

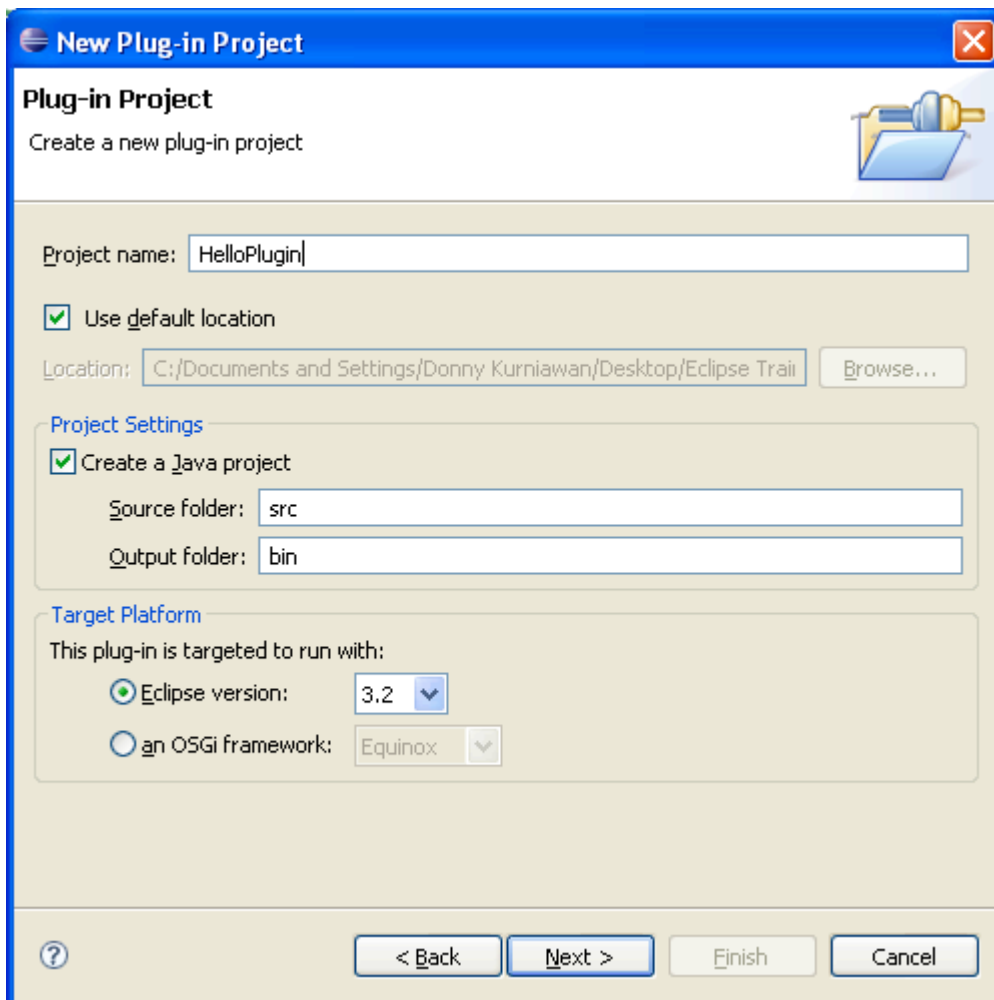
## Tutorial 06: A Simple Plug-in Example

Contents:

1. Creating a plug-in project.
2. Running an Eclipse application.
3. Inspecting the plug-in code.
4. Deploying the plug-in.

### Creating a plug-in project


1. Switch to the Plug-in Development perspective.
2. Click File > New > Project...
3. Select Plug-in Development > Plug-in Project.
4. Name the new project, HelloPlugin.



5. Enter the following properties, and click Next.

**New Plug-in Project** ✕

### Plug-in Content

Enter the data required to generate the plug-in. 

**Plug-in Properties**

Plug-in ID:

Plug-in Version:

Plug-in Name:

Plug-in Provider:

Classpath:

**Plug-in Options**

Generate an activator, a Java class that controls the plug-in's life cycle

Activator:

