Eclipse Preferences - Tutorial

Lars Vogel (c) 2009, 2016 vogella GmbH – Version 2.3, 06.07.2016

Table of Contents

1. Eclipse Preference basics
2. Workspace Location
3. Preferences and dependency injection
4. Preference handling in Eclipse 3.x
5. Persistence of part state
6. Assumptions
7. Setting preferences via plugin_customization.ini
8. Tutorial: Preferences via code
9. Exercise: Contribute a preference page to the Eclipse IDE
10. Tutorial: Secure storage of preferences
11. About this website
12. Links and Literature

Appendix A: Copyright and License

Eclipse Preferences. This article describes the usage of Eclipse preferences and preference pages. Within this tutorial Eclipse 4.5 (Mars) is used.

1. Eclipse Preference basics

1.1. Preferences and scopes

The Eclipse platform supports *preferences* for persisting data between application restarts. Preferences are stored as key / value pairs. The key is an arbitrary String. The value can be a boolean, String, int or another primitive type. For example the *user* key may point to the value *vogella*.

The preference support in Eclipse is based on the `Preferences` class from the `org.osgi.service.prefs` package. Eclipse preferences are very similar to the standard Java preferences API but use the Eclipse framework to save and retrieve the configuration and support *scopes*.

The scope defines how the preference data is stored and how it is changeable. The Eclipse runtime defines three scopes as explained in the following table.

Table 1. Eclipse Preference scope

Scope	Description
Instance scope	Preferences in this scope are specific to a single Eclipse workspace. If the user runs start same Eclipse application for different workspaces, the settings between the applications can be different.

Configuration scope	Settings for identical for the same installation. Preferences stored in this scope are shared across all workspaces.
Default scope	Default values can not be changed. This scope is not stored on disk at all but can be used to store default values for all your keys. These preferences are supplied via configuration files in plug-ins and product definitions.
BundleDefaultsScope	Similar to the default scope, these values are not written to disk. They are read from a file, typically named *preferences.ini*.

1.2. Storage of the preferences

Eclipse stores the preferences in the workspace of your application in the *.metadata/.plugins/org.eclipse.core.runtime/.settings/* directory in the *<nodePath>.prefs* file.

The *<nodePath>* is by default the Bundle-SymbolicName of the plug-in but can be specified via the preference API. The workspace is by default the directory in which the application starts.

You can configure the storage location of the preferences via the *-data path* launch parameter in Eclipse. To place the preferences in the user home directory use the *-data @user.home* parameter setting.

1.3. Eclipse preference API

You can create and manipulate preferences directly via Singletons provided by the Eclipse runtime. You have the `InstanceScope`, `ConfigurationScope` and `DefaultScope` classes which give access to the corresponding instance via the `INSTANCE` field.

Preference values are read and saved by `get()` and `put()` methods. In the `get()` method you specify a default value in case the key can not be found. The `clear()` method removes all preferences and the `remove()` method allows you to delete a selected preference value. Via the `flush()` method you persist the preferences to the file system.

JAVA

```
// We access the instanceScope
Preferences preferences = InstanceScope.INSTANCE
    .getNode("com.vogella.eclipse.preferences.test");

Preferences sub1 = preferences.node("node1");
Preferences sub2 = preferences.node("node2");
sub1.put("h1", "Hello");
sub1.put("h2", "Hello again");
sub2.put("h1", "Moin");
try {
    // forces the application to save the preferences
    preferences.flush();
    } catch (BackingStoreException e) {
        e.printStackTrace();
    }
}
```

```
    .getNode("com.vogella.eclipse.preferences.test");
Preferences sub1 = preferences.node("node1");
Preferences sub2 = preferences.node("node2");
sub1.get("h1", "default");
sub1.get("h2", "default");
sub2.get("h1", "default");
```


2. Workspace Location

When starting the Eclipse IDE first of all a workspace location has to be specified. There are several ways to specify a workspace location in an RCP application as well, which might be more suitable for an end user than having a dialog, which asks for a workspace location.

