

# Developing for Eclipse/RCP/CSS

---

Kay Kasemir, Ph.D., ORNL/SNS

[kasemirk@ornl.gov](mailto:kasemirk@ornl.gov)

July 2011 at KEK

## Contents

Introduction .....	2
Definitions.....	2
Is developing for RCP/CSS complicated? .....	2
RCP: Complex, but sound.....	3
Sources of Information .....	4
Books .....	4
Preconditions .....	4
Create (Plug-in) Project .....	6
Java command-line “Hello” .....	7
Standalone GUI “Hello” .....	9
RCP “Hello” View.....	12
Connect to PVs .....	20
Hooking into menus .....	24
PV Context menu ‘Probe’ .....	26
Look at details in org.cstudio.util.pvscript.....	26
Summary.....	27

## Introduction

CSS is meant to be reasonably easy to understand for end users, with integration between tools that was impossible with legacy EPICS tools.

For developers, this requires additional work. While it is easy to create a one-of, standalone application, it is naturally harder to develop code that collaborates with other code. A properly developed RCP plugin for CSS is started on demand within CSS by the user, then maybe closed, then re-opened, while at the same time other CSS tools are opened and closed within the same instance of CSS. This requires each CSS plugin to de-allocate its resources when closed down. A plugin should persist its state so that it can re-open as it was left when CSS closed. Instead of hard-coded settings or maybe using environment variables, utilize the Eclipse preference settings and offer a preference page to end users. Exchange Process Variable names with other CSS tools, without actually knowing those other CSS tools while the application is being developed.

Please contact me at [kasemirk@ornl.gov](mailto:kasemirk@ornl.gov) with comments on this tutorial.

## Definitions

- Eclipse IDE  
Development environment for Java (also C++, JavaScript, Android, ...)
- Rich Client Platform, RCP  
Originally to implement the IDE, but can be used to build other applications
- Plugins  
Fundamental RCP building blocks
- CSS  
An RCP application, plugins for control system tools

## Is developing for RCP/CSS complicated?

### Yes!

Compare to building a clock.

Everybody should once build a simple clock.

Building your own clock from scratch is easier than interfacing a complex clockwork. In fact most children learned to read a clock that way.

End users, however, will prefer a clock that actually tells the time without first having to adjust the clock to the correct time. They may even need a clock that displays several time zones, can indicated upcoming appointment times, the phase of the moon, or other time-related information.



Simple Clock

[http://catalog.newtrendsonline.com/big\\_timetrade\\_12hour\\_demonstration\\_learning\\_clock-p-48566.html](http://catalog.newtrendsonline.com/big_timetrade_12hour_demonstration_learning_clock-p-48566.html)

Creating such a clock for end users is obviously more difficult.

Similarly, developing code for Eclipse RCP / CSS is more involved than writing a standalone Java program.

But your users will be able to tell the difference as well.



Internals of a clock that actually keeps track of the time

[http://homepage.mac.com/d\\_halgren/WatchMvt2.jpg](http://homepage.mac.com/d_halgren/WatchMvt2.jpg)



Astronomical clock, Prague

<http://morfis.wordpress.com/2011/01/12/architectural-timepieces/>

## RCP: Complex, but sound

Anybody who looked at software engineering from 1994 until today has probably read a copy of the book “Design Patterns: Elements of Reusable Object-Oriented Software” by Erich Gamma et. al., which is in its 38th printing in 2010.

Erich Gamma was an initial and long-time key developer for Eclipse/RCP. While Eclipse/RCP is certainly complex, it is based on many sound design decisions, so it may be well worth the time required to understand it.

## Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma  
Richard Helm  
Ralph Johnson  
John Vlissides



Foreword by Grady Booch

ADDITION-WESLEY PROFESSIONAL COMPUTING SERIES

## Sources of Information

This document walks through the steps of creating a simple plugin for Eclipse RCP and CSS. It is meant to give an example for the general idea, and offer a comparison of the difference in complexity when going from a command-line tool to a standalone graphical tool to finally an RCP plugin.

This tutorial cannot replace a more in-depth study. Suggested sources of information:

- Books: Look for 'RCP' books, not basic usage of the IDE
- IDE Help: Help Content, Platform Plug-in Developer Guide.  
**In the following, I refer to this as the IDE help.**
- CSS Book:  
<http://cs-studio.sourceforge.net/docbook>,  
[http://cs-studio.sourceforge.net/docbook/css\\_book.pdf](http://cs-studio.sourceforge.net/docbook/css_book.pdf)  
**In the following, I refer to this as the CSS docbook.**
- Google: Many developers use RCP.  
A search will often lead to blog entries by Lars Vogel, <http://www.vogella.de/>

## Books

McAffer, Lemieux, Aniszczyk, "Eclipse Rich Client Platform" seems to be a good overall introduction to RCP at this time.

It is based on the earlier versions of Clayberg, Rubel, "eclipse: Building Commercial Quality Plug-ins" and later "eclipse Plug-ins".

McAffer, VanderLei, Archer also wrote "OSGi and Equinox" which has details on the plugin architecture at the basis of Eclipse.



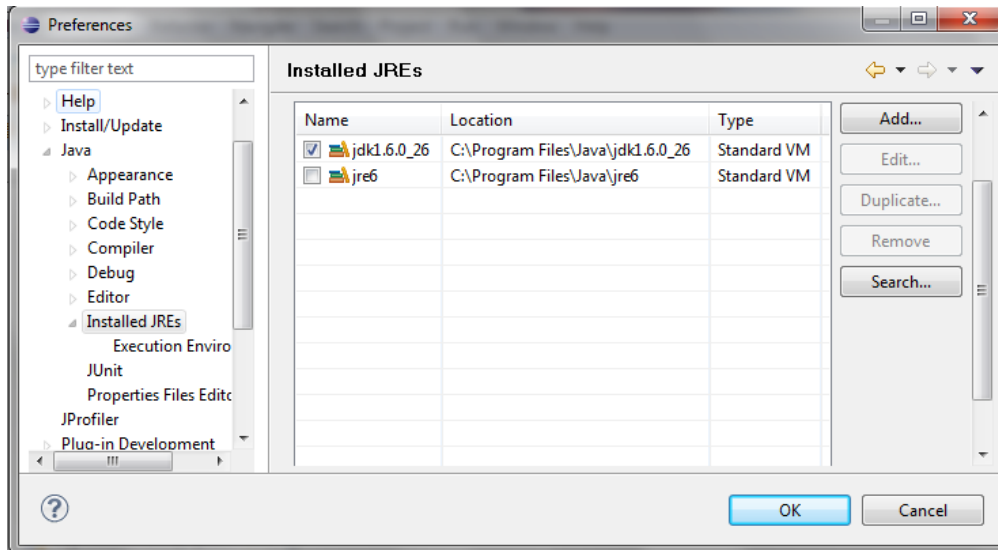
## Preconditions

To get started, you need the following (with version numbers at the time of this writing):

- Java JDK (1.6). You need a Java Development Kit. Your computer might already have a Java Runtime Environment JRE, but you want a full JDK.
- Eclipse IDE for RCP Developers (3.6.x) from <http://www.eclipse.org>
- Source snapshot for one of the CSS products (<http://www-linac.kek.jp/cont/css>, or <http://ics-web.sns.ornl.gov/css/>)

After starting Eclipse, open the menu Window/Preferences and assert that you have a JDK as the default choice under Java/Installed JREs:

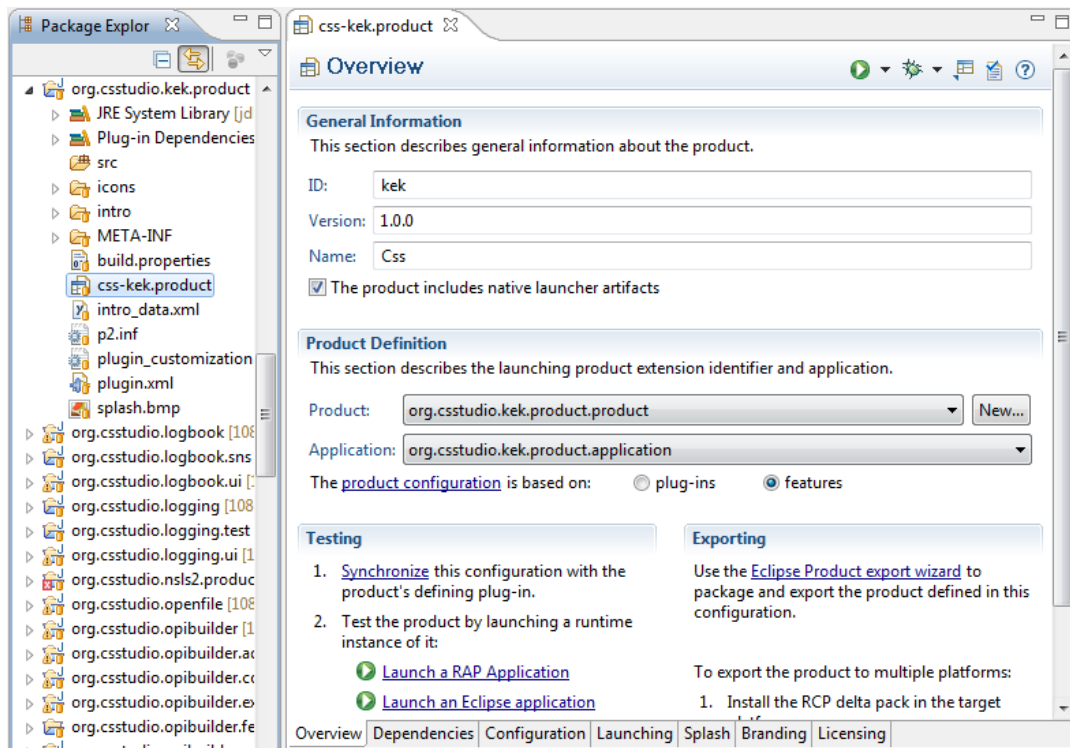
## Developing for Eclipse/RCP/CSS



You can go through the first 2 steps of the following tutorial with this setup. From then on, you will need the CSS sources, so you may as well prepare that now.

