This page is deprecated and may contain some information that is no longer relevant or accurate.

Back to Eclipse Corner Articles (/articles/index.php)
ADD THIS (http://www.addthis.com/bookmark.php) Printer-friendly version (/articles/printable.php?file=Article-Preferences/article.html)

To comment on this article, ask questions, or propose corrections, please **open a bug**.

Eclipse Corner Article

Preferences in the Eclipse Workbench UI

Summary
In the Eclipse Platform plug-in developers define preference pages for their plug-ins for use in the Workbench Preferences Dialog. This article explains when to use a preference and some of the features the Eclipse Platform provides to support preferences.
By Tod Creasey, OTI
August 15, 2002

Editor's note: **This article contains outdated information.** This article was originally published in December 2001 and described Eclipse release 1.0. This revision reflects the minor enhancements made to UI preferences in Eclipse release 2.0. A revision of this article is scheduled for development. For more information, or to contribute, please see **bug 43727 (https://bugs.eclipse.org/bugs/show_bug.cgi?id=143727)**.

Introduction

The Eclipse Platform has support for preferences that are persisted along with the workspace. This article will discuss what type of data should be stored as a preference and will show how to develop and register a user interface to allow the user to set these preferences as well as how to store them independent of the workbench by use of the import and export functions. It will also cover how to initialize and retrieve preferences for use by other other plug-ins that use your plug-in. This functionality will be shown using an example that searches files for bad words. We will set our preferences for this tool using two preference pages, one simple one to set a highlight color and one more complex one to set the list of words.

When to Use a Preference

A preference is data that is persisted between workspace sessions to allow the user to keep the state of a plug-in consistent between Eclipse sessions. As of 2.0 Eclipse offers two varieties of preference, UI preferences (the same as in 1.0) and Core Preferences. This article is concerned only with how to use the UI preference store. Typical UI preferences are default values for new instances, colors for editors and paths. Core preferences are used for values that are not part of the user interface.

Preferences are not intended to reference any resource currently defined in the workspace and instead should be used for editors, views or other objects that perform operations on a resource. Data persisted on a resource instance is better suited to be a property which will be discussed in a later article.

A preference can be made available to any plug-in that has your plug-in as a prerequisite. The usual way to do that is to provide API on your plug-in that allows for access to the preferences you want to make available. The values of these preferences are stored in the .metadata/.plugins directory of the workspace on a per plug-in basis. We demonstrate how to do this below.

The Preference Store and the Plug-in

Every plug-in has it's own preference store provided by the workspace. For this example we will define a plug-in and use its preference store for our preferences. As we are going to use this plug-in within the UI we define it as a subclass of AbstractUIPlugin. Our constructor (see 2) will create a singleton to allow easy access to the plug-in instance in the workbench. We also implement the method initializeDefaultPreferences() to set up our default values for our two preferences. We are defining a preference for the bad words and a preference for the color of the highlight. Each preference value is looked up using a given key. In the code below the keys we are using are defined by the constants in 1.

The default value should be set for all preferences to be sure that there is a value to use at all times. A default value also ensures that the UI can provide a way to reset a preference value back to a reasonable initial setting via the Restore Defaults button.The default value of the preference should be initialized in the plug-in so that it is set before any of the UI is created.

