IBM Lotus Symphony Developers Tutorial - Building a simple document workflow plug-in

Note: Before using this information and the product it supports, read the informationin Notice page.

Symphony Beta 4 Edition (Jan 2008)

This edition applies to release Beta 4 of IBM Lotus Symphony toolkit (license number L-AENR-7AYBE2) and to all subsequent releases and modifications until otherwise indicated in new editions.

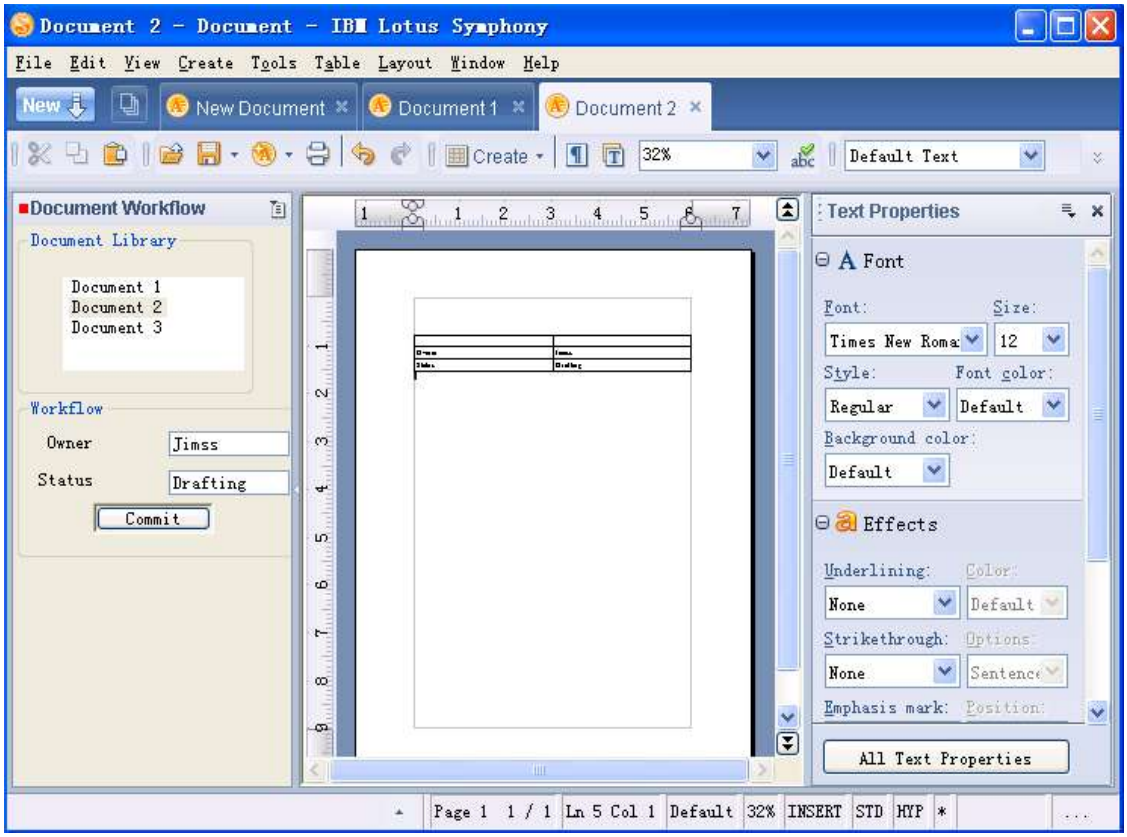
Table of contents

Overview

When many people think of IBM Lotus Symphony, they rightly think of it as a group of office productivity editors, but it fact it is much more. Because Symphony is based on OpenOffice.org technology and the Eclipse / Lotus Expeditor rich client platform, it's by definition an extensible product. Lotus Partners take advantage of this flexibility to add new features that enhance the base product's capabilities, delivered as a natural extension of Symphony's user interface. As you will learn in this lab, it is also possible to add business-specific capabilities beyond editing. Whether you are enhancing the base editing features or adding business-specific capabilities, the approach is the same: Develop a plug-in that “hooks into” Symphony. The purpose of this lab is to demonstrate how plug-in are used to extend the functionality of the office suite in a variety of ways.

In this lab you will create a simple document workflow plug-in. The document workflow application provides a way to apply a simple workflow to your office documents in document library. When a document is opened, the document is opened in Lotus Symphony window and the workflow information is displayed in the application. Workflow information can be changed and stored in the document. The document is built on a special template, and stored locally.

The following image shows the user interface with the Document Workflow plug-in.



The Document Workflow plug-in demonstrates key functionality, such as how to perform the following actions:

- Declare an extension to add an application to theside shelf area(Left pane above)
- Open a document with Lotus Symphony API
- Access and modify content of a document

You will create the plug-in from start to finish and use pre-created snippets of code in order to speed up the manual data entry process. This lab is intended for developers who are familiar with Java

programming and would like to try the plug-in development environment to extend Symphony. Eclipse and OpenOffice.org UNO programming knowledge are preferred but not necessary.

Outline

In order to create the Document Workflow application, you need to perform the following steps:

1. Verify Eclipse development environment
   - Run Lotus Symphony from Eclipse development environment
2. Creating Eclipse plug-in for Document Workflow application
   - Build a simple Eclipse plug-in which contributes a side shelf view to Lotus Symphony
   - Create the UI for the application with Eclipse's Standard Widget Toolkit(SWT)
3. Manipulating document in the library
   - Open document from library
4. Accessing and modifying document
   - Read workflow information when the document is loaded
   - Modify the workflow information