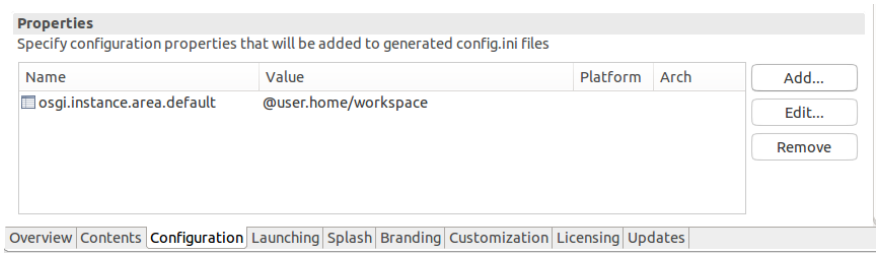
The default workspace location is the install location of the Eclipse RCP application.

2.1. Using the osgi.instance.area.default property

The `osgi.instance.area.default` property can be used to declaratively define an alternative workspace location.

The property can be specified in the *config.ini* file of the RCP application.

Usually properties of the *config.ini* file are specified in a product configuration:



It is also possible to use variables like `@user.home` as value. This is especially useful when an enviroment like Citrix is used, where several users might use the same application, but should have a different workspace.

2.2. Setting the workspace location programmatically

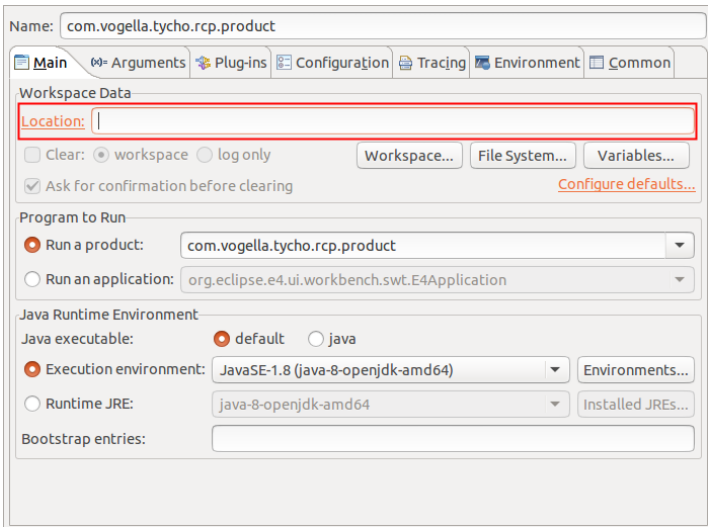
Many RCP applications have a login screen at startup and the actual workspace location should be set according to the user being logged in.

In E4 applications a login screen is usually created in a lifecycle class in the method annotated with `@PostContextCreate` .

```
import java.net.URL;

import org.eclipse.core.runtime.Platform;
import org.eclipse.e4.ui.workbench.lifecycle.PostContextCreate;

public class LifeCycleManager {

    @PostContextCreate
    public void postContextCreate() throws IllegalStateException,
IOException {
        // Show login dialog to the user
        String userName = // get username from login dialog;

        // check if the instance location is already set,
        // otherwise setting another one will throw an IllegalStateException
        if (!Platform.getInstanceLocation().isSet()) {
            String defaultPath = System.getProperty("user.home");

            // build the desired path for the workspace
            String path = defaultPath + "/" + userName + "/workspace/";
            URL instanceLocationUrl = new URL("file", null, path);
            Platform.getInstanceLocation().set(instanceLocationUrl, false);
        }
    }

}
```


The instance location can only be set once! Therefore the if statement with `if(Platform.getInstanceLocation().isSet())` is used.

When running the RCP application during development from the Eclipse IDE there usually already is a workspace in the run configuration. To be able to set the location during development the *Location* field must be empty.



3. Preferences and dependency injection

The Eclipse platform allows you to use dependency injection for preferences handling. To access preference you use the `@Preference` annotation as qualifier for the dependency injection annotation. This means that `@Preference` must be

The `@Preference` annotation allows you to specify the *nodePath* and the *value* as optional parameters.