IAbstractWorkbenchPlugin defines a method called initializeDefaultPreferences(IPreferenceStore) which is called when the preference store is created the first time. In this method (see 3) you should set the default value for all values that you will be using the preference store for. We set a default color using the helper methods in the PreferenceConverter which allows the plug-in developer to set and get values for a preference of commonly stored types like FontData, Point etc. This API is provided because preferences are stored and retrieved as Strings in a human readable format in order to leverage the java properties mechanism. Our more complex bad words preference is initialized using a set of preselected bad words defined in the format we are going to store them in as we do not have API on the PreferenceConvertor to store or retreive arrays of Strings.

```
Color color= Display.getDefault().getSystemColor(SWT.COLOR_BLUE);
PreferenceConverter.setDefault(store, HIGHLIGHT_PREFERENCE, color.getRGB());
```

```
public class BadWordCheckerPlugin extends AbstractUIPlugin {
        //The shared instance.
        private static BadWordCheckerPlugin plugin;

        //The identifiers for the preferences
1       public static final String BAD_WORDS_PREFERENCE = "badwords";
        public static final String HIGHLIGHT_PREFERENCE = "highlight";

        //The default values for the preferences
        public static final String DEFAULT_BAD_WORDS = "bug;bogus;hack;";
        public static final int DEFAULT_HIGHLIGHT = SWT.COLOR_BLUE;

        public BadWordCheckerPlugin(IPluginDescriptor descriptor) {
                super(descriptor);
2               plugin = this;
        }

        public static BadWordCheckerPlugin getDefault() {
                return plugin;
        }

        /**
         * Initializes a preference store with default preference values
         * for this plug-in.
         */
        protected void initializeDefaultPreferences(IPreferenceStore store) {
3               store.setDefault(BAD_WORDS_PREFERENCE, DEFAULT_BAD_WORDS);
                Color color= Display.getDefault().getSystemColor(DEFAULT_HIGHLIGHT);
                PreferenceConverter.setDefault(store,  HIGHLIGHT_PREFERENCE, color.getRGB());

        }
}
```

Defining Preference Pages in plugin.xml

Now that we have defined the preference we want to provide a way for the user to set the preference value. Preference pages for the workbench can be found in the preferences dialog. The preferences dialog is accessible via the Window->Preferences menu group. Plug-in developers should add their preference pages to this dialog using the plugin.xml of their plug-in in order to maintain a consistent look and feel with other Eclipse plug-ins. The definition of the preference pages within plugin.xml looks like this:

```
<extension point="org.eclipse.ui.preferencePages">
        <page id="BadWordsPreferencePage"
1               name="Bad Words"
2               class="org.eclipse.ui.articles.badwordchecker.BadWordsPreferencePage">
        </page>

        <page id="BadWordsColorPreferencePage"
                name="Colors"
                class="org.eclipse.ui.articles.badwordchecker.BadWordsColorPreferencePage"
3               category="BadWordsPreferencePage">
        </page>
</extension>
```

The definition above sets the name (1) of the preference page for use in the list of pages in the preference dialog and also specifies the class(2) to be instantiated for creating the preference page. This class must conform to IWorkbenchPreferencePage.

In the second definition there is a category (3) tag which is used to make one page the child of another in the list in the preferences dialog. Preference pages can be stored as the children of other pages. This is useful for keeping a series of pages together that are related to each other and also reduces the clutter in the workbench preferences page. A page can be made the child of another page by setting the id of the parent page as the value of the category field in the plugin.xml. A page with no parent is displayed as a child with no root.

With the above declarations in our plugin.xml the list of preference pages shown in the preference dialog will look like Figure 1.

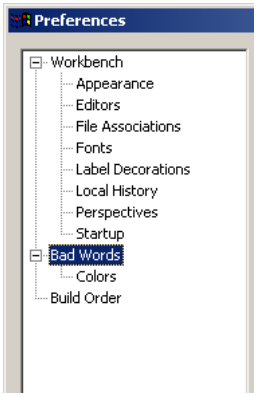

Figure 1:Preference dialog showing Bad Words and Colors preferences

The Color Preference Page

The color preference page is an example of a simple page that uses a single JFace field editor to manage its values. Initially a preference page class is defined. All classes used in the preference dialog must conform to IWorkbenchPreferencePage. Eclipse includes the class PreferencePage which implements most of the necessary API for a preference page. PreferenceDialog will save the preference store whenever OK is pressed - if you wish to use PreferencePages in places other than the default dialog in Window->Preferences be sure that you save the preference store after changes have been applied.