Part I

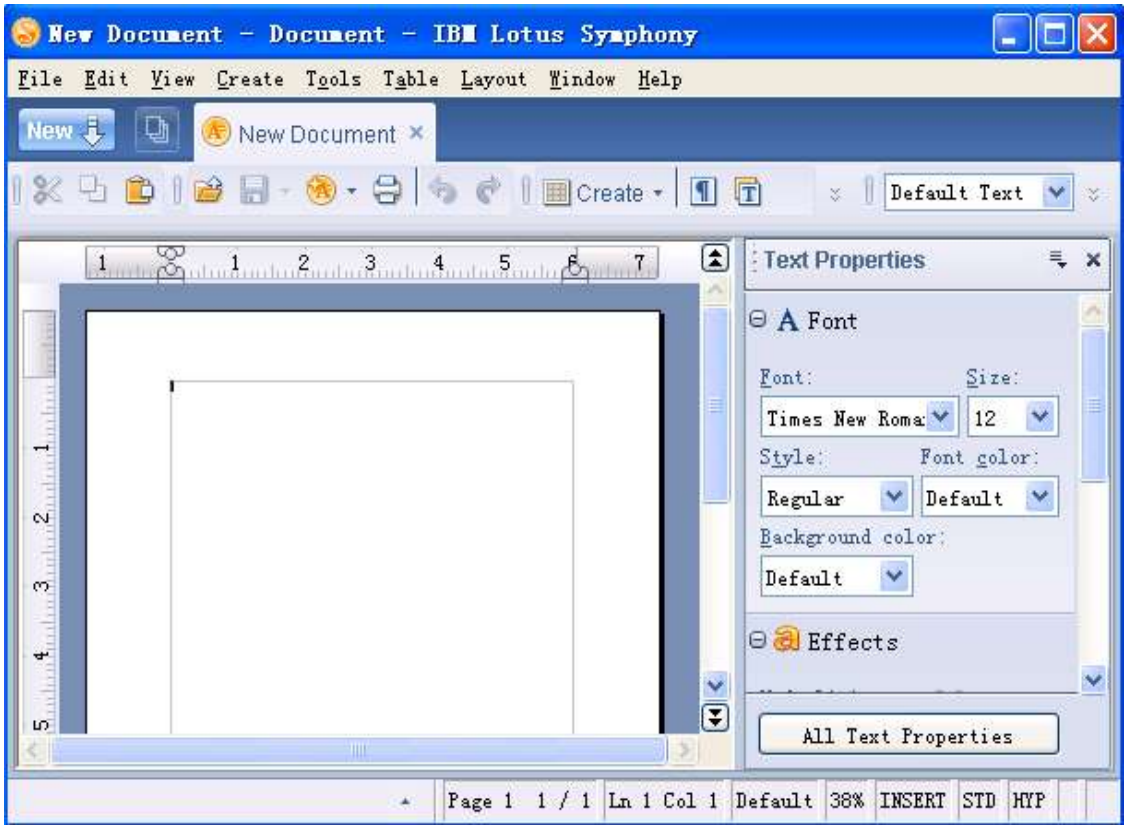
Verify Eclipse development environment

1. At first set up the Symphony development environment according **Symphony Developer's Guide** provided by the toolkit. Then switch to the Plug-in development environment by selecting **Window** > **Open Perspective** > **Other...**, then select **Plug-in Development**, click **OK**.

2. Click the "Run" button from the toolbar, Lotus Symphony should be able to be launched. If the Run option is disabled, select Run > Run... to open the runtime configuration dialog. Select Eclipse Application > Symphony and then the Run button. If asked if you want to clear the runtime workspace, select **No**.

Reminder: Once Symphony opens, close the Welcome page.

3. When Lotus Symphony window open, click File->New->Document, you will see the following window:



This is the standard Symphony document editor. In the next section you will add an Eclipse plug-in to the development environment and test that it works. Select **File** > **Exit** to close the runtime instance of Symphony before continuing.

Creating Eclipse plug-in for Document Workflow application

Create a new plug-in

1. Launch the Eclipse development environment
2. Click **File** > **New** > **Project**
3. Select "**Plug-in Project**", and click "**Next**"
4. Type **com.ibm.productivity.tools.sample.DocumentWorkflow** in the Project name field. Leave the rest of the default choices.
5. Click **Next**.
6. Type a descriptive name in the **Plug-in Name** field, e.g. **Document Workflow sample**. Keep the rest of the default values.
7. Click **Finish**.

Add the plug-in dependency

The following table lists some of the plug-in dependencies used by Document Library (plug-in names are abbreviated):

Plug-in	Description
org.eclipse.core.runtime, org.eclipse.ui	Eclipse core plug-ins
com.ibm.productivity.tools.ui.views	Lotus Symphony API plug-in
com.ibm.productivity.tools.core	
com.ibm.rcp.ui, com.ibm.rcp.jfaceex, com.ibm.rcp.swtex	RCP user interface APIs

Perform the following steps to add the plug-in dependency.

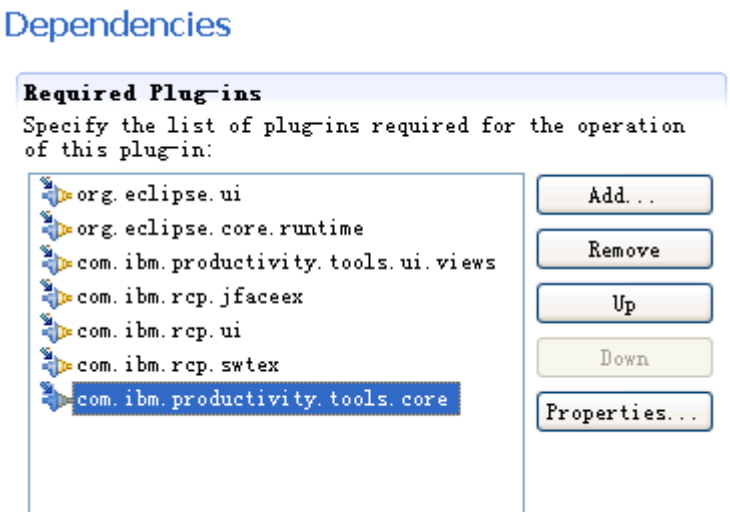
1 Click the **Dependencies** tab of the Document Workflow plug-in manifest.

2 Click **Add**.

3 Add the following plug-ins:

- **com.ibm.productivity.tools.ui.views**
- **com.ibm.productivity.tools.core**
- **com.ibm.rcp.ui**
- **com.ibm.rcp.jfaceex**
- **com.ibm.rcp.swtex**

Result: The screen should look like the following image (the plug-in order is not important)



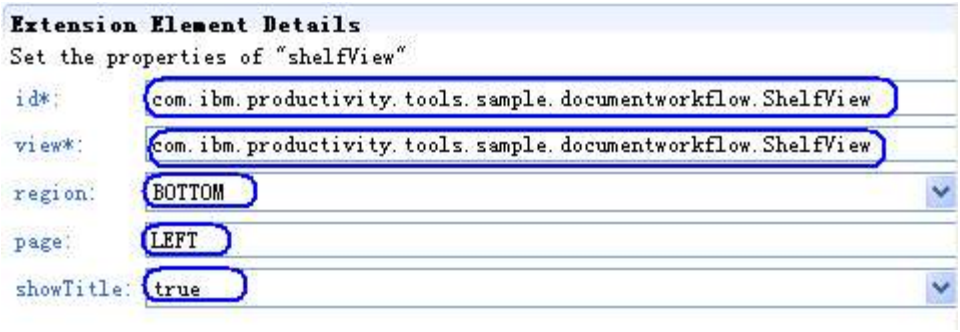
Note: Adding these plug-in dependencies adds the following to the MANIFEST.MF file, which defines the plug-in. You can see the contents of this file by turning to the Plug-in Manifest Editor's **MANIFEST.MF** tab:

```
Require-Bundle: org.eclipse.ui,
 org.eclipse.core.runtime,
 com.ibm.productivity.tools.ui.views,
 com.ibm.productivity.tools.core,
 com.ibm.rcp.jfaceex,
 com.ibm.rcp.ui,
 com.ibm.rcp.swtex
```

Contribute to side shelf

1. Click the **Extensions** tab.
2. Click **Add**.
3. Add the following extension: **com.ibm.rcp.ui.shelfViews**.
4. Click **Finish**.
5. Right click the added extension and select **New > shelfView**.

