chooser dialog box. Figure 17-4 shows some views from this dialog box.

[7]See http://docs.oracle.com/javase/tutorial/uiswing/components/filechooser.html.
[8]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JColorChooser.html.

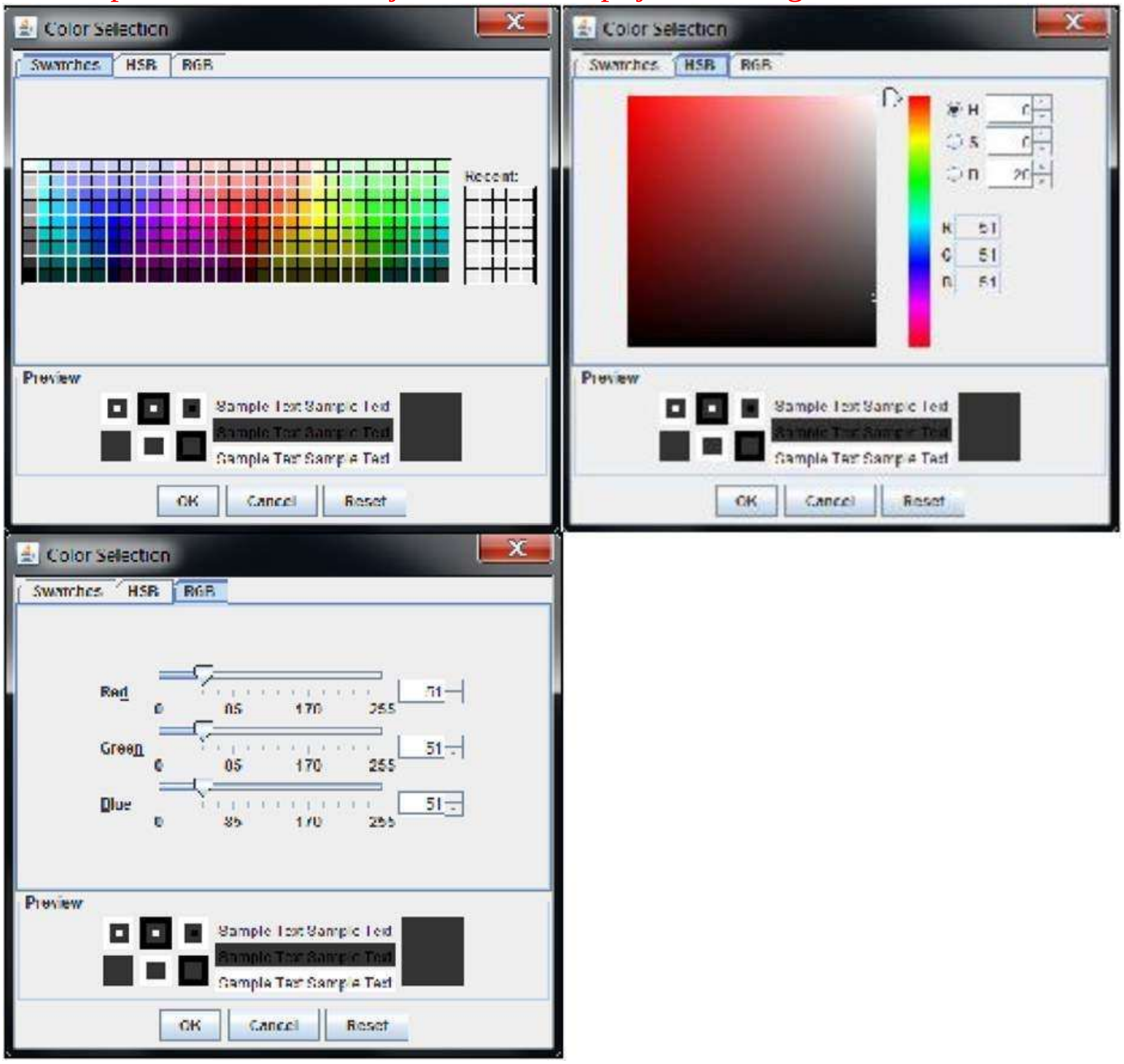


Figure 17-4. *Sample output from the ColorChooserDemo1.py script*

Listing 17-6 shows the showCC ActionListener event handler method from the ColorChooserDemo1.py script used to generate the output shown in Figure 17-4.

Listing 17-6. The showCC Method from the ColorChooserDemo1.py Script

```
def showCC( self, event ) :
result = JColorChooser().showDialog(
None, # Parent component
'Color Selection', # Dialog title
self.label.getForeground() # Initial color
)
if result :
message = 'New color: "%s"' % result.toString()
self.label.setForeground( result )
else :
message = 'Request canceled by user'
self.label.setText( message )
```

One of the interesting things about this color chooser dialog is the fact that the user has multiple ways of making a selection. As you can see in Figure 17-4, there are three tabs, any of which can be used by the user to pick a color. Aren't you glad that the Swing developers have gone through the effort of creating this? It is certainly much easier to use this than to create your own.

The javax.swing.colorchooser Package

In order to provide this functionality, the JColorChooser uses some support classes in the javax.swing.colorchooser package. You can determine this by taking a look at the JColorChooser constructors, as shown in Table 17-2. The first two constructors in this table are pretty straightforward.

Table 17-2. *JColorChooser Constructors*

Signature	**Description**
JColorChooser( )	Creates a color chooser pane with a default color of white.
JColorChooser(Color initialColor)	Creates a color chooser pane with the user-specified initial color.
JColorChooser (ColorSelectionModel model)	Creates a color chooser pane with the given ColorSelectionModel.

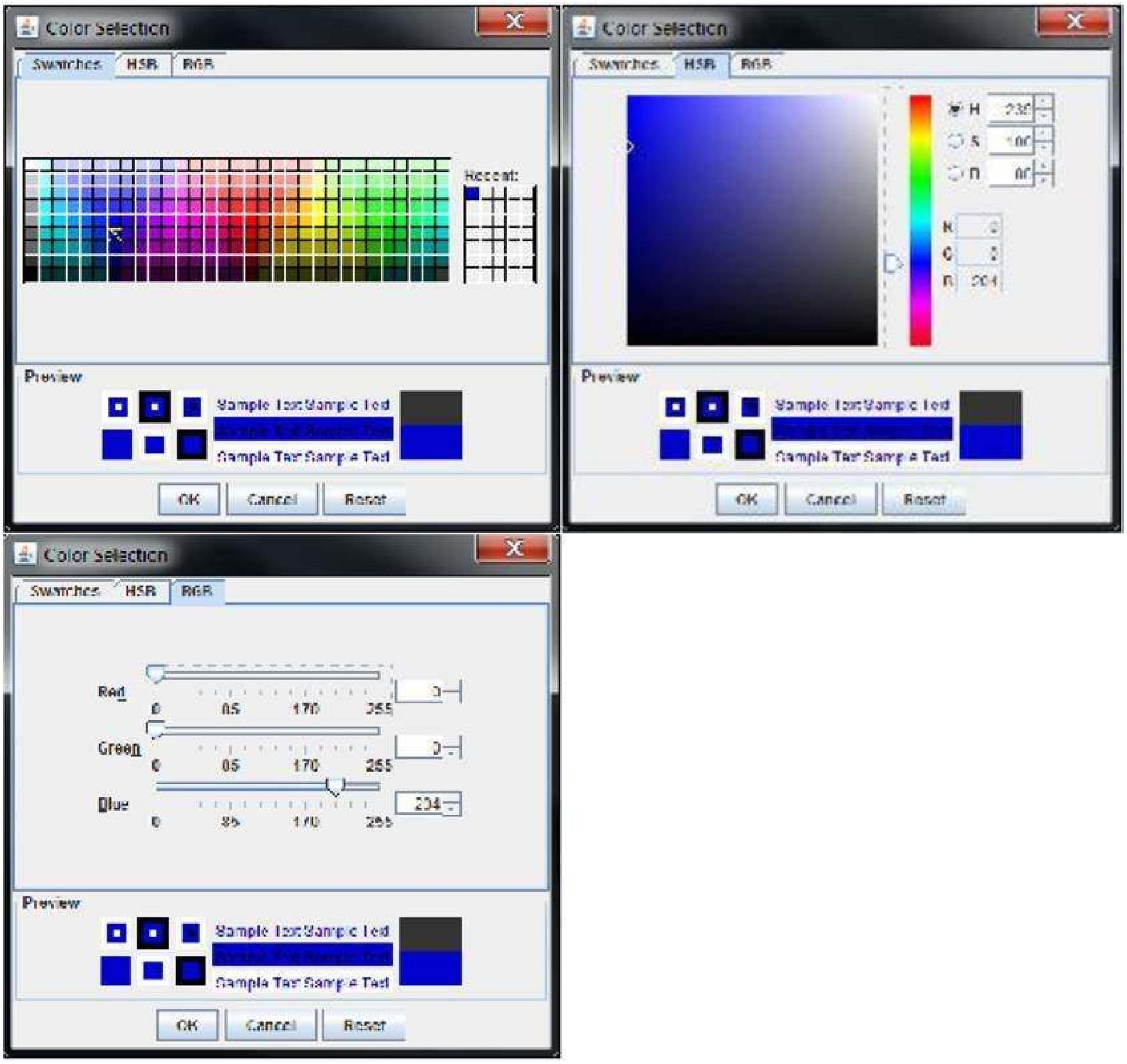
Paranoid developers might wonder if the default color of the first, no-argument constructor is in fact white. In fact, you can easily verify this using an expression like the one shown on lines 5 and 6 of Listing 17-7.

Listing 17-7. Verifing the Default JColorChooser Color

```
1|wsadmin>from java.awt import Color
2|wsadmin>from javax.swing import JColorChooser
3|wsadmin>
```

```
4|wsadmin>cc = JColorChooser()
5|wsadmin>cc.getColor() == Color.white
6|1
7|wsadmin>
8|wsadmin>csm = cc.getSelectionModel()
9|wsadmin>print csm.toString().split( '.' )[ -1 ]
10|DefaultColorSelectionModel@768b768b
11|wsadmin>
```

You might wonder about the last of the constructors shown in Table 17-2. What is a ColorSelectionModel?[9] The Javadoc says that it is an interface, and not a simple class. That same page says that the DefaultColorSelectionModel[10] class is the only implementation class that is provided by in the Swing hierarchy. In lines 8-10 of Figure 17-5, you can see that the default ColorSelectionModel that is associated with a JColorChooser instance is one of these DefaultColorSelectionModel instances.



H 230
S 100
B 80
R 0
G 0
B 204

Red 0
Green 0
Blue 204

Figure 17-5. *JColorChooser sample output after making a change*

Looking at the Javadoc for the DefaultColorSelectionModel class, you see that it allows you to listen for changes to the color. As you have seen before, this is the standard way that can be monitored. From this experience, you should be able to make an educated "guess" as to how the JColorChooser works. Take another look at the

[9]See http://docs.oracle.com/javase/8/docs/api/javax/swing/colorchooser/ColorSelectio
[10]See http://docs.oracle.com/javase/8/docs/api/javax/swing/colorchooser/DefaultColorS

ColorChooserDemo1.py sample script. What is the initial color that is displayed? Take a look at each of the tabs and you should see that the same color shown in the preview pane is represented on each tab in a consistent fashion. What happens when you pick another color and then select the other tabs? How do you think that this is done? Figure 17-5 shows some sample images after one color change has been made. You are encouraged to test this application yourself to see the JColorChooser in full size.

What Else Do You Need to Know About These “Special” Dialog Boxes?

One thing that may be useful to understand about these classes is the fact that they are based on the javax.swing. JComponent class, as you can see in Listing 17-8. What does this mean? The interesting thing is that since they are both JComponents, they aren’t limited to being used as modal dialog boxes. You can, if you want, add an instance of either to any container, just like you can for any JComponent. You don’t have to, but it’s something to consider.

Listing 17-8. JColorChooser and JFileChooser Class Hierarchy

```
wsadmin>from javax.swing import JColorChooser wsadmin>from javax.swing
import JFileChooser wsadmin>
wsadmin>classInfo( JColorChooser )
javax.swing.JColorChooser
| javax.swing.JComponent
| | java.awt.Container
| | | java.awt.Component
| | | | java.lang.Object
| | | | java.awt.image.ImageObserver
| | | | java.awt.MenuContainer
| | | | java.io.Serializable
| | java.io.Serializable
| javax.accessibility.Accessible
wsadmin>
wsadmin>classInfo( JFileChooser )
javax.swing.JFileChooser
| javax.swing.JComponent
| | java.awt.Container
| | | java.awt.Component
| | | | java.lang.Object
| | | | java.awt.image.ImageObserver
| | | | java.awt.MenuContainer
| | | | java.io.Serializable
| | java.io.Serializable
| javax.accessibility.Accessible
wsadmin>
```

Summary

This chapter covered some specialized dialog boxes that are highly functional and generally useful all at the same time. The ones you learned about here are related to user selection—selecting a color, a file, or a directory. Hopefully, you have seen just how easily they can be added to your scripts. Coming up in the next chapter, you'll learn how your applications can monitor progress and report it back to the users.

Chapter 18

Monitoring and Indicating Progress

As you've no doubt seen, many graphical applications include a progress indicator of some sort to convey how quickly something is happening. I don't know about you, but I tend to be somewhat impatient, so I find these progress bars very helpful. I hate it when a program just sits there—is it still working, has it hung, who knows? This chapter covers some different ways to measure progress and communicate it to your users.

Changing the Cursor

Sometimes you'll need to tell the users that the program is busy and let them know that they have to wait for a (hopefully short) amount of time. The easiest way to do this is to change the Cursor[1] from its default value to one that indicates that the application is busy. The Cursor class includes a number of predefined constants that can be used for this purpose. For this particular situation, you can use WAIT_CURSOR. How do you do that? Listing 18-1 shows a basic example of how to use it.

Listing 18-1. Using the WAIT_CURSOR Method from WaitCursor1.py

```
def wait( self, event ) :
source = event.getSource()
prev = source.getCursor()
source.setCursor(
Cursor.getPredefinedCursor( Cursor.WAIT_CURSOR )
)
sleep( 5 )
source.setCursor( prev )
```

This method is from the WaitCursor1.py script in the code\Chap_18 directory. If you execute the script and press the Wait button, the cursor will change from its default value to the WAIT_CURSOR for five seconds. Remember to change it back. Otherwise, you are likely to confuse your users into thinking that the script is still busy. Figure 18-1 shows some sample images from my system when I executed this script.

[1]Seehttp://docs.oracle.com/javase/8/docs/api/java/awt/Cursor.html.

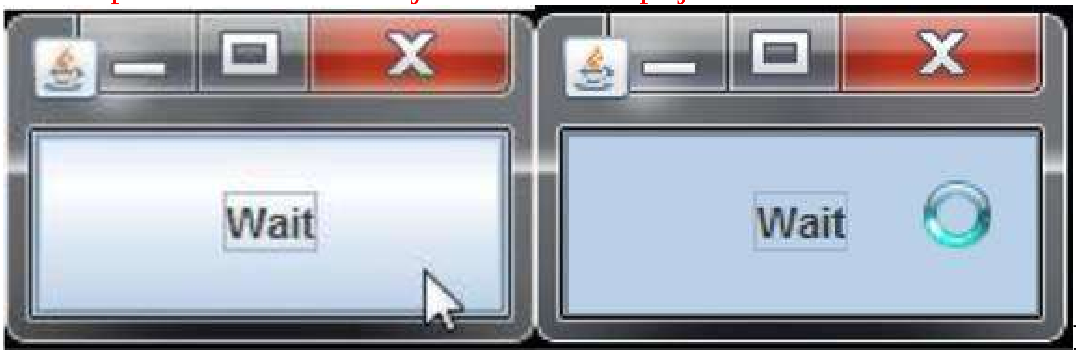


Figure 18-1. *WaitCursor images from a Windows 7 environment*

■ **Note** it is important to note that the way that the cursor looks is dependent on the operating system. it’s also important to remember that WAIT_CURSOR is visible only when the cursor is over the component for which the cursor was changed. Because of this, you may want to call the setCursor() method for your highest-level container (for example, your frame instance).

What if you want to enable WAIT_CURSOR in one part of your application and change it back somewhere else? You either have to save the original cursor setting somewhere, so it can be restored using this saved value, or you can simply use the Cursor.DEFAULT_CURSOR constant. Listing 18-2 demonstrates this constant using a JToggleButton. An even easier technique is to use None as an argument to the setCursor(...) method, which will force the specified component to use the cursor setting of its parent component.

Listing 18-2. Setting the Cursor Shape Based on isSelected() from WaitCursor2.py

```
def wait( self, event ) :
source = event.getSource()
cursor = [
Cursor.DEFAULT_CURSOR, # isSelected() == 0 (false)
Cursor.WAIT_CURSOR # isSelected() == 1 (true)
][ source.isSelected() ]
source.setCursor( Cursor.getPredefinedCursor( cursor ) )
```

When the ActionListener event handler is called, the cursor state will be based on the result of calling the JToggleButton isSelected() method (which returns zero for false and one for true).
Are there any problems with using the cursor to indicate that the application is busy? Think about it. When you use this technique, you are telling the user to wait. How happy do you think the users are when they are told to wait? I, for one, hate when I get a “wait” message. Since your applications are supposed to be event driven, you want them to be able to continue running even though a different part might be busy doing something. If your script needs to be interacting with the user while another part is doing something else, it needs to be doing these other operations on separate threads. Remember the SwingWorker class first mentioned in Chapter 6? That’s how you’re going to do it.

Showing a Progress Bar

Have you ever had an application appear to stop working and wonder what was going on? It happens to me all of the time. Even though I have a fairly capable laptop, there are times when every application appears to be hung. Sometimes the title bars show a “(Not Responding)” message, but that isn’t very reassuring. Unless an operation is expected to complete in less than half a second, it is best to consider communicating to the user some kind of indication of the expected wait time. That’s the purpose of a progress bar. Let’s take a look at the JProgressBar[2] class and see what it takes to make use of it. It’s always a good idea to begin by looking at the class constructors. Table 18-1 contains the constructors for the JProgressBar class.

Table 18-1. *JProgressBar Constructors* **Signature Description** JProgressBar()
JProgressBar(BoundedRangeModel newModel)
JProgressBar(int orient)
JProgressBar(int min, int max)

JProgressBar(int orient, int min, int max) Creates a horizontal progress bar that displays a border but no progress string.

Creates a horizontal progress bar that uses the specified model to hold the progress bar's data.

Creates a progress bar with the specified orientation, which can be SwingConstants.VERTICAL or SwingConstants. HORIZONTAL.

Creates a horizontal progress bar with the specified minimum and maximum.
Creates a progress bar using the specified orientation, minimum, and maximum.

As is often the case, at least for me, the description that is provided in the Javadoc isn't always clear. Take a look at the default JProgressBar instance. The first implementation can be found in ProgressBar0.py; sample images from this implementation are shown in Figure 18-2.