Unpack the CSS sources into some directory. Read the CSS docbook section on “Compiling, Running, Debugging CSS” for details on how to import the sources into your IDE workspace.

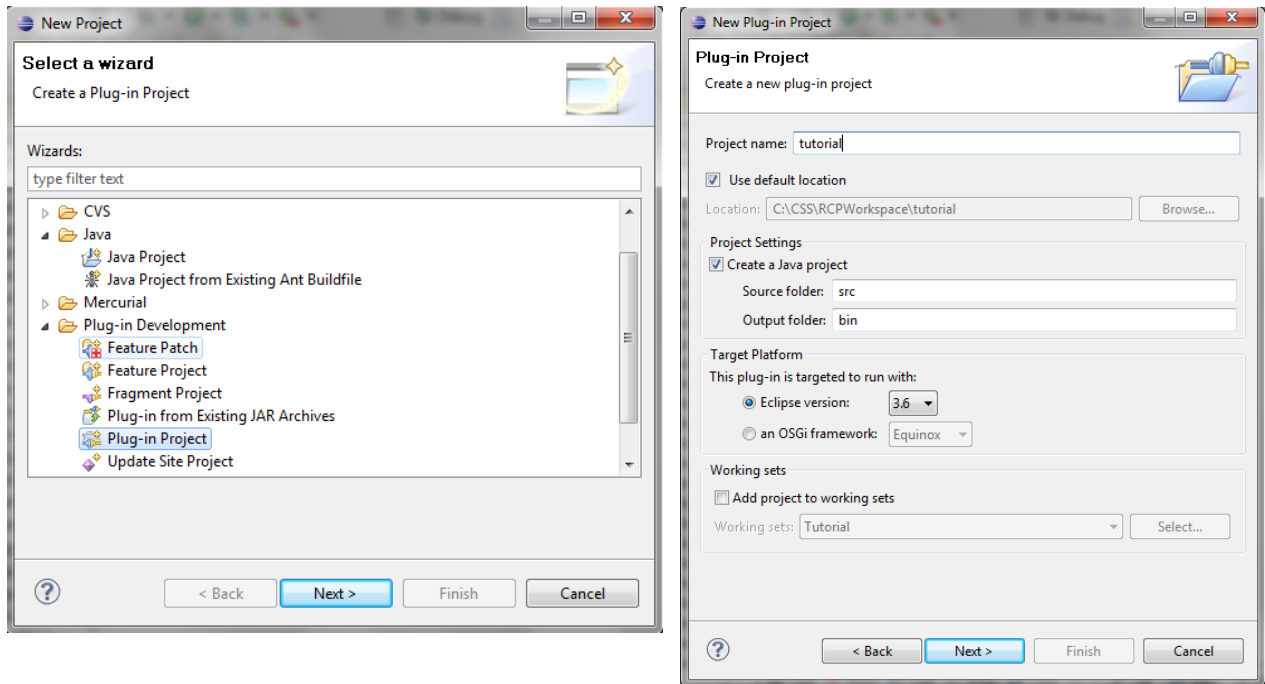
Then open for example the KEK \*.product file, select “1. Synchronize” and “2. Launch an Eclipse application” to start CSS from within the IDE:



# Tutorial Steps

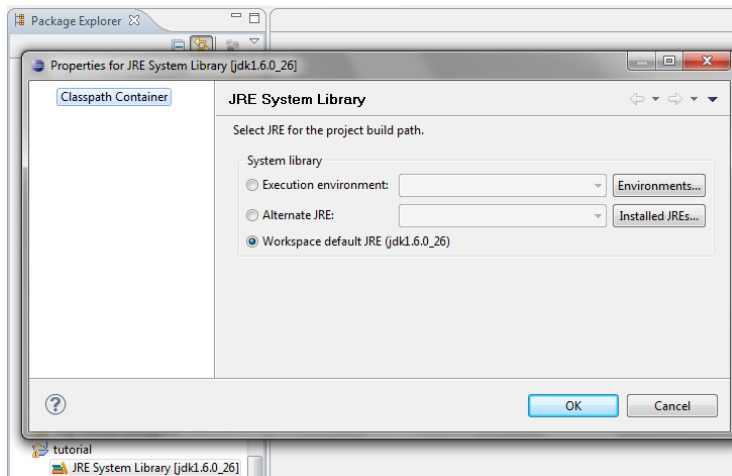
## Create (Plug-in) Project

Select menu File, New Project ...



Select a Plug-in Project, call it “tutorial”, click Next and Finish without changing any of the other settings.

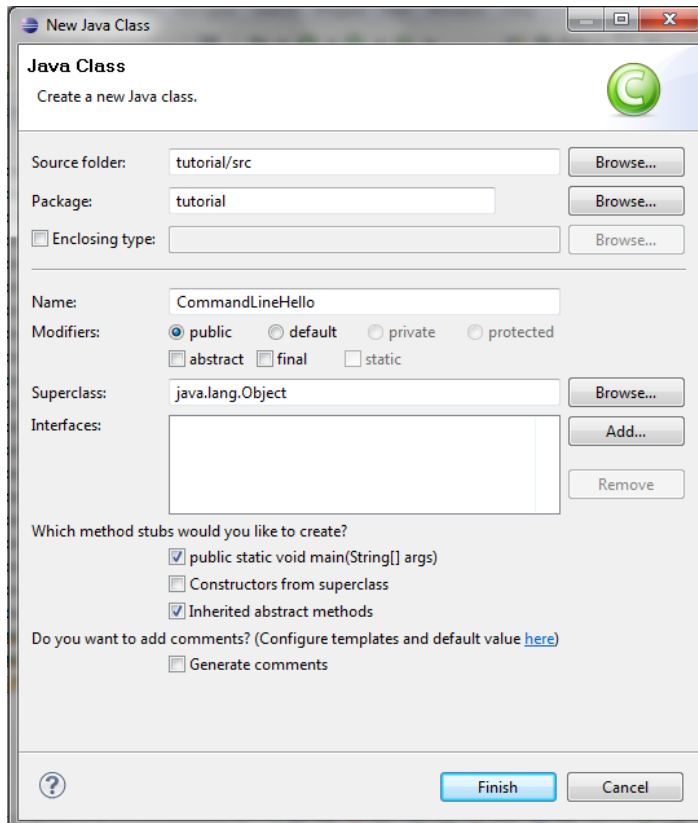
In the created tutorial project, right-click on the JRE System Library, select Properties and assert that it uses the “Workspace default”, which should be a JDK.



## Java command-line “Hello”

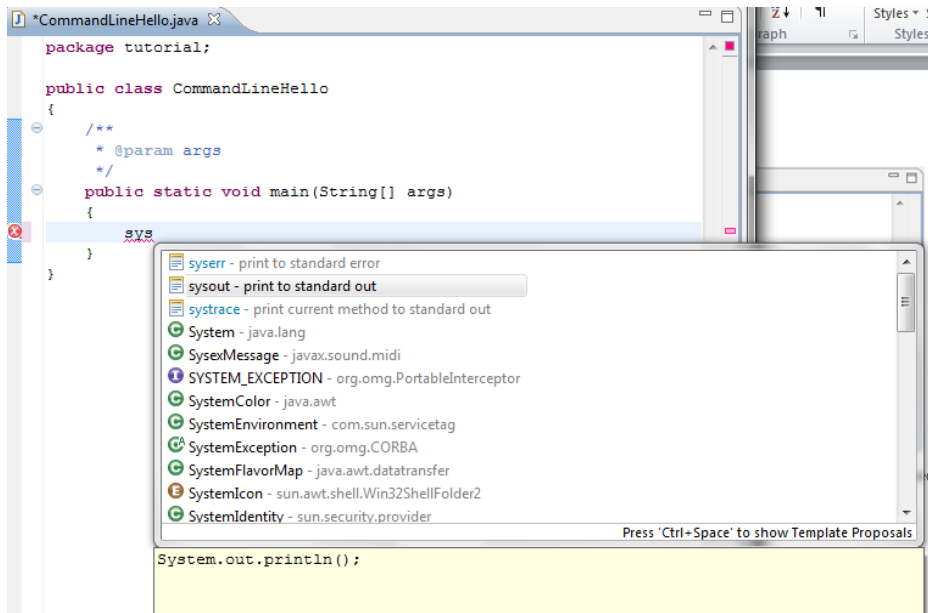
Right-click on the tutorial “src” directory, select “New”, “Package” to create a “tutorial” package unless there is already one.