Selecting this menu choice will add a shelfview element to the extension declaration. Select the newly added element and note the Extension Element Details is updated to show the possible attributes. Fill in the fields as shown below.



The asterisk (*) indicates a required attribute. One of particular importance is the **class** attribute; this indicates the Java class that will implement the shelfview's behavior (i.e., this class defines what the side shelf area will contain, how it will respond to user events, etc.).

The id used in this document: com.ibm.productivity.tools.sample.documentworkflow.ShelfView. You may copy and paste it.

6 Click the **plugin.xml** tab.

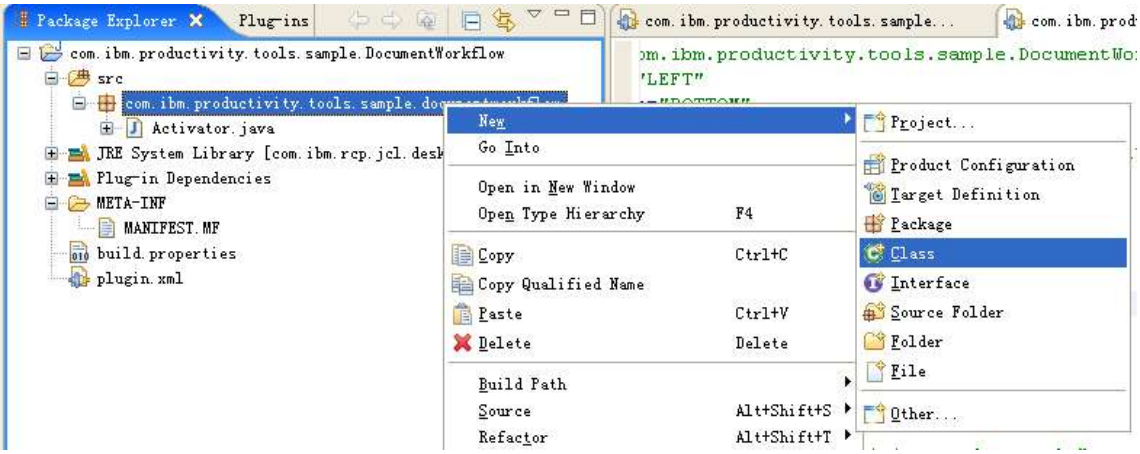
7 Copy and paste the following into the plugin.xml.

```
<extension
      point="org.eclipse.ui.views">

   <category
         name="Sample Category"
         id="com.ibm.productivity.tools.sample">
   </category>

   <view
         name="Document Workflow"
         icon=" "
         category="com.ibm.productivity.tools.sample"
         class="com.ibm.productivity.tools.sample.documentworkflow.ShelfView"
         id="com.ibm.productivity.tools.sample.documentworkflow.ShelfView">
   </view>

</extension>
```

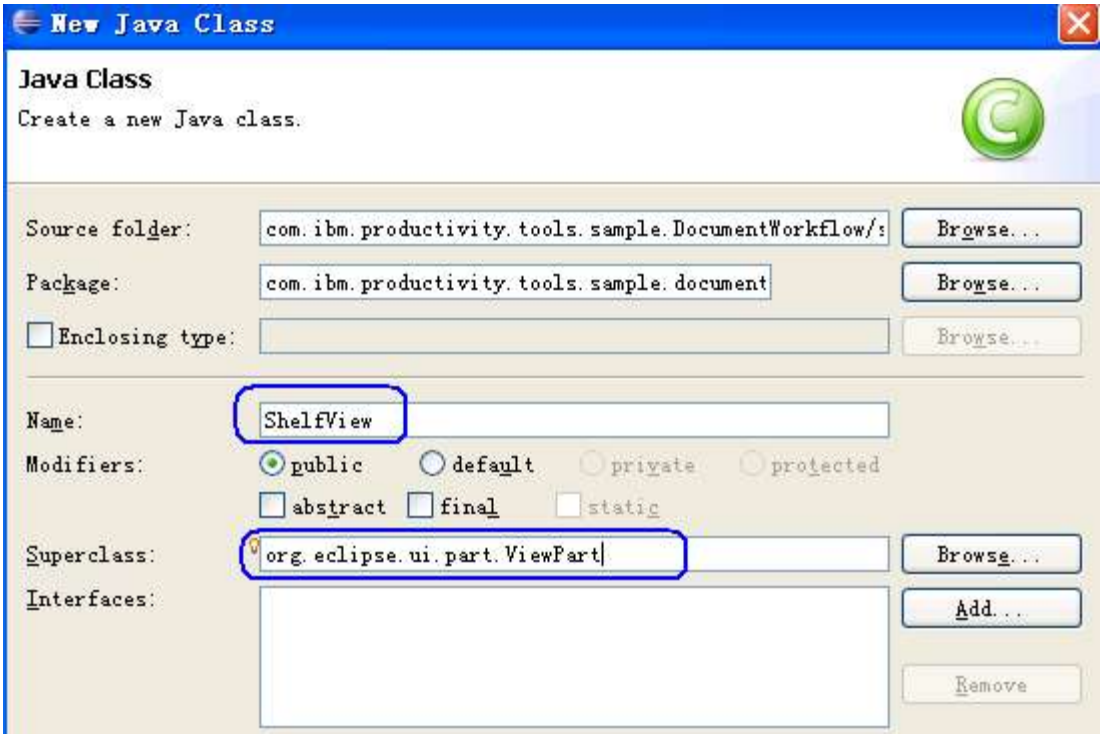
IMPORTANT: The view attribute of the <shelfView> tag in the com.ibm.rcp.ui.shelfViews extension must match the id attribute of the <view> tag in the org.eclipse.ui.views extension above exactly. That is, the side-shelf content is defined by the extensions' <view> / <shelfView> pairs.

The step above will contribute a new Eclipse ViewPart to the platform. You can create your plug-in extensions with Manifest Editor or enter the specifications directly in the plugin.xml.

8 Right click on the package "com.ibm.productivity.tools.sample.documentworkflow" in **Package Explorer**, select **New** > **Class**.



9 Input the class info as following, you can click the "**Browse…**" button to search the superclass of org.eclipse.ui.part.ViewPart.



Result: a new Eclipse ViewPart named ShelfView is created in the com.ibm.productivity.tools.samples.documentworkflow package.