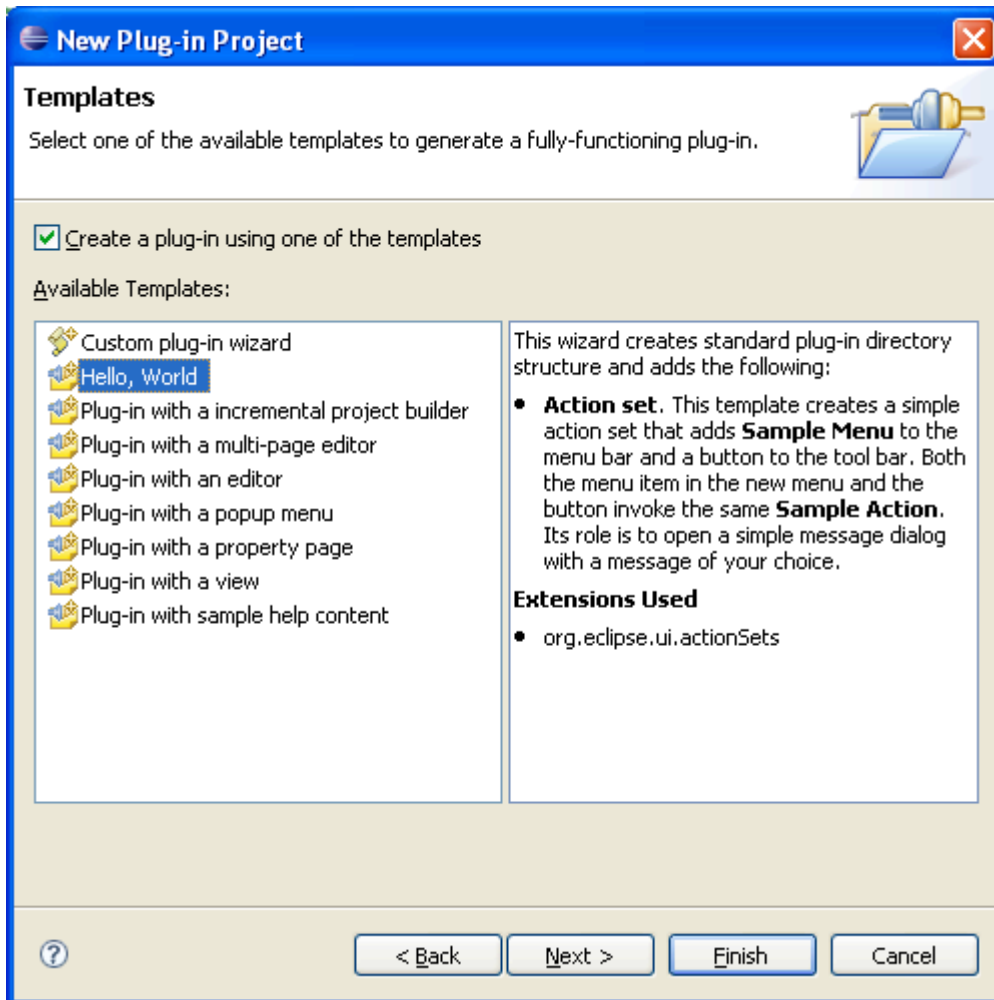
This plug-in will make contributions to the UI

**Rich Client Application**

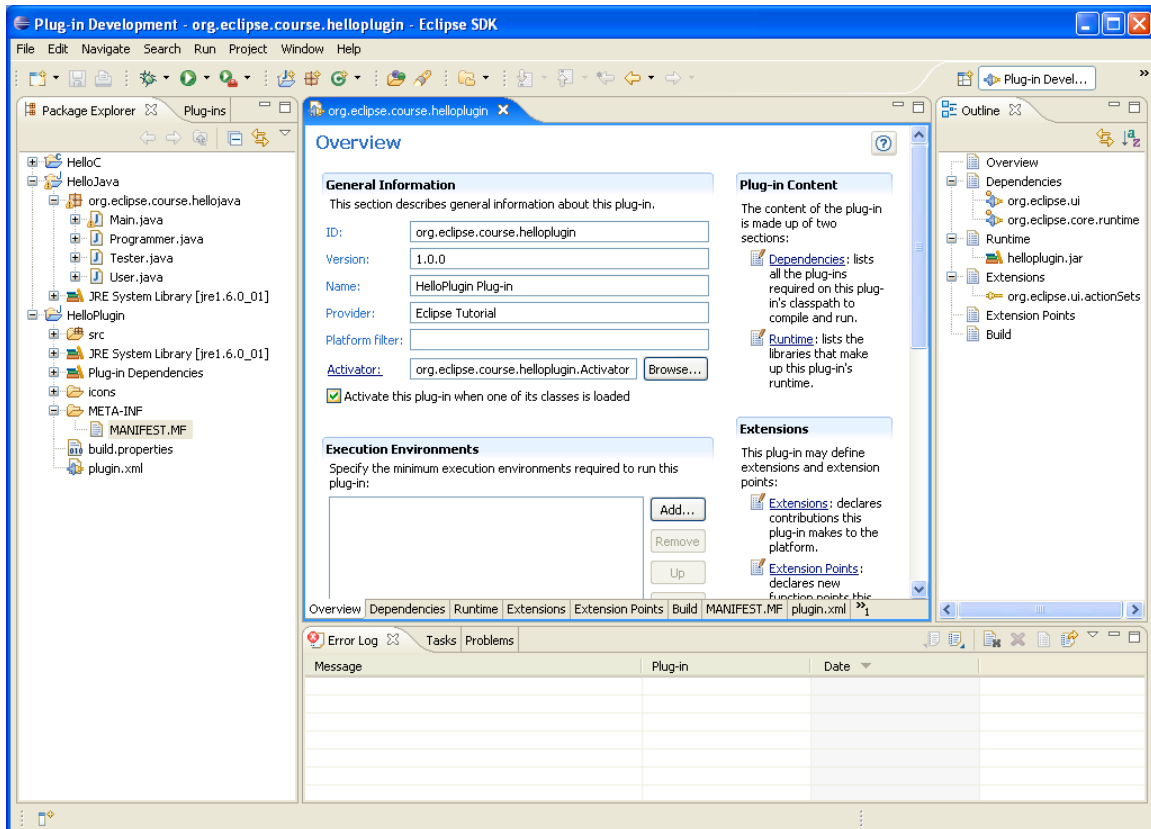
Would you like to create a rich client application?  Yes  No

?