Right-click on the “tutorial” package in the “src” directory, select “New”, “Class”.  
Enter a Name: “CommandLineHello”, select “public static void main”, press Finish.

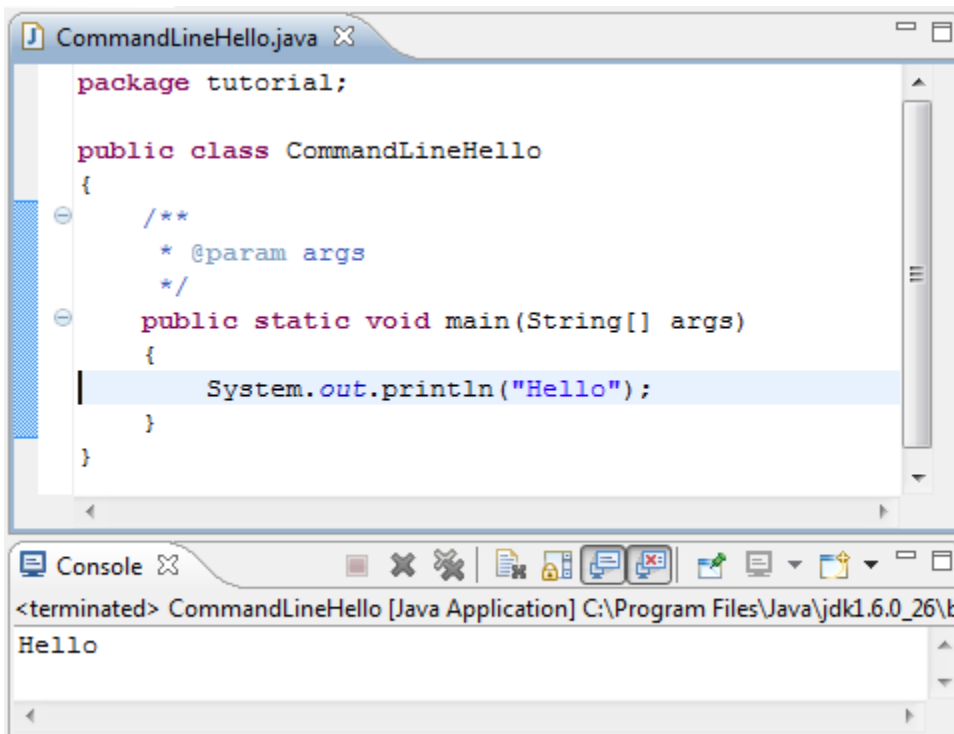


In the main() routine of the generated CommandLineHello.java source code file, enter a command to print hello. As a shortcut to typing “System.out.println”, you can simply type “sys” followed by Ctrl-space which will open the Eclipse content assist system.

## Developing for Eclipse/RCP/CSS



Select the “sysout” entry and complete the source code to look as shown below.



Right-click on the source file, select Run As, Java Application, and the output should appear in the “Console” view.

You can set a breakpoint on the System.out line by double-clicking in the left border of the editor. It should add a blue ball to the left border.



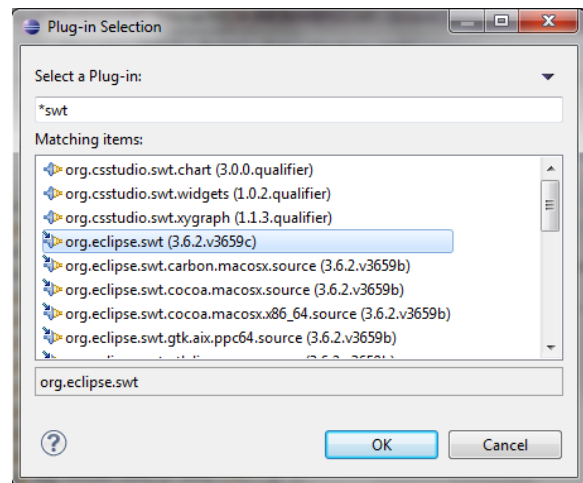
```
public static void main(String[] args)
{
    System.out.println("Hello");
}
```

When you now select “Debug As” instead of “Run As”, you will execute the code in the debugger, which will stop on that line, you can then single-step from there on etc.

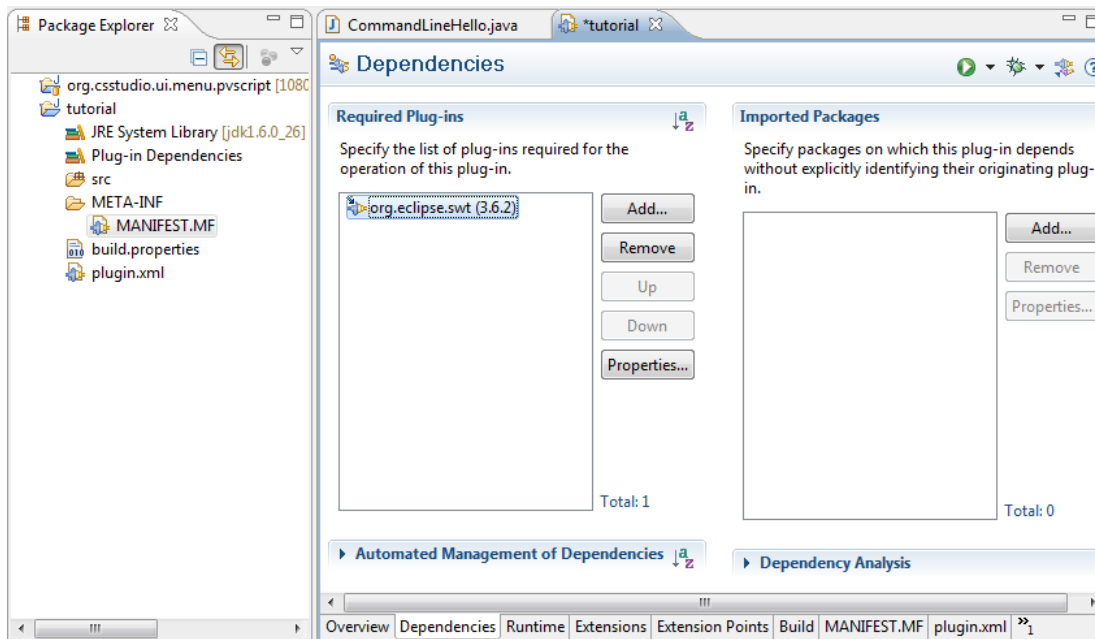
## Standalone GUI “Hello”

To create a version of “Hello” that opens a Window, i.e. has a graphical user interface, you need to use some Java window library. Eclipse/RCP uses SWT, the Standard Window Toolkit. The tutorial plugin needs to be configured to use that SWT library.

Open the MANIFEST.MF file. On the “Dependencies” tab of the Plug-in Manifest Editor, press “Add...” and select the org.eclipse.swt plugin. You can narrow the search to “\*swt” to make it easier to locate that plugin.



In the end, it should look as below with org.eclipse.swt listed as one of the Dependencies of your tutorial plugin:



## Developing for Eclipse/RCP/CSS

When you select the “MANIFEST.MF” tab you can see the raw file content that was created by the the Plug-in Manifest Editor:

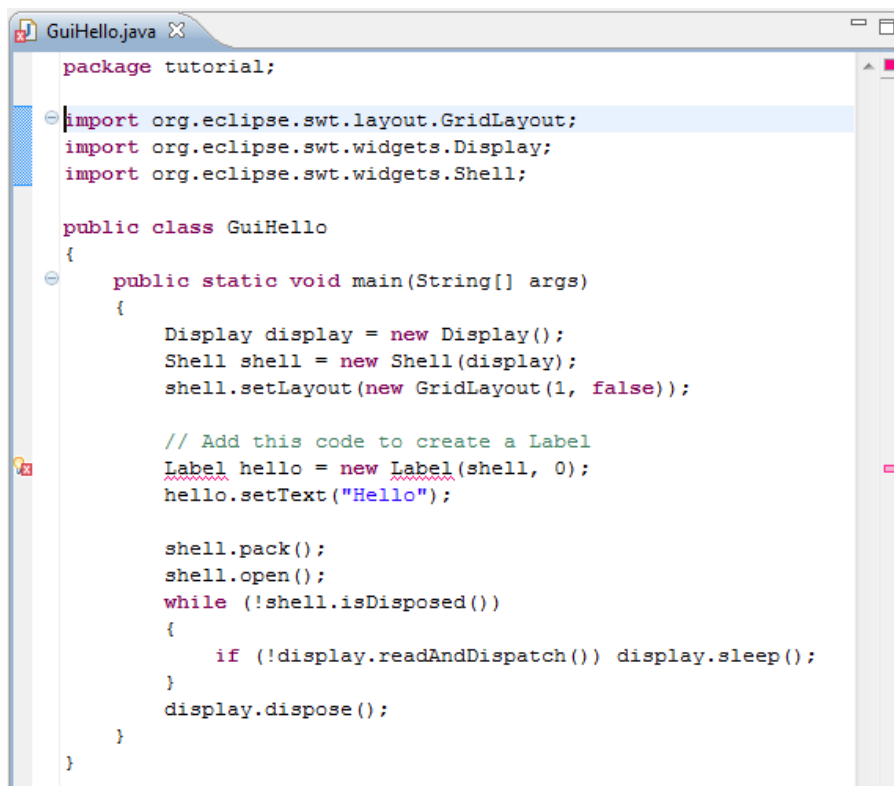
```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Tutorial
Bundle-SymbolicName: tutorial;singleton:=true
Bundle-Version: 1.0.0.qualifier
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Require-Bundle: org.eclipse.swt;bundle-version="3.6.2"
Bundle-Activator: tutorial.Activator
Bundle-ActivationPolicy: lazy
```

The key here is the “Require-Bundle: ...” line.