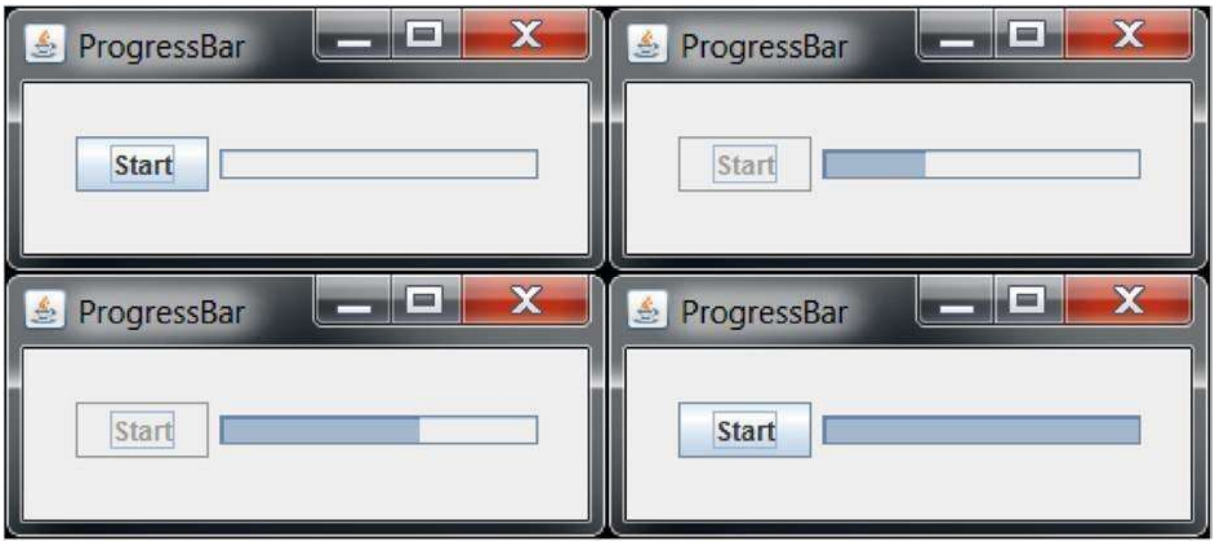


Figure 18-2. *Sample images from the ProgressBar0.py output*
[2]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JProgressBar.html.

How was this done? The primary (ProgressBar) class for this script is shown in Listing 18-3. There really shouldn't be too many surprises here. The only part that you might wonder about is using the setBorder(...) method call on lines 46-48.

It's sometimes confusing that you have to use the frame.getContentPane() method to access the particular pane to which you want to add an "empty" border. Note also the call to the BorderFactory[3].createEmptyBorder(...) method to create a little space or gap around the components in the JPanel. You might want to take a look at the "How to Use Borders" section of the Java Swing Tutorials[4] to obtain a better understanding about borders and how they can be used in Swing applications.

Listing 18-3. ProgressBar Class from ProgressBar0.py

```
38|class ProgressBar0( java.lang.Runnable ) :
39| def run( self ) :
40| frame = JFrame(
41| 'ProgressBar0',
42| size = ( 280, 125 ),
43| locationRelativeTo = None,
44| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
45| )
46| frame.getContentPane().setBorder(
47| BorderFactory.createEmptyBorder( 20, 20, 20, 20 )
48| )
49| panel = JPanel()
50| self.button = panel.add(
51| JButton(
52| 'Start',
53| actionPerformed = self.start
54| )
55| )
56| self.progressBar = panel.add( JProgressBar() )
57| frame.add(
58| panel,
59| BorderLayout.NORTH
60| )
61| frame.setVisible( 1 )
```

```
62| def start( self, event ) :
63| progressTask(
64| self.button,
65| self.progressBar
66| ).execute()
```

The ActionListener event handler routine, shown in lines 62-66 of Listing 18-3, creates the progressTask instance. Immediately after that, the thread starts executing. Details about this class are shown in Listing 18-4.

[3]Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/BorderFactory.html.

[4]See http://docs.oracle.com/javase/tutorial/uiswing/components/border.html.

Listing 18-4. The progressTask class from ProgressBar0.py

```
14|class progressTask( SwingWorker ) :
15| def __init__( self, button, progressBar ) :
16| self.btn = button # Save provided references
17| self.PB = progressBar
18| SwingWorker.__init__( self )
19| def doInBackground( self ) :
20| self.btn.setEnabled( 0 ) # Disable the "start" button
21| try :
22| random = Random()
23| progress = 0
24| self.PB.setValue( progress )
25| while progress < 100 :
26| sleep( ( random.nextInt( 1400 ) + 100 ) / 1000.0 )
27| progress = min(
28| progress + random.nextInt( 10 ) + 1, 100
29| )
30| self.PB.setValue( progress )
31| except :
32| Type, value = sys.exc_info()[ :2 ]
33| print 'Error:', str( Type )
34| print 'value:', str( value )
35| sys.exit()
36| def done( self ) :
37| self.btn.setEnabled( 1 ) # Enable the "start" button
```

Is this a great example of a SwingWorker descendent class? No, not really. For one thing, it needs to know too much about the invoking application. This should be obvious when you see that the constructor needs to have references to specific application components provided when the object is instantiated. This should be a dead giveaway. There has to be a better way, don't you think?

SwingWorker Progress

Are there any methods in the SwingWorker class related to "progress"? How can you find out? Chapter 4, introduced the classInfo utility class. Listing 18-5 shows the output of this function. It tells you that a getter method exists for a property named progress. Looking at the Javadoc for the SwingWorker[5] class shows you that a setter method exists, but it is protected, so any descendent classes that you define have to do something special to access that setter method.

Listing 18-5. SwingWorker "Progress" Methods

```
wsadmin>from javax.swing import SwingWorker wsadmin>
wsadmin>classInfo( SwingWorker, meth = 'progress' )
javax.swing.SwingWorker

getProgress
| java.lang.Object
| java.util.concurrent.RunnableFuture | | java.lang.Runnable
| | java.util.concurrent.Future wsadmin>
```

[5]See http://docs.oracle.com/javase/8/docs/api/javax/swing/SwingWorker.html.

How do I know this? When I tried to use a simple call to the SwingWorker setProgress(...) method, an AttributeError exception was raised and the setProgress name was the source of the error.[6]
So, what can you do? The Jython developers were kind enough to provide a way to access this kind of protected method. A Java programmer can use the @Override annotation to allow the doInBackground(...) method to use the setProgress(...) method. In Jython, you have to use the following syntax to call the setProgress(...) method. In the SwingWorker descendent class, use this:

```
self.super__setProgress( value )
```
[7]

Listing 18-6 shows the modified progressTask class from the working

ProgressBar2.py script. Note the simplifications, which include the fact that this class now has no references to the application components. That means it’s completely contained, meaning that all references to the class attributes exist only in the class.

Listing 18-6. The progressTask Class from ProgressBar2.py

```
15|class progressTask( SwingWorker ) :
16| def __init__( self ) :
17| SwingWorker.__init__( self )
18| def doInBackground( self ) :
19| try :
20| random = Random()
21| progress = 0
22| self.super__setProgress( progress )
23| while progress < 100 :
24| sleep( ( random.nextInt( 1400 ) + 100 ) / 1000.0 )
25| progress = min(
26| progress + random.nextInt( 10 ) + 1, 100
27| )
28| self.super__setProgress( progress )
29| except :
30| Type, value = sys.exc_info()[ :2 ]
31| print 'Error:', str( Type )
32| print 'value:', str( value )
33| sys.exit()
34| def done( self ) :
35| pass
```

Listing 18-7 shows the modified ProgressBar class from this same script. Since the SwingWorker descendent class knows nothing about the class components defined here, you have to make this class a descendent of the PropertyChangeListener class, as shown on lines 36 and 65-70.

[6]The script containing the “first attempt” is the ProgressBar1.py file found in the code\Chap_18 directory. [7]Yes, there really are two underscores between super and setProgress.

Listing 18-7. The ProgressBar Class from ProgressBar2.py

```
class ProgressBar2( java.lang.Runnable, PropertyChangeListener ) :
def run( self ) :
frame = JFrame(
'ProgressBar2',
size = ( 280, 125 ),
locationRelativeTo = None,
defaultCloseOperation = JFrame.EXIT_ON_CLOSE
)
frame.getContentPane().setBorder(
BorderFactory.createEmptyBorder( 20, 20, 20, 20 )
)
panel = JPanel()
self.button = panel.add(
JButton(
'Start',
actionPerformed = self.start
)
)
self.progressBar = panel.add( JProgressBar() )
frame.add(
panel,
BorderLayout.NORTH
)
frame.setVisible( 1 )
def start( self, event ) :
self.button.setEnabled( 0 )
task = progressTask()
task.addPropertyChangeListener( self )
task.execute()
def propertyChange( self, event ) :
if event.getPropertyName() == 'progress' :
progress = event.getNewValue()
self.progressBar.setValue( progress )
if progress == 100 :
self.button.setEnabled( 1 )
```

Showing Progress Details

So far, this is a pretty good way to display progress, albeit a bit vague. Wouldn’t it be nice if you were able to display the percentage complete on the progress bar? The developers of the Swing classes thought about this, and have made it really easy for you to do. All you have to do is enable the stringPainted property of the JProgressBar instance. In Jython, this is as simple as adding the stringPainted keyword to the JProgressBar constructor call. Listing 18-8 shows this process.

Listing 18-8. ProgressBar Constructor Call with the stringPainted Keyword

```
self.progressBar = panel.add(
JProgressBar( stringPainted = 1 )
)
```

After making this change, you can see that the progress bar includes a string representation of the completion as a percentage value. Figure 18-3 shows some sample images from ProgressBar4.py, which incorporates this change.

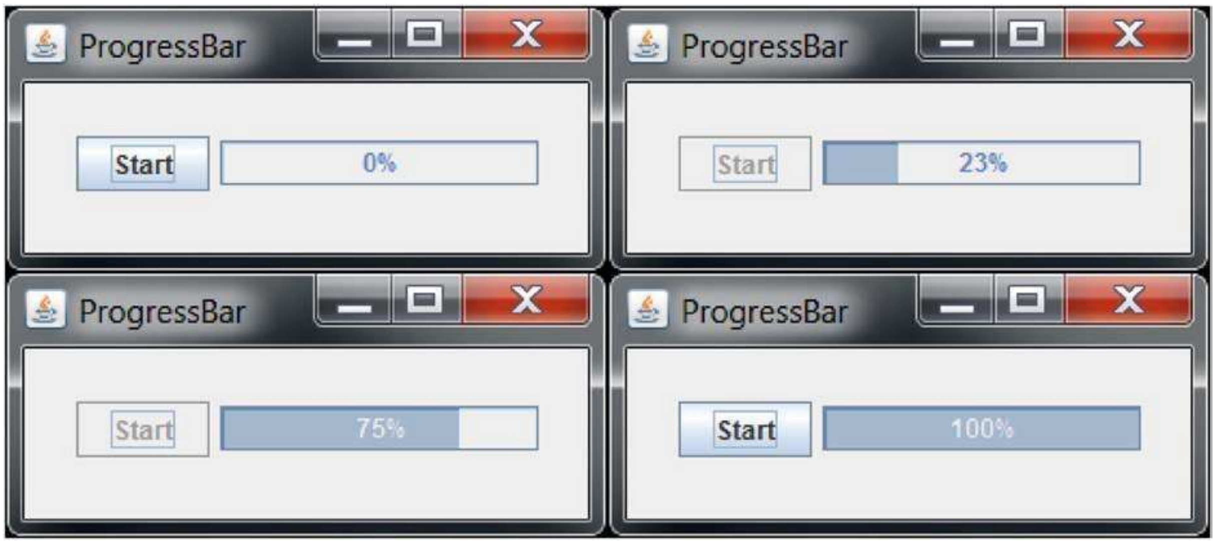


Figure 18-3. *Sample images with stringPainted enabled from ProgressBar4.py*

Specifying a Progress Bar Range

Some of the JProgressBar constructors shown in Table 18-1 allow you to define the minimum and maximum values for your progress bar. What does this do, and what does this mean as far as the SwingWorker progress?

If you specify minimum and maximum values for a progress bar, you become responsible for tracking its updates. The separation of the progressTask (SwingWorker) instance from the ProgressBar class means that only the progress bar is aware of the new minimum and maximum values. In the progressTask instance, it is measuring a percentage of the progress, as shown in Figure 18-3.

So, if your application needs to use non-default minimum and maximum values in a progress bar instance, it needs to take this into account. Listing 18-9 shows one approach that you might want to consider using. Regardless of the minimum and maximum values you choose for the JProgressBar instance, it uses the progress value of the progressTask instance as a percentage complete value to determine the value that's assigned to the JProgressBar instance (as shown on line 74).

Listing 18-9. propertyUpdate() Method from ProgressBar5.py

```
68| def propertyUpdate( self, event ) :
69| if event.getPropertyName() == 'progress' :
70| progress = event.getNewValue() # integer % complete
71| lo = self.progressBar.getMinimum()
72| hi = self.progressBar.getMaximum()
73| here = int( ( hi - lo ) * 0.01 * progress ) + lo
74| self.progressBar.setValue( here )
75| if progress == 100 :
76| self.button.setEnabled( 1 )
```

Indeterminate ProgressBar Range

Sometimes, you won't know how long an action will take. For example, if your application needs to communicate with a remote host to transfer data from there to here, there is likely to be some delay while communication is being established. Once that has occurred, the amount of data can be provided before the copying initiates. While this is happening, it is considered good practice to tell the users that something is happening.

For this purpose, the JProgressBar class includes an indeterminate attribute. When this attribute[8] is enabled, the progress bar will show movement, but no change in progress completion percentage. The ProgressBar6. py sample script shows one way that this attribute can be used. It includes a random delay in the progressTask doInBackground(...) method to simulate some kind of initialization delay before progress changes occur. Figure 18-4 shows some images of the progress bar when the indeterminate attribute is true.

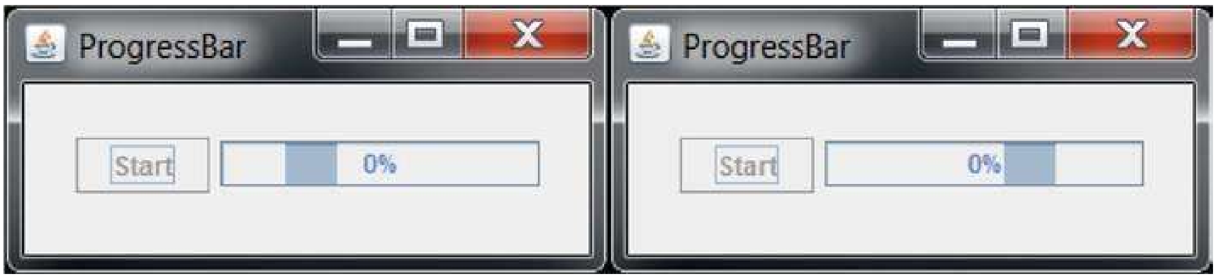


Figure 18-4. *Sample images of indeterminate progress*

Listing 18-10 shows the lines in ProgressBar6.py that relate to using the indeterminate attribute in your application progress bar instance. In the ActionListener event handler invoked when the Start button is selected, you see how the indeterminate attribute is enabled. Then, in the PropertyChangeListener event handler in line 74, you can see how the state of the indeterminate attribute is disabled and the actual progress is determined.

Listing 18-10. Indeterminate Related Code from ProgressBar6.py

```
65| def start( self, event ) :
66| self.button.setEnabled( 0 )
67| self.progressBar.setIndeterminate( 1 )
68| task = progressTask( self.propertyUpdate )
69| task.execute()
70| def propertyUpdate( self, event ) :
71| if event.getPropertyName() == 'progress' :
72| progress = event.getNewValue() # integer % complete
73| PB = self.progressBar
74| PB.setIndeterminate( 0 )
75| lo = PB.getMinimum()
76| hi = PB.getMaximum()
77| here = int( ( hi - lo ) * 0.01 * progress ) + lo
78| PB.setValue( here )
79| if progress == 100 :
80| self.button.setEnabled( 1 )
```

[8] Interestingly enough, the JProgressBar Javadoc doesn’t explicitly identify indeterminate as an actual attribute or field. But if you use the classInfo function from Chapter 4, you can see that it does exist and your Jython scripts can access it directly, even though this practice is discouraged.

ProgressMonitor Objects

In many ways, instances of the ProgressMonitor[9] class appear to be similar to the dialog boxes described in Chapters 16 and 17. This is a bit misleading, though. Listing 18-11 shows that the ProgressMonitor class is a descendent of the java.lang.Object class. JDialog descends from the java.awt.Dialog class, and JColorChooser and JFileChooser both descend from the javax.swing.JComponent class, as you learned in Chapter 17.

Listing 18-11. ProgressMonitor Class Hierarchy

```
wsadmin>from javax.swing import ProgressMonitor wsadmin>
wsadmin>classInfo( ProgressMonitor )
javax.swing.ProgressMonitor
| java.lang.Object
| javax.accessibility.Accessible
wsadmin>
```

What does this mean for your applications? For one, ProgressMonitor instances are going to act very different from descendants of the JComponent class. Additionally, you can't add a ProgressMonitor instance to a Swing container. This provides additional information as to why this dialogue-like box is being discussed here, instead of in Chapter 17. How do you use a ProgressMonitor object? Let's begin with the constructor and then discuss how the instance can and should be used. Take a look at the ProgressMonitor constructors shown in Table 18-2.

Table 18-2. *ProgressMonitor Constructor*

Signature	Description
ProgressMonitor(Component parentComponent, Object message, String note, int min, int max)	Instantiates an object for the purpose of showing progress.

It’s kind of interesting to see that this class has only one constructor. Most of the previous Swing classes you’ve learned about had many more. The constructor parameters shouldn’t be too much of a surprise. The kinds of questions that I had when I first encountered this constructor were mostly concerned with the message and note arguments. Mostly, I wondered about how they differ and how they should be used.

Listing 18-12 shows how simple it can be to demonstrate a progress monitor. If you want to enter this example in your own interactive session, remember that the last (i.e., empty) line is significant. It tells the Jython interpreter that the while statement is complete, and that the contents of the loop should be executed.

[9]See http://docs.oracle.com/javase/8/docs/api/javax/swing/ProgressMonitor.html.

Listing 18-12. Interactive wsadmin Session Showing ProgressMonitor

```
wsadmin>from javax.swing import ProgressMonitor
wsadmin>from java.util import Random
wsadmin>from time import sleep
wsadmin>
wsadmin>random = Random()
wsadmin>progress = 0
wsadmin>
wsadmin>pm = ProgressMonitor( None, 'Message text', None, 0, 100 )
wsadmin>while progress < 100 :
wsadmin> sleep( ( random.nextInt( 1400 ) + 100 ) / 1000.0 ) wsadmin> progress
= min( 100, progress + random.nextInt( 10 ) ) wsadmin> pm.setProgress(
progress )
wsadmin>
```

Figure 18-5 shows some images of a ProgressMonitor instance.