Run the application

1. Check your plug-in

Before run the application, take a look at the plugin.xml and the newly created class:

The plugin.xml is like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
   <extension
         point="com.ibm.rcp.ui.shelfViews">
      <shelfView
            id="com.ibm.productivity.tools.sample.documentworkflow.ShelfView"
            page="LEFT"
            region="BOTTOM"
            showTitle="true"
            view="com.ibm.productivity.tools.sample.documentworkflow.ShelfView"/>
   </extension>

   <extension
         point="org.eclipse.ui.views">
      <category
            name="Sample Category"
            id="com.ibm.productivity.tools.sample">
      </category>
      <view
            name="Document Workflow"
            icon=" "
            category="com.ibm.productivity.tools.sample"
            class="com.ibm.productivity.tools.sample.documentworkflow.ShelfView"
            id="com.ibm.productivity.tools.sample.documentworkflow.ShelfView">
```

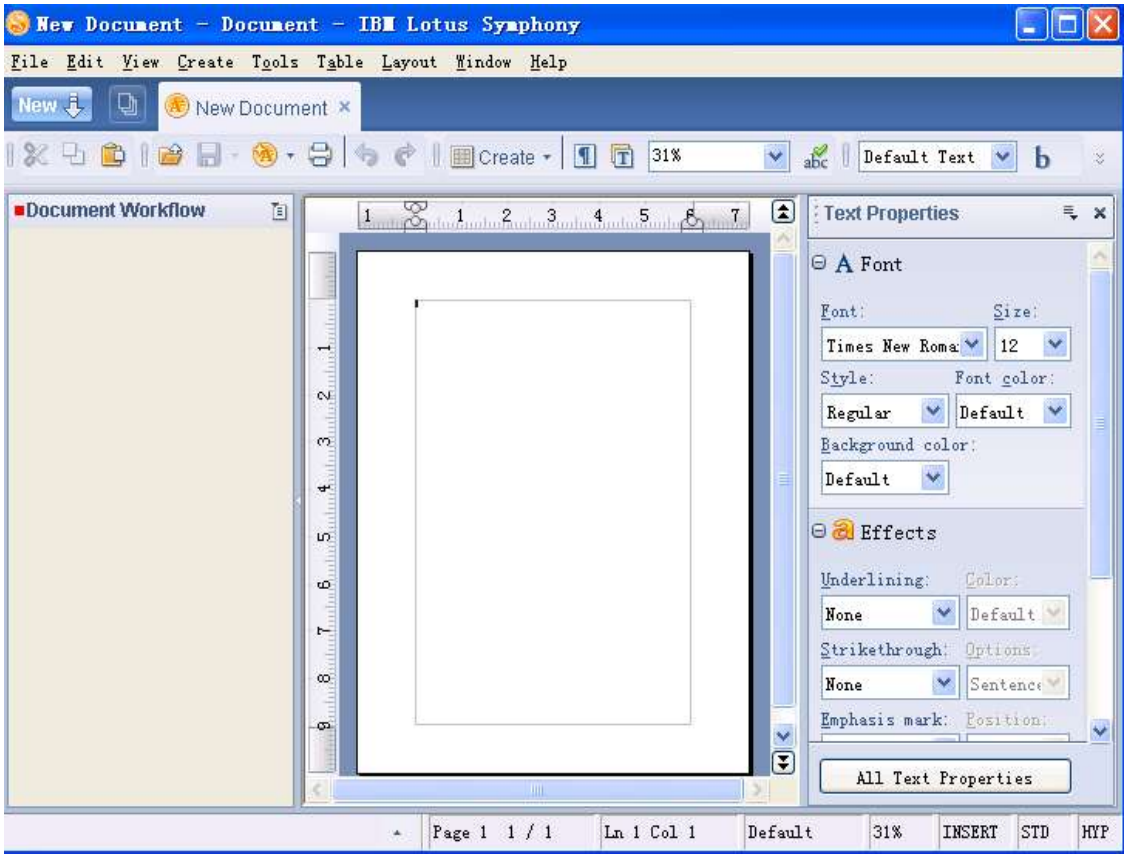
```
      </view>
   </extension>
</plugin>
```

Double Click the ShelfView.java in **Package Explorer**, the ShelfView.java looks like the following:

```
package com.ibm.productivity.tools.sample.documentworkflow;

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

public class ShelfView extends ViewPart {

    public void createPartControl(Composite arg0) {
           // TODO Auto-generated method stub

    }

    public void setFocus() {
           // TODO Auto-generated method stub

    }

}
```

2. Click the “Run” button from toolbar:

3. Lotus Symphony will be launched; Click File -> New -> Document, your screen should look similar to the following image:



Hint: If the newly contributed view does not display, check the Console for a message like “org.eclipse.ui.PartInitException: Could not create view: XXX” and confirm that XXX = the view id. The view attribute of the <shelfView> tag in the com.ibm.rcp.ui.shelfViews extension must match the id attribute of the <view> tag in the org.eclipse.ui.views extension.

Congratulations! You have reserved space in the Lotus Symphony side shelf for your application. Next, you will open document from the document library, and access, modify the document.

Part II

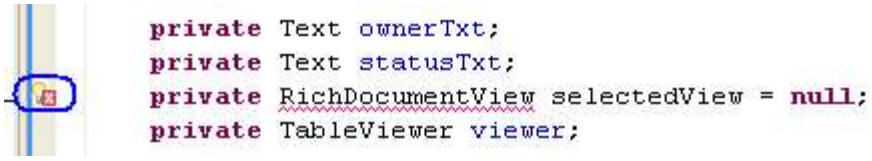
Before start Lotus Symphony APIs, let's add all the variables and methods that will be required for the Document Workflow application, and some helper methods that are not specific to Lotus Symphony. We can then turn to the code that is specific to extending Lotus Symphony. All the following code are within ShelfView.java.

Define variables

The variables are shown below. Copy and paste it into the beginning of ShelfView.java.

```
private Text ownerTxt;
private Text statusTxt;
private RichDocumentView selectedView = null;
private TableViewer viewer;
```

Note: Use the menu choice **Source > Organize Imports** (Ctrl+Shift+O) to add the needed import statements after pasting a code snippet. For example, in the following snippet, you will need to import com.ibm.productivity.tools.ui.views.RichDocumentView package.



IMPORTANT: TableViewer and Text are defined by several packages. Be certain to select org.eclipse.jface.viewers.TableViewer and org.eclipse.swt.widgets.Text.

Complete the createPartControl method

The Document Workflow contributes a viewable area to side shelf of Lotus Symphony. This area is defined in the createPartControl method using Eclipse's standard widgets, called SWT (Standard Widget Toolkit).

The SWT API is used to create user interface elements in Eclipse platform. Copy the following code and replace the createPartControl method in ShelfView.java.

The following code snippet creates a Document Library group and a Workflow group in side shelf, and other SWT controls.