The *nodePath* is the file name used to save the preference values to disk. By default this is the Bundle-SymbolicName of the plug-in. The *value* parameter specifies the preference key for the value which should be injected.

Eclipse can also inject the `IEclipsePreference` object. You can use this object for storing values. If you use the *value* parameter, Eclipse injects the value directly. Use the *value* parameter for read access, while for storing or changing values, use the `IEclipsePreference` object.

The following code snippet demonstrates how to put values into the preferences store. Please note that `@Preference` is used in combination with `@Execute`.

JAVA

```
// get IEclipsePreferences injected to change a value
@Execute
public void execute
    (@Preference(nodePath = "com.example.e4.rcp.todo") IEclipsePreferences
prefs) {
    // more stuff...
    prefs.put("user", "TestUser");
    prefs.put("password", "Password");
    // Persists
    try {
    prefs.flush();
        } catch (BackingStoreException e) {
            e.printStackTrace();
        }
}
```

The next snippet demonstrates the read access of preference values. This time the preference annotation is used a qualifier for `@Inject`.

JAVA

```
@Inject
@Optional
public void trackUserSettings
    (
    @Preference(nodePath = "com.example.e4.rcp.todo",
    value = "user")
    String user) {
    System.out.println("New user: " + user);
}

@Inject
@Optional
public void trackPasswordSettings
    (
    @Preference(nodePath = "com.example.e4.rcp.todo",
    value = "password")
    String password) {
    System.out.println("New password: " + password);
}
```

The Eclipse platform automatically tracks the values and re-injects them into fields and methods if they change. Eclipse tracks changes of preferences in the `InstanceScope` scope. Preference values in the `ConfigurationScope` and `DefaultScope` are not tracked.

If you use the injected `IEclipsePreference` to store new preference values, these values are stored in the instance scope.

4. Preference handling in Eclipse 3.x

4.1. Preference Page

Eclipse 3.x provides a standard dialog to display and change preference values via a preference dialog. Please note that this dialog does not work for Eclipse 4 RCP applications.

To add a page to this preference dialog a plug-in must provide an contribution to the `org.eclipse.ui.preferencePages` extension point.

This extension point defines a class which is responsible for creating a user interface and storing the preference values. This class must implement `IWorkbenchPreferencePage` and must have a non-parameter constructor. The *keywordReference id* attribute in this extension point can be used to define search terms for the Eclipse IDE search field for preferences.

The `PreferencePage` class or one of its subclasses can get extended; a good template is usually `FieldEditorPreferencePage`.

To open the Preference dialog, you can use the `org.eclipse.ui.window.preferences` command.

4.2. Secure storage of preferences

Eclipse allows to encrypt preference values via the `org.eclipse.equinox.security` plug-in.

The key / value pairs will be stored in the *secure.storage* file in the *.eclipse/org.eclipse.equinox.security* folder of the users home directory. Eclipse uses a class of type `PasswordProvider` for encrypting the preferences and has a default class registered.

Via the `org.eclipse.equinox.security.secureStorage` extension point you can register your own PasswordProvider.

4.3. Access Preferences in different plug-ins

You can access preferences in other plug-ins via the `PreferenceService` service.

For example, to access the "MySTRING1" preference in the "de.vogella.preferences.page" plug-in, you can use the following:

JAVA

```
String text = Platform.getPreferencesService().
    getString("de.vogella.preferences.page", "MySTRING1", "hello", null);
```

4.4. Reacting to changes in the preferences

You can register `IPropertyChangeListener` instances to changes in the preference values. These listener are called by the Eclipse framework if the reference value changes.

```
        @Override
        public void propertyChange(PropertyChangeEvent event) {
            if (event.getProperty() == "MySTRING1") {
                String value = event.getNewValue().toString()
                // do something with the new value
            }
        }
    });
```


5. Persistence of part state

Eclipse provides the `@PersistState` annotation. This annotation can be applied to a method in a class referred to a part.