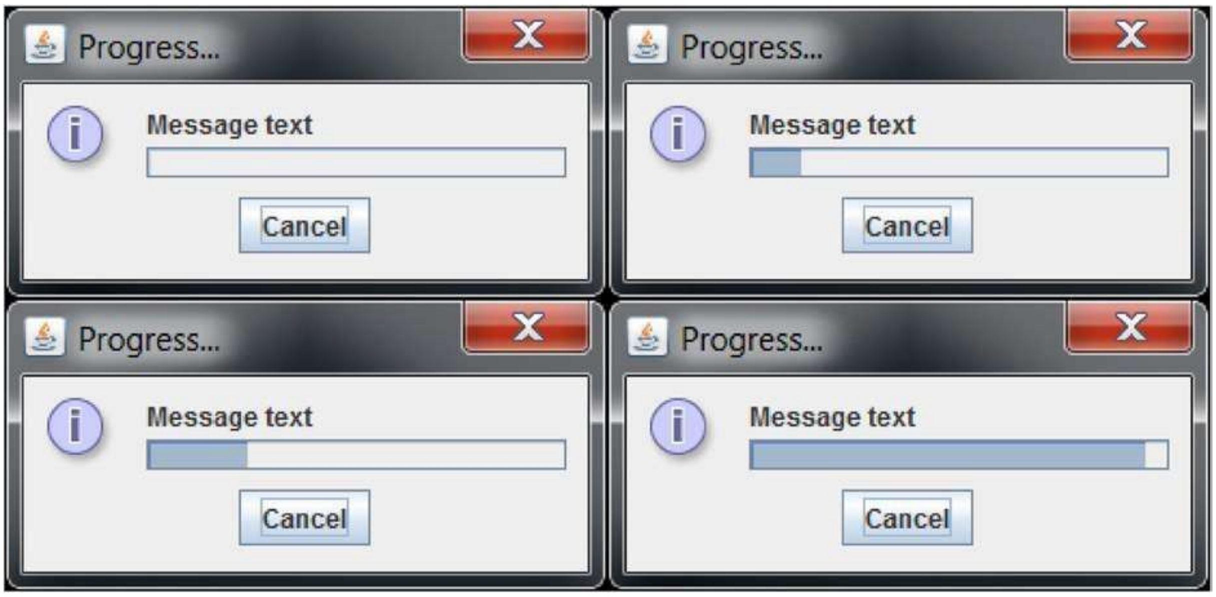


Figure 18-5. *Images from the interactive session ProgressMonitor instance in Listing 18-12*

■**Note** aProgressMonitor instance shouldn't be reused. a new instance should be instantiated instead.

From these, you can see the following about the dialog box:

• A title of Progress...
• A Close icon in the upper-right corner
• An Information icon
• The user-supplied message text
• A progress bar
• A Cancel (or local equivalent) button

Based on this first attempt and on the previous investigation into JDialog instances, you may be surprised to learn that you have no control over:

• The title of the ProgressMonitor dialog box.
• The icon that is displayed
• The buttons that are displayed
• The text that is shown on the button

Some other big differences between ProgressMonitor instances and the other dialog boxes that you've seen are:

• ProgressMonitor dialog box instances are not modal.
• The ProgressMonitor class doesn't have any listeners associated with it.

You may be wondering why I used an interactive wsadmin session, as shown in Listing 18-12, to demonstrate the ProgressMonitor class. One big reason was to show that once the progress reaches the maximum value (100%), the dialog is automatically hidden.

What is required to create a simple script that demonstrates the ProgressMonitor class? Listing 18-13 shows the class that is defined in ProgressMonitor1.py.[10] Notice how the ProgressMonitor is instantiated in the ActionListener event handler, that is, in the start(.) method in lines 65-75.

Listing 18-13. The ProgressMonitor1 Class from ProgressMonitor1.py

```
42|class ProgressMonitor1( java.lang.Runnable ) :
43| def run( self ) :
44| frame = JFrame(
45| 'ProgressMonitor',
46| size = ( 280, 125 ),
47| locationRelativeTo = None,
48| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
49| )
50| frame.getContentPane().setBorder(
51| BorderFactory.createEmptyBorder( 20, 20, 20, 20 )
52| )
53| panel = JPanel()
54| self.button = panel.add(
55| JButton(
56| 'Start',
57| actionPerformed = self.start
58| )
59| )
60| frame.add(
61| panel,
62| BorderLayout.NORTH
63| )
64| frame.setVisible( 1 )
65| def start( self, event ) :
66| self.button.setEnabled( 0 )
67| self.PM = ProgressMonitor(
68| None, # parentComponent 69| 'Message text', # message
None, # note
0, # minimum value
100 # maximum value
)
task = progressTask( self.propertyUpdate )
task.execute()
```

[10]Note that the class name can’t be the same as the Java Swing class you are trying to use.

Listing 18-14 shows the PropertyChangeListener event handler that is defined in the propertyUpdate() method. Note its similarity to the PropertyChangeListener event handler in the progress bar scripts.

Listing 18-14. PropertyChangeListener from ProgressMonitor1.py

```
def propertyUpdate( self, event ) :
if event.getPropertyName() == 'progress' :
progress = event.getNewValue() # integer % complete
PM = self.PM
lo = PM.getMinimum()
hi = PM.getMaximum()
here = int( ( hi - lo ) * 0.01 * progress ) + lo
PM.setProgress( here )
if progress == 100 :
self.button.setEnabled( 1 )
```

ProgressMonitor Cancellation

Using ProgressMonitor1.py, you can easily demonstrate the cancellation process. Start the application and press the Start button. Watch what happens when the ProgressMonitor dialog appears. You shouldn’t be too surprised that the dialog box goes away. You might wonder why the Start button is still disabled ... at least for a short time.

What is happening? Consider that, when the ProgressMonitor was canceled, the progressTask continued to execute on the separate thread. Until this task completes, PropertyChangeEvents will continue to be generated, which will result in the PropertyChangeListener method being called. Once the progress reaches 100, the Start button will be enabled. Is this the right way to handle the cancellation event? I don’t think so.

In order to properly react to the cancellation of a ProgressMonitor, you need to be able to determine that it has occurred. Fortunately, the Swing designers have provided an isCanceled() method as part of the ProgressMonitor API. You can use it to detect the situation. What should you do when it occurs? Well, one of the important things that you should do is terminate the progressTask that was created to perform the job being monitored. Again, the Swing designers come to the rescue. The SwingWorker API includes a cancel() method that allows developers to terminate the thread.

Is that all you need to worry about? No, you also have to include code in the progressTask class and the PropertyChangeListener event handler to deal with the possible cancellation of these different objects.
Listing 18-15 shows the modifications found in the second script—ProgressMonitor2.py—to deal with the cancellation of a ProgressMonitor object.
Listing 18-15. Differences Found in ProgressMonitor2.py

```
class progressTask( SwingWorker ) :
...
def doInBackground( self ) :
try :
random = Random()
progress = 0
self.super__setProgress( progress )
sleep( ( random.nextInt( 1400 ) + 100 ) / 1000.0 )
while progress < 100 :
sleep( ( random.nextInt( 1400 ) + 100 ) / 1000.0 )
progress = min(
progress + random.nextInt( 10 ) + 1, 100
)
self.super__setProgress( progress )
except KeyboardInterrupt, ki :
pass
except :
Type, value = sys.exc_info()[ :2 ]
print 'Error:', str( Type )
print 'value:', str( value )
sys.exit()
...
class ProgressMonitor2( java.lang.Runnable ) :
...
def start( self, event ) :
```

```
68| self.button.setEnabled( 0 )
69| self.PM = ProgressMonitor(
70| None, # parentComponent
71| 'Message text', # message
72| None, # note
73| 0, # minimum value
74| 100 # maximum value
75| )
76| self.task = progressTask( self.propertyUpdate )
77| self.task.execute()
78| def propertyUpdate( self, event ) :
79| if event.getPropertyName() == 'progress' :
80| progress = event.getNewValue() # integer % complete
81| PM, task = self.PM, self.task
82| lo, hi = PM.getMinimum(), PM.getMaximum()
83| here = int( ( hi - lo ) * 0.01 * progress ) + lo
84| PM.setProgress( here )
85| done = task.isDone()
86| if PM.isCanceled() or done :
87| if not done :
88| task.cancel( 1 )
89| self.button.setEnabled( 1 )
```

Table 18-3 identifies the changes that you nee to make in order to deal with the cancellation of ProgressMonitor or progressTask more appropriately. It is interesting to see how little code is needed in order to adequately handle these cancellation events.[11]

Table 18-3. *Explanation of Changes in ProgressMonitor2.py*

Lines	**Description**
35-36	The except clause in the progressTask doInBackground() method now silently ignores KeyboardInterrupt exceptions. All others will continue to display information about the exception before the application is terminated.
76	A reference is saved in the object instance to the task object. The PropertyChangeListener event handler needs this reference.
86-88	If the ProgressMonitor was canceled, you might need to cancel the progressTask instance.
89	If the ProgressMonitor was canceled or if the progressTask instance is completed, the Start button will be enabled.

The ProgressMonitor Message

Take another look at the ProgressMonitor constructor in Table 18-2. What data type is the message parameter? It's an object. What does that mean and why isn't it a String? According to the Javadoc, the message argument is an object so that it can be used in different ways, as described in the JOptionPane.message[12] documentation. Additionally, it is important to note that the message portion of the ProgressMonitor object will not change during the life of the ProgressMonitor. Let's see what you can do with the message.

The ProgressMonitor3.py script uses the same technique as shown in Chapter 16 to specify the message as an ImageIcon. I tried this approach first because I wanted to see if I could use this technique to display a different icon on the ProgressMonitor. Unfortunately, as you can see in Figure 18-6, the ImageIcon is displayed in addition to the Information icon.

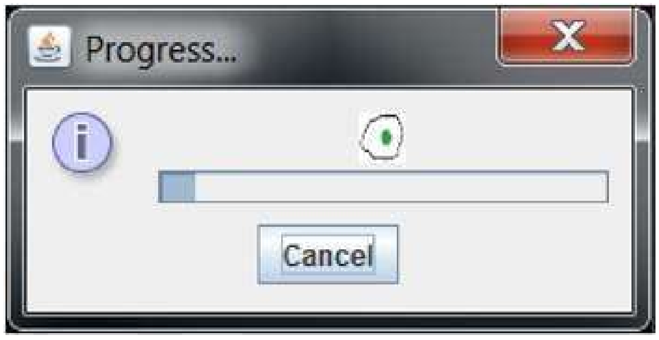


Figure 18-6. *ProgressMonitor with an ImageIcon message*

Can you use HTML in the message? According to the JOptionPane.message documentation, you would expect strings to be displayed in a JLabel. Since the JLabel class allows HTML text, you might be able to do some interesting things. Figure 18-7 shows what happens when you try using an HTML message, as well as an ordered and unordered list.[13] Unfortunately, it appears that an excessively "long" HTML message can cause some problems. From the little testing that I did, it appears to be related to the horizontal space that is allowed for the message. I guess you need to determine just how much information, and in what form, should be displayed in this message field.

[11] A Java application would need to catch a java.lang.InterruptedException instead.

[12]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JOptionPane.html#message

[13]All of these output images were generated by ProgressMonitor4.py. You will

have to edit the source script and uncomment the desired assignment statement in the start() method to duplicate these different outputs.

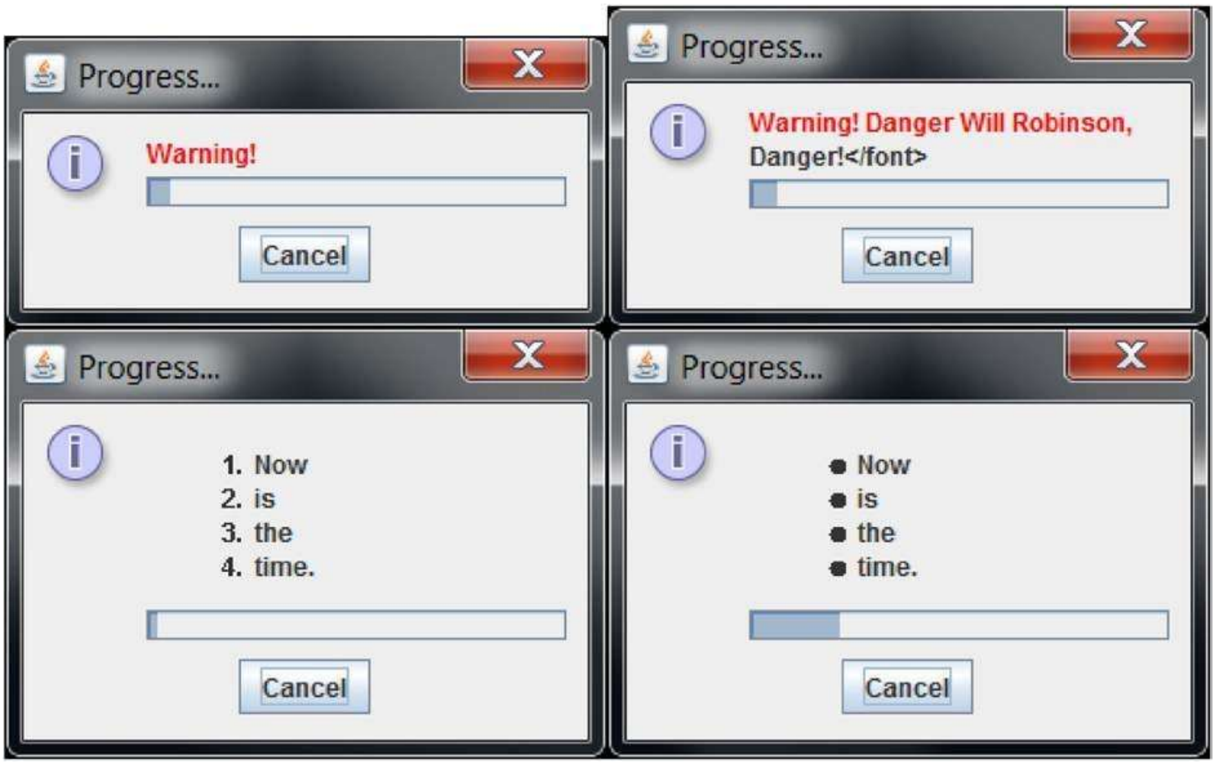


Figure 18-7. *ProgressMonitor with HTML message strings*

The ProgressMonitor Note

In the ProgressMonitor constructor in Table 18-2, you can see that, in addition to the message parameter, a note parameter is also defined. What's the difference between the message and note parameters? As you just saw, the message parameter is an object that is represented as a JLabel, so you can use HTML tags to control how it is displayed (within limits). Additionally, it is static for the life of the ProgressMonitor. The note argument is a string that is used to initialize a note attribute. If a value of None (the Jython equivalent to the Java null value) is specified in the constructor, the value of the note attribute shouldn't change.

On the other hand, you can use the note field on the ProgressMonitor to provide information related to the progress that is being made. According to the Javadoc, you only need to call the setNote(...) method to update this portion of the ProgressMonitor. Figure 18-8 shows the output of the ProgressMonitor5.py script, which uses the note field to provide a numeric percentage indicating the progress.

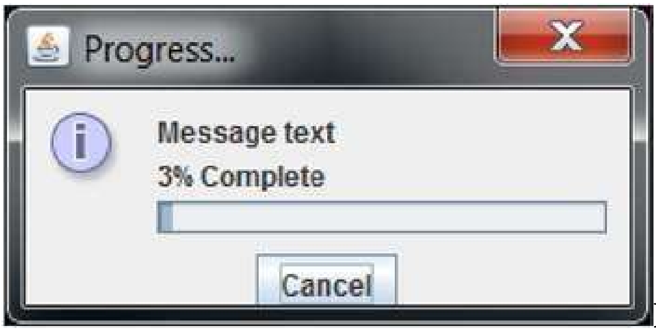


Figure 18-8. *ProgressMonitor with a note attribute from ProgressMonitor5.py*

Does anything about this catch your eye? When I first saw this, I was more than a little surprised by the fact that bottom of the Cancel button was encroaching on the bottom edge of the dialog box. After a bit of testing, I realized what was happening. It appears that the progress monitor's size is based on the initial parameter values. If an empty note string is provided in the ProgressMonitor constructor, then no vertical space is allocated to hold the string when it is changed. How do you fix this? One simple fix is to provide a note string containing one or more blanks. Figure 18-9 shows the output of this same script when the note is initialized with a single blank. I don't know about you, but I think that this looks significantly better.

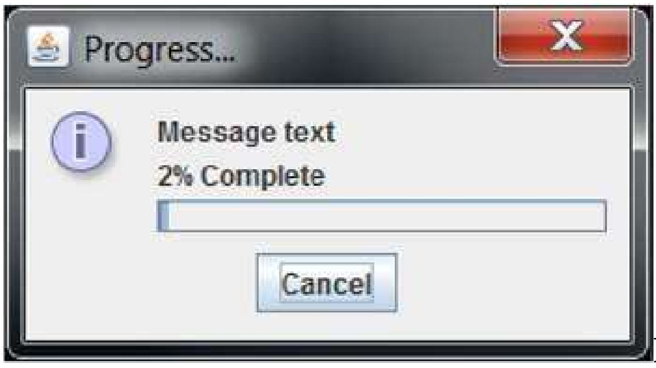


Figure 18-9. *ProgressMonitor with note initialized with* ' '

Can the note string contain HTML text? Yes it can, but again, I caution you to be careful about the kind of text that you use. It is important to remember that the ProgressMonitor class hierarchy, as shown in Listing 18-11, is based on an object and not a JComponent. So, you don't have the same kind of control over it that you do with normal Swing components, which means that you can't resize it after it is instantiated.

The ProgressMonitor parentComponent

Up to now, you have passed None as the value for the parentComponent argument. What role is this parameter supposed to play? Is there any reason for providing something other than None? The ProgressMonitor6.py script attempts to answer this question. In all of the other ProgressMonitor scripts, None was specified as the parentComponent value. This script references the application by using the self.frame value. Now, regardless of where the frame is positioned on the screen, when you press the Start button, the ProgressMonitor will be positioned with respect to the parentComponent window. This parameter provides the only control over how the ProgressMonitor appears on the screen.

Other ProgressMonitor Properties

One difference that exists between ProgressMonitor objects and other dialog boxes is that there is a delay between when the object is created and when it appears on the screen. Why is that? It's possible for there to be no reason for the ProgressMonitor to be displayed. Certain properties of the ProgressMonitor determine when and if a ProgressMonitor object should be displayed.

If you again take a look at the classInfo function introduced in Chapter 4, as shown in Listing 18-16, you'll see that there are some attributes defined in the ProgressMonitor class that aren't listed in the Javadoc. **Listing 18-16.** ProgressMonitor Attributes

```
wsadmin>from javax.swing import ProgressMonitor wsadmin>
wsadmin>classInfo( ProgressMonitor, attr = " ) javax.swing.ProgressMonitor

canceled, maximum, millisToDecideToPopup, millisToPopup minimum, note,
progress
| java.lang.Object
* class
| javax.accessibility.Accessible
* accessibleContext
wsadmin>
```

Table 18-4 identifies and describes these attributes. This should help you understand any delay in the appearance of the ProgressMonitor dialog. What happens is that when a progress value is set, the millisToDecideToPopup value determines how long the ProgressMonitor instance waits before trying to determine if the ProgressMonitor should be visible. If the estimated time to completion is less than the millisToPopup value, the ProgressMonitor dialog will not be displayed. After millisToPopup milliseconds, the dialog will appear. Once the progress value reaches the maximum value, the ProgressMonitor is hidden.