```
   public void createPartControl(Composite parent) {
          parent.setLayout(new RowLayout());
          // Group of Document Library
          int y = 10 ;
          Group doclibGrp = new Group(parent, SWT.NULL);
          doclibGrp.setText("Document Library");
          doclibGrp.setSize(390, 72);
          doclibGrp.setLocation(10, y);

          viewer = new TableViewer(doclibGrp, SWT.MULTI
                        | SWT.H_SCROLL | SWT.V_SCROLL);
          viewer.getTable().setSize(120, 60);
          viewer.getTable().setLocation(30, y + 20);
          viewer.setContentProvider(new LibraryContentProvider());
          viewer.setLabelProvider(new LibraryLabelProvider());
          viewer.setInput(this );
          hookDoubleClickAction();


          // Group of user information
          y = 10 + doclibGrp.getBounds().height + 7;
          Group workflowGrp = new Group(parent, SWT.NULL);
          workflowGrp.setText("Workflow");
          workflowGrp.setSize(390, 72);
          workflowGrp.setLocation(10, y);

          Label ownerLbl = new Label(workflowGrp, SWT.SHADOW_NONE
                        | SWT.RIGHT);
          ownerLbl.setText("Owner");
          ownerLbl.setSize(40, 22);
          ownerLbl.setLocation(10, 22);

          Label statusLbl = new Label(workflowGrp, SWT.SHADOW_NONE
                        | SWT.RIGHT);
          statusLbl.setText("Status");
          statusLbl.setSize(40, 22);
          statusLbl.setLocation(10, 46);

          ownerTxt = new Text(workflowGrp, SWT.BORDER | SWT.LEFT);
          ownerTxt
                        .setBackground(new Color(this.getSite().
                                getShell().getDisplay(), 255, 255, 255));
          ownerTxt.setSize(80, 16);
          ownerTxt.setLocation(100, 21);

          statusTxt = new Text(workflowGrp, SWT.BORDER | SWT.LEFT);
          statusTxt.setBackground(new Color(this.getSite().
                        getShell().getDisplay(), 255,
                        255, 255));
          statusTxt.setSize(80, 16);
          statusTxt.setLocation(100, 46);

          Button commitBtn = new Button( workflowGrp, SWT.BORDER
                        | SWT.PUSH );
          commitBtn.setText("Commit");
          commitBtn.setSize(80, 20);
          commitBtn.setLocation(50, 67);
          commitBtn.addSelectionListener(new SelectionListener(){

                 public void widgetDefaultSelected(SelectionEvent
arg0) {
                        // TODO Auto-generated method stub

                 }

                 public void widgetSelected(SelectionEvent arg0) {


                 }

          });

   }
```

Note: Use the menu choice **Source > Organize Imports** (Ctrl+Shift+O) to add the needed import statements after pasting a code snippet. If there are methods and classes missing, please just continue copy and paste in the following document.

IMPORTANT: Label and SelectionEvent are defined by several packages. Be certain to select the ones defined in the SWT packages.

You will need to import the following packages, you may check the list or just copy and paste the following into import area;

```
import java.io.IOException;

import org.eclipse.core.runtime.FileLocator;
import org.eclipse.core.runtime.Platform;
import org.eclipse.jface.viewers.DoubleClickEvent;
import org.eclipse.jface.viewers.IDoubleClickListener;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.jface.viewers.IStructuredContentProvider;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.jface.viewers.ITableLabelProvider;
import org.eclipse.jface.viewers.LabelProvider;
import org.eclipse.jface.viewers.TableViewer;
import org.eclipse.jface.viewers.Viewer;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.events.SelectionListener;
import org.eclipse.swt.graphics.Color;
import org.eclipse.swt.graphics.Image;
import org.eclipse.swt.layout.RowLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Group;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Text;
import org.eclipse.ui.part.ViewPart;
import org.osgi.framework.Bundle;
import com.ibm.productivity.tools.ui.views.RichDocumentView;
```

Helper methods and classes

SWT is the low-level widgets that are common across different platforms. For example, a label, text entry field, button, and so on. Eclipse developers can use another framework called JFace to simplify the widget code by mapping “widget friendly” data types like strings into “application friendly” objects like Clients, Documents, and similar high-level classes. For example, the TableViewer works with “helper” objects that handle the mapping of higher-level classes like Documents into lower-level data types expected by widgets like strings. The next two classes define these helper classes.

The following methods and classes are required; copy each of them into end of ShelfView.java.

LibraryContentProvider

The ContentProvider provides the file name of for sample documents listed in Document Library:

```
  class LibraryContentProvider implements IStructuredContentProvider {
         public void inputChanged(Viewer v, Object oldInput,
                        Object newInput) {
         }
         public void dispose() {
         }
         public Object[] getElements(Object parent) {
                return new String[] { "Document 1", "Document 2",
                               "Document 3" };
         }
  }
```

LibraryLabelProvider

The LabelProvider provides the viewable text for file name of each sample document in Document Library:

```
  class LibraryLabelProvider extends LabelProvider implements
         ITableLabelProvider {
         public String getColumnText(Object obj, int index) {
```

```
                return getText(obj);
        }
        public Image getColumnImage(Object obj, int index) {
                return getImage(obj);
        }
        public Image getImage(Object obj) {
                return null;
        }
}
```

IMPORTANT: Image is defined by more than one package. Be certain to select the ones defined in the SWT package.

hookDoubleClickAction

The method will handle the double click happened in the Document Library :

```
private void hookDoubleClickAction() {
        viewer.addDoubleClickListener(new IDoubleClickListener() {
                public void doubleClick(DoubleClickEvent event) {
                        ISelection selection = viewer.getSelection();
                        Object obj = ((IStructuredSelection)selection).
                                getFirstElement();

                        String displayName = obj.toString();
                        String url = getDocumentURL( displayName );



                }
        });
}
```

getDocumentURL

The method translates the display name of each sample document in Document Library into absolute URL:

```
private String getDocumentURL( String displayName )
{
        String url = "";

        String res = "docs/" + displayName + ".odt";

        url = getResolvedPath( res );

        return url;
}

private String getResolvedPath(String file){
        String resolvedPath = null;
            Bundle bundle = Platform.getBundle
              ( "com.ibm.productivity.tools.sample.DocumentWorkflow" );
            if (bundle != null) {
                 java.net.URL bundleURL = bundle.getEntry( "/" );
                 if (bundleURL != null) {
                   try {
                     resolvedPath = FileLocator.resolve(
                               bundleURL ).getFile();
                   } catch (IOException e) {
                e.printStackTrace();
                 }
           }
      }