The class definition for our preference page is:

```
class BadWordsColorPreferencePage
        extends PreferencePage
        implements IWorkbenchPreferencePage
```

Once we have defined the page we want to initialize it. IWorkbenchPreferencePage specifies a message init(IWorkbench) for this purpose. We will not use the Workbench argument for this page. Our implementation only sets the preference store for the page.

```
public void init(IWorkbench workbench) {
        //Initialize the preference store we wish to use
        setPreferenceStore(BadWordCheckerPlugin.getDefault().getPreferenceStore());
}
```

The other required method we must implement is createContents(). All we are going to do is use a ColorFieldEditor to set our preference. It is also suggested that performDefaults is implemented so that the current state can be reset to the defaults defined in the plug-in. We also need to implement performOK so that the settings defined by the user are stored in the preference store for our plug-in. Our implementation is simple as the ColorFieldEditor has the code to load defaults and store the results of an apply for a preference already defined and performOK and performDefaults can call the corresponding methods on ColorFieldEditor. See figure 2 for the Colors preference page.

```
protected void performDefaults() {
        colorEditor.loadDefault();
}
/**
 * Save the color preference to the preference store.
 */
public boolean performOk() {
        colorEditor.store();
        return super.performOk();
}
```

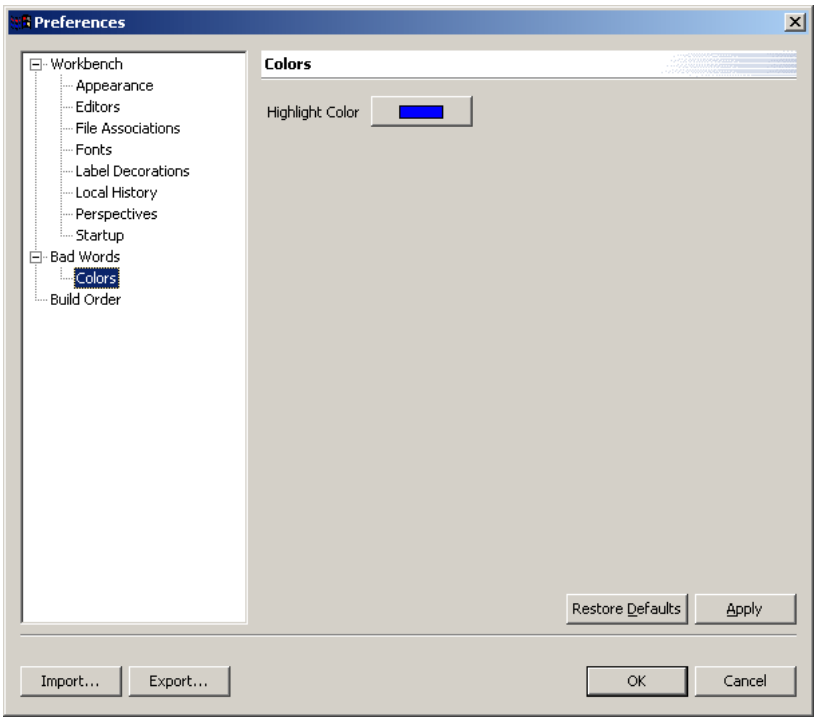

Figure 2: Preference dialog showing Colors preference page

The Bad Words Preference Page

We have seen how to do a simple preference page with just a color and categorize it. Now we will show how to use a complex object as a preference and still have it persisted by the preference store and editable in a preference page. For this example we are going to add a bad words preference which is an array of Strings.

As the PreferenceConverter does not have API for conversion of arrays of Strings we will implement it ourselves in the BadWordCheckerPlugin. By implementing it in the plug-in we put the API for the use of the preference in a place visible to all objects that have access to this plug-in. Normally we would use the PreferenceConverter for conversion to and from the storage format.