Table 18-4. *ProgressMonitor Attributes, Explained* **Attribute Name**
canceled
maximum
millisToDecideToPopup millisToPopup
minimum
note

progress

Description
Boolean value indicating if the Cancel button has been used.
Integer value initialized by constructor; accessible via getter and setter methods.
Integer value defaulting to 500 (0.5 sec); accessible via getter and setter methods. Integer value defaulting to 2000 (2 sec); accessible via getter and setter methods. Integer value initialized by constructor; accessible via getter and setter methods.

String value initialized by constructor; accessible via getter and setter methods.
Note: If this is initialized to None, any changes made via setNote() are ignored.
Write-only integer value modified via the setProgress() setter method. Values are limited by the values of minimum and maximum.
Note: When a progress is set to the maximum, the ProgressMonitor is hidden.

One question that keeps coming up with respect to ProgressMonitor dialog boxes relates to the fact that the dialog box is hidden when the process value is set to the maximum value. Generally, developers wonder whether there is any way to get the dialog box to stay visible once the maximum value is set. Unfortunately not. However, you can pass the setProgress(...) method a value in the range from minimum to maximum - 1. Unfortunately, if the range is small (e.g., 0 .. 10), a value of maximum - 1 might leave a visible gap at the upper end of the progress bar. In this case, you might want to consider using some kind of multiplier so that the value of maximum - 1 won’t be discernable.

Another possible annoyance related to doing this exists. Once the dialog is hidden, if you use the setProgress(...) method to cause it to reappear, it will not be located where the user left it. It will be located based on the original position, relative to the value of the parentComponent attribute. So, if the user moves the ProgressMonitor dialog box, the dialog box will appear to jump from where it is back to its original screen location.

ProgressMonitorInputStream Objects

There is another class, much like ProgressMonitor, that can use a ProgressMonitor object to determine the progress while data is read from an InputStream. As with ProgressMonitor objects, the ProgressMonitorInputStream[14] processing (reading data from the InputStream) should be performed by a separate (SwingWorker) type of task.

Unfortunately, space and time limitations don't allow me to provide more detail about this particular class.

Summary

This chapter covered monitoring and communicating progress to the user with a variety of techniques. The really nice thing that this shows you is how much control application developers have over the way they convey progress to their users. It's important to understand the implications and consequence of each choice you make, so keep this in mind. In next chapter, you'll learn about internal frames, which allow you to create even more interesting applications.

[14]See http://docs.oracle.com/javase/8/docs/api/javax/swing/ProgressMonitorInputStrea

Chapter 19

Internal Frames

Up to this point, the applications you've seen have been able to create and use individual windows. Almost every sample has used a single JFrame to display information and interact with the users. Now you're going to look at using internal frames. First you will take a quick look at a collection of inner frames, because there are times when applications can use a variety of views in order to convey different information to the users. The chapter begins by comparing internal frames to the JFrame used in the previous applications. Then you'll build an application that uses internal frames that display information in a variety of ways.

Looking at Inner Frames

Figure 19-1 shows the output of a simple application that displays three inner frames.

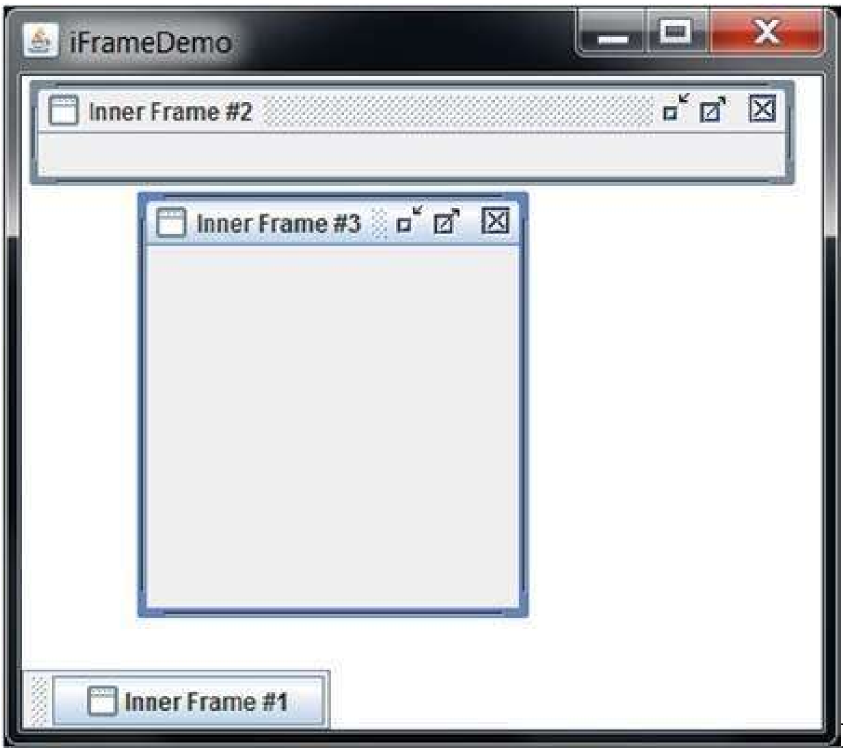


Figure 19-1. *Simple application with three inner frames*

Listing 19-1 shows the code required to produce this kind of simple inner frame. As you can see, even though very little code is required, it is quite functional.[1] The script, which you can find in the code\Chap_19\iFrameDemo.py file, shows that each of the inner frames can be moved, resized, closed, iconified, restored, and even maximized within the available space.

Listing 19-1. iFrameDemo Class Within iFrameDemo.py

```
class iFrameDemo( java.lang.Runnable ) :
def run( self ) :
screenSize = Toolkit.getDefaultToolkit().getScreenSize()
w = screenSize.width >> 1 # 1/2 screen width
h = screenSize.height >> 1 # 1/2
```

```
screen height 13| x = ( screenSize.width - w ) >> 1
14| y = ( screenSize.height - h ) >> 1
15| frame = JFrame(
16| 'iFrameDemo',
17| bounds = ( x, y, w, h ), # location & size 18| defaultCloseOperation =
JFrame.EXIT_ON_CLOSE 19| )
20| desktop = JDesktopPane()
21| for i in range( 3 ) :
22| inner = JInternalFrame(
23| 'Inner Frame #%d' % ( i + 1 ),
24| 1, # Resizeable 25| 1, # Closeable 26| 1, # Maximizable 27| 1, # Iconifiable
28| visible = 1, # setVisible( 1 ) 29| bounds = ( i * 25 + 25, i * 25 + 25, 250, 250
) 30| )
31| desktop.add( inner )
32| frame.setContentPane( desktop )
33| frame.setVisible( 1 )
```

Table 19-1 describes each of the steps in Listing 19-1. You will be learning about these steps in more detail throughout this chapter.

[1]Please note, however, that the iFrameDemo.py script output requires user manipulation to look like the image seen in Figure 19-1. This is discussed further in section 19-2.

Table 19-1. *iFrameDemo Class, Explained*

Lines	**Description**
10	Determines the size of the current screen.
11	Makes the width of the application use half the physical width of the screen.
12	Makes the height of the application use half the physical height of the screen.
13-14	Computes the upper-left corner of the application to center the application window.
15-19	Instantiates the frame using the specified parameters.
20	Creates a JDesktopPane onto which the InternalFrames will be added.
21-31	Creates three InternalFrames, one at a time, and adds them to the desktop. **Note:** The bounds keyword argument is used to size and position the inner frame on the JDesktopPane.
32	Replaces the frame ContentPane with the populated desktop.
33	Makes the application frame visible.

Before you investigate the JDesktopPane[2] class shown in Listing 19-1, it’s a good idea to learn more about layers and about the JLayeredPane[3] on which this class is based.

Layers

Let’s take a quick look at a simple script that demonstrates layered components using a trivial component—the JLabel object. Figure 19-2 shows a group of seven overlapping colored labels. Note how the text is centered across the top of each label, and how the labels are stacked to make all of the text visible.

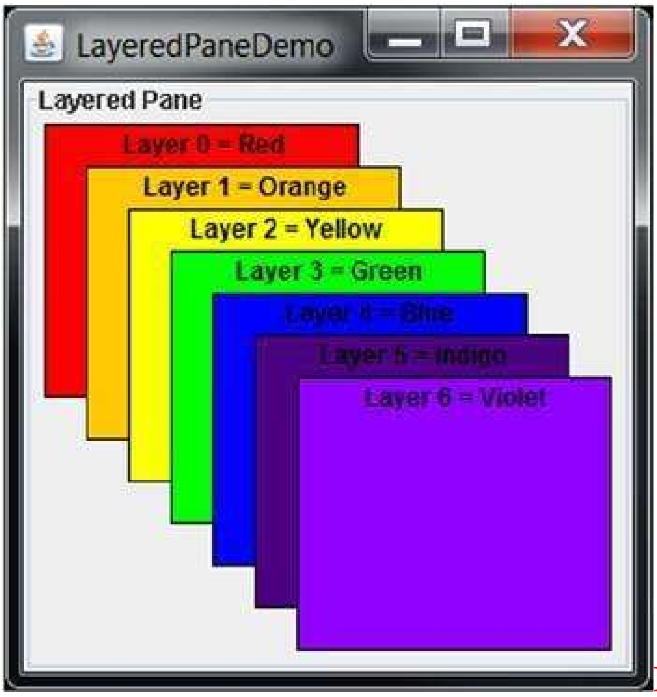


Figure 19-2. *Layered labels*

[2]http://docs.oracle.com/javase/8/docs/api/javax/swing/JDesktopPane.html.

[3]http://docs.oracle.com/javase/8/docs/api/javax/swing/JLayeredPane.html.

Listing 19-2 shows the relevant part of the LayeredPaneDemo class from the LayeredPaneDemo.py script.

Listing 19-2. LayeredPaneDemo Class, Part 1

```
class LayeredPaneDemo( java.lang.Runnable ) :
def run( self ) :
...
frame.setContentPane( self.createLayeredPane() )
frame.pack()
frame.setVisible( 1 )
def createColoredLabel( self, text, color ) :
```

```
22| return JLabel(
23| text,
24| opaque = 1,
25| size = ( 150, 130 ),
26| background = color,
27| foreground = Color.black,
28| verticalAlignment = JLabel.TOP,
29| horizontalAlignment = JLabel.CENTER,
30| border = BorderFactory.createLineBorder( Color.black )
31| )
```

Unfortunately, this class is too long to fit onto one page, so it has to be split into pieces. Table 19-2 describes the portion of the LayeredPaneDemo shown in Listing 19-2.

Table 19-2. *Comments About the LayeredPaneDemo Class, Part 1*

Lines	**Description**
12-20	This portion of the run() method should be familiar to you by now. The only new part is where the default ContentPane is replaced by the result of calling the createLayeredPane() in line 18.
21-31	The createColoredLabel(...) method creates and returns a JLabel containing the specified text and using the specified color. Each label object is roughly square, and the text is positioned in the center of the top, with a black border.

The remainder of the LayeredPaneDemo class, the createLayeredPane(...) method, is shown in Listing 19-3. The purpose of this method is to create and return a layered pane object populated with a collection of colored labels.

Listing 19-3. LayeredPaneDemo Class, Part 2

```
32| def createLayeredPane( self ) :
33| colors = [
34| ( 'Red' , Color.red ),
35| ( 'Orange', Color.orange ),
36| ( 'Yellow', Color.yellow ),
37| ( 'Green' , Color.green ),
38| ( 'Blue' , Color.blue ),
39| ( 'Indigo', Color( 75, 0, 130 ) ),
40| ( 'Violet', Color( 143, 0, 255 ) )
41| ]
42| result = JLayeredPane(
43| border = BorderFactory.createTitledBorder( 44| 'Layered Pane'
45| ),
46| preferredSize = Dimension( 290, 280 ) 47| )
48| position, level = Point( 10, 20 ), 0 49| for name, color in colors :
50| label = self.createColoredLabel( 51| 'Layer %d = %s' % ( level, name ), 52|
color
53| )
54| label.setLocation( position )
55| position.x += 20
56| position.y += 20
57| result.add( label, level, 0 )
58| level += 1
59| return result
```

Table 19-3 describes the statements found in this method.

Table 19-3. *Comments About the LayeredPaneDemo Class, Part 2*

Lines	**Description**
32-59	The createLayeredPane(...) method is used to create, populate, and return a JLayeredPane object instance.
33-41	The colors array holds information about the text and colors to be used to create each of the label objects.
42-47	Creates the JLayeredPane object with the specified preferred size and a title.
48	Each label will be positioned in an overlaid fashion, at a different layer on the pane, starting with these values.
49-58	Loops over the available colors (defined in 33-41), creating and positioning the labels accordingly.
57	This statement adds the current label to the layered pane at the specified layer and proper position. I will discuss this method in more detail shortly.
58	Increments the level variable used to indicate the layer number of the next label.

59 Returns the populated JLayeredPane object instance.
Position Within the Layer

You might wonder what the third parameter on the add(...) method (line 57 in Listing 19-3) is used for. It identifies the position, within the layer, of the component being added. Figure 19-3 shows what happens when you don't include the position parameter. Notice how in Figure 19-2, the top portion of each label (including the text) was visible.

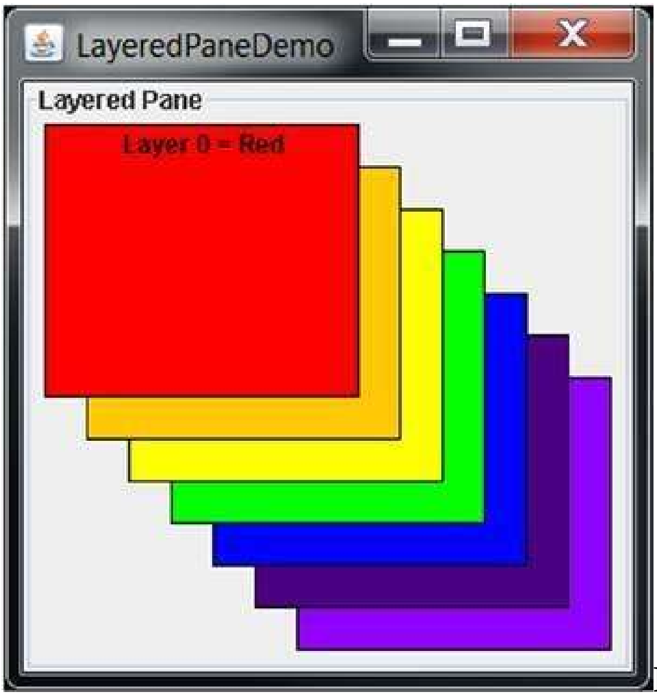


Figure 19-3. *Layered Labels without the position parameter specified*

As you can see, the layers are now stacked in reverse order, with the red label on top. If you add a print statement after the add() method statement and display the value of level (the intended level of the component being added) and the result of calling the getLayer(...) method for the label component on the layered panel, you'll see that an unintended add(...) method is being invoked and you aren't specifying the level, as expected. Listing 19-4 shows the proposed statement changes.

Listing 19-4. Changes to the createLayeredPane() Method

```
| ...
57| result.add( label, level, 0 )
```

```
58| print level, result.getLayer( label )
| ...
```

The output of the getLayer(...) method call for each label will show that every label is, in fact, being added to layer zero. With the three argument add(...) method calls, the output of the getLayer(...) method call will show the expected layer value.

So, what is the position supposed to do and when does it come into play? It is used when multiple components are at the same layer. It is used to determine the relationship of components on the same level. The position value should be an integer from -1 to N - 1, where N is the number of components in layer. The larger the value (the closer to N) of the position, the deeper the component.

The exception to this is -1, which is considered the same as N - 1, which means that it will be the deepest component on the layer. That’s why you specify a value of zero on the add(...) method when you add a component to a layer. This will position the newest component as the uppermost position within the layer (closest to the user).

The JDesktopPane Class

Now that you’ve seen how layers work, you’ll be better able to understand the JDesktopPane class. It is generally best to have internal frames added to a JDesktopPane instance in order to show and manipulate them. If you look back at Figure 19-1, you should notice familiar JFrame icons manipulating the internal frames. It should be clear how different these are from something as simple as a label.

There was another reason for discussing the JLayeredPane before moving onto the JDesktopPane, and that is the add(...) method as shown in Listing 19-1, line 31. If you take another look at Figure 19-1, you’ll see that the internal frames have been moved around so that they don’t overlap. This is because the initial output of the iFrameDemo script has the layering in an unexpected order. Unexpected until you understand which of the add(...) methods should have been used. The single argument add(...) method is used in Listing 19-1. If, on the other hand, you had used the three argument version, as shown in line 57 of Listing 19-3, the first internal frame (Inner Frame #3) would have been completely visible. Figure 19-4 shows the ordering of the internal frames when using the single and multiple argument add(...) methods.[4] As a user of GUI applications, which output would you prefer?

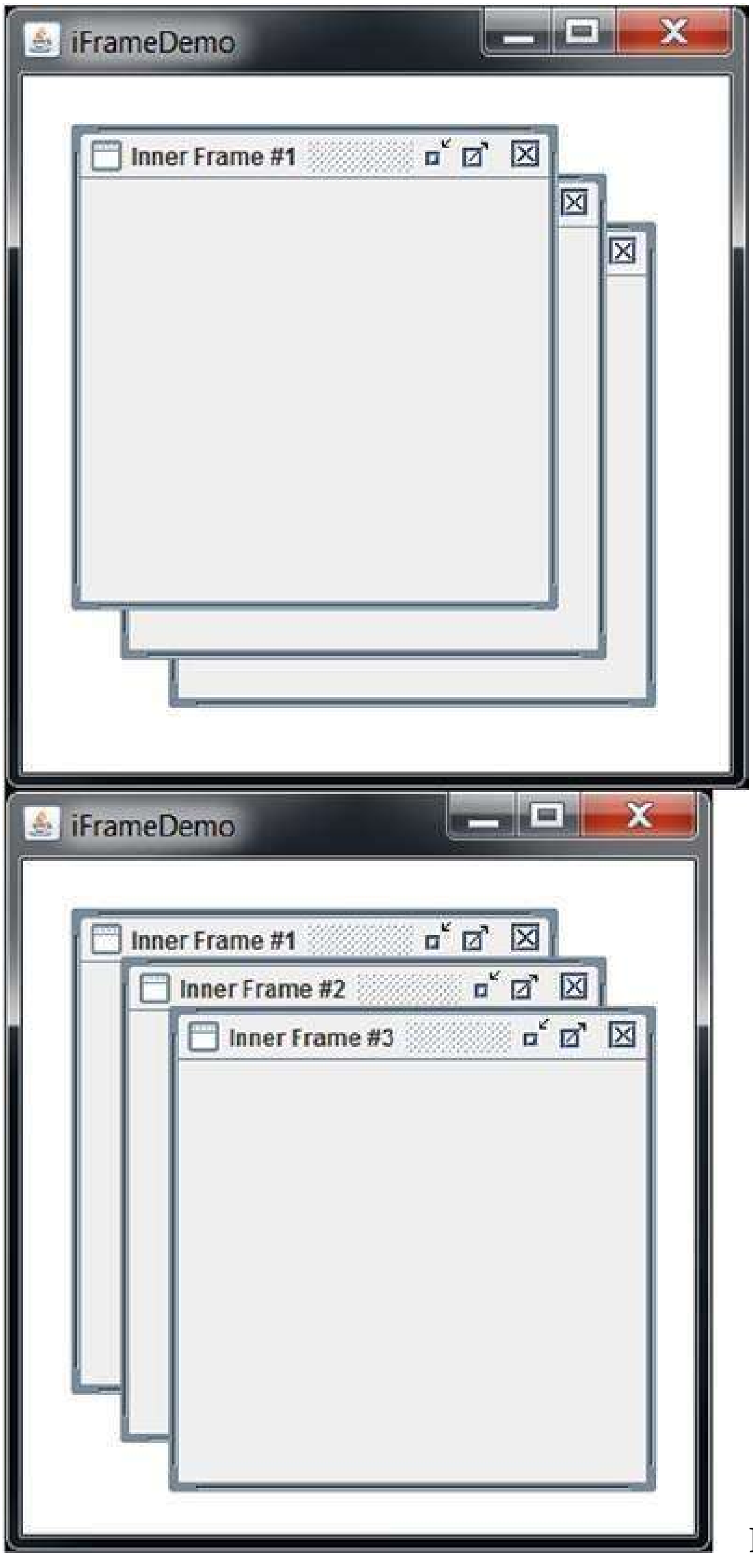


Figure 19-4. *iFrameDemo using single (top) and multiple (bottom) argument* add(...) *methods*

[4] The three argument add(...) method call can be seen as a comment in the iFrameDemo.py sample script. The Javadoc for which is http://docs.oracle.com/javase/8/docs/api/java/awt/Container.html#add%28java.av %20java.lang.Object,%20int%29.