      String ret = resolvedPath+file;
      ret = ret.substring(1);
      ret = ret.replace('/', '\\');
      return ret;
}
```

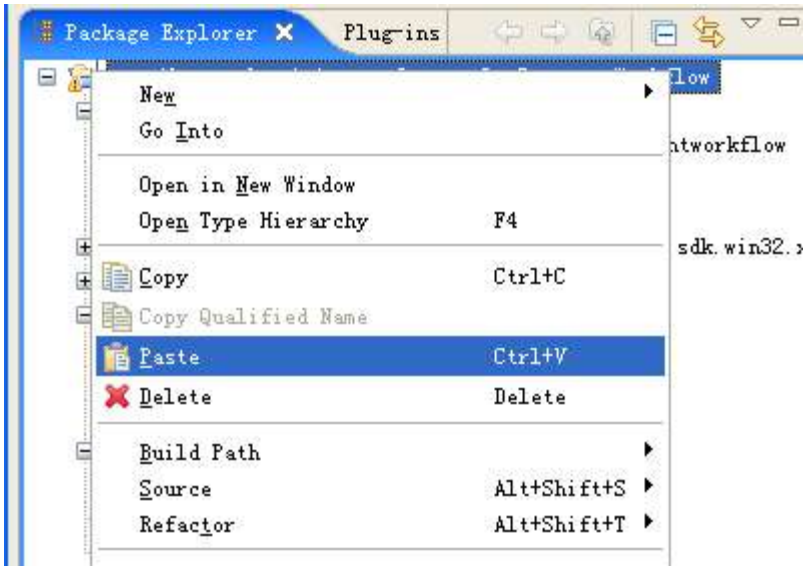
IMPORTANT: IOException and Platform are defined by more than one package. Be certain to select the ones defined in the Java and Eclipse packages.

Create the library

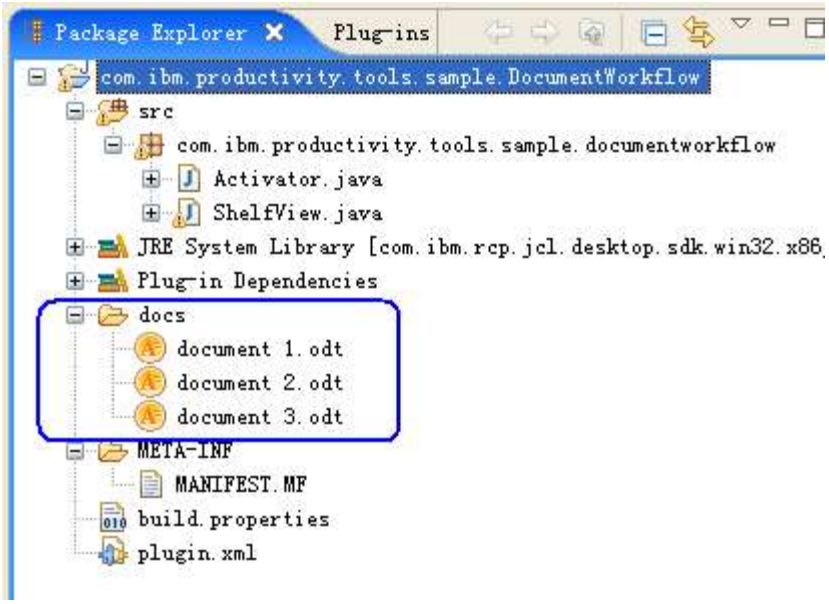
We use a dummy library in this tutorial, all files are local files. Please follow the following steps to create the library:

1. Switch back to **Windows Explorer**, locate the tutorial directory, right click on the **docs** directory, select **Copy.**

2. Switch to Eclipse environment, right click on the com.ibm.productivity.tools.sample.DocumentWorkflow plug-in, and select **Paste.**



Result: The docs directory is created as following:

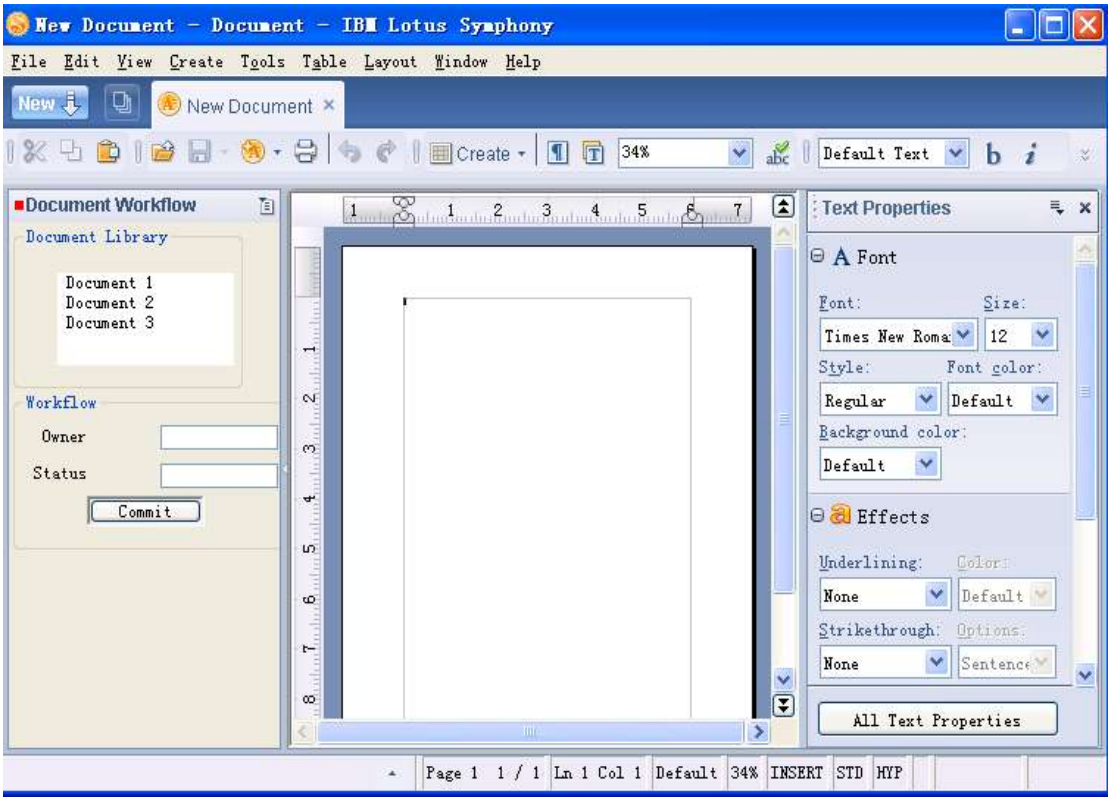


Run the application

1 Confirm it is correct, and click **File** > **Save All**

2 Click the **Run** button from toolbar

Result: The application is launched; click **File** > **New > Document,** your screen show should be similar to the following.



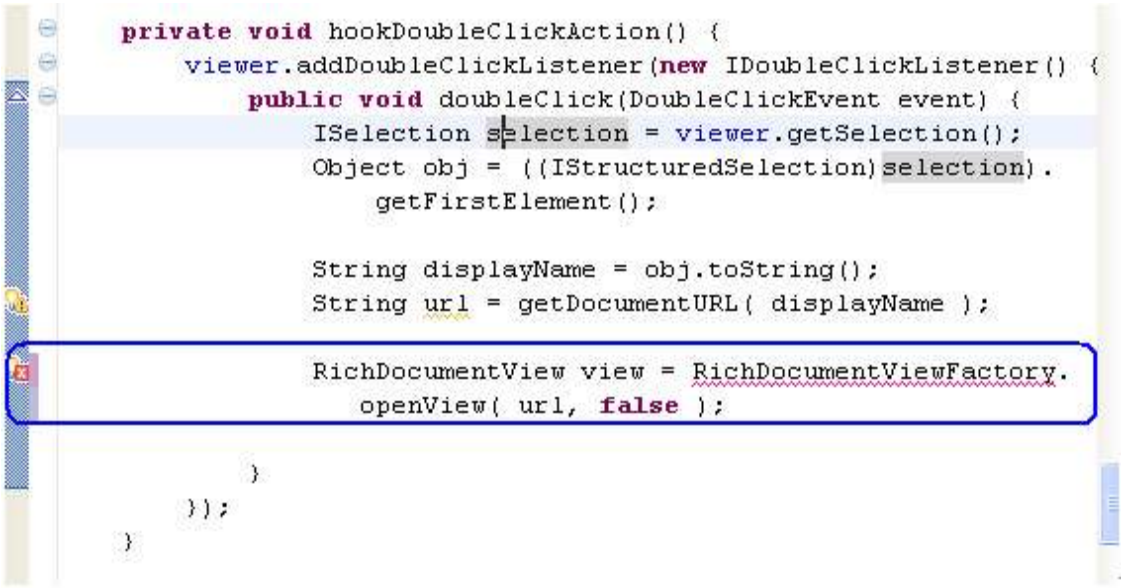
Part III

Opening document from the library

Locate hookDoubleClickAction method in ShelfView.java, and add the following line in the end of this function. When double click a file entry in the document library, the document will be opened with the following code automatically:

```
RichDocumentView view = RichDocumentViewFactory.
                            openView( url, false );
```

Note: Use the menu choice **Source > Organize Imports** (Ctrl+Shift+O) to add the needed import statements after pasting a code snippet. You will need to import com.ibm.productivity.tools.ui.views.RichDOcumentViewFactory package. Your screen looks like the following:



Using SelectionService and accessing content of document

In Eclipse, the selection service provided by the Eclipse workbench allows efficient linking of different parts within the workbench window. Each workbench window has its own *selection service* instance. The service keeps track of the selection in the currently active part and propagates selection changes to all registered listeners. Such selection events occur when the selection in the current part is changed or when a different part is activated. Both can be triggered by user interaction or programmatically.

Each Lotus Symphony view register the selection provider, so it is possible to monitor the selection change event happened in Lotus Symphony.

The following code demonstrates how to use selection provider in Lotus Symphony. Please perform the following steps:

- **Create an instance of ISelectionListener**

The selection listener will be invoked when user switch between opened views. Copy and paste the code into ShelfView.java. It will perform the following tasks at runtime:

1. Get the selected RichDocumentView

2. Get the UNO model of current selected view

3. Query the workflow information from the document with UNO API

4. Update the User Interface of Document Workflow application:

```
private ISelectionListener selectionListener = new ISelectionListener()
{
```

```
        public void selectionChanged(IWorkbenchPart arg0, ISelection arg1) {

                IAdaptable adaptable = ( IAdaptable ) arg1;

                RichDocumentViewSelection selection = ( RichDocumentViewSelection )
                        adaptable.getAdapter(RichDocumentViewSelection.class);

                selectedView = selection.getView();
                Object model = selectedView.getUNOModel();

                XTextTablesSupplier tableSupplier = ( XTextTablesSupplier )
                  UnoRuntime.queryInterface( XTextTablesSupplier.class, model );

                XNameAccess nameAccess = tableSupplier.getTextTables();

                try {
                        XTextTable table = ( XTextTable )UnoRuntime.queryInterface
                           ( XTextTable.class,nameAccess.getByName("Workflow"));
                        XCell cell = table.getCellByName( "B2" );
                        XText text = ( XText ) UnoRuntime.queryInterface
                           (XText.class, cell);
                        String owner = text.getString();
                        cell = table.getCellByName("B3");
                        text = ( XText ) UnoRuntime.queryInterface(XText.class, cell);
                        String reviewer = text.getString();
                        updateWorkflowInfo( owner, reviewer );

                } catch (NoSuchElementException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                } catch (WrappedTargetException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }

};
```

IMPORTANT: NoSuchElementException and XText are defined by more than one package. Be certain to select the ones defined in the Sun packages (not the Java packages).

IMPORTANT: Please notice the following line in this step; it is used to get the UNO document model of current document. The UNO document model is the entry point to access and modify document content. You can get almost every object within the document. In this lab, you will learn how to access a pre-defined table named as “workflow” in the document.

```
Object model = selectedView.getUNOModel();
```

Note: Use the menu choice **Source > Organize Imports** (Ctrl+Shift+O) to add the needed import statements after pasting a code snippet. You will need to import more packages. The following classes are listed here. Please check your own screen:

```
import org.eclipse.core.runtime.IAdaptable;
import com.sun.star.container.NoSuchElementException;
import com.sun.star.container.XNameAccess;
import com.sun.star.lang.WrappedTargetException;
import com.sun.star.table.XCell;
import com.sun.star.text.XText;
import com.sun.star.text.XTextTable;
import com.sun.star.text.XTextTablesSupplier;
import com.sun.star.uno.UnoRuntime;
import com.ibm.productivity.tools.ui.views.RichDocumentViewSelection;
import org.eclipse.ui.ISelectionListener;
import org.eclipse.ui.IWorkbenchPart;
```

- **updateWorkflowInfo method**

The following code updates the User Interface of Document Workflow application :

```
private void updateWorkflowInfo(String owner, String reviewer) {

        ownerTxt.setText(owner);
        statusTxt.setText(reviewer);
```

```
}
```

- **Register the selection listener**

To make the selection listener work, you need to register the listener into the selection service. Please copy the following code into end of createPartControl() method:

```
        IWorkbenchWindow window = PlatformUI.getWorkbench().
            getActiveWorkbenchWindow();
        ISelectionService service = window.getSelectionService();
        service.addSelectionListener(selectionListener);
```

Note: Use the menu choice **Source > Organize Imports** (Ctrl+Shift+O) to add the needed import statements after pasting a code snippet. You will need to import more packages. The following import are listed here. Please check your own screen:

```
import org.eclipse.ui.ISelectionService;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.PlatformUI;
```

Modifying current document

Until now, you have read the workflow information from the document. In this part, you will learn how to modify the current Lotus Symphony document with content from Document Workflow application. Perform the following steps to commit workflow information into the Lotus Symphony document.

- **Add function to commit button**

Locate the commitBtn in createPartControl, and change them as following. commitWorkflowInfo method is invoked when widget selected.