Similar to the CommandLineHello, create a GuiHello class by right-clicking on the “tutorial” package in the “src” directory, select “New”, “Class”, entering a name of “GuiHello”, again selecting the “public static void main” option, then press Finish.

In the empty main method, create an SWT main loop. This is easiest done by typing “mainloop”, then pressing Ctrl-space to open the Eclipse content assist system and selecting the suggested SWT main loop code.

Around the middle of that main loop code, add the Label code as shown below:



```
GuiHello.java X
package tutorial;

import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;

public class GuiHello
{
    public static void main(String[] args)
    {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setLayout(new GridLayout(1, false));

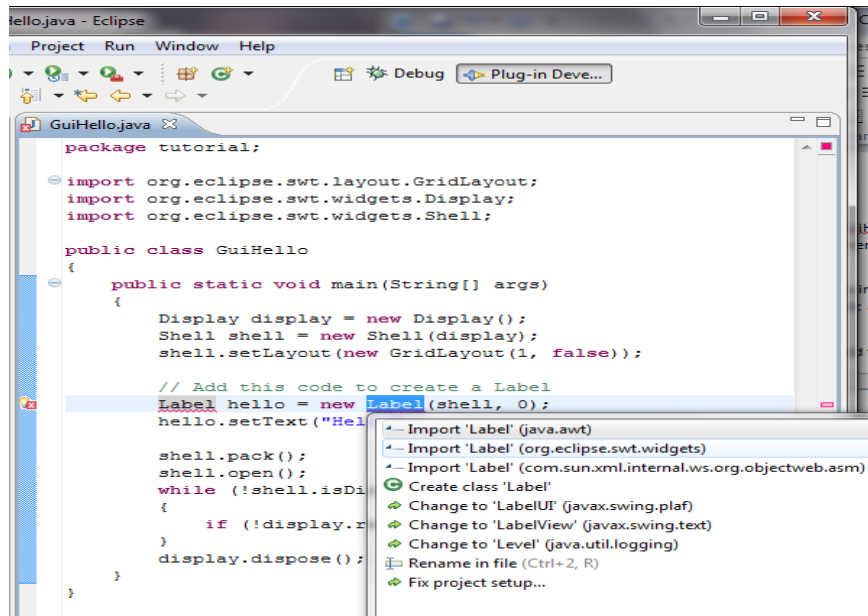
        // Add this code to create a Label
        Label hello = new Label(shell, 0);
        hello.setText("Hello");

        shell.pack();
        shell.open();
        while (!shell.isDisposed())
        {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

## Developing for Eclipse/RCP/CSS

There should be an error indicator on the Label... line. When you hover your mouse pointer over the error light-bulb in the left column, a popup will indicate “Label cannot be resolved to a type” because the compiler is unclear what “Label” type to use.

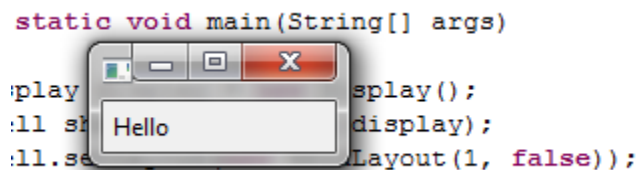
If you right-click on the error light-bulb and select “Quick Fix”, a menu will appear that allows you to pick “Import Label org.eclipse.swt.widgets”. Do not select the “awt” version of the Label type, pick the “swt” type!



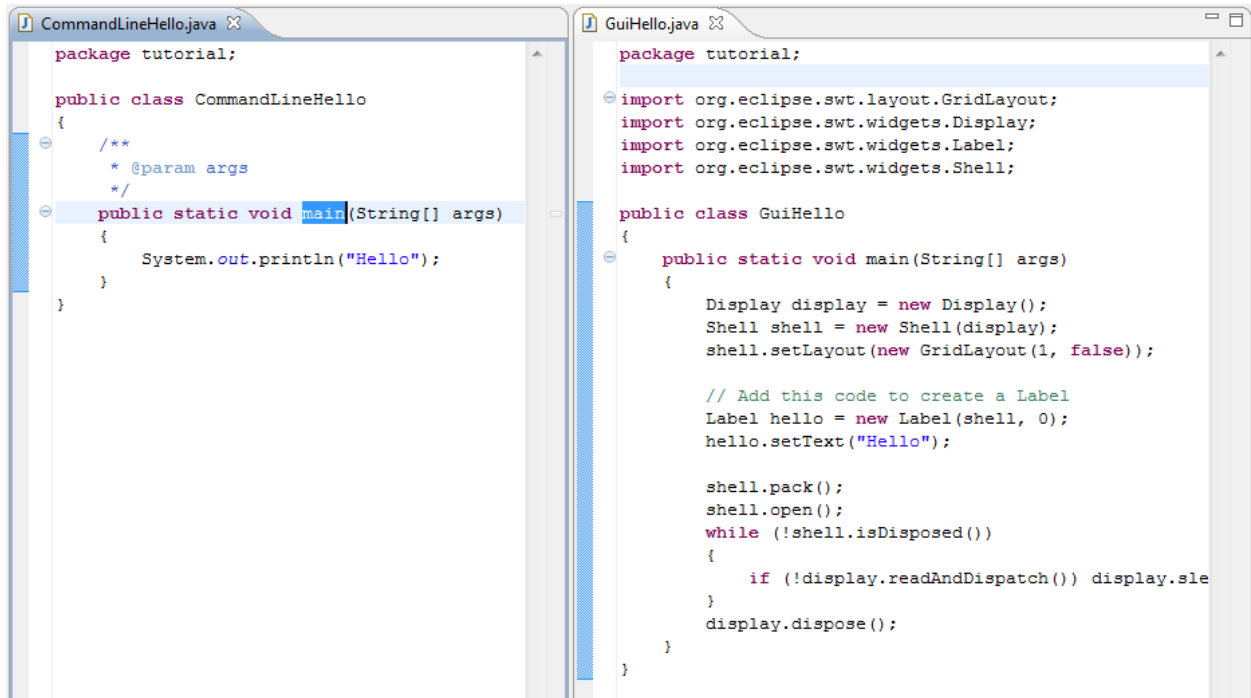
If your code already contains a statement “import java.awt...”, delete that.

If you do not see the “mainloop” content assist, if you do not see “org.eclipse.swt.widgets” offered for the label, check again that your plugin has org.eclipse.swt as a dependency and there are no other errors indicated on the plugin project.

Finally, select Run As, Java Application. A new window will open that contains “Hello” as a text. You can move that window around, finally close it.



So now we created both a command-line and a simple GUI version of a “Hello” program. The GUI version clearly requires more code.



```
package tutorial;

public class CommandLineHello
{
    /**
     * @param args
     */
    public static void main(String[] args)
    {
        System.out.println("Hello");
    }
}

package tutorial;

import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Shell;

public class GuiHello
{
    public static void main(String[] args)
    {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setLayout(new GridLayout(1, false));

        // Add this code to create a Label
        Label hello = new Label(shell, 0);
        hello.setText("Hello");

        shell.pack();
        shell.open();
        while (!shell.isDisposed())
        {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

The next step will be a GUI version of Hello that integrates with Eclipse RCP/CSS.

## RCP “Hello” View

Instead of opening a new, standalone window, this version will display “Hello” within Eclipse/CSS, as a View similar to the Console view. A view that we can move around within CSS. When restarting CSS, it will remember the last location of our view, unlike the standalone window that always appears at some default location, not remembering its last position and size.