Such an annotated method can be used to store the instance state of the part. The Eclipse framework calls such a method whenever the part or the application closes. The stored information can be used in the method annotated with the `@PostConstruct` annotation. A typical use case for such a method would be to store the state of a checkbox.

The usage of this annotation is demonstrated in the following example code.

JAVA

```
@PostConstruct
public void createControl(MPart part) {
    Map<String, String> state = part.getPersistedState();
    String value = state.get("key");
    ...
}

@PersistState
public void persistState(MPart part) {
    Map<String, String> state = part.getPersistedState();
    state.put("key", "newValue");
    ...
}
```

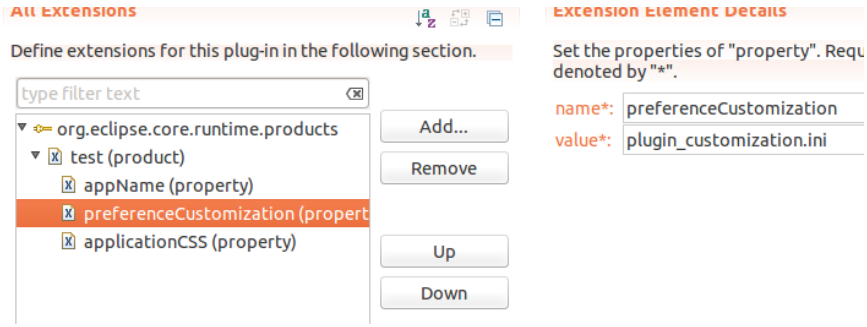
6. Assumptions

The following assumes that you know how to create simple Eclipse RCP (http://www.vogella.com/tutorials/RichClientPlatform/article.html) applications and that you know how to create commands (http://www.vogella.com/tutorials/EclipseCommands/article.html) and add them to the menu. If not please have a look at the Eclipse RCP (http://www.vogella.com/tutorials/RichClientPlatform/article.html) and Eclipse Commands (http://www.vogella.com/tutorials/EclipseCommands/article.html) tutorial .

7. Setting preferences via plugin_customization.ini

You can use a file to set the default values of preferences. The file which contains these defaults is typically named *plugin_customization.ini*.

Such a file needs to be registered via the *preferenceCustomization* property on the product extension point in the *plugin.xml* file. This is demonstrated in the following screenshot.



The format to use is `<plugin id>/<setting>=<value>` , e.g. *com.example.e4.rcp.todo/user=vogella*.

To find the correct keys, just start your Eclipse application, switch to the *.metadata* directory in your workspace directory (by default the directory your application is starting in) and search for files ending with *.pref*.

8. Tutorial: Preferences via code

You can create, store and retrieve preference values directly via your coding. The following gives an example for this. Create an Eclipse composite with three buttons. Use this composite in one of your parts (View or Editor).

The first `Button` will set the preference values. The next will display the values and the last will clear the preference values.

```
import org.eclipse.core.runtime.preferences.ConfigurationScope;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.osgi.service.prefs.BackingStoreException;
import org.osgi.service.prefs.Preferences;

public class ButtonComposite extends Composite {

    public ButtonComposite(Composite parent, int style) {
        super(parent, style);
        Button write = new Button(parent, SWT.PUSH);
        write.setText("Write");
        write.addSelectionListener(new SelectionAdapter() {
            @Override
            public void widgetSelected(SelectionEvent e) {

                Preferences preferences = ConfigurationScope.INSTANCE
                        .getNode("de.vogella.preferences.test");
                Preferences sub1 = preferences.node("node1");
                Preferences sub2 = preferences.node("node2");
                sub1.put("h1", "Hello");
                sub1.put("h2", "Hello again");
                sub2.put("h1", "Moin");

                try {
                    // forces the application to save the preferences
                    preferences.flush();
                } catch (BackingStoreException e2) {
                    e2.printStackTrace();
                }
            }
        });
        Button read = new Button(parent, SWT.PUSH);
        read.setText("Read");
        read.addSelectionListener(new SelectionAdapter() {
            @Override
            public void widgetSelected(SelectionEvent e) {
                Preferences preferences = ConfigurationScope.INSTANCE
                        .getNode("de.vogella.preferences.test");
                Preferences sub1 = preferences.node("node1");
                Preferences sub2 = preferences.node("node2");
                System.out.println(sub1.get("h1", "default"));
                System.out.println(sub1.get("h2", "default"));
                System.out.println(sub2.get("h1", "default"));

            }
        });

        Button clear = new Button(parent, SWT.PUSH);
        clear.addSelectionListener(new SelectionAdapter() {
            @Override
            public void widgetSelected(SelectionEvent e) {
                Preferences preferences = ConfigurationScope.INSTANCE
                        .getNode("de.vogella.preferences.test");
                Preferences sub1 = preferences.node("node1");
                Preferences sub2 = preferences.node("node2");
                // Delete the existing settings
                try {
                    sub1.clear();
                    sub2.clear();
                    preferences.flush();
                } catch (BackingStoreException e1) {
                    e1.printStackTrace();
                }
            }
        });
        clear.setText("clear");
    }
}
```

