Part 1: Basics

Chapter 4. Listening, and Other Techniques

Topics: Window Listeners; Office Manipulation with JNA; Dispatching; Robot Keys

Example folders: "Office Tests" and "Utils"

This chapter concludes the general introduction to Office programming by looking at several techniques that will reappear periodically in later chapters: the use of window listeners, the manipulation of Office using the JNA library, dispatching messages to the Office GUI, and the Java Robot package.

Once again the examples come from the "Office Tests" directory in the code download associated with this book, and make liberal use of the classes in the "Utils" directory. For details please visit http://fivedots.coe.psu.ac.th/~ad/jlop/.

1. Listening to a Window

I haven't previously mentioned Office's listener interfaces (there are about 140 of them), because they work in the same way as listeners in Java.

Probably the easiest way of obtaining a list of them all is to visit the LibreOffice documentation for XEventListener (use `lodoc XEventListener`). The tree diagram at the top of the page shows that every listener interface is a subclass of XEventListener, and you can click on a subclass box to jump to its documentation.

One nice syntactic feature of listeners is that almost all their names end with “Listener”. This makes them easy to find when searching through indices of class names, such as the “Class Index” page at http://api.libreoffice.org/docs/idl/ref/classes.html.

The top-level document window can be monitored for changes using XTopWindowListener, which responds to modifications of the window’s state, such as when it is opened, closed, minimized, and made active.

The DocWindow.java example illustrates how to use the listener:

```
public class DocWindow implements XTopWindowListener
{

  public DocWindow(String fnm)
  {
    XComponentLoader loader = Lo.loadOffice();

    XExtendedToolkit tk = Lo.createInstanceMCF(
                XExtendedToolkit.class, "com.sun.star.awt.Toolkit");
    if (tk != null)
      tk.addTopWindowListener(this);

    XComponent doc = Lo.openDoc(fnm, loader);
    if (doc == null) {
      System.out.println("Could not open " + fnm);
      Lo.closeOffice();
      return;
    }

    GUI.setVisible(doc, true);

     // various window manipulation code; see below
     //     :

    Lo.closeDoc(doc);
    Lo.closeOffice();
  } // end of DocWindow()



  // --------- 7 XTopWindowListener methods -----------

  public void windowOpened(EventObject event)
  {
    System.out.println("WL: Opened");
    XWindow xWin = Lo.qi(XWindow.class, event.Source);
    GUI.printRect( xWin.getPosSize());
  }  // end of windowOpened()

  public void windowActivated(EventObject event)
  { System.out.println("WL: Activated");
    System.out.println("  Title bar: \"" + GUI.getTitleBar() + "\"");
  }  // end of windowActivated()

  public void windowMinimized(EventObject event)
  { System.out.println("WL: Minimized");  }

  public void windowNormalized(EventObject event)
  { System.out.println("WL: Normalized");  }

  public void windowDeactivated(EventObject event)
  { System.out.println("WL: De-activated");  }

  public void windowClosing(EventObject event) // never called (?)
  { System.out.println("WL: Closing");  }

  public void windowClosed(EventObject event)
  { System.out.println("WL: Closed");  }

  // --------- XEventListener method ------------

  public void disposing(EventObject event)   // never called (?)
  { System.out.println("WL: Disposing");  }

}  // end of DocWindow class
```

The class implements seven methods for XTopWindowListener, and disposing() inherited from XEventListener.

The DocWindow object is made the listener for the window by accessing the XExtendedToolkit interface, which is part of the Toolkit service. Toolkit is utilized by Office to create windows, and XExtendedToolkit adds three kinds of listeners: XTopWindowListener, XFocusListener, and the XKeyHandler listener.

When an event arrives at a listener method, one of the more useful things to do is to transform it into an XWindow instance:

```
XWindow xWin = Lo.qi(XWindow.class, event.Source);
```

It's then possible to access details about the frame, such as its size.

Events are fired when GUI.setVisible() is called in the DocWindow() constructor. An opened event is issued, followed by an activated event, triggering calls to windowOpened() and windowActivated(). Rather confusingly, both these methods are called twice.

When Lo.closeDoc() is called at the end of the constructor, a single de-activated event is fired, but two closed events are issued. Consequently, there's a single call to windowDeactivated() and two to windowClosed(). Strangely, there's no window closing event trigger of windowClosing(), and Lo.closeOffice() doesn't cause disposing() to fire.

2. Office Manipulation with JNA

Although XTopWindowListener can detect the minimization and re-activation of the document window, there's no way to trigger these changes from the Office API.

Fixing this requires a trip outside Office because I need a library that can manipulate a window controlled by the OS. One very nice API that fits my needs is Java Native Access (JNA, https://github.com/twall/jna), which supports OS programming without writing anything but Java. JNA comes with numerous classes, representing OS data structures, such as window handles and process IDs.