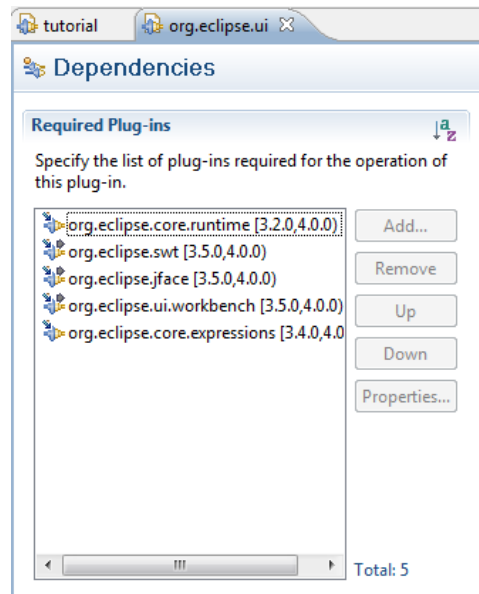
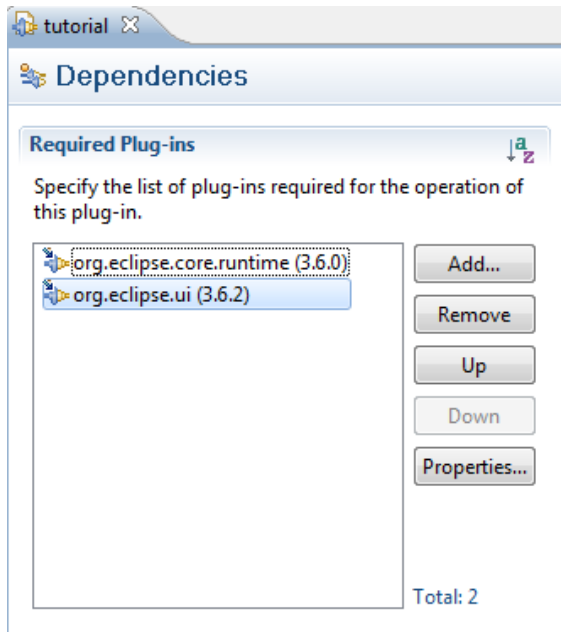
Instead of opening a display and top-level shell in our own code, we depend on Eclipse/CSS to open the main window. We also let Eclipse handle the main loop. All we contribute is a View for Eclipse to display.

The way to interface with RCP, with other plugins in general, is through Extension Points. In this example, we use an Extension Point “Views” offered by the Eclipse user interface to add a view.

In the tutorial Plug-in Manifest Editor, select the “Dependencies” tab.

Remove the dependency on the org.eclipse.swt plugin. Instead, add dependencies to

- org.eclipse.core.runtime
- org.eclipse.ui

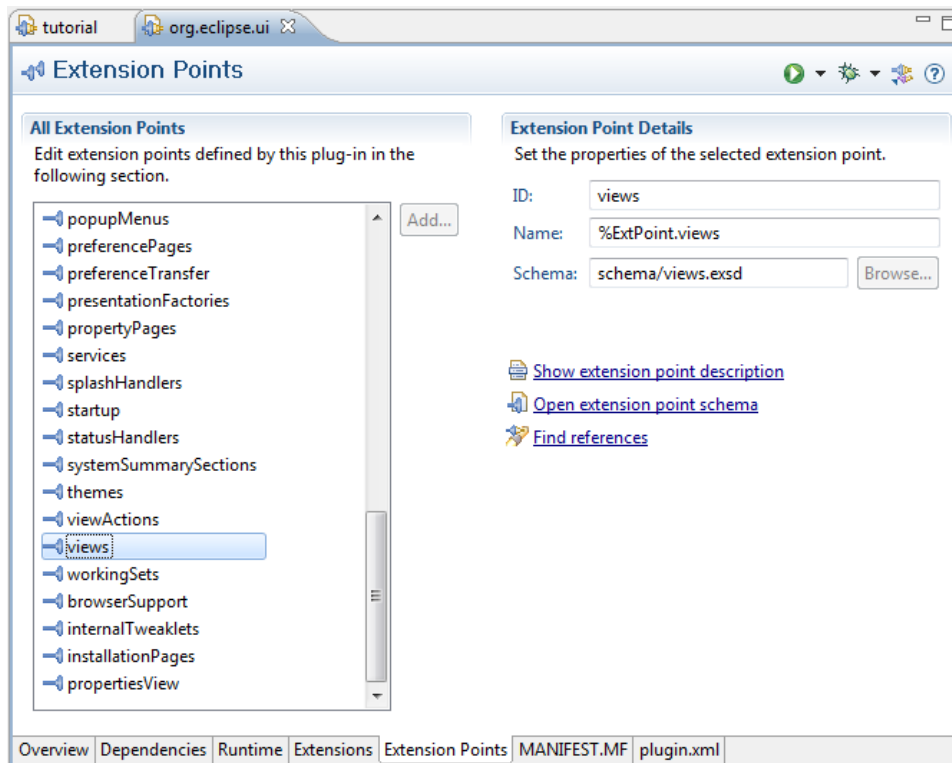


It should look similar to the left screenshot shown above.

When you double-click on the org.eclipse.ui entry, Eclipse opens the Plug-in Manifest Editor for the org.eclipse.ui plugin, as shown to the right above. In the Dependencies tab of org.eclipse.ui, notice that it depends on org.eclipse.swt.

So we have this dependency hierarchy: Tutorial -> org.eclipse.ui -> org.eclipse.swt

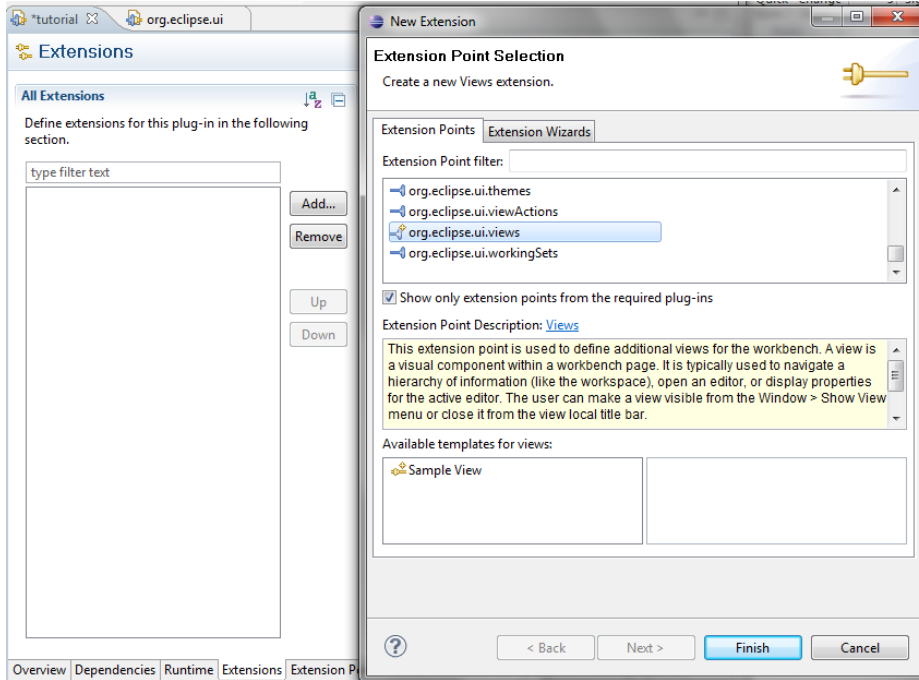
We no longer need a direct entry for org.eclipse.swt because we get that from depending on the org.eclipse.ui plugin.



Look at the Extension Points tab of org.eclipse.ui. Locate the “views” extension point. You may click “Show extension point description” and browse through that. In principle, it has all the information for using that extension point. But it can be hard to understand for newcomers.

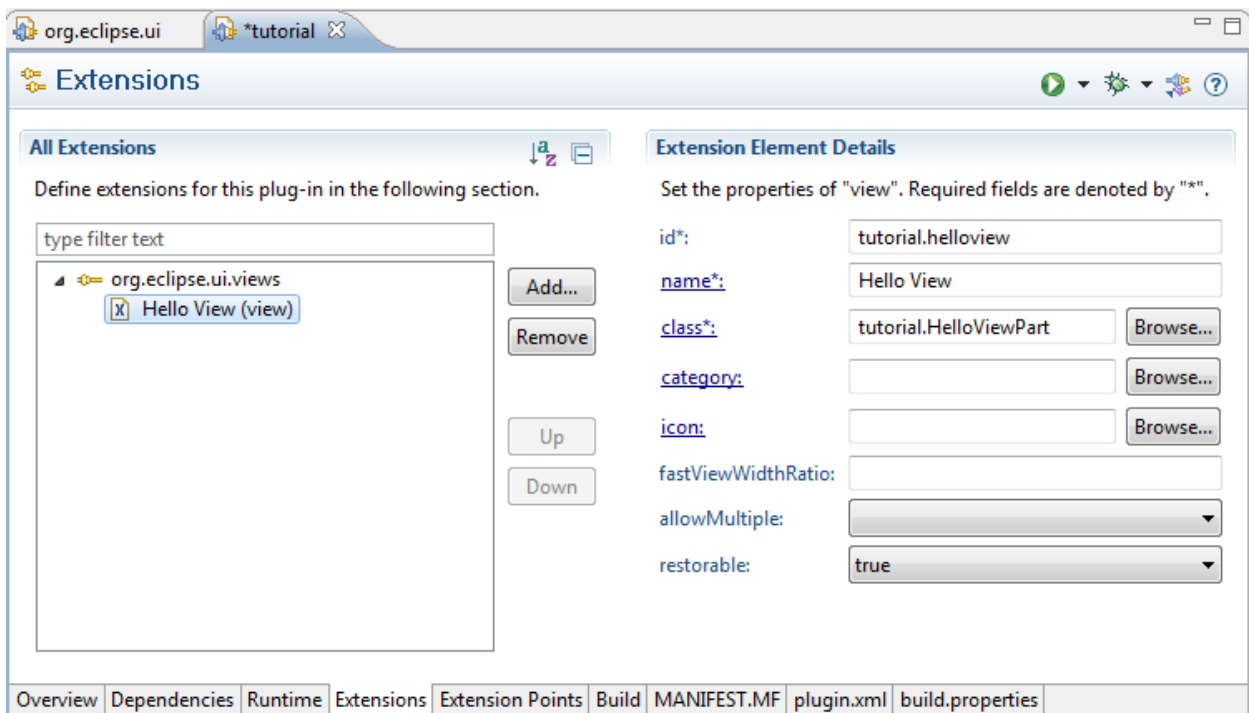
Going back to the Plug-in Manifest Editor for the tutorial plugin, open the Extensions tab.