Methods for getting the default value of the preference and a getter and a setter are defined first - getBadWordsDefaultPreference (which returns an array of Strings), getBadWordsPreference (which also returns an array of Strings) and setBadWordsPreference which takes an array of Strings as its argument. The String array is stored in the preference store as a single string separated by semicolons. We choose semicolons as this character is only ever used as punctuation and will therefore never be part of a word we are searching for.

```
/**
 * Return the bad words preference default.
 */
public String[] getDefaultBadWordsPreference(){
        return convert(getPreferenceStore().getDefaultString(BAD_WORDS_PREFERENCE));
}

/**
 * Returns the bad words preference.
 */
public String[] getBadWordsPreference() {
        return convert(getPreferenceStore().getString(BAD_WORDS_PREFERENCE));
}

/**
 * Converts PREFERENCE_DELIMITER delimited String to a String array.
 */
private String[] convert(String preferenceValue) {
        StringTokenizer tokenizer =
                new StringTokenizer(preferenceValue, PREFERENCE_DELIMITER);
        int tokenCount = tokenizer.countTokens();
        String[] elements = new String[tokenCount];
        for (int i = 0; i < tokenCount; i++) {
                elements[i] = tokenizer.nextToken();
        }

        return elements;
}

/**
 * Sets the bad words preference.
 */
public void setBadWordsPreference(String[] elements) {
        StringBuffer buffer = new StringBuffer();
        for (int i = 0; i < elements.length; i++) {
                buffer.append(elements[i]);
                buffer.append(PREFERENCE_DELIMITER);
        }
        getPreferenceStore().setValue(BAD_WORDS_PREFERENCE, buffer.toString());
}
```

There is no field editor defined in JFace for editing String arrays so we will define a list that shows the items with widgets to add and remove them. Our performOK method will send the current contents of the list to the setBadWordsPreference method and the performDefaults method will reset the list of strings to be the result of getDefaultBadWordsPreference. Both methods are defined in BadWordCheckerPlugin. As a List widget takes an array of Strings as its content we can use the results of these helper methods directly in conjunction with the methods we defined for the bad words preference in the plug-in. The performOK and performDefaults for this preference page use these methods to update the preference and reset the values in the list widget respectively. See Figure 3 for the Bad Words preference page.

```
/**
 * Sets the contents of the nameEntry field to be the default
 */
protected void performDefaults() {
        badWordList.setItems(BadWordCheckerPlugin.getDefault().getDefaultBadWordsPreference());
}
/**
 * Saves the author name to the preference store.
 */
public boolean performOk() {
        BadWordCheckerPlugin.getDefault().setBadWordsPreference(badWordList.getItems());
        return super.performOk();
}
```