The JNA-powered methods in my JNAUtils.java utilities class fall roughly into three groups: functions for accessing windows via their handles, methods for accessing buttons inside windows via their handles, and functions that map between handles and process IDs.

In the DocWindow.java example, the document window’s handle is retrieved first, then minimization and re-activation methods change the window:

```
// part of DocWindow.java
HWND handle = JNAUtils.getHandle();
System.out.println("Handle: \"" +
                       JNAUtils.handleString(handle) + "\"");

JNAUtils.winMinimize(handle);
                  // triggers minimized and de-activated events
Lo.delay(3000);

JNAUtils.winRestore(handle);
                 // triggers normalized and activated events
Lo.delay(3000);
```

The calls to Lo.delay() slow down the window changes at run time so the user can see what's happening.

JNAUtils.getHandle() employs JNA's User32 library function FindWindow(). It finds the handle associated with a specific application window's class name, which is "SALFRAME" for both LibreOffice and OpenOffice. JNAUtils.handleString() converts the HWND handle into a hexadecimal string, which prints nicely.

JNAUtils.winMinimize() and JNAUtils.winRestore() call User32.ShowWindow() with different state parameters.

The JNA User32 class contains many functions related to window manipulation. For example, it's easy to implement window movement and resizing using SetWindowPos(). However, that functionality is already available in Office's XWindow.setPosSize().

Incidentally, it's possible for Office to access a window’s handle as an integer by means of XSystemDependentWindowPeer (e.g. see the code in GUI.getWindowHandle()), but there are no Office methods which utilize this integer. JNA methods require a HWND instance, so the handle needs to be obtained using JNA.

3. Detecting Office Termination

Office termination is most easily observed by attaching a listener to the Desktop object, as in DocMonitor.java:

```
public class DocMonitor
{
  public DocMonitor(String fnm)
  {
    XComponentLoader loader = Lo.loadOffice();

    XDesktop xDesktop = Lo.getDesktop();
    xDesktop.addTerminateListener( new XTerminateListener()
    {
       public void queryTermination(EventObject e)
                                throws TerminationVetoException
       {  System.out.println("TL: Starting Closing");   }

       public void notifyTermination(EventObject e)
       {  System.out.println("TL: Finished Closing"); }

       public void disposing(EventObject e)
       {  System.out.println("TL: Disposing");  }
    });

    XComponent doc = Lo.openDoc(fnm, loader);
    if (doc == null) {
      System.out.println("Could not open " + fnm);
      Lo.closeOffice();
      return;
    }

    GUI.setVisible(doc, true);

    System.out.println("Waiting for 5 secs before closing doc…");
    Lo.delay(5000);
    Lo.closeDoc(doc);

    System.out.println("Waiting for 5 secs before closing Office…");
    Lo.delay(5000);
    Lo.closeOffice();
  }  // end of DocMonitor()
}
```

An XTerminateListener is attached to the XDesktop instance. The program's output is:

```
> run DocMonitor algs.odp
Loading Office...
Opening algs.odp
Waiting for 5 secs before closing doc...
Closing the document
Waiting for 5 secs before closing Office...
Closing Office
TL: Starting Closing
TL: Finished Closing
Office terminated
```

XTerminateListener’s queryTermination() and notifyTermination() are called at the start and end of the Office closing sequence. As in the DocWindow.java example, disposing() is never triggered.

4. Bridge Shutdown Detection

There's another way to detect Office closure: by listening for the shutdown of the UNO bridge between the Java and Office processes. This can be useful if Office crashes independently of your Java code. However, I was only able to get this approach to work when using a socket-based link to Office.

The modified parts of DocMonitor.java are:

```
// in DocMonitor()
XComponentLoader loader = // Lo.loadOffice();
                          Lo.loadSocketOffice();
     :  // more code

XComponent bridgeComp = Lo.getBridge();
if (bridgeComp != null) {
  System.out.println("Found bridge");
  bridgeComp.addEventListener( new XEventListener()
  {
    public void disposing(EventObject e)
    { /* remote bridge has gone down, because
         office crashed or was terminated. */
      System.out.println("Office bridge has gone!!");
      System.exit(1);
    }
  });
}

XComponent doc = Lo.openDoc(fnm, loader);
     :   // more code
```

Since the disappearance of the Office bridge is a fatal event, disposing() finishes by calling System.exit() to kill Java.

The output of the revised DocMonitor.java is:

```
> run DocMonitor algs.odp
Loading Office...
Office process created
Found bridge
Opening algs.odp
Waiting for 5 secs before closing doc...
Closing the document
Waiting for 5 secs before closing Office...
Closing Office
TL: Starting Closing
TL: Finished Closing
Office terminated
Office bridge has gone!!
```

This output shows that bridge closure follows the call to Lo.closeOffice(), as you'd expect. However, if I make Office crash while DocMonitor is running, then the output becomes:

```
> run DocMonitor algs.odp
Loading Office...
Office process created
Found bridge
Opening algs.odp
Waiting for 5 secs before closing doc...
Office bridge has gone!!
```

I killed Office while the Java program was still running, so it never reached its Lo.closeOffice() call which triggers the XTerminateListener methods. However, the XEventListener attached to the bridge did fire. (If you're wondering, I killed Office by opening the Task manager on my test machine, and stopped the soffice process.)