## Developing for Eclipse/RCP/CSS



“Add” an extension to the org.eclipse.views extension point. In the extension point selector, you could select the “Sample View” template, but for this tutorial press “Finish” without selecting it.

To implement the extension point, the editor will indicate that you must provide an id, name and class. Enter the names shown below:



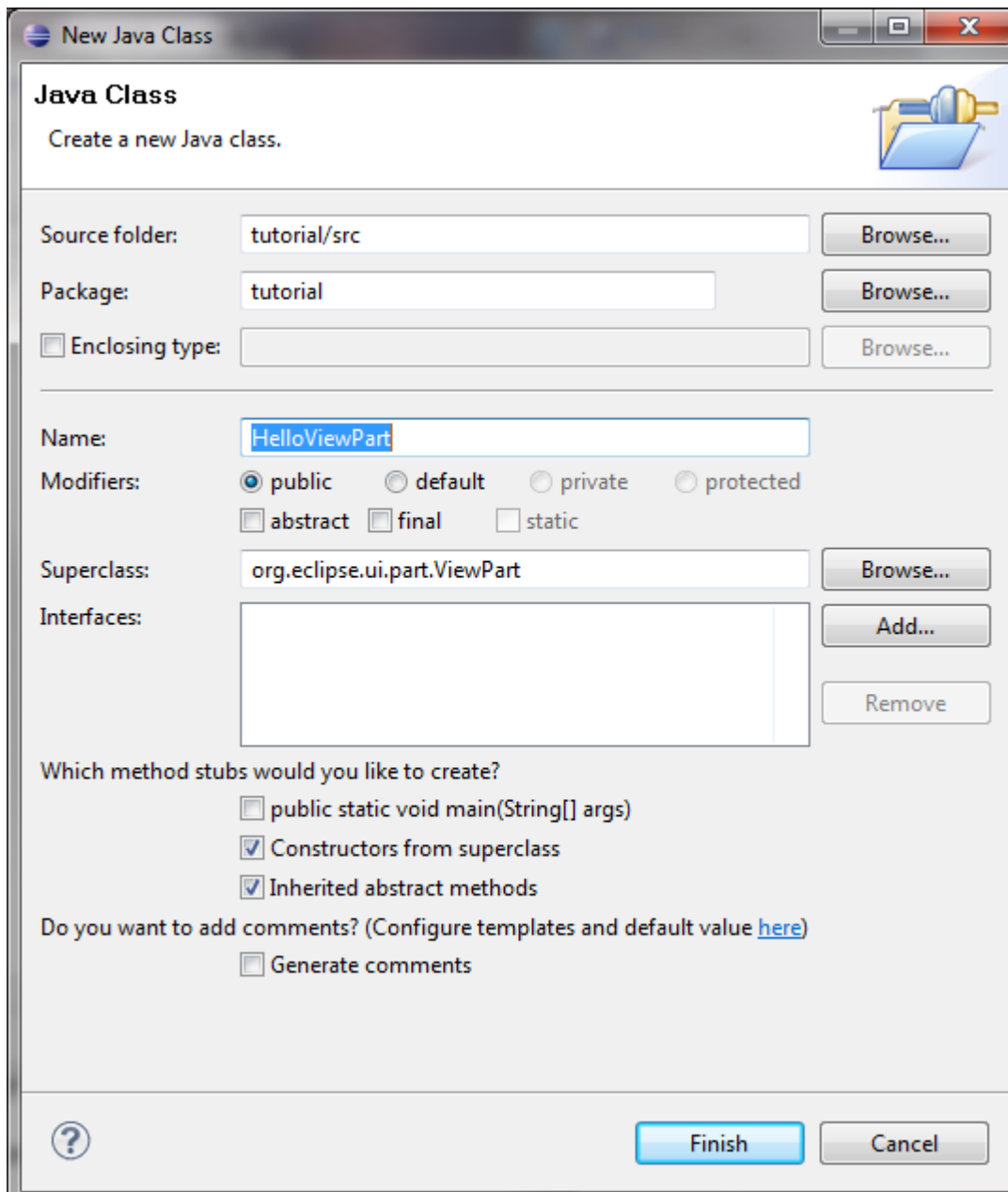
You can also check the plugin.xml tab of the editor. It will show the raw XML description of our extension point, and it should look like this:

```
<plugin>
  <extension point="org.eclipse.ui.views">
    <view id="tutorial.helloview"
          class="tutorial.HelloViewPart"
          name="Hello View"
          restorable="true">
    </view>
  </extension>
</plugin>
```

This XML content is explained in the extension point description. Usually, however, it is sufficient to use the other tabs of the editor to view and modify the extension point info, so return to the “Extensions” tab.

When you click on the blue “class\*.” link, a new class wizard will open because that class “tutorial.HelloViewPart” does not exist, yet.





If you had carefully read the extension point description, you would remember that the class provided for the views extension point should extend the `org.eclipse.ui.part.ViewPart` class. Note how the class wizard is already pre-populated with `...ViewPart` as the Superclass. Press Finish.

Edit the code to look like this:

```
package tutorial;

import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.ui.part.ViewPart;
```

## Developing for Eclipse/RCP/CSS

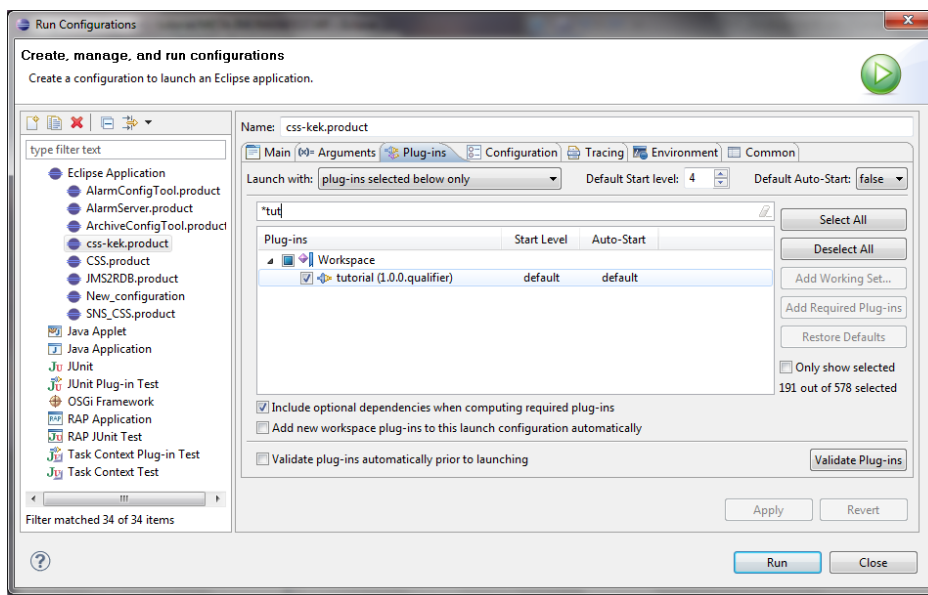
```
public class HelloViewPart extends ViewPart
{
    @Override
    public void createPartControl(Composite parent)
    {
        // Create the label, similar to the GuiHello example,
        // using the 'parent' provided by Eclipse/RCP
        Label hello = new Label(parent, 0);
        hello.setText("Hello");
    }

    @Override
    public void setFocus()
    {
        // Nothing to do
    }
}
```

Note that you fundamentally just add the Label... code inside createPartControl(). If there was a HelloViewPart constructor, you can delete it. The setFocus() must remain, even though it is empty.

*How do we run our tutorial plugin within Eclipse/CSS?*

You need to have at least once executed some CSS product, for example the KEK version of CSS, as mentioned in the Preconditions section on page 4. From the menu Run, Run Configurations..., locate the run configuration for your product. On its Plug-Ins tab, add the tutorial plugin. This will be easier after entering a filter like “\*tut” to reduce the very long list of available plugins to the one you are looking for:



You can press “Validate Plug-Ins”, there should be no error. Press “Run”.