6. Select “Hello, World”, and click Next.



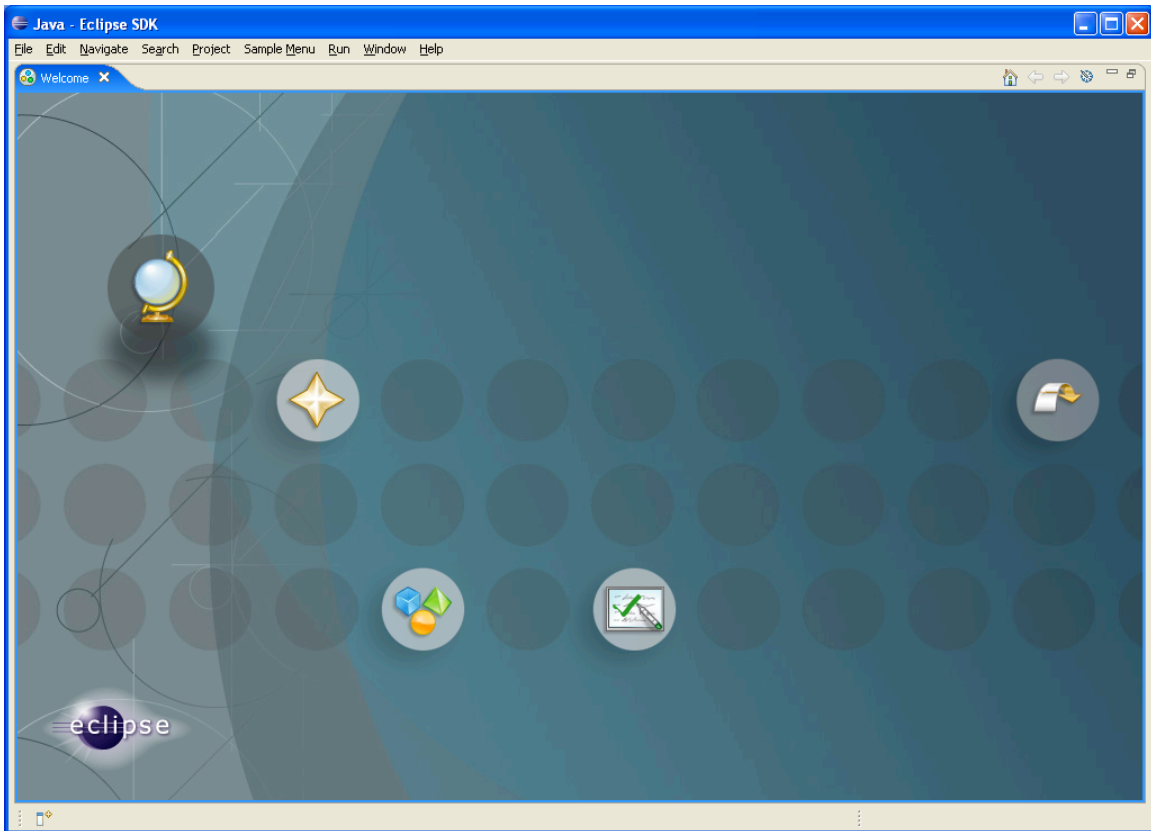
7. Accept the default options and click Finish.



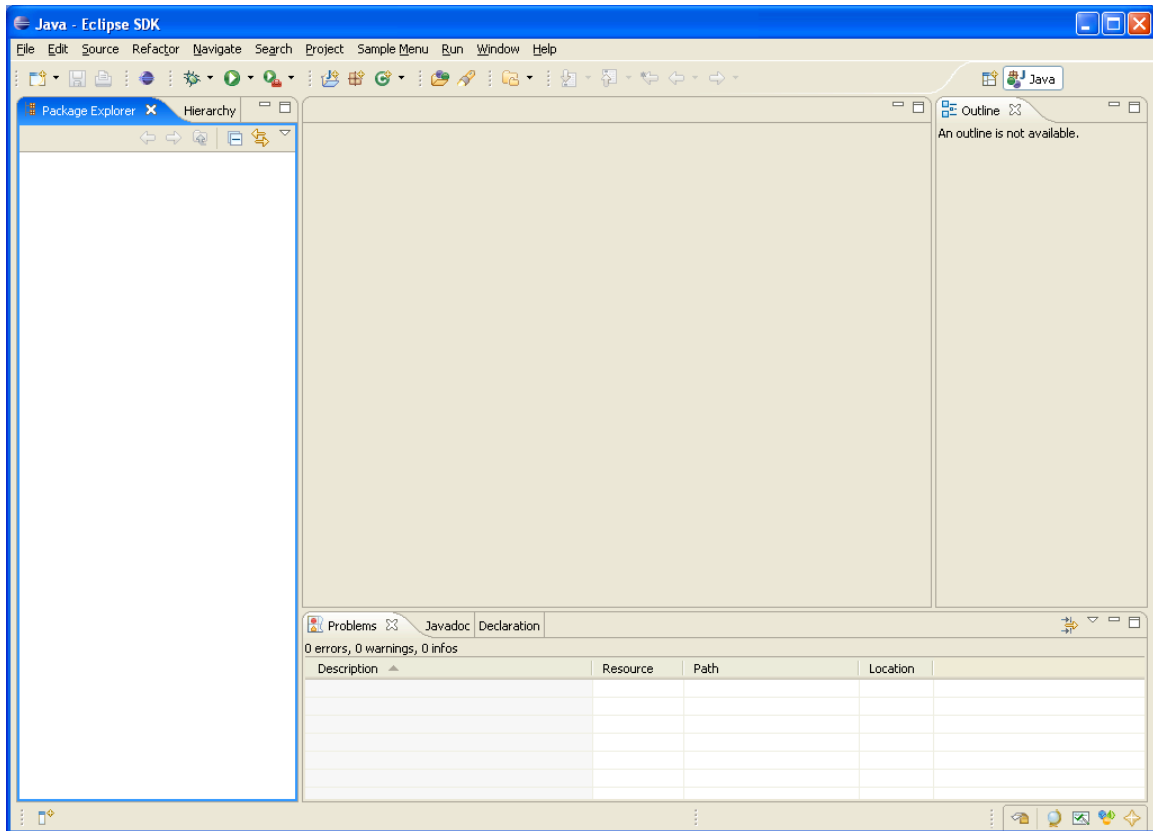
## Running an Eclipse application

1. To test the plug-in, select and right-click HelloPlugin in the Package Explorer view, and select Run As > Eclipse Application.