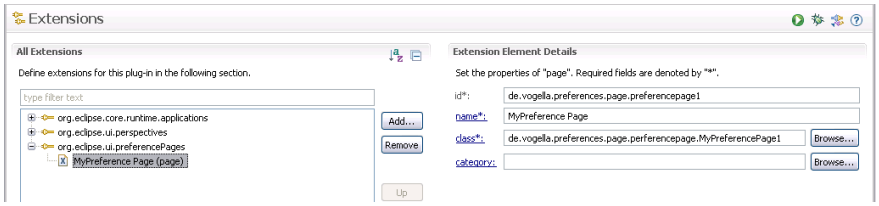
9. Exercise: Contribute a preference page to the Eclipse IDE

This exercise is specific to the Eclipse IDE and will not work for Eclipse 4 RCP applications. See for an Eclipse 4 implementation (https://github.com/opcoach/e4Preferences).

In the following exercise you create a plug-in with a preference pages which allows the user to maintain certain settings.

Create a new simple plug-in project called `com.vogella.preferences.page`. No activator is needed and you do not have to use a template.

Open the *MANIFEST.MF* editor and click on the *Extensions* link on the *Overview* tab. On this tab, add the extension `org.eclipse.ui.preferencePages` with the following settings.

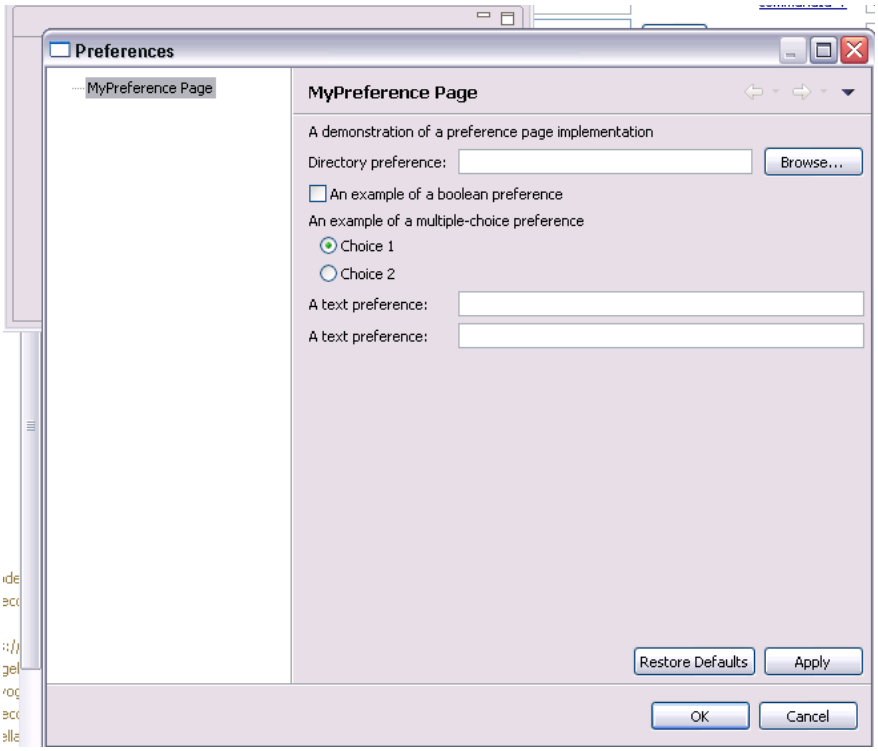


Enter the following code for your class `MyPreferencePage1`. Method `init()` sets the preferences store and the method `createFieldEditors()` registers pre-defined editors for values. `checkState()` allows to perform a validations. To get notified about value changes you need to override the propertyChange method.

```
import org.eclipse.jface.preference.BooleanFieldEditor;
import org.eclipse.jface.preference.DirectoryFieldEditor;
import org.eclipse.jface.preference.FieldEditorPreferencePage;
import org.eclipse.jface.preference.RadioGroupFieldEditor;
import org.eclipse.jface.preference.StringFieldEditor;
import org.eclipse.ui.IWorkbench;
import org.eclipse.ui.IWorkbenchPreferencePage;

import de.vogella.preferences.page.Activator;

public class MyPreferencePage1 extends FieldEditorPreferencePage implements
        IWorkbenchPreferencePage {

    public MyPreferencePage1() {
        super(GRID);

    }

    public void createFieldEditors() {
        addField(new DirectoryFieldEditor("PATH", "&Directory preference:",
                getFieldEditorParent()));
        addField(new BooleanFieldEditor("BOOLEAN_VALUE",
                "&An example of a boolean preference",
getFieldEditorParent()));

        addField(new RadioGroupFieldEditor("CHOICE",
                "An example of a multiple-choice preference", 1,
                new String[][] { { "&Choice 1", "choice1" },
                        { "C&hoice 2", "choice2" } },
getFieldEditorParent()));
        addField(new StringFieldEditor("MySTRING1", "A &text preference:",
                getFieldEditorParent()));