Another thing to note about these sample scripts is that when the inner frames are created, their position needs to be specified so that all of the inner frames aren't stacked on top of each other, thereby hiding the frames underneath.

One thing that you haven't learned about in much detail yet is the JDesktopPane instance onto which all of these inner frames are added. In fact, I have only mentioned its add(...) method. There have to be more, don't there? Of course there are more. In fact, there are far too many to discuss in this chapter. In fact, the sample scripts in this chapter don't use anything but the add(...) method. That doesn't mean that they aren't useful. I just don't have the space or time required to do each of them justice.

More complex scripts are likely to use the JDesktopPane methods to manipulate the inner frames. To give you a little idea what some of the methods can do, I've listed a few of the most useful in Table 19-4. Remember that this is only a partial list.

Table 19-4. *Some Useful JDesktopPane Methods*

Returned Value	**Method Signature and Role**
JInternalFrame[]	getAllFrames() Returns all JInternalFrames currently present in the desktop container.
JInternalFrame[]	getAllFramesInLayer(int layer) Returns all JInternalFrames currently contained at the specified layer of the desktop object.
JInternalFrame	getSelectedFrame() Returns the currently active JInternalFrame in the desktop, or None if none is selected.
void	remove(int index) Removes the specified component from the container.
void	removeAll()

Removes all the components from the container.

JinternalFrame selectFrame(boolean forward)
Selects the next component in the container if the value of forward is true (1) or the previous component if the forward value is false (0).

void setSelectedFrame(JInternalFrame f)
Makes the specified JInternalFrame active.

JFrame or JInternalFrame?

When trying to decide how to implement an application, the similarity of the JFrame and JInternalFrame[5] classes might lead you to think that you can start with one and easily switch to the other should the need arise. Unfortunately, it's not quite as simple as that. It's better to pick one from the start and stick with it. It's possible that you could pick one way and soon realize that the other technique would have been better. If you haven't invested too much time or effort, it might be better to start over at that point. Here are some questions to consider to help you decide which approach is more advantageous for your needs.

• How contained and well defined is your application? Does it make more sense for you to see one, or multiple kinds or views of data at the same time? If you only need to see one view, then a single frame, possibly with something like tabbed panes might make more sense.

• How much data sharing will you need to do for the different portions of your application? If each aspect of your application is separate and distinct, there might be some advantage to seeing multiple views or pieces of data. If this is the case, multiple internal frames may be a better approach.

• Do multiple instances of the same kind of view make better sense, or is a single perspective sufficient? Might multiple views add complexity or confusion? If multiple, simultaneous views are advantageous, using multiple internal frames might be the best approach. However, try to consider the additional complexity that might be required by this choice.

One of the really powerful things about using Jython to produce or prototype your graphical application is how easy it is to create a proof of concept. This allows you to start your development and improve your knowledge about what exactly you want your program to do. This might help you decide which approach is best—a single frame or multiple internal frames.

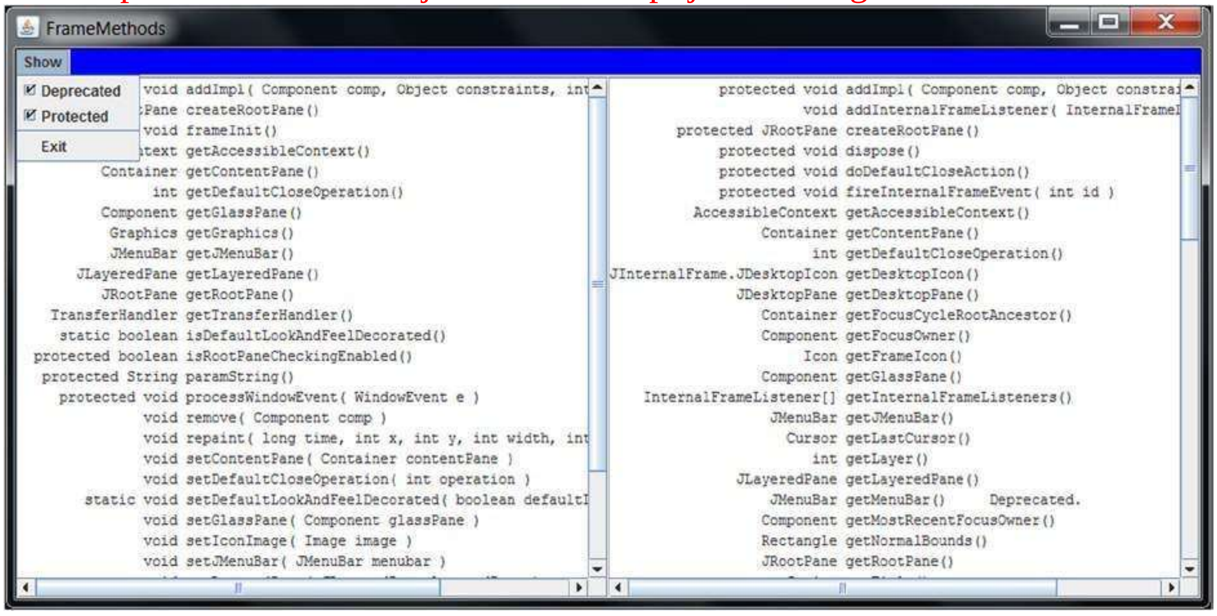
The JInternalFrame Class

One way to determine the difference between the JInternalFrame and JFrame classes is by comparing the methods available to each. A quick glance shows that the JFrame class has about 30 methods and the JInternalFrame class has about 80. This would appear to be a huge difference, which in some ways, it is. Looking a bit closer at the methods, you'll see that the JInternalFrame class has a large number of methods (46) that are identified as protected, but the JFrame class has only eight. Additionally, the JInternalFrame class has some deprecated methods.

JFrame and JInternalFrame Methods

While on the topic of graphical applications, consider what it would take to create a small application that showed the JFrame and JInternalFrame methods side-by-side and determined whether the protected and deprecated methods should be viewable. Figure 19-5 shows an image from this simple application. The menu selections allow you to hide the deprecated and protected methods.

[5]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JInternalFrame.html.



```
                 void addImpl( Component comp, Object constraints, int
                :Pane createRootPane()
                 void frameInit()
                :text getAccessibleContext()
            Container getContentPane()
                  int getDefaultCloseOperation()
            Component getGlassPane()
             Graphics getGraphics()
             JMenuBar getJMenuBar()
         JLayeredPane getLayeredPane()
            JRootPane getRootPane()
      TransferHandler getTransferHandler()
       static boolean isDefaultLookAndFeelDecorated()
    protected boolean isRootPaneCheckingEnabled()
     protected String paramString()
       protected void processWindowEvent( WindowEvent e )
                 void remove( Component comp )
                 void repaint( long time, int x, int y, int width, int
                 void setContentPane( Container contentPane )
                 void setDefaultCloseOperation( int operation )
          static void setDefaultLookAndFeelDecorated( boolean defaultI
                 void setGlassPane( Component glassPane )
                 void setIconImage( Image image )
                 void setJMenuBar( JMenuBar menubar )
```

```
                protected void addImpl( Component comp, Object constrai
                          void addInternalFrameListener( InternalFrameI
          protected JRootPane createRootPane()
                protected void dispose()
                protected void doDefaultCloseAction()
                protected void fireInternalFrameEvent( int id )
             AccessibleContext getAccessibleContext()
                     Container getContentPane()
                           int getDefaultCloseOperation()
JInternalFrame.JDesktopIcon getDesktopIcon()
                  JDesktopPane getDesktopPane()
                     Container getFocusCycleRootAncestor()
                     Component getFocusOwner()
                          Icon getFrameIcon()
                     Component getGlassPane()
       InternalFrameListener[] getInternalFrameListeners()
                      JMenuBar getJMenuBar()
                        Cursor getLastCursor()
                           int getLayer()
                  JLayeredPane getLayeredPane()
                      JMenuBar getMenuBar()      Deprecated.
                     Component getMostRecentFocusOwner()
                     Rectangle getNormalBounds()
                     JRootPane getRootPane()
```

Figure 19-5. *FrameMethod application images*

To create this fairly simple application, you first have to figure out where the data exists and in what format. Based on this, you need to figure out how to get the data into your application. For simplicity's sake, I created two text files, one for each class, and one line for each method. Each line has three fields:

- The optional modifier and return type of the method
- The method signature (the method name and parameter list)
- The method abstract

The fields are separated by the | delimiter. So, the application needs to load the text from the file and process each line. Listing 19-5 contains some utility methods from the FrameMethods.py script.[6] The textFile(...) method returns a string containing the file contents and the parse(...) method formats each line to contain only the first two fields from the file, with all of the method names aligned on the first character in the method name.

Listing 19-5. FrameMethods.py Utility Methods

```
def parse( self, text ) :
data = [ line.split( ' | ' ) for line in text.splitlines() ]
width = max(
[ len( result ) for result, sign, desc in data ]
)
return '\n'.join(
[
'%*s %s' % ( width, result, sign )
for result, sign, desc in data
]
)
...
def textFile( self, filename ) :
result = "
try :
f = open( filename )
result = f.read()
f.close()
except :
Type, value = sys.exc_info()[ :2 ]
result = '%s\n%s' % ( Type, value )
return result
```

[6]See ...\code\Chap_19\FrameMethods.py in the sample directories.

Listings 19-6 and 19-17 show the run(...) method from the FrameMethods class, which is used to create the application frame, define the layout, load the files, and populate the panes and the application menu. The contents and structure of this method should be quite familiar to you by now, so I won’t bother describing it in detail.

Listing 19-6 shows the run(...) method of the FrameMethods.py script. In it, you can see how the frame is composed of two JTextArea sections (lines 47-53 and lines 58-64), each of which is contained in a JScrollPane (lines 54 and 65). Each text area is populated from one of the text files used by this script.

Listing 19-6. FrameMethods run() Method

```
15|class FrameMethods( java.lang.Runnable ) :
| ...
36| def run( self ) :
37| frame = JFrame(
38| 'FrameMethods',
39| size = ( 1000, 500 ),
40| locationRelativeTo = None,
41| layout = GridLayout( 0, 2 ),
42| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
43| )
44| self.one = self.parse(
45| self.textFile( 'JFrame Methods.txt' )
46| )
47| self.left = JTextArea(
48| self.one,
49| 20,
50| 40,
51| editable = 0,
52| font = Font( 'Courier' , Font.PLAIN, 12 )
53| )
54| frame.add( JScrollPane( self.left ) ) 55| self.two = self.parse(
56| self.textFile( 'JInternalFrame Methods.txt' ) 57| )
58| self.right = JTextArea(
59| self.two,
60| 20,
61| 40,
editable = 0,
font = Font( 'Courier' , Font.PLAIN, 12 )
)
frame.add( JScrollPane( self.right ) )
frame.setJMenuBar( self.makeMenu() )
frame.setVisible( 1 )
```

In case you are interested, the instance variables one and two are used by the event handler methods to determine the text that will be displayed in each text area. Listing 19-7 shows the makeMenu() method from the same script. As you can see, this method creates, populates, and returns the menu bar and its entries, which have the associated actionPerformed event handling methods assigned as the showItems() method.

Listing 19-7. FrameMethods makeMenu() Method

```
def makeMenu( self ) :
menuBar = JMenuBar(
background = Color.blue,
foreground = Color.white
)
showMenu = JMenu(
'Show',
background = Color.blue,
foreground = Color.white
)
self.deprecated = JCheckBoxMenuItem(
'Deprecated',
1,
actionPerformed = self.showItems
)
showMenu.add( self.deprecated )
self.protected = JCheckBoxMenuItem(
'Protected',
1,
actionPerformed = self.showItems
)
showMenu.add( self.protected )
showMenu.addSeparator()
showMenu.add(
JMenuItem(
'Exit',
actionPerformed = self.exit
)
)
menuBar.add( showMenu )
return menuBar
```

Listing 19-8 shows the showItems(...) method used as the menu item ActionListener event handler. It, in turn, may use the findNot(...) method to filter the data, removing all lines containing the specified text (either “deprecated” or “protected”).

Listing 19-8. The showItems() and findNot() Methods

```
def findNot( self, data, text ) :
return '\n'.join(
[
line for line in data.splitlines()
if line.lower().find( text ) < 0
]
)
...
def showItems( self, event ) :
item = event.getActionCommand()
one = self.one
two = self.two
if not self.deprecated.isSelected() :
one = self.findNot( one, 'deprecated' )
two = self.findNot( two, 'deprecated' )
if not self.protected.isSelected() :
one = self.findNot( one, 'protected' )
two = self.findNot( two, 'protected' )
self.left.setText( one )
self.right.setText( two )
```

I’ll admit that I created this application quickly. It was one of those “rapid prototyping” opportunities that took about two hours, all told. Most of that time was spent creating the input files. It only required about 30 minutes to create the actual application, once I had the data in place. Not bad for a quick “proof of concept,” wouldn’t you say? The really neat part was that it is more than adequate, as is, to deal with the menu selection events. At first, I wasn’t sure how rapidly the application would update after a menu selection was made.

Watching it in action, though, I was pleasantly surprised and am very happy with the responsiveness.

Does that mean that this is the best way to go? Absolutely not. That’s one of the great things about being able to create a quick proof of concept script like this. It can help you understand the problem better, as well as give you some ideas about improvements and additional features. For example, based on this example, I can imagine a more useful application that:

• Adds menu items to identify the classes to be compared, such as allows users to specify input sources (URL or filename).

• Uses jsoup (Chapter 15) to retrieve and process the Javadoc URLs.
• Uses the JList class (Chapter 9) to allow a method to be selected. If the same method signature exists in the other list, highlights it as well.

• Adds selection logic to highlight the corresponding method in the other pane, if one exists.

In fact, the more I think about it, this might be a really good enhancement to the javadocInfo script in Chapter 15. What do you think? However, since we’re discussing internal frames maybe that would be a good way to do it. . . It’s a thought.

More JFrame and JInternalFrame Differences

One reason for looking at the differences between the JFrame and JInternalFrame classes is to understand the extent. Besides looking at their methods, you might also wonder about the differences in their class hierarchies. Listing 19-9 shows the output of the classInfo function, which was introduced in Chapter 4. It emphasizes that even though the class names are similar, they are very different. One of the most important differences was pointed out in Chapter 1, where the JFrame class was identified as a top-level container.

Listing 19-9. JFrame and JInternalFrame Class Hierarchies

```
wsadmin>from javax.swing import JFrame, JInternalFrame wsadmin>
wsadmin>classInfo( JFrame )
javax.swing.JFrame
| java.awt.Frame
| | java.awt.Window
| | | java.awt.Container
| | | | java.awt.Component
| | | | | java.lang.Object
| | | | | java.awt.image.ImageObserver
| | | | | java.awt.MenuContainer
| | | | | java.io.Serializable
| | | javax.accessibility.Accessible
| | java.awt.MenuContainer
| javax.swing.WindowConstants
| javax.accessibility.Accessible
| javax.swing.RootPaneContainer
wsadmin>
wsadmin>classInfo( JInternalFrame )
javax.swing.JInternalFrame
| javax.swing.JComponent
| | java.awt.Container
| | | java.awt.Component
| | | | java.lang.Object
| | | | java.awt.image.ImageObserver
| | | | java.awt.MenuContainer
| | | | java.io.Serializable
| | java.io.Serializable
| javax.accessibility.Accessible
| javax.swing.WindowConstants
| javax.swing.RootPaneContainer
wsadmin>
```

The differences between these two classes are significant, even though the function they provide is quite similar in nature. One area that differs significantly is the type of events recognized by each class.

JInternalFrame Events

In order to work well with internal fames, you need to understand the events that can occur in order to decide how your application should react when these events occur. The Javadoc for the JInternalFrame class identifies the InternalFrameListener[7] class as the base class for dealing with these kinds of events. Since it is an identified as an interface, it is probably better idea to consider using the InternalFrameAdapter[8] class as a base class.

What would it take to create an application that uses internal frames to help you better understand internal frame events? You can use one perpetual internal frame to display details about the internal frame events when they occur.[9] All of the others can be simple internal frames that can be created, minimized, restored, maximized, and closed.

The perpetual inner frame should contain a scrollable text area. This would allow your application to display the details and information about the events as they occur. Taking a quick look back to Listing 19-1, you can make some decisions about what arguments should be used to create this special internal frame. It would probably make the application much more complex if you allowed zero or multiple instances of this special internal frame to exist. So let's start simply by allowing only one perpetual instance in your application.

Listing 19-10 shows the first attempt of implementing a logging internal frame class. As you can see, it's really simple and straightforward. In fact, there really shouldn't be anything surprising about it.

Listing 19-10. The eventLogger Class, First Attempt from iFrameEvents1.py

```
class eventLogger( JInternalFrame ) :
def __init__( self ) :
JInternalFrame.__init__(
self,
'eventLogger',
1, # Resizeable - yes
0, # Closeable - no
0, # Maximizable - no
1, # Iconifiable - yes
visible = 1,
bounds = ( 0, 0, 250, 250 )
)
self.textArea = JTextArea(
20, # rows
40, # columns
editable = 0 # read-only
)
self.add( JScrollPane( self.textArea ) )
```

When instantiated, this class creates an internal frame, positions it, defines its size, and adds a scrollable pane containing a read-only text area to it. Wow, that was easy. Figure 19-6 shows the top-left portion of an application showing this internal frame.