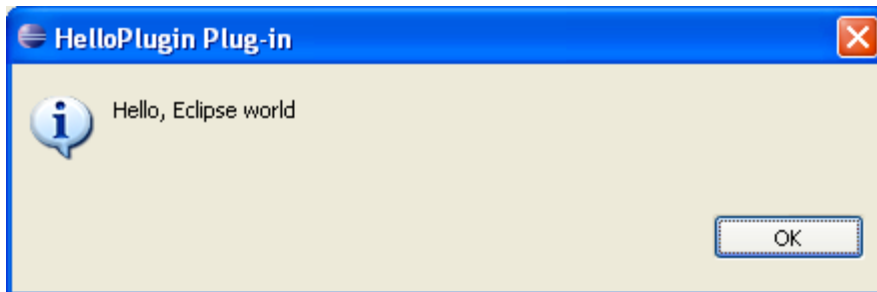
Eclipse is able to run another instance of Eclipse within itself (Eclipse Application). It simplifies the development of Eclipse plug-ins. It allows us to test the plug-ins immediately.



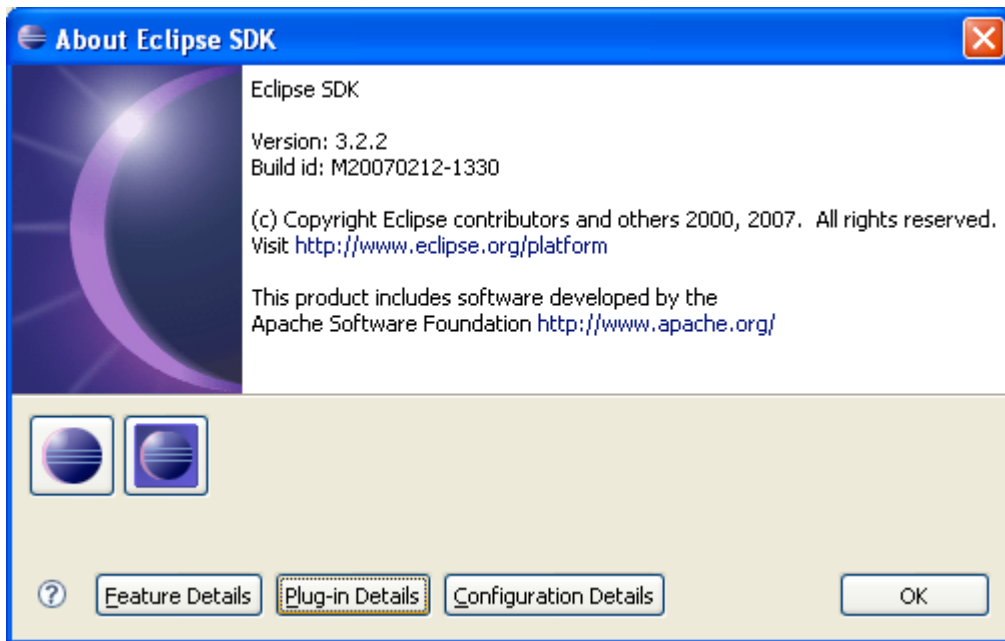
2. Click the arrow on the right to go to the workspace.