## Developing for Eclipse/RCP/CSS

Your CSS product should start and look like before.

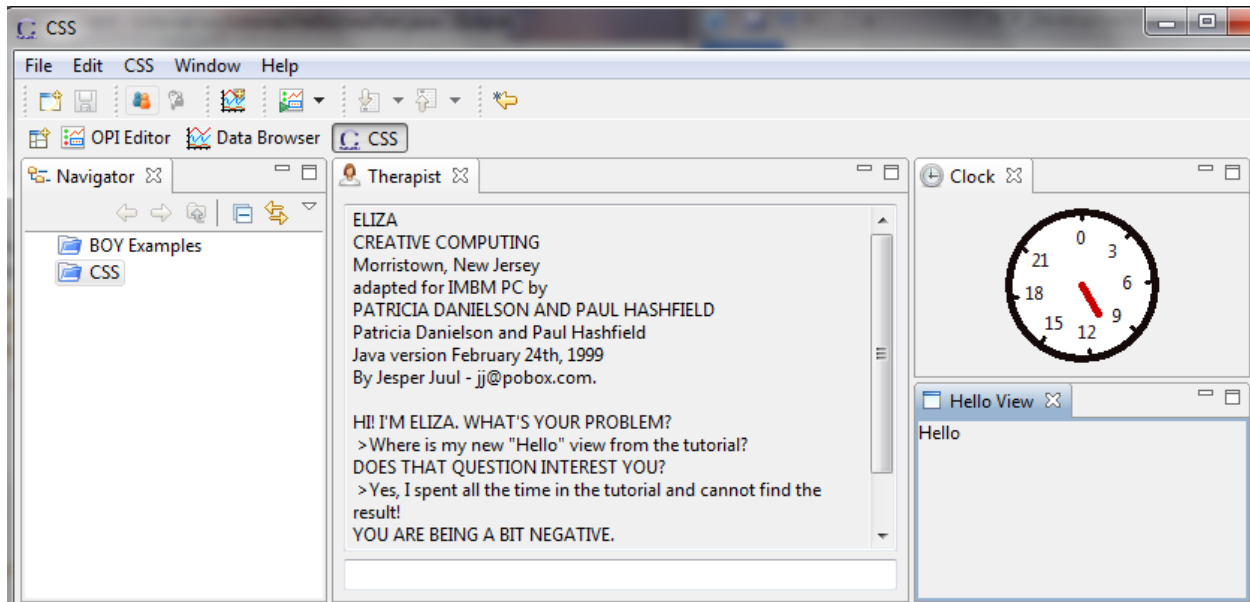
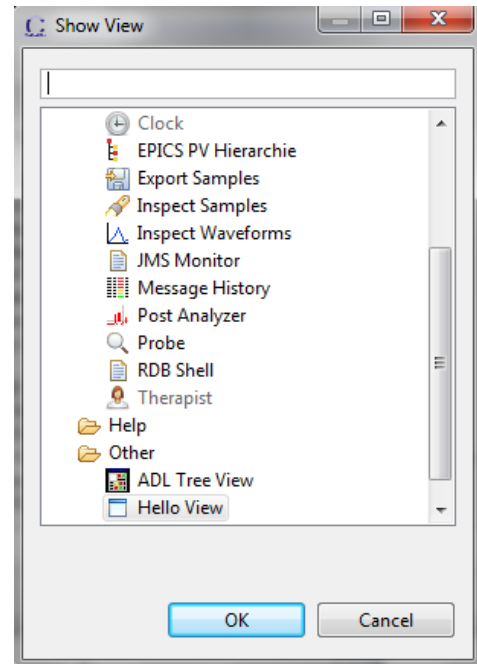
*So where is the new "Hello" View?*

Your tutorial plugin offers a new "Hello" view to Eclipse/RCP/CSS. But nothing is automatically opening it. A user has to actually request to see your view.

This can be done via the menu Window, Show View, Other... where the "Hello View" will appear towards the end under "Other".

If we had defined a Category for our view, we could have placed our view in the CSS category. Check the plugin.xml file of for example org.csstudio.utility.clock which defines a category for its view.

After opening your "Hello" view, it should now appear within CSS. You can move it around just like any other Eclipse view. When restarting CSS, it will remember the location and size of the Hello view:



This step in the tutorial is quite long, and it certainly covers a lot of RCP detail: plugin dependencies, the 'views' extension point and how to implement it, finally adding a plugin to a run configuration and opening a view.

```
package tutorial;

import org.eclipse.swt.layout.GridLayout;

public class GuiHello
{
    public static void main(String[] args)
    {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setLayout(new GridLayout(1, false));

        // Add this code to create a Label
        Label hello = new Label(shell, 0);
        hello.setText("Hello");

        shell.pack();
        shell.open();
        while (!shell.isDisposed())
        {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}

package tutorial;

import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.ui.part.ViewPart;

public class HelloViewPart extends ViewPart
{
    @Override
    public void createPartControl(Composite parent)
    {
        // Create the label, similar to the GuiHello example,
        // using the 'parent' provided by Eclipse/RCP
        Label hello = new Label(parent, 0);
        hello.setText("Hello");
    }

    @Override
    public void setFocus()
    {
        // Nothing to do
    }
}

<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
  <extension
    point="org.eclipse.ui.views">
    <view
      class="tutorial.HelloViewPart"
      id="tutorial.helloview"
      name="Hello View"
      restorable="true">
    </view>
  </extension>
</plugin>
```

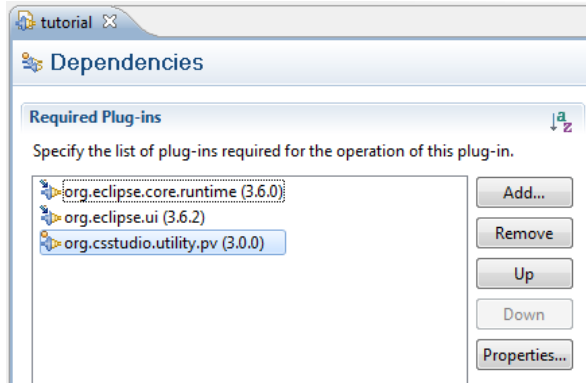
When you compare the standalone `GuiHello.java` to the code required to implement the view, there is not much more code required. The essential `Label...` section is essentially the same. The learning curve for creating the view is steep, but at the same time the result is something that integrates with other views, where the framework restores the size and location on restarts.

## Connect to PVs

The Hello view example was static. In this section, we will extend it to display the changing value of a PV. For simplicity, the PV name will be fixed.

CSS offers several ways of accessing PVs. A straight forward API for creating individual PVs and listening to their updates is the `utility.pv` API described in the docbook Part II. Plug-in Reference, PV Access - `org.csstudio.utility.pv`.

First, you need to add a dependency on the `org.csstudio.utility.pv` plugin to the tutorial plugin:



In HelloViewPart.java, change the ....Label ... code such that the label is a field of the class that we can later update with the current value of the PV.

You can most easily accomplish this by right-clicking on the original “hello” Label variable name, selecting Refactor, “Convert local variable to field” and entering value\_display as the field name.

Or edit it the hard way. In the end, it should look like this:

```
private Label value_display;

@Override
public void createPartControl(Composite parent)
{
    value_display = new Label(parent, 0);
    value_display.setText("Hello");
}
```

Now add the code to connect to a PV, using this (i.e. the HelloViewPart class) as the listener:

```
try
{
    PV pv = PVFactory.createPV("sim://ramp");
    pv.addListener(this);
    pv.start();
}
catch (Exception e)
{
    e.printStackTrace();
}
```

There will be an error on the addListener() call because the HelloViewPart class is not a valid PVListener. Right-click on the error light-bulb, select Quick-Fix, “Let ‘HelloViewPart’ implement ‘PVListener’”.

Next there will be an error on the HelloViewPart class because it is missing the actual implementation of the PVListener interface. Again use the Quick-Fix to “Add unimplemented methods”.

This will add the skeleton for the methods pvValueUpdate and pvDisconnected.

Implement them like this:

```
@Override
public void pvValueUpdate(PV pv)
{
    final String value = "PV " + pv.getName() +
        " has value " + pv.getValue().toString();
    value_display.setText(value);
}

@Override
public void pvDisconnected(PV pv)
{
    // Ignored
}
```

When you now run CSS and open the Hello View, we might expect to see the “Hello” text replaced with a text like “PV sim://ramp has value 2011/07/06 13:13 2.0”. In fact, there will be no updates. You can close CSS again. Check the “Console” view of the IDE. There will be this type of error:

```
Exception in thread "ramp" org.eclipse.swt.SWTException:
  Invalid thread access
  at org.eclipse.swt.SWT.error(SWT.java:4083)
  at org.eclipse.swt.SWT.error(SWT.java:3998)
  at org.eclipse.swt.SWT.error(SWT.java:3969)
  at org.eclipse.swt.widgets.Widget.error(Widget.java:468)
  at org.eclipse.swt.widgets.Widget.checkWidget(Widget.java:359)
  at org.eclipse.swt.widgets.Label.setText(Label.java:387)
  at tutorial.HelloViewPart.pvValueUpdate(HelloViewPart.java:42)
  at org.csstudio.utility.pv.simu.BasicPV.changed(BasicPV.java:84)
```

Similar to most GUI toolkits (AWT, Swing, Qt,...), SWT only allows access to the user interface elements (Label, Shell, ...) from the main thread that also creates the GUI and executes the main loop. The PV value updates on the other hand can arrive on other threads, for example an EPICS Channel Access client thread.