        addField(new StringFieldEditor("MySTRING2", "A &text preference:",
                getFieldEditorParent()));
    }

    @Override
    public void init(IWorkbench workbench) {
        setPreferenceStore(Activator.getDefault().getPreferenceStore());
        setDescription("A demonstration of a preference page
implementation");
    }
}
```

Start a runtime Eclipse IDE with your plug-in and ensure that you see your preference page in the preference settings of the Eclipse IDE.

Validate that maintained values are stored even if you restart your application.



Add a command `showPreferenceValues` with the following handler to the menu. This command demonstrates how to access preferences values from the `preferencePage`.

JAVA

```
public class ShowPreferenceValues {

    @Execute
    public Object execute(Shell shell) {
        String myPrefString = Activator.getDefault().getPreferenceStore()
                .getString("MySTRING1");
        MessageDialog.openInformation(shell, "Info", myPrefString);
        Boolean myPrefBoolean = Activator.getDefault().getPreferenceStore()
                .getBoolean("BOOLEAN_VALUE");
        // RadioGroupFieldEditor can get access
        String choice =
Activator.getDefault().getPreferenceStore().getString("CHOICE");
        System.out.println(choice);
        MessageDialog.openInformation(shell, "Info",
myPrefBoolean.toString());
        // I assume you get the rest by yourself
    }

}
```

To set the default values for preferences use the extension point `org.eclipse.core.runtime.preferences`. Create a new initializer with the following class `de.vogella.preferences.page.preferencepage.MyInitializer`.

```
import org.eclipse.core.runtime.preferences.AbstractPreferenceInitializer;
import org.eclipse.jface.preference.IPreferenceStore;

import de.vogella.preferences.page.Activator;

public class MyInitializer extends AbstractPreferenceInitializer {

    public MyInitializer() {
    }

    @Override
    public void initializeDefaultPreferences() {
        IPreferenceStore store =
Activator.getDefault().getPreferenceStore();
        store.setDefault("MySTRING1", "http://www.vogella.com");
    }

}
```

Finally create a new view to show one of the preference values. Also register a `PropertyChangeListener` to the preferences store to get informed in case the preference settings change.