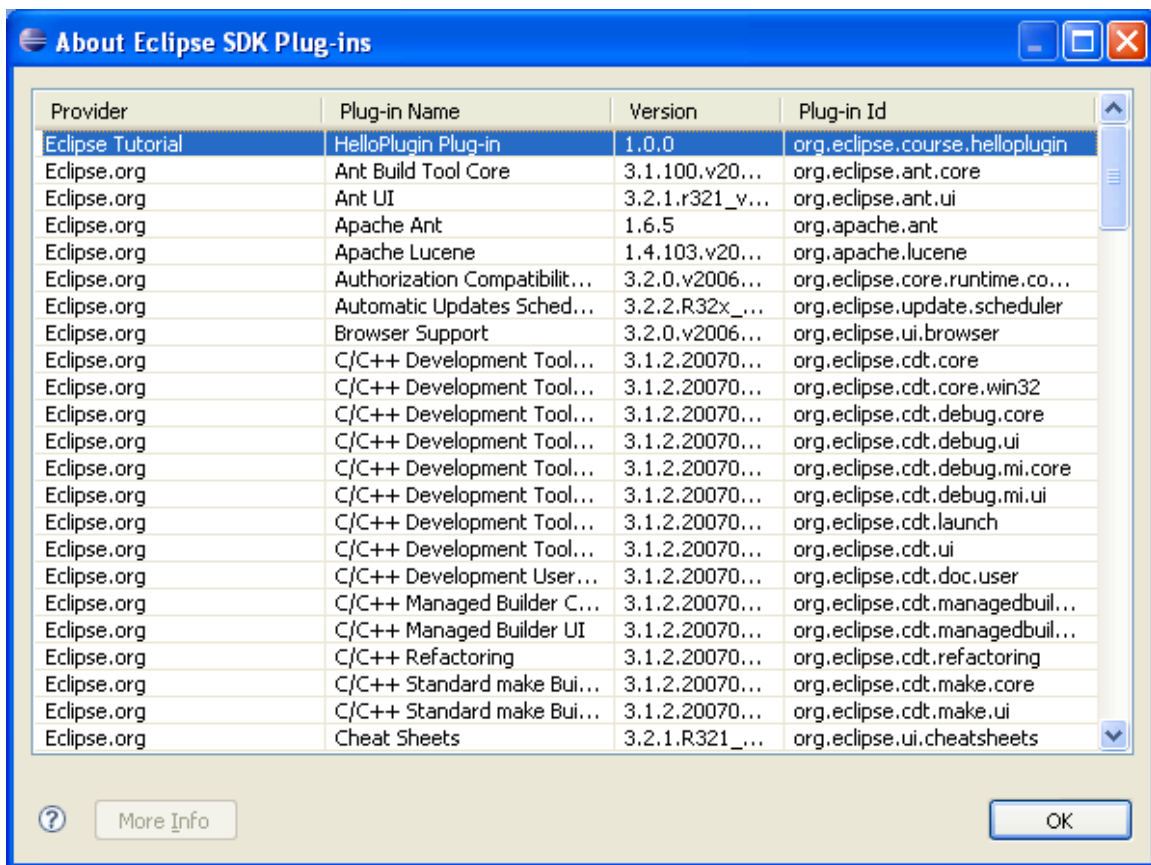
3. There is an Eclipse icon on the toolbar (and a new menu, Sample Menu). Click the icon.



4. To check the plug-in information, click Help > About Eclipse SDK.



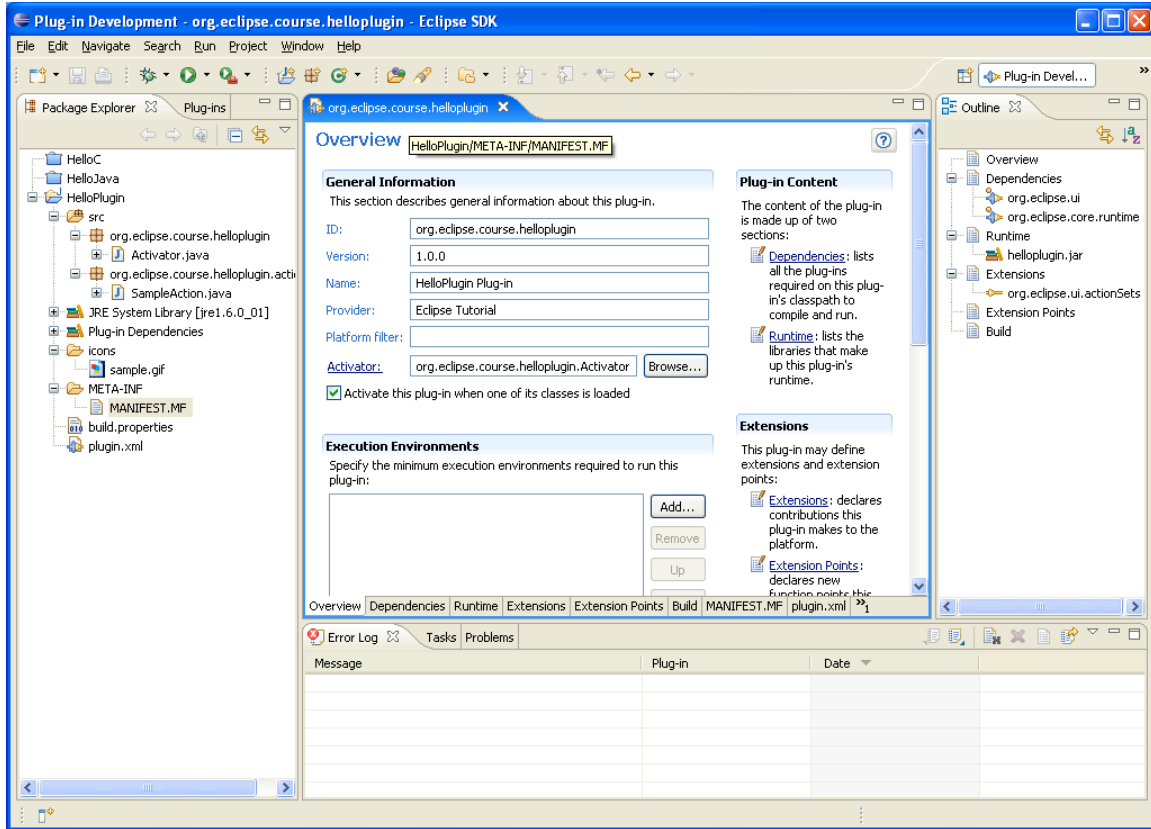
5. Click the Plug-in Details.