```
commitBtn.addSelectionListener(new SelectionListener(){

        public void widgetDefaultSelected(SelectionEvent arg0) {
                // TODO Auto-generated method stub

        }

        public void widgetSelected(SelectionEvent arg0) {

                commitWorkflowInfo();

        }

});
```

- **commitWorkflowInfo method**

This method will read content from Document Workflow application and invoke function to write data into current document. Copy and paste it into ShelfView.java:

```
private void commitWorkflowInfo( )
{
        String owner = ownerTxt.getText();
        String reviewer = statusTxt.getText();

        writeWorkflowInfo( owner, reviewer );
}
```

- **writeWorkflowInfo**

This method will commit workflow data into current document. Please copy and paste the following function into ShelfView.java.

```
private void writeWorkflowInfo(String owner, String reviewer) {
        Object model = selectedView.getUNOModel();

        XTextTablesSupplier tableSupplier = ( XTextTablesSupplier )
          UnoRuntime.queryInterface( XTextTablesSupplier.class, model );

        XNameAccess nameAccess = tableSupplier.getTextTables();

        try {
```

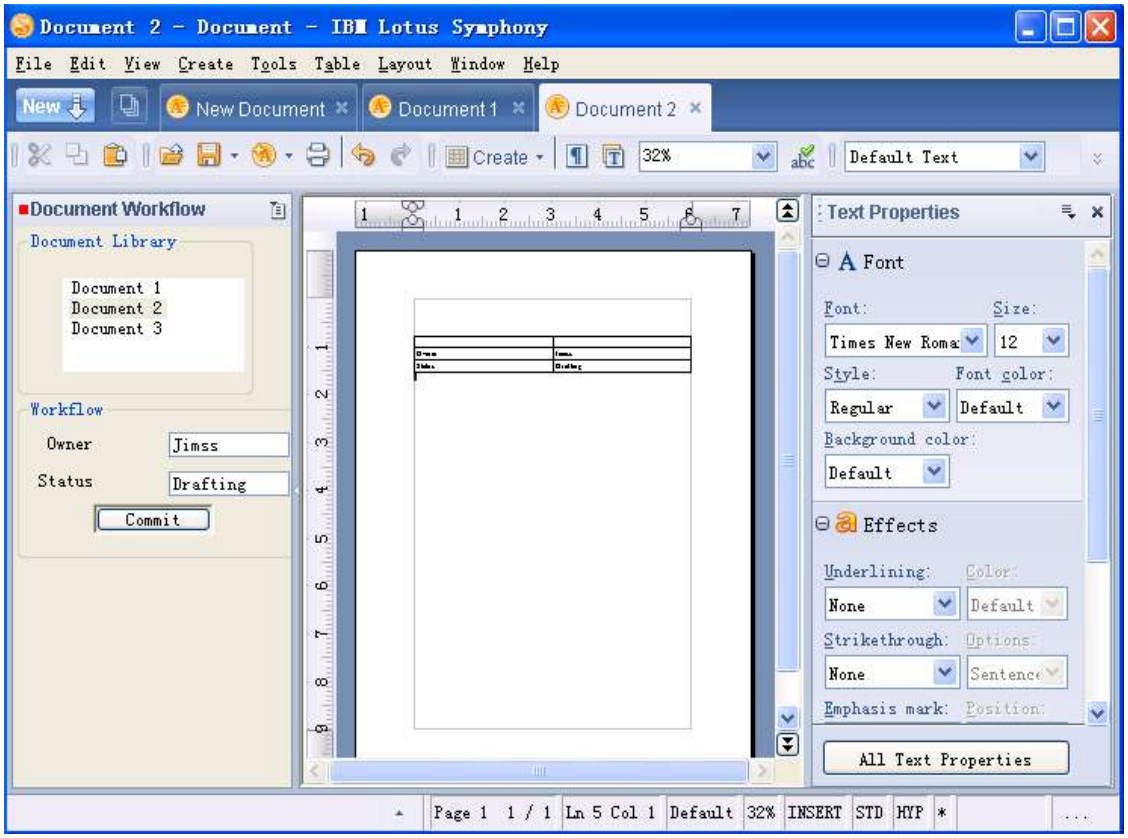
```
                XTextTable table = ( XTextTable )UnoRuntime.queryInterface
                  ( XTextTable.class,nameAccess.getByName("Workflow"));
                XCell cell = table.getCellByName( "B2" );
                XText text = ( XText ) UnoRuntime.queryInterface(
                                XText.class, cell);
                text.setString(owner);
                cell = table.getCellByName("B3");
                text = ( XText ) UnoRuntime.queryInterface(XText.class,
                                cell);
                text.setString(reviewer);

        } catch (NoSuchElementException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        } catch (WrappedTargetException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }

}
```

Run the application

1. Confirm it is correct, and click **File** > **Save ALL**

2. Click the **Run** button from toolbar

Result: The application is launched; click **File** > **New > Document,** your screen show should be similar to the following:



3. Double click each document in the Document Library

Result: the document will be opened in new tab window automatically.

4. Switch between “Document 1” “Document 2” and “Document 3”

Result: the owner and status in Workflow group was updated automatically.

5. Change the text of owner and status, click **Commit** button.

Result: the modified content is updated into Lotus Symphony document automatically.

Conclusion

This tutorial leads you through the creation of a Lotus Symphony side shelf extension. You learned how to create a plug-in, plug-in extensions, and how to using Java APIs supported by Lotus Symphony to read and update content of Lotus Symphony documents.

Appendix . Notices

Notices

The information contained in this publication is provided for informational purposes only. While efforts were made to verify the completeness and accuracy of the information contained in this publication, it is provided AS IS without warranty of any kind, express or implied. In addition, this information is based on IBM's current product plans and strategy, which are subject to change by IBM without notice. IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, this publication or any other materials. Nothing contained in this publication is intended to, nor shall have the effect of, creating any warranties or representations from IBM or its suppliers or licensors, or altering the terms and conditions of the applicable license agreement governing the use of IBM software.

Copyright

Lotus Software
IBM Software Group
One Rogers Street
Cambridge, MA 02142

Revision History:
Original material produced for IBM Lotus Symphony Release Beta 4.

List of Trademarks

IBM, the IBM logo, AIX, DB2, Domino, iSeries, i5/OS, Lotus, Lotus Notes, LotusScript, Notes, Quickplace, Sametime, WebSphere, Workplace, z/OS, and zSeries are trademarks or registered trademarks of IBM Corporation in the United States, other countries, or both.

Additional IBM copyright information can be found at: http://www.ibm.com/legal/copytrade.shtml

This information also refers to products built on Eclipse™ (http://www.eclipse.org)

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

The Graphics Interchange Format© is the Copyright property of CompuServe Incorporated. GIF(sm) is a Service Mark property of CompuServe Incorporated.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product and service names may be trademarks or service marks of others.