5. Dispatching

This book is about the Java Office API, which manipulates UNO data structures such as services, interfaces, and components. There's an alternative programming style, based on the dispatching of messages to Office. These messages are mostly related to menu items, so, for example, the messages ".uno:Copy", ".uno:Cut", ".uno:Paste", and ".uno:Undo" duplicate commands in the "Edit" menu. The use of messages tends to be most common when writing macros (scripts) in Basic, because Office's built-in Macro recorder automatically converts a user's interaction with menus into dispatches.

One drawback of dispatching is that it isn't a complete programming solution. For instance, copying requires the selection of text, which has to be implemented with the Office API.

LibreOffice has a comprehensive webpage listing all the dispatch commands (https://wiki.documentfoundation.org/Development/DispatchCommands). An older source is the UICommands.ods spreadsheet, put together by Ariel Constenla-Haile in 2010, at https://arielch.fedorapeople.org/devel/ooo/UICommands.ods.

Another resource is chapter 10 of "OpenOffice.org Macros Explained" by Andrew Pitonyak (free online at http://www.pitonyak.org/book/)

Creating a dispatcher in Java is a little complicated since XDispatchProvider and XDispatchHelper instances are needed. XDispatchProvider is obtained from the frame (i.e. window) where the message will be delivered, which is almost always the Desktop's frame (i.e. Office application's window). XDispatchHelper sends the message via its executeDispatch() method. It's also possible to examine the result status in an DispatchResultEvent object, but that seems a bit flakey – it reports failure when the dispatch works, and raises an exception when the dispatch really fails.

The code is wrapped up in Lo.dispatchCmd() , which is called twice in the DispatchTest.java example:

```
public DispatchTest(String fnm)
{
  XComponentLoader loader = Lo.loadOffice();

  XComponent doc = Lo.openDoc(fnm, loader);
  if (doc == null) {
    System.out.println("Could not open " + fnm);
    Lo.closeOffice();
    return;
  }

  GUI.setVisible(doc, true);
  Lo.delay(100);
  toggleSlidePane();

  Lo.dispatchCmd("HelpIndex");      // show online Help

  Lo.dispatchCmd("Presentation");  // start slideshow

  //Lo.closeDoc(doc);
  //Lo.closeOffice();
}  // end of DispatchTest()
```

The Lo.dispatchCmd() string doesn't require an ".uno"" prefix. The first call sends ".uno:HelpIndex" to open Office's help window, and the second (".uno:Presentation") starts an Impress slideshow. This latter message only works if the loaded file is an Impress document.

DispatchTest() doesn't close the document or Office at the end, so the user must explicitly exit the slideshow and close the Office application himself by clicking on its close box.

6. Robot Keys

Another menu-related approach to controlling Office is to programmatically send menu shortcut key strokes to the currently active window. For example, a loaded Impress document is often displayed with a slide selection pane. This can be closed using the menu item View > Slide Pane, which is assigned the shortcut keys ALT-v ALT-l.

toggleSlidePane() 'types' these key strokes with the help of Java's Robot class:

```
// in DispatchTest.java

private void toggleSlidePane()
// send ALT-v and then ALT-l to foreground window;
// makes slide pane appear/disappear in Impress
{
  try {
    Robot robot = new Robot();
    robot.setAutoDelay(250);
    robot.keyPress(KeyEvent.VK_ALT);

    robot.keyPress(KeyEvent.VK_V);
    robot.keyRelease(KeyEvent.VK_V);

    robot.keyPress(KeyEvent.VK_L);
    robot.keyRelease(KeyEvent.VK_L);

    robot.keyRelease(KeyEvent.VK_ALT);
  }
  catch(AWTException e)
  {  System.out.println("sendkeys slidePane exception: " + e); }
}   // end of toggleSlidePane()
```

This technique has a few drawbacks – one is that Robot can only send key strokes to the currently active window on the OS' desktop. I've assumed this is Office, because I just made Office visible with a call to GUI.setVisible() back in the DispatchTest() constructor. Of course, the OS can do whatever it likes, such as suddenly pop up an instant messaging window or a software update alert, which would cause the characters to head to the wrong place.

Another problem is that toggleSlidePane() is in the dark about the current state of the GUI. If the slide pane is visible then the keys will make it disappear, but if the pane is not currently on-screen then these keys will bring it up. Also, what if the current GUI isn't the one for Impress? Do these combination of keys do something deadly in one of Office's other applications?

One nice feature is that there's lots of documentation on keyboard shortcuts for Office in its User guides (downloadable from https://th.libreoffice.org/get-help/documentation/), and these can be easily translated into key presses and releases in Robot.