Every GUI toolkit has some mechanism that allows arbitrary threads to schedule GUI updates on the correct main thread. With SWT, you need to use the Display.asyncExec(Runnable) API. A corrected version of the pvValueUpdate looks like this:

```
@Override
public void pvValueUpdate(PV pv)
{
    final String value = "PV " + pv.getName() +
        " has value " + pv.getValue().toString();
    // Perform GUI update on display thread
```

```
value_display.getDisplay().asyncExec(new Runnable()
{
    @Override
    public void run()
    {
        value_display.setText(value);
    }
});
}
```

When you now start CSS and open the Hello view, it should display changing PV data.

You will still, however, notice a problem when you close the Hello View. Since we do not stop the PV, our code will continue to attempt updates to our view even though the view is long gone. When exactly this happens can vary. Eclipse will not right away delete your view classes when a user closes the view, because after all the user may re-open the view soon. But eventually Eclipse will delete the view, and then errors like this start to occur:

```
Exception in thread "ramp" org.eclipse.swt.SWTException: Widget is disposed
    at org.eclipse.swt.SWT.error(SWT.java:4083)
    at org.eclipse.swt.SWT.error(SWT.java:3998)
    at org.eclipse.swt.SWT.error(SWT.java:3969)
    at org.eclipse.swt.widgets.Widget.error(Widget.java:468)
    at org.eclipse.swt.widgets.Widget.getDisplay(Widget.java:582)
    at tutorial.HelloViewPart.pvValueUpdate(HelloViewPart.java:43)
```

A properly written plugin needs to clean up when it is closed. One way to do this is by adding a dispose listener that notifies us when the view is closed.

In completeness, our view code with PV updates could look like this:

```
public class HelloViewPart extends ViewPart implements PVListener
{
    private Label value_display;

    @Override
    public void createPartControl(Composite parent)
    {
        value_display = new Label(parent, 0);
        value_display.setText("Hello");

        final PV pv;
        // Create and start PV
        try
        {
            pv = PVFactory.createPV("sim://ramp");
            pv.addListener(this);
            pv.start();
        }
        catch (Exception e)
        {
            // For tutorial, just print error
            e.printStackTrace();
            return;
        }
    }
}
```

```
// When the view is closed, stop the PV
parent.addDisposeListener(new DisposeListener()
{
    @Override
    public void widgetDisposed(DisposeEvent e)
    {
        pv.stop();
    }
});

@Override
public void setFocus()
{
    // NOP
}

@Override
public void pvValueUpdate(PV pv)
{
    final String value = "PV " + pv.getName() +
        " has value " + pv.getValue().toString();
    // Perform GUI update on display thread
    value_display.getDisplay().asyncExec(new Runnable()
    {
        @Override
        public void run()
        {
            value_display.setText(value);
        }
    });
}

@Override
public void pvDisconnected(PV pv)
{
}
}
```

## Hooking into menus

To invoke our view, we have to use the generic menu Window, Show View, Other...

Most CSS views have an entry in the more obvious CSS menu, for example CSS, Utilities, Clock to open the clock.

To add a menu, you again use Eclipse extension points, this time `org.eclipse.ui.menus`. The process has actually three steps:

1. Add the GUI element entry (main menu, context menu, toolbar, ...) to invoke a command. This tells eclipse where and how to display the desired item to the user.



## Developing for Eclipse/RCP/CSS

2. Create the command. This is an abstract description of what you want to accomplish, like “open a view”. The same command can be invoked by many means: Menu, toolbar, keyboard shortcut, programmatically.
3. Implement the handler. The command only describes what to do. It doesn’t do anything. A handler performs the actual action. Eclipse allows more than one handler, and there are ways to select which handler to use.

Overall, this separation of GUI, command, handler is very flexible, but initially it can be very confusing. The CSS docbook chapter in Part II. Plug-in Reference, CSS menus - org.csstudio.ui.menu gives brief examples for adding menu entries to the CSS menu.

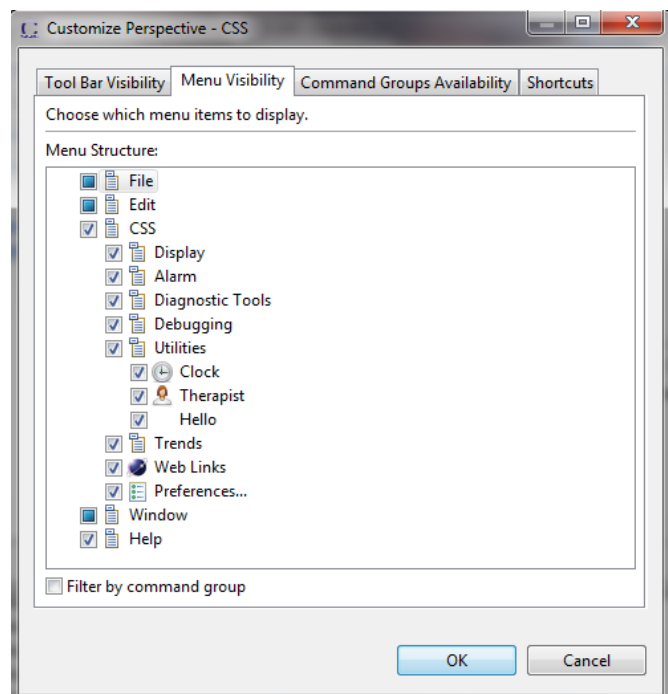
For our purpose, we want to add an entry to the CSS Utilities menu, which has a menu path of menu: utility. The command to open a view is actually already pre-defined by Eclipse, and so is a handler for it. We only need to refer to the existing Eclipse command, which takes the name of the view to open as a parameter.

In short, adding this to the tutorial plugin.xml file allows opening our view from the CSS/Utilities menu:

```
<extension point="org.eclipse.ui.menus">
  <menuContribution
    allPopups="false"
    locationURI="menu:utility">
    <command
      commandId="org.eclipse.ui.views.showView"
      label="Hello"
      style="push">
      <parameter
        name="org.eclipse.ui.views.showView.viewId"
        value="tutorial.helloview">
      </parameter>
    </command>
  </menuContribution>
</extension>
```

Similar to adding a view, this is not really much code: We obviously need to add some type of “menu” entry, specify where it should appear (menu:utility), and what it should do (showView, the one with ID tutorial.helloview).

The approach is very flexible. The CSS application code defines where exactly “menu:utility” appears and how. The main



CSS/Utilities menu entry could be moved to a different location, the name “Utility” could be renamed to a localized text. Our plugin will still work.

At runtime, users can right-click on the CSS toolbar, select “Customize Perspective” and then disable our menu entry if they prefer not to see it. Or they can define a keyboard shortcut if they want even faster access to our view.

But there is certainly a steep learning curve until a developer knows all the details behind those few lines of XML markup in the plugin.xml file to define a menu.

## PV Context menu ‘Probe’

Another very powerful feature of Eclipse is the idea of menu contributions to popup (context) menus based on the currently selected data type.

For example, “Probe” appears in the context menus of BOY widget, Data Browser channel names, ... It appears in the context menu of any CSS application that deals with process variables. When selected, Probe will be started, receiving the PV name.

To learn about this, refer to the CSS docbook chapter in Part II. Plug-in Reference, CSS menus - org.csstudio.ui.menu, the section called “Process Variable popup-menu”, and compare that to the plugin.xml code of org.csstudio.diag.probe.

## Look at details in org.csstudio.util.pvscript

The plugin org.csstudio.util.pvscript is very small but it includes several interesting features:

1. Online help  
The tool contributes online help in a straight forward way.
2. PV Script appears in context menus for Process Variables, similar to Probe.  
While Probe, however, always just creates a “Probe” entry in the context menu, i.e. one static entry, the PV Script tool creates zero or more entries, one for each configured script.  
It implements a dynamic context menu.
3. PV Script has preference settings and a preference GUI. Refer to the CS docbook Part I, Hierarchical Preferences. Usually, preferences are simple true/false, numbers or string settings.  
The PV Script util on the other hand has a complex preference: A variable-length list of script names with descriptions. It needs to encode and decode this list from the plain string that is stored in the preferences.

The PV Script plugin does not contain much code, so it can be a good example for studying the above items.

## Summary

In summary, Eclipse/RCP/CSS is complex but very powerful. Any attempt to learn it in “10 easy steps” is futile. It would be like building the toy clock mentioned above.

This was supposed to show some of the ideas, and then you need to study an RCP book and examples.

Please contact me at [kasemirk@ornl.gov](mailto:kasemirk@ornl.gov) with comments on this tutorial.