6. Notice that our plug-in has been loaded as one of the Eclipse Application plug-ins.

## Inspecting the plug-in code

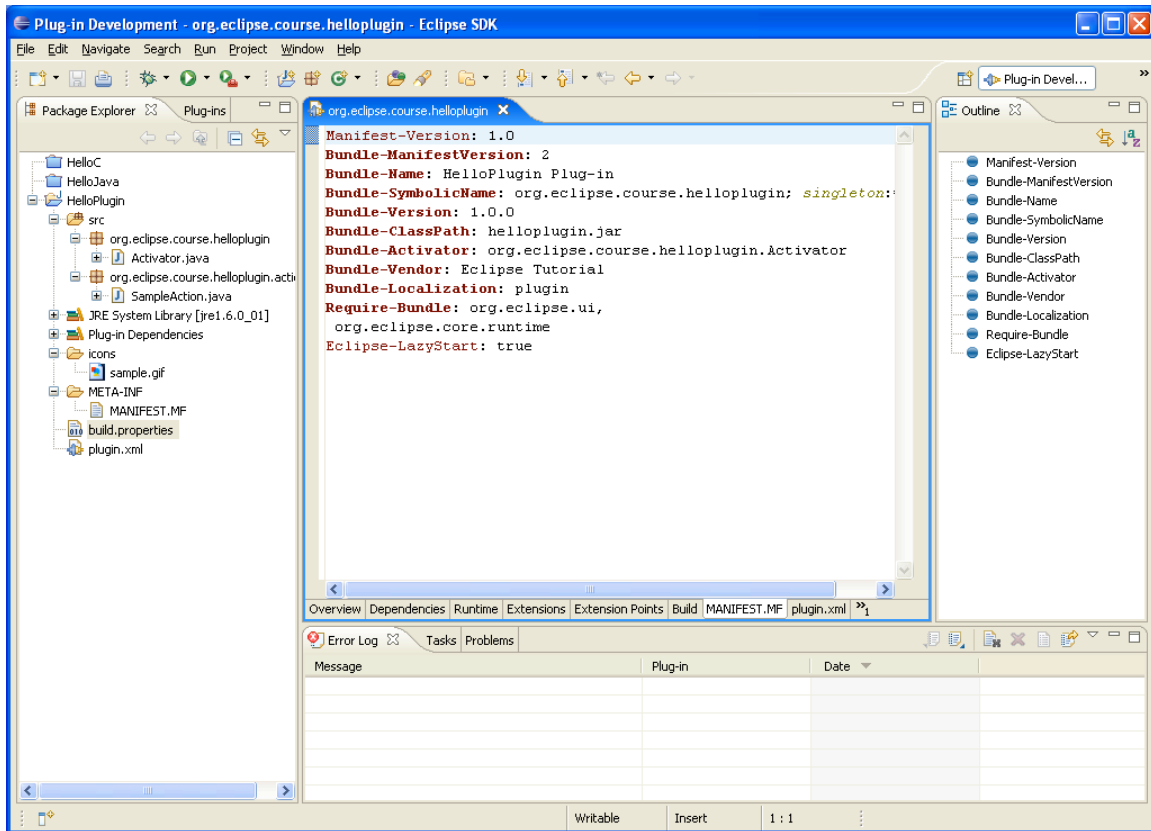
1. Go back to the Eclipse IDE.
2. Open up the plug-in manifest editor by double-clicking META-INF/MANIFEST.MF or plugin.xml.



Plug-in information is stored in two files: META-INF/MANIFEST.MF and plugin.xml, which define how this plug-in relates to all the others in the system. The build.properties file describes the build information for the plug-in.

3. Click the MANIFEST.MF tab to display the source of the META-INF/MANIFEST.MF.





The META-INF/MANIFEST.MF file defines the runtime aspects of this plug-in. The first two lines define it as an OSGi manifest file. Subsequent lines specify plug-in name, identifier, version, and classpath. All these aspects are editable using other pages in the plug-in manifest editor.

Eclipse employs a lazy-loading mechanism; it loads a plug-in when it is required. If you want to load the plug-in when Eclipse starts, change the Eclipse-LazyStart to false.