[7] See http://docs.oracle.com/javase/8/docs/api/javax/swing/event/InternalFrameListene
[8]See http://docs.oracle.com/javase/8/docs/api/javax/swing/event/InternalFrameAdapte
[9]See http://docs.oracle.com/javase/8/docs/api/javax/swing/event/InternalFrameEvent.h

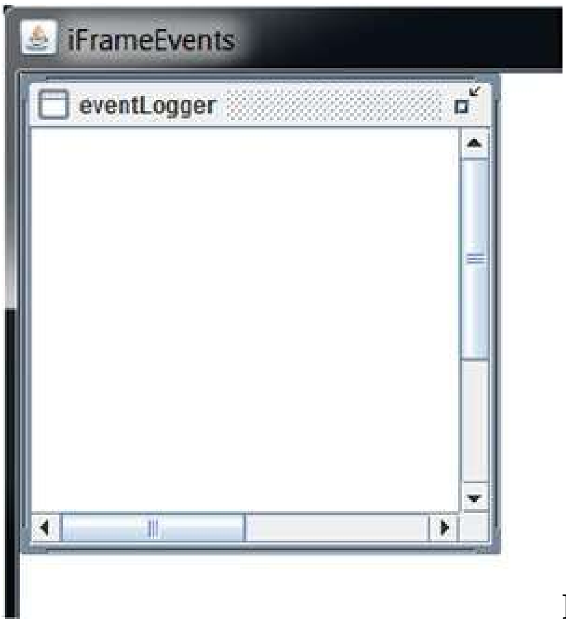


Figure 19-6. *iFrameEvents1.py initial display*

That's nice, but how is it supposed to log an InternalFrameEvent? How is the application supposed to create any other internal frames?
The first issue is easy enough to solve. You can have the eventLogger class instantiate an InternalFrameAdapter instance and provide a getter method to access it. This means that when any InternalFrameEvent occurs, the event handler will be able to access the event logger frame and add text to it.
The second issue is solved by adding a simple menu that allows you to create more internal frames.
Most of the revised script is in code\Chap_19\iFrameEvents2.py and is shown in the following listings. Listing 19-11 shows the utility methods for this class, which should look quite familiar by now.

Listing 19-11. Beginning of the iFrameEvents Class from iFrameEvents2.py

```
56|class iFrameEvents2( java.lang.Runnable ) :
57| def addIframe( self, event ) :
58| desktop = self.desktop
59| self.iFrameCount += 1
60| i = self.iFrameCount % 10
61| inner = JInternalFrame(
62| 'Inner Frame #%d' % self.iFrameCount,
63| 1, # Resizeable
64| 1, # Closeable
65| 1, # Maximizable
66| 1, # Iconifiable
67| bounds = ( i * 20 + 20, i * 20 + 20, 200, 200 )
68| )
69| inner.addInternalFrameListener( self.logger.getListener() )
70| inner.setVisible( 1 )
71| desktop.add( inner, i, 0 )
72| def exit( self, event ) :
73| sys.exit()
74| def menuBar( self ) :
75| result = JMenuBar()
76| newMenu = result.add( JMenu( 'New' ) ) 77| newMenu.add(
78| JMenuItem(
79| 'InnerFrame',
80| actionPerformed = self.addIframe 81| )
82| )
83| newMenu.addSeparator()
84| newMenu.add(
85| JMenuItem(
86| 'Exit',
87| actionPerformed = self.exit 88| )
89| )
90| return result
```

The addIframe(...) method, shown in lines 57-71, is called by the menu entry event handler to create a new internal frame. It is important to note that this method depends on the self.logger instance attribute value (see line 69), which is initialized in the run() method.

Listing 19-12 shows the run() method, which:
• Creates the initial application frame of the initial size and location
• Creates and adds the menu bar
• Creates a desktop instance to hold the internal frames
• Creates the special internal frame, which is an instance of the eventLogger class
• Adds the special eventLogger instance to the desktop
• Replaces the initial frame ContentPane with the desktop
• Makes the application visible

Listing 19-12. The run() Method from the iFrameEvents Class from iFrameEvents2.py

```
91| def run( self ) :
92| screenSize = Toolkit.getDefaultToolkit().getScreenSize()
93| w = screenSize.width >> 1 # 1/2 screen width
94| h = screenSize.height >> 1 # 1/2 screen height
95| x = ( screenSize.width - w ) >> 1
96| y = ( screenSize.height - h ) >> 1
97| frame = JFrame(
98| 'iFrameEvents2',
99| bounds = ( x, y, w, h ), # location & size 100| defaultCloseOperation =
JFrame.EXIT_ON_CLOSE 101| )
102| frame.setJMenuBar( self.menuBar() )
103| self.desktop = desktop = JDesktopPane()
104| self.logger = eventLogger()
105| desktop.add( self.logger, 0, 0 )
106| frame.setContentPane( desktop )
107| self.iFrameCount = 0
108| frame.setVisible( 1 )
```

The event listener that is added to each internal frame was instantiated by the run() method and is an instance of the eventLogger class, as shown in Listing 19-13. In fact, I added a getter for the eventAdapter instance, as shown in lines 54 and 55.

Listing 19-13. The eventLogger Class from iFrameEvents2.py

```
34|class eventLogger( JInternalFrame ) :
35| def __init__( self ) :
```

```
JInternalFrame.__init__(
self,
'eventLogger',
1, # Resizeable - yes
0, # Closeable - no
0, # Maximizable - no
1, # Iconifiable - yes
visible = 1,
bounds = ( 0, 0, 250, 250 )
)
self.textArea = JTextArea(
'', # Inital text
20, # rows
40, # columns
editable = 0 # read-only
)
self.eventListener = eventAdapter( self.textArea )
self.add( JScrollPane( self.textArea ) )
def getListener( self ) :
return self.eventListener
```

Listing 19-14 shows the eventAdapter class that is used by the eventLogger class. It is important to note how the constructor, in lines 16-18, requires the text area to be updated by the messages that this class needs to log. The log(...) utility method, found on lines 19-26, uses a regular expression (RegExp) to extract the name for the kind of event being logged and updates the specified text area with a line identifying the title of the internal frame for which the event occurred, as well as the kind of event.

■**Note** notice how all of the event handler methods are trivial calls to the log() method, which then does all of the real work.

Listing 19-14. The eventAdapter Class from iFrameEvents2.py

```
15|class eventAdapter( event.InternalFrameAdapter ) :
16| def __init__( self, textArea ) :
17| self.textArea = textArea
18| self.regexp = re.compile( '\[INTERNAL_FRAME_(\w+)]' )
19| def log( self, event ) :
20| title = event.getInternalFrame().getTitle()
```

```
mo = re.search( self.regexp, event.toString() )
if mo :
Type = mo.group( 1 ).capitalize()
else :
Type = 'unknown'
self.textArea.append( '\n%s : %s' % ( title, Type ) )
def internalFrameActivated( self, ife ) : self.log( ife )
def internalFrameClosed( self, ife ) : self.log( ife )
def internalFrameClosing( self, ife ) : self.log( ife )
def internalFrameDeactivated( self, ife ) : self.log( ife )
def internalFrameDeiconified( self, ife ) : self.log( ife )
def internalFrameIconified( self, ife ) : self.log( ife )
def internalFrameOpened( self, ife ) : self.log( ife )
```

I'll let you play with the sample application to get a feel for the kinds of events that are generated and the order in which they can occur.

More JInternalFrame Topics

Unfortunately, internal frames, and the manipulation thereof, are not simple or easy. There are lots of things to deal with. For example, you can use the simple iFrameEvents2.py script to get a better idea.

1. Start the script.
2. Choose New ➤ InnerFrame to create Inner Frame #1.
3. Use the minimize icon on Inner Frame #1 to iconify it.
4. Resize the application to hide the iconified inner frame.

How are you supposed to access the iconified inner frame without making the application larger? This is a simple example of one of the many kinds of things that you need to consider when you are dealing with inner frames. It may also help you understand the need for some of the JDesktopPane methods listed in Table 19-4.

Building an Application from Scratch

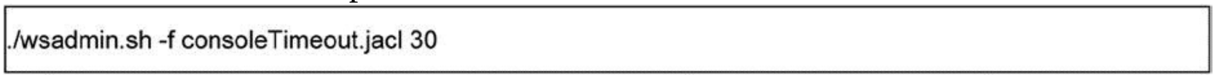
This section shows what it takes to build an application that uses the JInternalFrame class. Rather than showing the end result, the section iterates with fairly simple steps to create the script from absolutely nothing, to the point where you'll have a better understanding of what it takes to put these Swing components together into some semblance of a complete application.

Where should you start? Let's look for something, hopefully simple, that you can use to make an interesting application. From the WebSphere Application Server online documentation, you can find a simple script written in Jacl that can be used to change the console timeout value.[10] This value determines the length of time (in minutes) that the administrator console will remain active without user input.

Since the script in the documentation is written in Jacl, you'll begin by taking a look at it. In the code directory for this chapter, you can find the consoleTimeout_00.jacl script (see code\Chap_19\consoleTimeout_00.jacl). It is slightly different from the online documentation. It is different in that it allows users to specify the new timeout value on the wsadmin command line. The script in the documentation requires you to edit the script file and replace the two instances of <timeout value> with the desired timeout value (in minutes).

[10]See http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=/com.ibm.websphere.nd.doc/info/ ae/isc/cons_sessionto.html.

Figure 19-7 shows an example invocation of this wsadmin script, which presumes that the current working directory contains the WebSphere Application Server wsadmin.sh script.



```
./wsadmin.sh -f consoleTimeout.jacl 30
```

Figure 19-7. *Using the consoleTimeout_00.jacl script*

Simple Non-GUI Jython Version of the consoleTimeout Script

I don't know about you, but I find it a challenge to write and modify Jacl scripts.[11] So, let's start by converting this Jacl script to a roughly equivalent Jython script. Listings 19-14 and 19-15 show the consoleTimeout routine from this converted script. During the conversion, a little usability enhancement was added to display the current timeout value if no command-line parameters are provided. Additionally, if too many parameters are specified, some usage information is displayed.

Listing 19-15. The consoleTimeout Routine from consoleTimeout_01.py, Part 1

```
def consoleTimeout_01( cmdName = 'consoleTimeout_01' ) :
 argc = len( sys.argv ) # Number of args
 if argc > 1 : # Too many?
 Usage( cmdName ) # show Usage info
```

```
value = None
if argc == 1 :
value = sys.argv[ 0 ]
if not re.search( re.compile( '^\d+$' ), value ) :
print nonNumeric % locals()
Usage( cmdName )
dep = AdminConfig.getid( '/Deployment:isclite/' )
if not dep :
print noISCLite % locals()
Usage( cmdName )
appDep = AdminConfig.list( 'ApplicationDeployment', dep )
appConfig = AdminConfig.list( 'ApplicationConfig', appDep )
if not appConfig :
appConfig = AdminConfig.create(
'ApplicationConfig',
appDep,
[]
)
sesMgmt = AdminConfig.list( 'SessionManager', appDep )
if not sesMgmt :
sesMgmt = AdminConfig.create(
'SessionManager',
appConfig,
[]
)
```

This script should be pretty straightforward to those with a modicum of wsadmin scripting experience. It verifies that the user-supplied valid numeric input then uses some calls to the wsadmin scripting objects to locate the session manager configuration object so that its tuning parameters can be checked. Listing 19-16 deals with creating tuning parameters in order to set the invalidationTimeout attribute value. One interesting thing about this simple script is the fact that almost 60 lines are needed to provide this simple functionality.

[11]I'll admit it, it took me much longer than it should have to figure out how to modify the consoleTimeout.jacl script to accept and use command-line parameters. I am not proficient with Jacl since I use it so infrequently.

■ **Note** In case you are interested, you can look at the complete source code to see that the lines that aren't shown in these listings are comments, initialization statements, and a Usage(...) method, which displays appropriate information about how the script can and should be used.

Listing 19-16. The consoleTimeout Routine from consoleTimeout_01.py, Part 2

```
tuningParams = AdminConfig.showAttribute(
sesMgmt,
'tuningParams'
)
if value :
if not tuningParams :
AdminConfig.create(
'TuningParams',
sesMgmt,
[[ 'invalidationTimeout', value ]]
)
else :
AdminConfig.modify(
tuningParams,
[[ 'invalidationTimeout', value ]]
)
else :
if not tuningParams :
print noTPobj % locals()
else :
timeout = AdminConfig.showAttribute(
tuningParams,
'invalidationTimeout'
)
print currentVal % locals()
if AdminConfig.hasChanges() :
print saveConfig % locals()
AdminConfig.save()
```

First GUI Jython Version of the consoleTimeout Script

The first Jython Swing version of this script is shown in Listings 19-16 and 19-17. Remember that first impressions can be deceiving. Even though there appear to be many more statements in the Swing version (consoleTimeout_02.py), that’s simply because of the number of statements displayed on multiple lines so that they fit in the space available in the listings in this publication.

Listing 19-17 contains most of the run(...) method from the ConsoleTimeout class in this script. Note that very little effort was put into any complex layout of the Swing components. If you make the frame wider, I think that you’ll agree that it doesn’t look as nice as it could.

Listing 19-17. The ConsoleTimeout Class run(...) Method from consoleTimeout_02.py

```
class consoleTimeout_02( java.lang.Runnable ) :
def run( self ) :
frame = JFrame(
'consoleTimeout_02',
layout = FlowLayout(),
size = ( 180, 120 ),
locationRelativeTo = None,
defaultCloseOperation = JFrame.EXIT_ON_CLOSE
)
dep = AdminConfig.getid( '/Deployment:isclite/' )

...
if not self.tuningParms :
timeout = ''
messageText = "tuningParams object doesn't exist."
else :
timeout = AdminConfig.showAttribute(
self.tuningParms,
'invalidationTimeout'
)
messageText = ''
frame.add( JLabel( 'Timeout:' ) )
self.text = frame.add(
JTextField(
3,
text = timeout,
actionPerformed = self.update
)
)
frame.add( JLabel( 'minutes' ) )
```

```
self.message = frame.add( JLabel( messageText ) ) | ...
frame.setVisible( 1 )
```

Listing 19-18 contains the update(...) method from the same ConsoleTimeout class, which is almost identical to the previous code. It is important to remember that this method is an event handler.

Listing 19-18. The ConsoleTimeout Class update(...) Method from consoleTimeout_02.py

```
def update( self, event ) :
value = self.text.getText()
if not re.search( re.compile( '^\d+$' ), value ) :
text = 'Invalid numeric value: "%s"' % value
else :
if not self.tuningParms :
try :
AdminConfig.create(
'TuningParams',
self.sesMgmt,
[[ 'invalidationTimeout', value ]]
)
AdminConfig.save()
text = 'The TuningParams object has ' + \
'been created successfully'
except :
text = 'A problem was encountered while ' + \
'creating the TuningParams object.'
else :
try :
AdminConfig.modify(
self.tuningParms,
[[ 'invalidationTimeout', value ]]
)
AdminConfig.save()
text = 'Update successful.'
except :
text = 'A problem was encountered while ' + \
'updating the TuningParams object.'
self.message.setText( text )
```

It’s important to realize that this version of the script does a very bad thing. What is that, you ask? Well, take a look at the update(...) method again. It is the event handler code that verifies the user-specified input and attempts to modify the invalidation timeout attribute for the administration console application.

The big mistake is the fact that the event handler routine contains calls to wsadmin scripting object (AdminConfig) methods. These kinds of calls should always be done on a separate (SwingWorker) thread. Why? It's because they are likely to require a non-trivial amount of time (the first call took about five seconds) to complete the requested action and return. In the meantime, the application can't respond to any user events. That's why it is not a good way to implement GUI scripts.

What does this look like when you run the script? Figure 19-8 shows a sample image created by this script. Go ahead and test it out yourself. See what happens when you enter a value and press Enter. Is there any kind of indication on the GUI that something is happening? Nope, at least not until the calls to the AdminConfig objects are complete and control returns to the application. Is this how you want your GUI applications to work and interact with your users? Absolutely not; this kind of behavior is unacceptable.

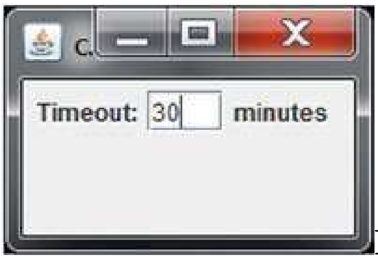


Figure 19-8. *Output from consoleTimeout_02.py*

Adding SwingWorker Instance to the Mix

Listing 19-19 shows how easy it is to take the statements from the update(...) method shown in consoleTimeout_02.py (that's Listing 19-18) and put them into an separate SwingWorker class. It also shows how simple the event handler update(...) method then becomes (see lines 122 and 122).

Listing 19-19. The WSAStask Class from consoleTimeout_03.py

```
10|class WSAStask( SwingWorker ) :
11| def __init__( self, app ) :
12| self.app = app # application reference
13| self.messageText = ''
14| SwingWorker.__init__( self )
15| def doInBackground( self ) :
16| problem = 'A problem was encountered while %s ' + \
'the TuningParams object.'
messageText = self.messageText
self.app.textField.setEnabled( 0 )
self.app.message.setText(
'<html>working...' + ( '&nbsp;' * 20 )
)
value = self.app.textField.getText() # JTextField value
if not re.search( re.compile( '^\d+$' ), value ) :
messageText = 'Invalid numeric value: "%s"' % value
else :
if not self.app.tuningParms :
try :
AdminConfig.create(
'TuningParams',
self.app.sesMgmt,
[[ 'invalidationTimeout', value ]]
)
AdminConfig.save()
messageText = 'The TuningParams object' + \
'has been created successfully'
except :
messageText = problem % 'creating'
else :
try :
AdminConfig.modify(
self.app.tuningParms,
[[ 'invalidationTimeout', value ]]
)
AdminConfig.save()
messageText = 'Update complete.'
except :
messageText = problem % 'updating'
def done( self ) :
self.app.textField.setEnabled( 1 )
self.app.message.setText( self.messageText )
class consoleTimeout_03( java.lang.Runnable ) :
...
def update( self, event ) :
WSAStask( self ).execute()
```

One thing that you should notice when this SwingWorker thread is performing an update is the fact that application text input field is disabled. This kind of attention to detail adds significantly to the user experience and should be considered essential when you are developing graphical applications. Additionally, a status message is displayed to show that the thread is executing. It is only when this separate processing is complete that the text input field is enabled and an updated status message is displayed.

Adding Menu Items

As you've previously seen, you can add significant usability to your applications with simple menu items. Figure 19-9 shows the impact of making some small changes to this script.

