokect in the runtime environment. (If one does not exist, you can create one. For my test, I created a file named test.file). Right click on this element to see it's context menu and select our action named Our Test Action.

If we look at our console in our development instance of eclipse (not the runtime environment), we'll see that the preference is not set or does not exist. If we then go to `Window->Preferences` and select the tab for our plug-in's preferences, and fill in each table with some different data, apply the changes and click ok, we can try the action again.

Trying the action again, if we added a target element to the bottom list in the preferences, we should see that the preference is indeed set now.

8.6. Notes

The rest of the example just includes plenty of SWT and JFace controlls and examples of laying them out.