4. Click the plugin.xml tab to display the source of the plug-in.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>

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

</plugin>
```

The plugin.xml file defines the extension aspects of this plug-in. The first line declares this to be an XML file, while subsequent lines specify plug-in extensions. Making changes to any page other than the plugin.xml and MANIFEST.MF pages may cause the manifest editor to reformat the source. If you are particular about the formatting of either manifest file, then either use only the plugin.xml and MANIFEST.MF pages to perform editing or use another editor.

5. Double-click the file org.eclipse.course.helloplugin.Activator.java and examine its code. It is the class that controls the plug-in life cycle.

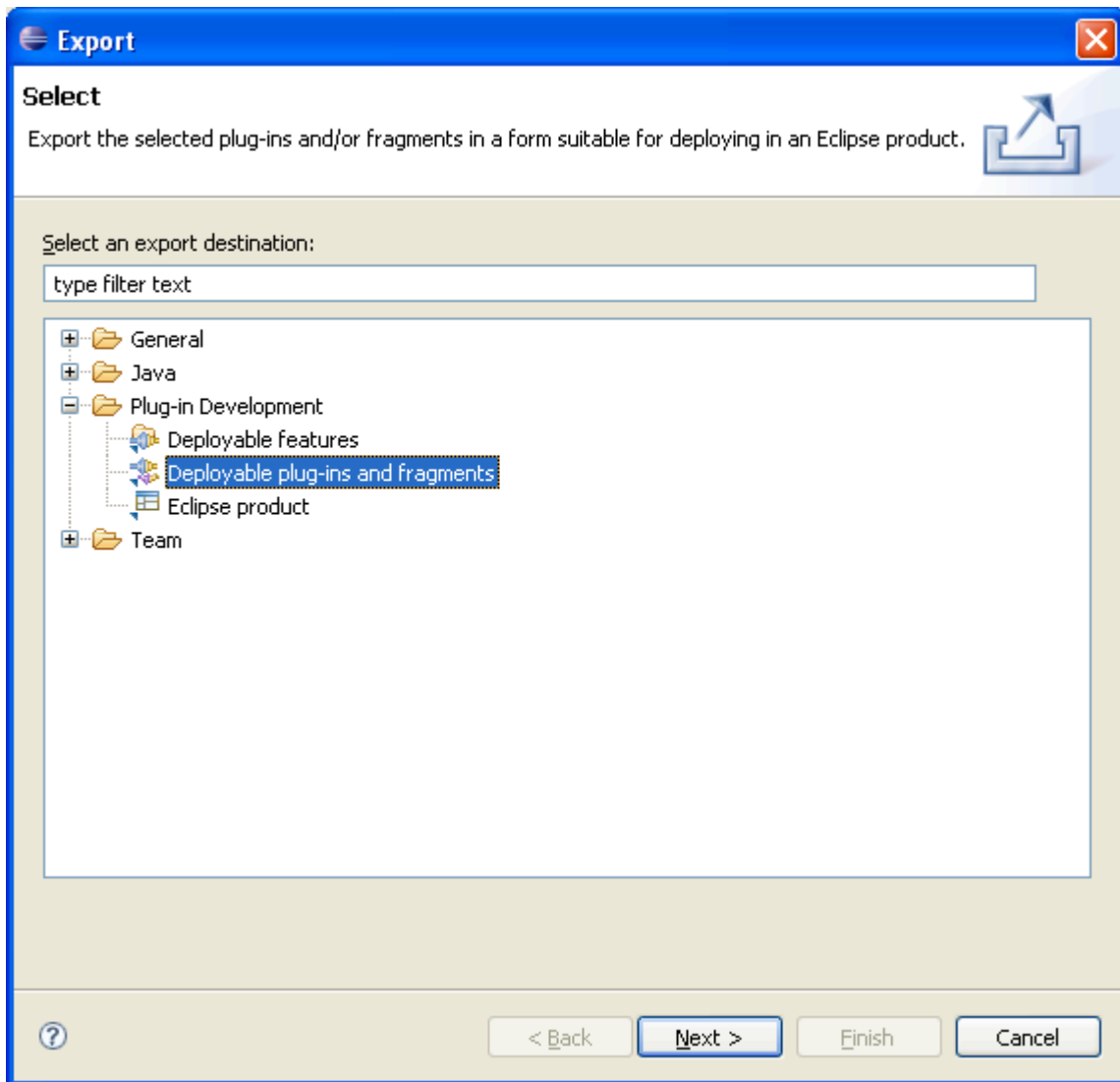
When the plug-in is activated, the Eclipse system instantiates the plug-in class before loading any other classes in it. This single plug-in class instance is used by the Eclipse system throughout the life of the plug-in and no other instance is created.