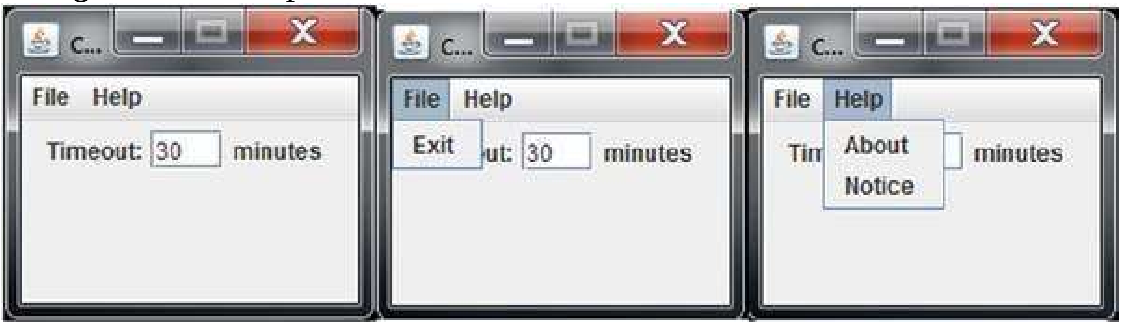


Figure 19-9. *Images from consoleTimeout_04.py*

Listing 19-20 shows the new and modified code from the application class in this updated script. The menuBar() method creates and populates the application menu bar. Three new methods were added to respond to each of the new menu actions. And that is pretty much it. All in all, this is a good return on investment for about 50 lines of code. I think that you'll agree that the additional user value is significant.

In case you're wondering, the about(...)and notice(...) method adds some nice functionality. The former serves the same kind of role normally performed by a usage message that a non-graphical application uses to tell users how the program should be used, and the latter is useful as a valuable disclaimer message.

Listing 19-20. New and Modified Code from consoleTimeout_04.py

```
87|class consoleTimeout_04( java.lang.Runnable ) :
| ...

110| def run( self ) :
```

```
| ...
119| frame.setJMenuBar( self.menuBar() )
| ...
178| frame.setVisible( 1 )
179| def Exit( self, event ) :
180| sys.exit( 0 )
181| def about( self, event ) :
182| text = __doc__.replace( '<', '&lt;'
183| ).replace( '>', '&gt;'
184| ).replace( ' ', '&nbsp;'
185| ).replace( '\n', '<br>' )
186| JOptionPane.showMessageDialog( 187| self.frame,
188| JLabel(
189| '<html>' + text, 190| font = monoFont 191| ),
192| 'About',
193| JOptionPane.PLAIN_MESSAGE 194| )
195| def notice( self, event ) : 196| JOptionPane.showMessageDialog( 197|
self.frame,
198| disclaimer,
199| 'Notice',
200| JOptionPane.WARNING_MESSAGE 201| )
202| def update( self, event ) : 203| WSAStask( self ).execute()
```

Changing from JFrame to JInternalFrame

For the next iteration, you’ll see how to convert the script to use internal frames. Begin by taking a look at the revised consoleTimeout class from the revised consoleTimeout_05.py script. Listing 19-21 shows the important methods from this class. As you saw earlier, internal frames need to be added to an instance of the JDesktopPane class. Overall, it shouldn’t be terribly new to you. You may, however, wonder about the call to the setSelected(...) method on line 186.

This isn’t an error. Table 19-3 shows that the JDesktopPane class has a similar method called setSelectedFrame(...). This is a method from the JInternalFrame class, and the parameter is a boolean value indicating that the specified object instance is being selected (in which case, a value that is recognized as true should be specified). What happens if you comment out this line in the script and execute it?

The internal frame doesn’t obtain the focus, which means that the current timeout value isn’t obtained and the input field remains empty. At least until you put focus on the internal frame, such as by clicking somewhere on it. Then you see that the input field is populated with the current timeout value.

Listing 19-21. Revised consoleTimeout Class from consoleTimeout_05.py, Part 1

```
167|class consoleTimeout_05( java.lang.Runnable ) :
168| def run( self ) :
169| self.frame = frame = JFrame(
170| 'consoleTimeout_05',
171| size = ( 428, 474 ),
172| locationRelativeTo = None,
173| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
174| )
175| frame.setJMenuBar( self.MenuBar() )
176| desktop = JDesktopPane()
177| if globals().has_key( 'AdminConfig' ) :
178| self.timeout = self.initialTimeout()
179| self.inner = TextField( self ) # JTextField only
180| desktop.add( self.inner )
181| else :
182| self.inner = self.noWSAS() # WebSphere not found 183| desktop.add( self.inner )
184| frame.add( desktop )
185| frame.setVisible( 1 )
186| self.inner.setSelected( 1 )
```

The initialTimeout() method (see line 178) isn’t shown because it contains the same steps that you’ve seen before to obtain the initial value, if one exists. The most important change in this script is shown in Listing 19-22. It contains the class used by all of the subsequent internal frame instances for this application. The first time you look at it, however, you may wonder why it is descended from InternalFrameListener and not from InternalFrameAdapter. That way, you wouldn’t need to have empty methods (ones like internalFrameClosed(...) that consist of a single pass statement).

Listing 19-22. InternalFrame Class from consoleTimeout_05.py, Part 1

```
class InternalFrame( JInternalFrame, InternalFrameListener ) :
def __init__( self,
title,
outer,
size,
location = None,
layout = None
) :
if location == None :
location = Point( 0, 0 )
if layout == None :
layout = FlowLayout()
JInternalFrame.__init__(
self,
title,
0, # resizeable = false
0, # closable = false
size = size,
internalFrameListener = self,
layout = layout
)
self.setLocation( location ) # keyword parm doesn't exist
self.outer = outer # application object
def internalFrameActivated( self, e ) :
self.outer.inner = e.getInternalFrame()
def internalFrameClosed( self, e ) :
pass
def internalFrameClosing( self, e ) :
pass
def internalFrameDeactivated( self, e ) :
self.outer.inner.message.setText( '' )
def internalFrameDeiconified( self, e ) :
pass
def internalFrameIconified( self, e ) :
pass
def internalFrameOpened( self, e ) :
pass
def getValue( self ) :
print 'InternalFrame.getValue() - not yet implemented'
return None
def setValue( self, value ) :
```

```
131| print 'InternalFrame.setValue() - not yet implemented' 132| def working( self ) :
133| print 'InternalFrame.working() - not yet implemented' 134| def finished( self ) :
135| print 'InternalFrame.finished() - not yet implemented'
```

Listing 19-23 shows what happens when you try to base an InternalFrame class on the JInternalFrame and the InternalFrameAdapter classes. Jython complains about this issue. Why doesn’t it complain about basing a class on JInternalFrame and the InternalFrameListener classes? This is allowed because the InternalFrameListener is actually an interface, not a class that can be instantiated.

Listing 19-23. Multiple Inheritance Issue

```
wsadmin>from javax.swing import JInternalFrame
wsadmin>from javax.swing.event import InternalFrameAdapter wsadmin>
wsadmin>class InternalFrame( JInternalFrame, InternalFrameAdapter ) :
wsadmin> def __init__( self, title ) :
wsadmin> JInternalFrame.__init__( self, title )
wsadmin>
WASX7015E: Exception running command: ""; exception information:

com.ibm.bsf.BSFException: exception from Jython:
Traceback (innermost last):
File "<input>", line 1, in ?
TypeError: no multiple inheritance for Java classes:
javax.swing.event.InternalFrameAdapter and javax.swing.JInternalFrame
wsadmin>
```

Listing 19-24 shows the TextField class from the consoleTimeout_05.py script. In it, you can see that the constructor (in lines 137-154) is responsible for populating the internal frame and providing the input components. Additionally, it includes instruction for the methods that are needed by this object. It is important to note that the internalFrameActivated(...) method, which is invoked when the object instance is activated, is responsible for updating the application attribute value that identifies the active internal frame, obtains the current timeout value from the application, and calls the object setValue(...) method to initialize the timeout value for this object instance.

Listing 19-24. TextField Class from consoleTimeout_05.py

```
136|class TextField( InternalFrame ) :
137| def __init__( self, outer ) :
138| InternalFrame.__init__(
139| self,
140| 'TextField',
141| outer,
142| size = ( 180, 85 ),
143| location = Point( 5, 5 )
144| )
145| self.add( JLabel( 'Timeout (minutes):' ) ) 146| self.text = self.add(
147| JTextField(
148| 3,
149| actionPerformed = outer.update 150| )
151| )
152| self.message = self.add( JLabel() ) 153| self.setVisible( 1 )
154| self.text.requestFocusInWindow() 155| def internalFrameActivated( self, e )
: 156| self.outer.inner = e.getInternalFrame() 157| self.setValue(
self.outer.timeout ) 158| def getValue( self ) :
159| return self.text.getText()
160| def setValue( self, value ) :
161| self.value = value
162| self.text.setText( value )
163| def working( self ) :
164| self.text.setEnabled( 0 )
165| def finished( self ) :
166| self.text.setEnabled( 1 )
```

All this is possible because of the framework provided by the InternalFrame class on which it is based. There are some important things to note about this class and how it fits into the application. One of these is the way in which this constructor begins by calling its base class constructor to initialize itself. One of the important steps performed by the base class constructor (InternalFrame.__init__) call, shown on lines 138-144, is the fact that it saves a reference to the application frame in an instance attribute called self.outer (see lines 90-110 in Listing 19-22).

What purpose does this application instance variable serve? It allows each inner frame to easily obtain access to this reference so that the event handlers can manipulate the appropriate fields. How does it do this? Take a look at the internalFrameActivated(...) method in lines 155-157. When an internal frame is activated, the application variable called self.outer is used to update the self.inner application variable, which identifies the active inner frame. By defining and updating this variable, the utility routines can easily access the "current" or "active" internal frame values with simple statements. All you have to do is ensure that every internal frame class implements these values the same way.

For example, this reference (self.inner) allows the object instance to specify the application update(...) method as the ActionListener event handler for the text field on this inner frame. This application update(...) method, as shown in Listing 19-25, is extremely simple. As an event handler routine, it should be simple.

Listing 19-25. The update(...) Method from consoleTimeout_05.py

```
272| def update( self, event ) :
273| self.timeout = self.inner.getValue()
274| WSAStask( self ).execute()
```

There are, however, some subtle changes needed in the WSAStask class in this script that is instantiated to create a separate thread to modify the timeout value. Listing 19-26 shows this modified WSAStask class from the consoleTimeout_05.py script. The important changes are all related to the fact that this class needs to work with the active internal frame that initiated the creation of the task.

Take a look at line 51 where a local variable, frame, is used to hold a reference to the active inner frame. This reference is then used to invoke the appropriate working() method (line 52), the appropriate getValue() method (line 53), the appropriate message.setText(...) method (lines 54-56 and 89), and the appropriate finished() method (line 88).

The rest of this chapter shows how easily you can create other classes to take advantage of this design simplification.
Listing 19-26. WSAStask Class from consoleTimeout_05.py

```
46|class WSAStask( SwingWorker ) :
```

```
def __init__( self, app ) :
self.app = app # application reference
self.messageText = ''
SwingWorker.__init__( self )
def doInBackground( self ) :
messageText = self.messageText
frame = self.app.inner # Active Inner Frame
frame.working()
value = frame.getValue()
frame.message.setText(
'<html>working...' + ( '&nbsp;' * 20 )
)
if not re.search( re.compile( '^\d+$' ), value ) :
messageText = 'Invalid numeric value: "%s"' % value
else :
TPobj = 'TuningParams'
success = 'The %s object was created successfully.'
problem = 'A problem was encountered %s the %s object.'
if not self.app.tuningParms :
try :
self.tuningParms = AdminConfig.create(
TPobj,
self.app.sesMgmt,
[[ 'invalidationTimeout', value ]]
)
AdminConfig.save()
messageText = success % TPobj
except :
messageText = problem % ( 'creating', TPobj )
else :
try :
AdminConfig.modify(
self.app.tuningParms,
[[ 'invalidationTimeout', value ]]
)
AdminConfig.save()
messageText = 'Update complete.'
except :
messageText = problem % ( 'updating', TPobj )
```

```
def done( self ) :
frame = self.app.inner
frame.finished()
frame.message.setText( self.messageText )
```

What does this mean as far as the application's appearance is concerned? Figure 19-10 shows part of the application output created by this script. The remainder of this chapter deals with iterations of this script that will fill in the blanks as far as this application output is concerned.

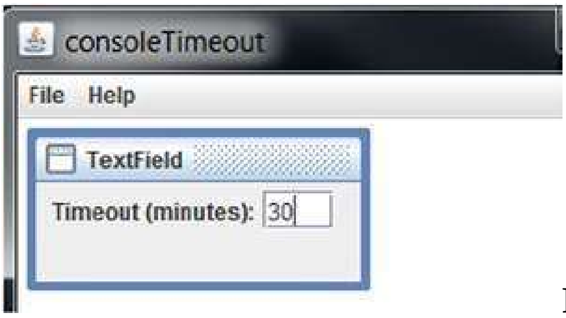


Figure 19-10. *Output from consoleTimeout_05.py*

Adding a Second Internal Frame Class

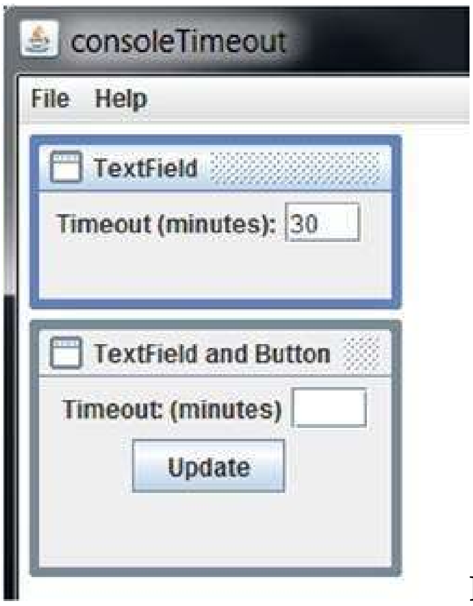
Until now, it really looks like a lot of code is required to create a fairly simple application with one internal frame. Was it worth it? In order to answer that question, take a look at what is required to add a second, different, internal frame. Listing 19-27 shows is the code required to define another class—the TextandButton class, which is based on the TextField class shown in Listing 19-24. It is almost trivial due to the fact that all of the class methods, other than the constructor, are in fact identical to those defined in the base TextField class. Isn't inheritance wonderful? You bet it is!

Listing 19-27. The TextandButton Class from consoleTimeout_06.py

```
class TextandButton( TextField ) :
def __init__( self, outer ) :
InternalFrame.__init__(
self,
'TextField and Button',
outer,
size = ( 180, 125 ),
location = Point( 5, 95 )
)
```

```
177| self.add( JLabel( 'Timeout: (minutes)' ) )
178| self.text = self.add(
179| JTextField(
180| 3,
181| actionPerformed = outer.update
182| )
183| )
184| self.button = self.add(
185| JButton(
186| 'Update',
187| actionPerformed = outer.update
188| )
191| self.setVisible( 1 )
192| self.text.requestFocusInWindow() 193|class consoleTimeout_06(
java.lang.Runnable ) : 194| def run( self ) :
|...
207| desktop.add( TextandButton( self ) )
```

This listing also shows, in line 207, the code needed to instantiate an object using this class and add it to the application desktop. Figure 19-11 shows the initial output of this iteration of the script. It's important to notice that you now have two internal frames, only one of which is active.



Timeout (minutes): 30

Timeout: (minutes)

Figure 19-11. *Output from consoleTimeout_06.py*

Adding a Third Internal Frame Class

Let’s see what it takes to add a very different internal frame classes to this application. Listings 19-28, 19-29, and 19-30 show another class based on the TextField class that adds a group of RadioButtons for commonly used timeout values. In addition, an “other”RadioButton is defined that allows the users to specify an uncommon value using an associated text field.

Note, however, that this class demonstrates a solution to an issue that may not be obvious without explanation. In Listing 19-28, line 228, you can see that the self.setting variable is defined and initialized in the class constructor. As the comment indicates, this value is used in the stateChange(...) method (found in Listing 19-30) and the setValue(...) method (shown in Listing 19-29). The purpose of this variable is to deal with a possible race condition that might occur. It is possible that the stateChange(...) method will be invoked before the setValue(...) method has completed. If this occurs, the state of the text field may be indeterminate, so the stateChange(...) method exits without making any changes if the value of self.setting is non-zero.

This is the kind of thing that can occur in event-driven applications. So you should be aware of this possibility and be on the lookout for it. You may want to see what happens to the script without this and other issues related to interrupt-driven applications.

This is a good time for me to bring up an “oops” moment. At this point, while testing the current script, I found a significant flaw. You can see what happens with the previous version of the script—consoleTimeout_06.py. I haven’t gone back to correct either of the previous iterations, just to give you a chance to see the flaw and compare the current version of the script to see how it handles things better.

What’s the error? It’s most obvious in the previous script because of the presence of multiple inner frames. Enter an invalid numeric value in the text input field and press Enter. A message is displayed indicating that the value is bad. So far, so good. Now put focus on the other inner frame and see what happens. The bad value is displayed as the current value in the selected inner frame. Is this that you want to happen? I don’t think so. This is exactly the kind of thing that you need to consider when using inner frames that can gain and lose focus at unexpected times.

So, how do you fix it? Well, you have to make some changes to a few of the existing methods. The biggest change needs to be made in the setValue(...) methods so that only valid values are saved. Note that the setValue(...) method only exists in the TextField class. The TextAndButton class reuses the base method, thereby limiting and simplifying the necessary changes.

It is important to note, however, that the new RadioButtons class has to implement this method because of the different data components in this new inner frame. The presence of the “other” radio button associated with a text input field means that you also have to change the update(...) method to properly deal with invalid input values.

Take a look at the resulting changes, focusing first on the new RadioButtons class. Listing 19-28 shows the constructors (__init__()) and getValue(...) methods. Notice how these are more complicated because they use RadioButtons and a text input field to display the current value.

One thing that is all too easy to overlook with Jython Swing code is how easily event handler methods can be assigned. Take another look at the class constructor, specifically line 210. This line uses a keyword argument to identify the ChangeListener event handler to each RadioButton object as the stateChange(...) method. The code for this method is shown in Listing 19-30.

Listing 19-28. RadioButtons Class from consoleTimeout_07.py, Part 1

```
205|class RadioButtons( TextField ) :
206| def __init__( self, outer ) :
207| InternalFrame.__init__(
| ...
213| )
214| self.add( JLabel( 'Timeout (minutes):' ) )
215| buttons = {}
216| self.bg = ButtonGroup()
217| for name in '0,15,30,60,Other'.split( ',' ) :
218| button = JRadioButton(
219| name,
220| itemStateChanged = self.stateChange
221| )
222| self.bg.add( button )
223| self.add( button )
```

```
buttons[ name ] = button
self.r00 = buttons[ '0' ]
self.r15 = buttons[ '15' ]
self.r30 = buttons[ '30' ]
self.r60 = buttons[ '60' ]
self.rot = buttons[ 'Other' ]
self.text = self.add(
JTextField(
",
3,
actionPerformed = outer.update
)
self.setting = 0 # see stateChange() and setValue()
self.setVisible( 1 )
def getValue( self ) :
if self.r00.isSelected() :
result = '0'
elif self.r15.isSelected() :
result = '15'
elif self.r30.isSelected() :
result = '30'
elif self.r60.isSelected() :
result = '60'
elif self.rot.isSelected() :
result = self.text.getText()
try :
int( result )
except :
messageText = badNumber % result
self.message.setText( messageText )
else :
result = None
return result
```

Listing 19-29 shows the setValue(...) method, which is also a bit more complex because it deals with multiple components to set the console timeout value.