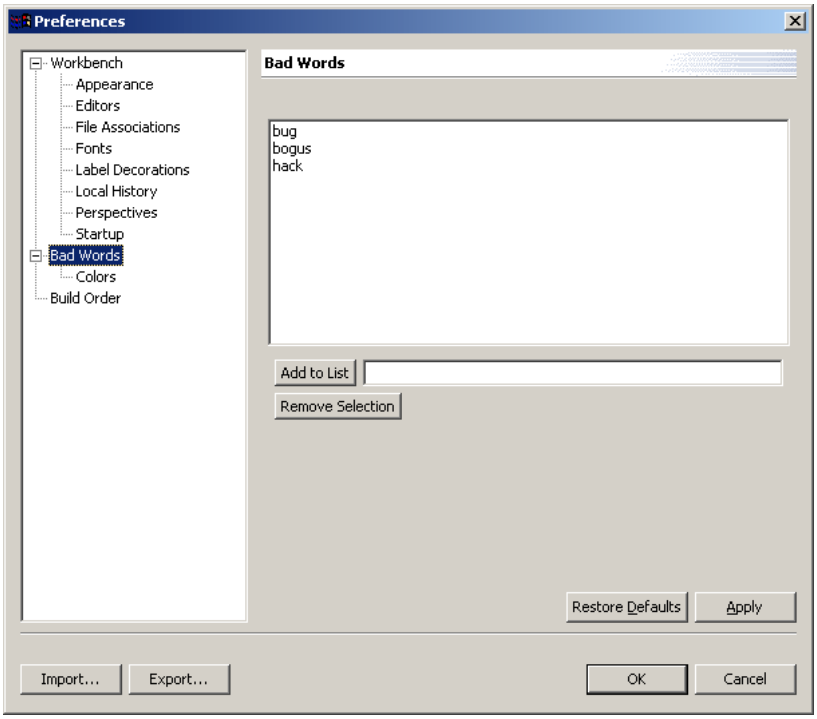

Figure 3: Preference dialog showing Bad Words preference page

Propagating Values With IPropertyChangeListener

Frequently a preference is used to set a value in another object or needs to be applied to an open editor or view. When this is required you can listen for these changes with an IPropertyChangeListener. IPropertyChangeListener is a class that is used to add a listener to an IPropertyStore so that the listener is informed whenever a change is made. Change notifications are issued whenever a preference is changed in the preference store with setValue(); this typically happens when the user hits OK or Apply in a preference dialog, or when a previously saved preference setting is imported.

In the bad word checker example we have implemented a view that displays the bad words for a file highlighted in the selected Color (see attached code). This view has a IPropertyChangeListener defined on it that updates the display color when it changes (see 1). When the view is created it is added as an IPropertyChangeListener on the preference store in the init(IViewSite) method (see 2). We also remove it as a listener on the preference store when we dispose of it so that any further changes to the preference store do not try and refer to a disposed page (see 3). The init(IViewSite) method is defined on IViewPart and the dispose() method is defined on IWorkbenchPart.

```
1 new IPropertyChangeListener() {
        public void propertyChange(PropertyChangeEvent event) {
                if (event.getProperty().equals(BadWordCheckerPlugin.HIGHLIGHT_PREFERENCE)) {
                        //Update the colors by clearing the current color,
                        //updating the view and then disposing the old color.
                        Color oldForeground = foreground;
                        foreground = null;
                        setBadWordHighlights(text.getText());
                        oldForeground.dispose();
                }
                if (event.getProperty().equals(BadWordCheckerPlugin.BAD_WORDS_PREFERENCE))
                        //Only update the text if only the words have changed
                        setBadWordHighlights(text.getText());
        }
};
```

```
2 public void init(IViewSite site) throws PartInitException {
        super.init(site);
        site.getPage().addSelectionListener(...);
        BadWordCheckerPlugin
                .getDefault()
                .getPreferenceStore()
                .addPropertyChangeListener(preferenceListener);}
```

```
3 public void dispose() {
        getSite().getPage().removeSelectionListener(...);
        BadWordCheckerPlugin
                .getDefault()
                .getPreferenceStore()
                .removePropertyChangeListener(preferenceListener);
        if (foreground != null)
                foreground.dispose();
        super.dispose();
}
```

Importing and Exporting Preference Settings

As of Eclipse 2.0 there is now a facility to import and export your preferences so that you can reload them when you get a new workspace. This can be done by use of the Import and Export buttons on the preferences dialog. The currently set preferences are stored in a .epf file that you specify when you export. Any preference that is still set to it's default value will not be saved in this file. When you import from a .epf file any preferences defined in that file will be set to the value stored. Preferences not stored in the .epf file are not affected.

If your preferences just use the preference store for storing and retrieving values then there is no more work to do to when writing your preferences as they will be saved and restored as part of the import and export support for preference stores. Should you need to have extra functionality executed when preferences are imported you can use an IPropertyChangeListener

Conclusions

In this article we have demonstrated how to use the preferences store and preferences pages provided by Eclipse to allow a plug-in to maintain and update preferences between Eclipse sessions and to import and export them using the preference dialog. By use of the preference store in conjunction with the preferences dialog and provided field editors a plug-in developer can quickly put together a user interface for managing preferences. To find out more about the preferences that Eclipse provides see the Platform Plug-in Developers Guide in the Help Perspective. The help information is in the Programmers Guide Preferences and Properties section.

The full implementation of the example in this article can be found in **preferences.zip (preferences.zip)**.

Back to the top