6. Double-click the file org.eclipse.course.helloplugin.actions.SampleAction.java and examine its code. It is the class that displays the dialog box.

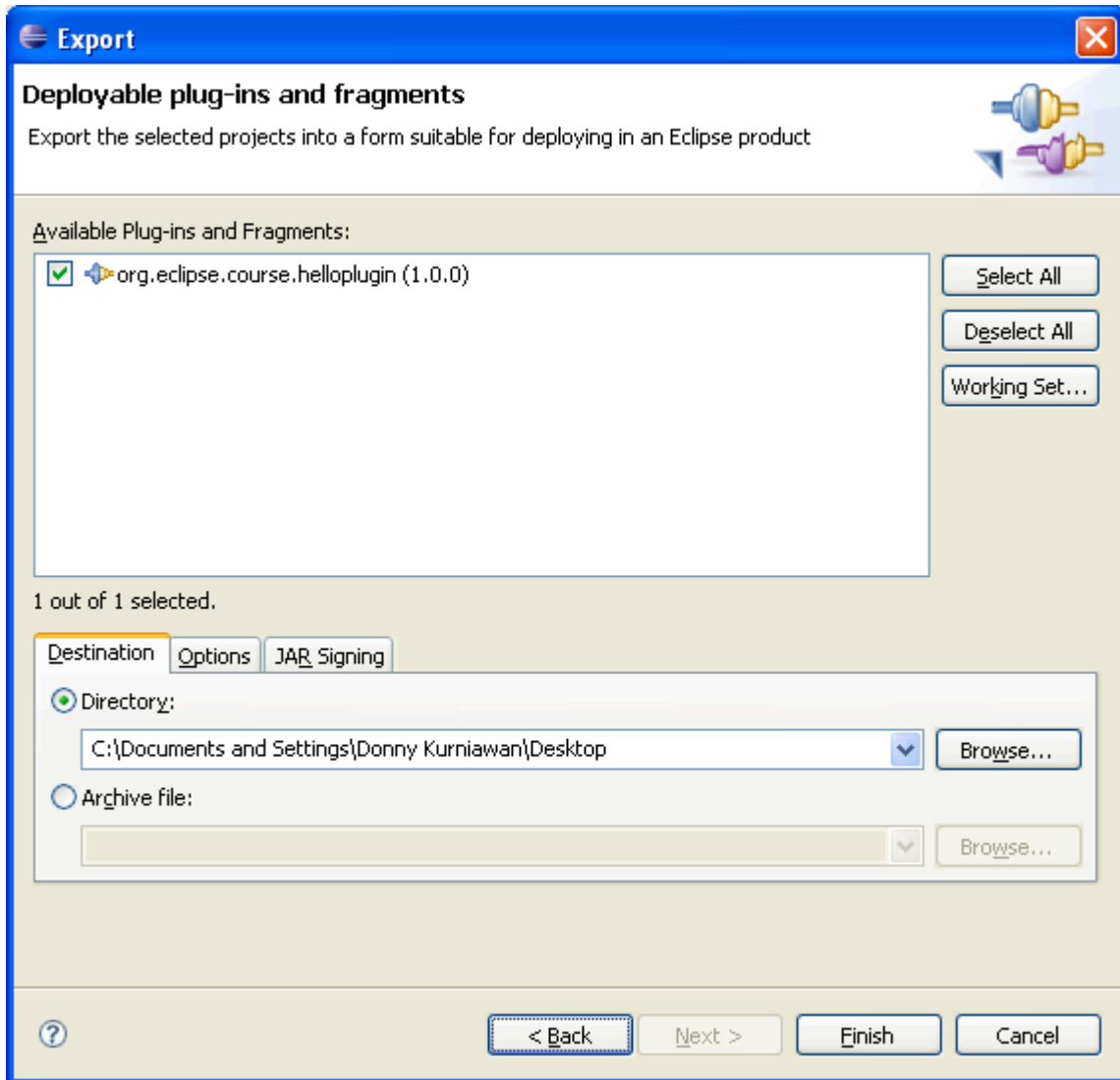
Take a look at plugin.xml. The HelloPlugin plug-in declares an extension to the org.eclipse.ui plug-in using the org.eclipse.ui.actionSets extension point by providing an additional menu (labeled Sample Menu) and an action (labeled Sample Action). The classname org.eclipse.course.helloplugin.actions.SampleAction is provided in plugin.xml.

## Deploying the plug-in

1. Select HelloPlugin in the Package Explorer view.
2. Click File > Export.
3. Select Plug-in Development > Deployable plug-ins and fragments.

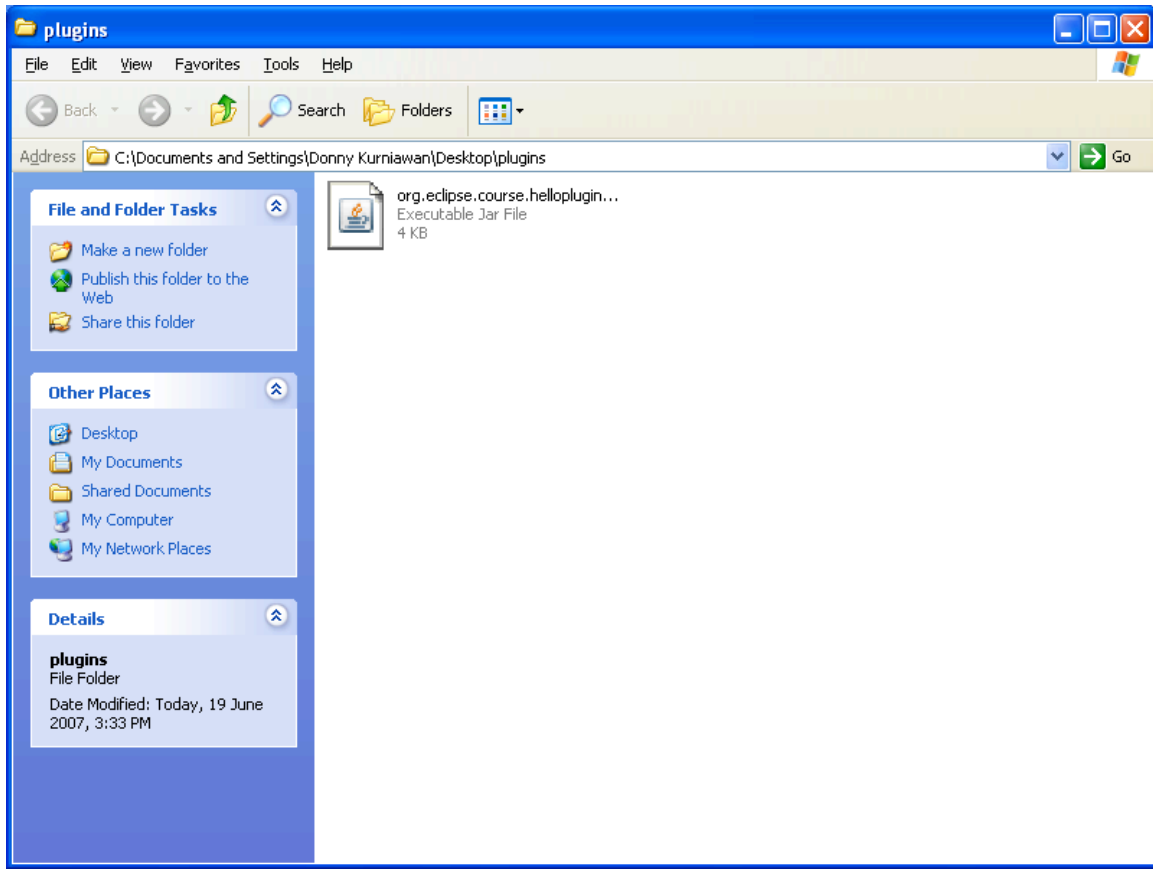


4. Accept the default options, and choose the destination directory.



5. Click Finish.

6. Open Windows Explorer, and move the jar file to 'Desktop\eclipse\plugins'.



7. Restart Eclipse.

8. Change the extension of the jar file, from '.jar' to '.zip' and uncompress the file.