JAVA

```
package de.vogella.preferences.page;

import org.eclipse.jface.preference.IPreferenceStore;
import org.eclipse.jface.util.IPropertyChangeListener;
import org.eclipse.jface.util.PropertyChangeEvent;
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.ui.part.ViewPart;

public class View extends ViewPart {

    private Label label;

    public void createPartControl(Composite parent) {
        IPreferenceStore preferenceStore = Activator.getDefault()
                .getPreferenceStore();
        String string = preferenceStore.getString("MySTRING1");

        label = new Label(parent, SWT.NONE);
        label.setLayoutData(new GridData(SWT.BEGINNING, SWT.CENTER, false,
                false));
        label.setText(string);
        // add change listener to the preferences store so that we are
notified
        // in case of changes
        Activator.getDefault().getPreferenceStore()
                .addPropertyChangeListener(new IPropertyChangeListener() {
                    @Override
                    public void propertyChange(PropertyChangeEvent event) {
                        if (event.getProperty() == "MySTRING1") {
                            label.setText(event.getNewValue().toString());
                        }
                    }
                });
    }

    public void setFocus() {
    }
}
```

10. Tutorial: Secure storage of preferences

`org.eclipse.equinox.security` plug-in as a dependency to it.

Change the code of your view to the following.

JAVA

```
package de.vogella.preferences.security;

import org.eclipse.equinox.security.storage.ISecurePreferences;
import org.eclipse.equinox.security.storage.SecurePreferencesFactory;
import org.eclipse.equinox.security.storage.StorageException;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionAdapter;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.ui.part.ViewPart;

public class View extends ViewPart {
    public void createPartControl(Composite parent) {
        Button buttonPut = new Button(parent, SWT.PUSH);
        buttonPut.setText("Save values");
        buttonPut.addSelectionListener(new SelectionAdapter() {
            @Override
            public void widgetSelected(SelectionEvent e) {
                ISecurePreferences preferences = SecurePreferencesFactory
                        .getDefault();
                ISecurePreferences node = preferences.node("info");
                try {
                    node.put("user", "vogella", true);
                    node.put("password", "123", true);
                } catch (StorageException e1) {
                    e1.printStackTrace();
                }
            }
        });
        Button buttonGet = new Button(parent, SWT.PUSH);
        buttonGet.setText("Get values");
        buttonGet.addSelectionListener(new SelectionAdapter() {
            @Override
            public void widgetSelected(SelectionEvent e) {
                ISecurePreferences preferences = SecurePreferencesFactory
                        .getDefault();
                if (preferences.nodeExists("info")) {
                    ISecurePreferences node = preferences.node("info");
                    try {
                        String user = node.get("user", "n/a");
                        String password = node.get("password", "n/a");
                        System.out.println(user);
                        System.out.println(password);
                    } catch (StorageException e1) {
                        e1.printStackTrace();
                    }
                }
            }
        });

    }

    /**
     * Passing the focus request to the viewer's control.
     */
    public void setFocus() {
    }
}
```

11. About this website

(http://www.vogella.com/support.html) (http://www.vogella.com/) (http://www.vogella.com/code/index.html)

12. Links and Literature

Nothing listed.

12.1. vogella GmbH training and consulting support

TRAINING (http://www.vogella.com/training/)	**SERVICE & SUPPORT** (http://www.vogella.com/consulting/)
The vogella company provides comprehensive training and education services (http://www.vogella.com/training/) from experts in the areas of Eclipse RCP, Android, Git, Java, Gradle and Spring. We offer both public and inhouse training. Whichever course you decide to take, you are guaranteed to experience what many before you refer to as "The best IT class I have ever attended" (http://www.vogella.com/training/).	The vogella company offers expert consulting (http://www.vogella.com/consulting/) services, development support and coaching. Our customers range from Fortune 100 corporations to individual developers.

Appendix A: Copyright and License

Version 2.3
Last updated 2017-05-03 21:28:46 +02:00