Listing 19-29. setValue(...) Method in the RadioButtons Class from consoleTimeout_07.py

```
def setValue( self, value ) :
```

```
260| self.setting = 1
261| if value == '0' :
262| self.r00.setSelected( 1 )
263| self.r00.requestFocusInWindow()
264| self.text.setText( '' )
265| self.text.setEnabled( 0 )
266| elif value == '15' :
267| self.r15.setSelected( 1 )
268| self.r15.requestFocusInWindow()
269| self.text.setText( '' )
270| self.text.setEnabled( 0 )
271| elif value == '30' :
272| self.r30.setSelected( 1 )
273| self.r30.requestFocusInWindow()
274| self.text.setText( '' )
275| self.text.setEnabled( 0 )
276| elif value == '60' :
277| self.r60.setSelected( 1 )
278| self.r60.requestFocusInWindow()
279| self.text.setText( '' )
280| self.text.setEnabled( 0 ) 281| else :
282| self.rot.setSelected( 1 ) 283| self.text.setText( value ) 284|
self.text.setEnabled( 1 ) 285| self.text.requestFocusInWindow() 286| self.value =
value
287| self.setting = 0
```

You may note that this setValue(...) method doesn't bother to include code to check for an invalid value in the text field. How can you get away with this? Isn't this an oversight? Not really, because the update(...) method already includes that kind of code, so you're covered.

Listing 19-30 shows the rest of the methods from this class. Note how easily you can enable and disable all of the RadioButton components using a simple loop that iterates over the list of RadioButton components.
Note also line 312, where the RadioButtons inner frame is instantiated and added to the desktop frame.

Listing 19-30. Utility Methods for the RadioButtons Class from consoleTimeout_07.py

```
def working( self ) :
for obj in [
self.r00, self.r15, self.r30,
self.r60, self.rot, self.text
] :
obj.setEnabled( 0 )
def finished( self ) :
for obj in [
self.r00, self.r15, self.r30,
self.r60, self.rot
] :
obj.setEnabled( 1 )
self.text.setEnabled( self.rot.isSelected() )
def stateChange( self, event ) :
item = event.getItem()
if not self.setting :
if item.getText() == 'Other' :
self.text.setEnabled( item.isSelected() )
else :
self.text.setEnabled( 0 )
self.text.setText( '' )
value = self.getValue()
if value :
self.outer.update( event )
class consoleTimeout_07( java.lang.Runnable ) :
def run( self ) :
...
desktop.add( RadioButtons( self ) )
```

I think that the functionality added by just over 100 lines of code in this class is pretty impressive. I hope that you agree. The most important point about this is that defining the way in which the classes interact and use the framework makes each additional class significantly easier to implement.

What does this look like as far as the application output is concerned? Well, take a look at Figure 19-12, which should help you understand the component configuration.

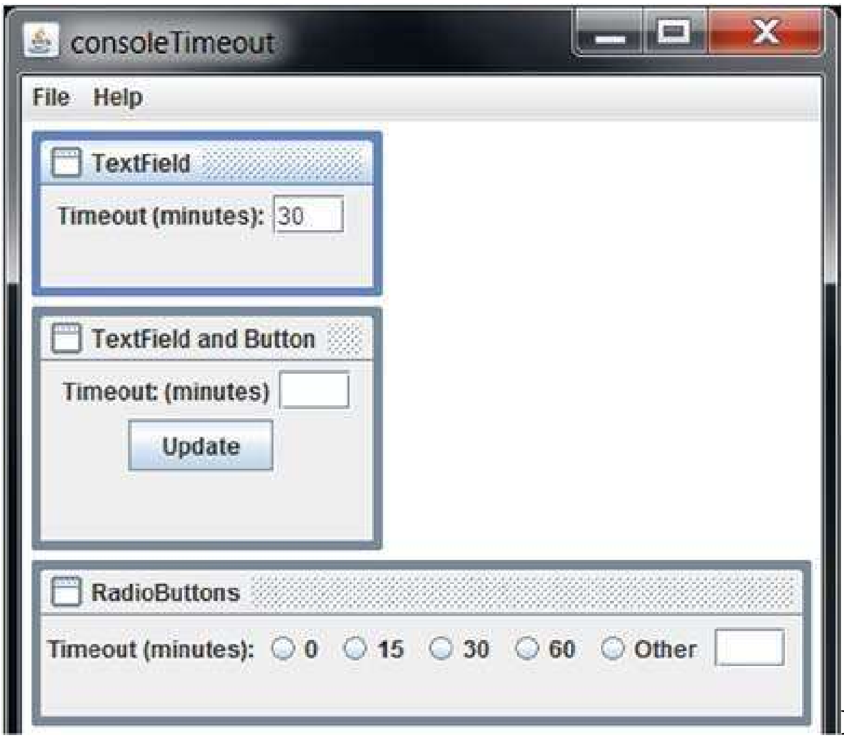
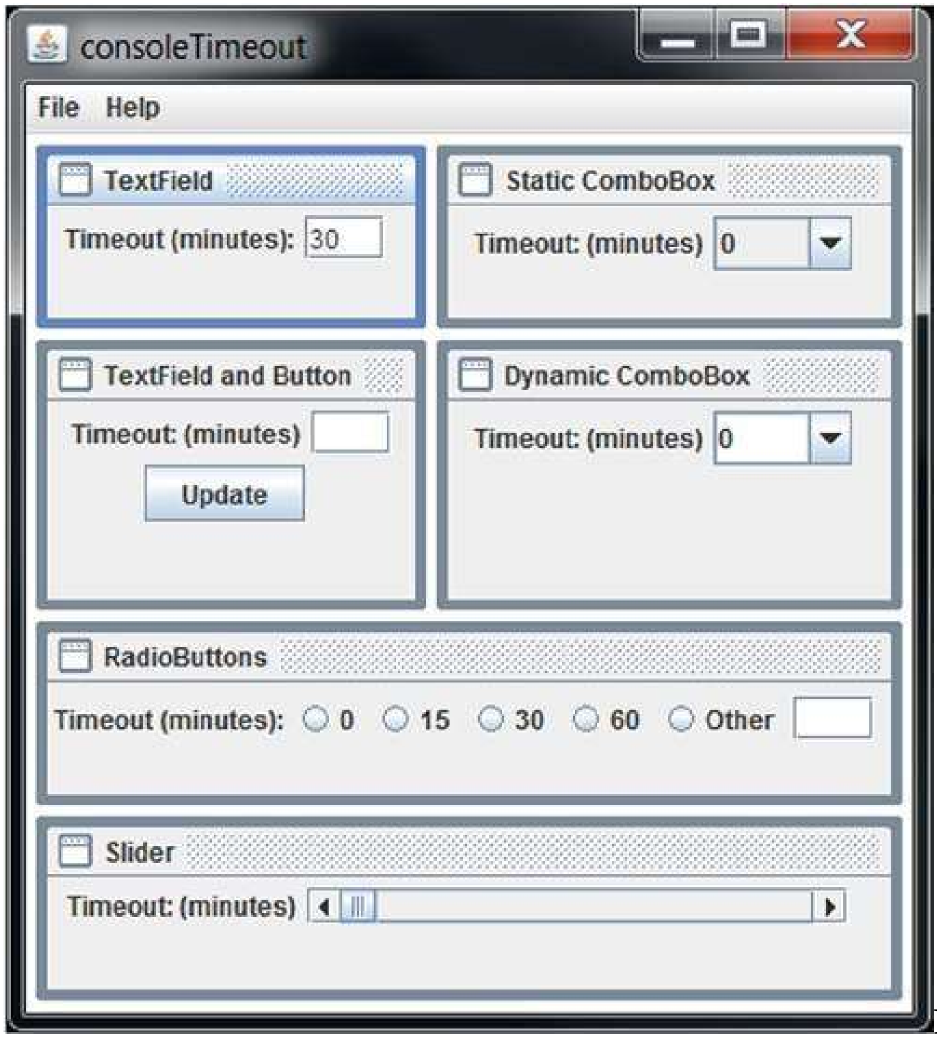


Figure 19-12. *Output from consoleTimeout_07.py*

Filling in the Blanks

Rather than continue showing every simple InternalFrame class definition, I'll cut to the chase[12] and simply show the end result of adding a few more internal frame descendant classes. The complete source is found in the consoleTimeout_08.py script file. Figure 19-13 shows the initial appearance of the final result of the application.

[12]See http://en.wikipedia.org/wiki/Cut_to_the_chase.



Timeout (minutes): 30

Timeout: (minutes) 0

Timeout: (minutes)

Timeout: (minutes) 0

Timeout (minutes): ○ 0 ○ 15 ○ 30 ○ 60 ○ Other

Timeout: (minutes)

Figure 19-13. *Output from consoleTimeout_08.py*

Remember that this is merely a sample use of internal frames. Your applications are much more likely to have a variety of internal frame types, depending on your requirement. This sample is simply provided to demonstrate some of the ways in which an application that uses internal frames can deal with issues.

Summary

Internal frames can make your applications dynamic and interesting. There is a tradeoff in that events may contribute to increased complexity, especially when the frames need to share data in some way. This should have been quite obvious in the chapter example, even though only one data value needed to be shared between the inner frames. You get to decide if the increased complexity is worth the investment of time and effort. In the next chapter, you'll use the same iterative approach to build an application that interacts with the WebSphere scripting object to create an application that can be used to more easily display the help text for the objects and their methods.

Chapter 20

Building a Graphical Help Application

In Chapter 15, you learned how to retrieve information from a web page and use Swing objects to find and display only the details in which you were interested. Now you're going to use the same kind of techniques to build an application that allows you to more easily find and manipulate the Help text that is available for the wsadmin scripting objects.

In Chapter 7 of the book titled *WebSphere Application Server Administration Using Jython*,[1,2] you learned about the wsadmin scripting objects. It describes how to use these objects and explained how to use their help(...) methods to obtain information about the methods that each object provides. Unfortunately, the interactive sessions and sample scripts provided in that book aren't as useful as they could be if an interactive graphical application were available. In this chapter, you'll remedy that situation.

Showing the Help Text

To begin, consider what you need to display the output returned by a call to the Help.help() method. You'll need to keep these issues in mind:

• You should display the text in a scrollable pane
• If you want the text displayed correctly, don't use a proportional font

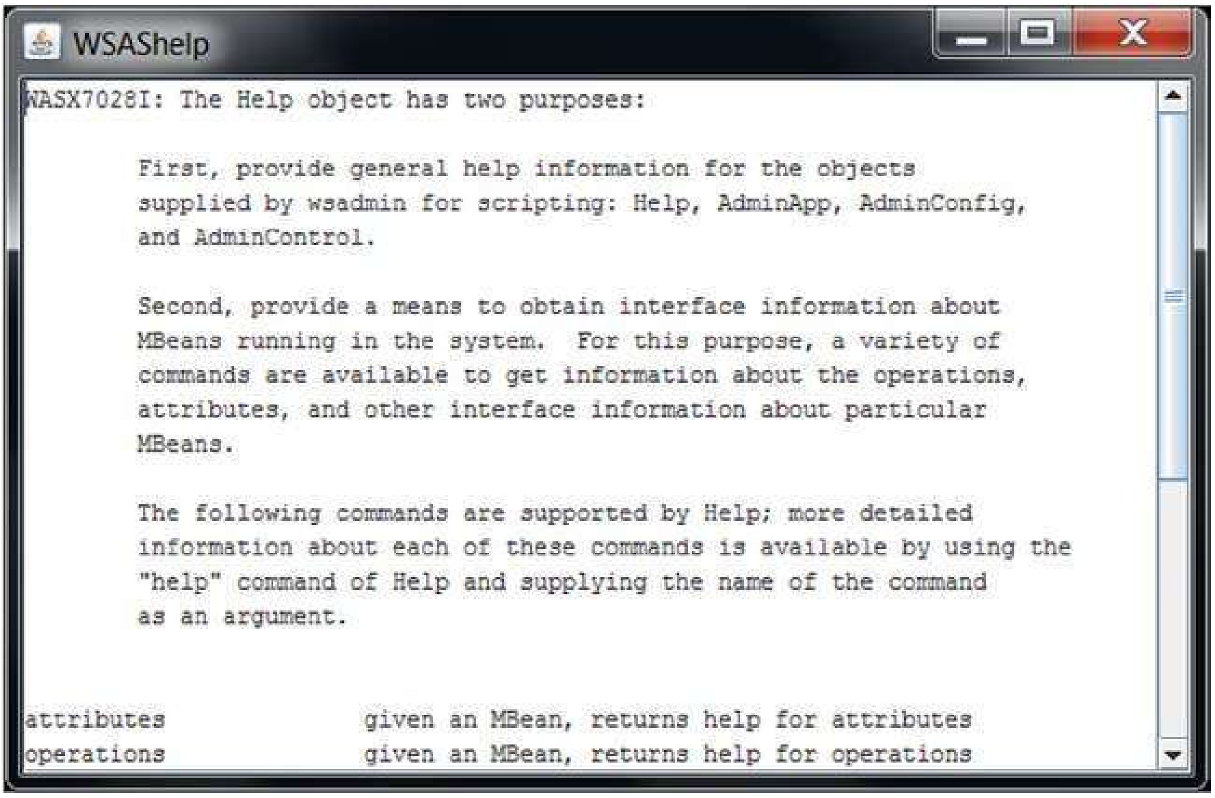
Only a tiny amount of effort is required to produce the first iteration, the main class of which is shown in Listing 20-1. One of the things that I hadn't remembered was that the Help text contains a number of tab characters (see '\t') to align the text. To simplify things, the text of the call to the help() method is immediately passed to the string's expandtabs() method, which is provided by Jython (see line 19).

Listing 20-1. WSAShelp_01.py Help.help() in a Scrollable Pane

```
9|class WSAShelp( java.lang.Runnable ) :
10| def run( self ) :
11| frame = JFrame(
12| 'WSAShelp',
13| locationRelativeTo = None,
14| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 15| )
16| frame.add(
17| JScrollPane(
18| JTextArea(
19| Help.help().expandtabs(),
20| 20,
21| 80,
22| font = Font( 'Courier' , Font.PLAIN, 12 ) 23| )
24| )
25| )
26| frame.pack()
27| size = frame.getSize()
28| loc = frame.getLocation()
29| loc.x -= ( size.width >> 1 )
30| loc.y -= ( size.height >> 1 )
31| frame.setLocation( loc )
32| frame.setVisible( 1 )
```

[1]See http://www.ibmpressbooks.com/store/websphere-application-server-administration-using-jython-9780137009527. [2]Hereafter referred to as “the WAuJ book” for simplicity’s sake.

The other statements that may cause you some pause are found in lines 27-31. When the frame in constructed (lines 11-15), the locationRelativeTo keyword argument places the upper-left corner of the frame in the center of the screen. Lines 27-31 adjust the frame to the left by half the frame width (line 29) and up by half the frame height (line 30) to center the frame in the screen. Figure 20-1 shows the simple result of the WSAShelp_01.py script.



WSAShelp

```
WASX7028I: The Help object has two purposes:

        First, provide general help information for the objects
        supplied by wsadmin for scripting: Help, AdminApp, AdminConfig,
        and AdminControl.

        Second, provide a means to obtain interface information about
        MBeans running in the system.  For this purpose, a variety of
        commands are available to get information about the operations,
        attributes, and other interface information about particular
        MBeans.

        The following commands are supported by Help; more detailed
        information about each of these commands is available by using the
        "help" command of Help and supplying the name of the command
        as an argument.


attributes              given an MBean, returns help for attributes
operations              given an MBean, returns help for operations
```

Figure 20-1. *WSAShelp_01.py sample output*

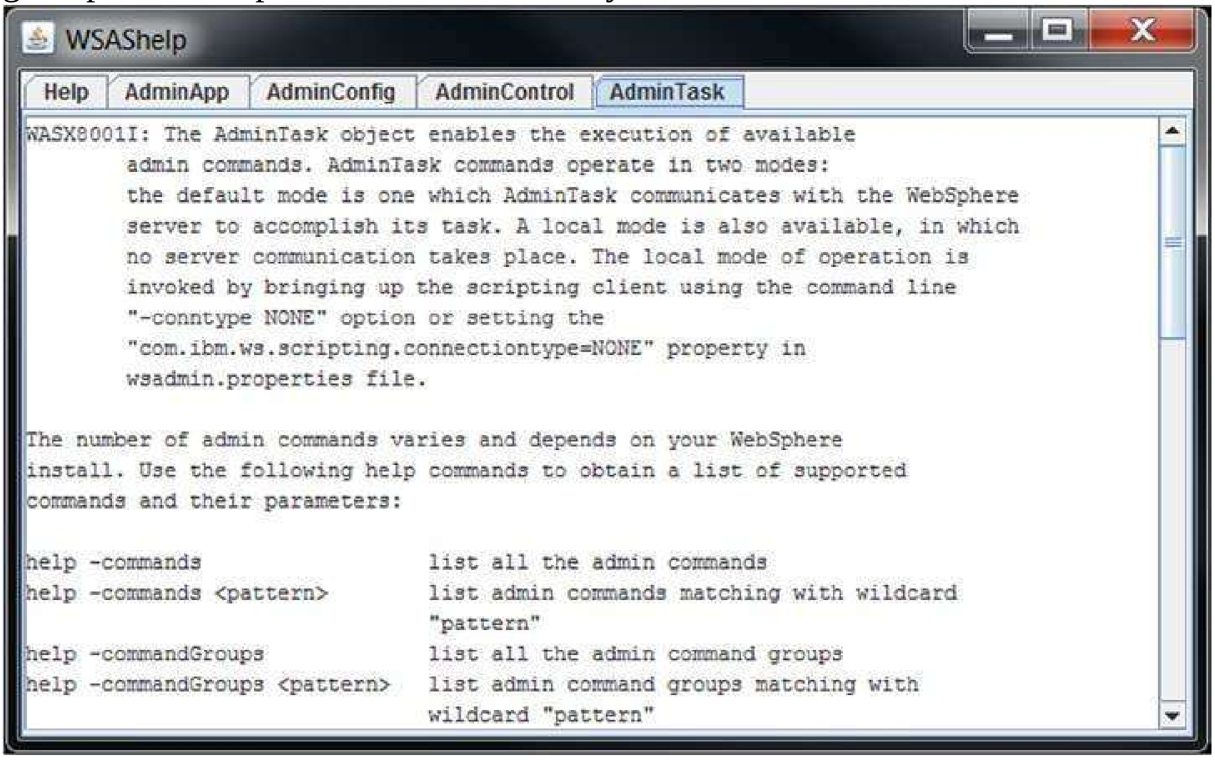
Using a Tabbed Pane

This next iteration uses tabbed panes to display the help text of the five wsadmin scripting objects (Help, AdminApp, AdminConfig, AdminControl, and AdminTask). Listing 20-2 demonstrates how a simple list (lines 28-34) can be used to easily identify the name of the tab and the associated scripting object about which help should be displayed.

Listing 20-2. Modifications to the WSAShelp Class in WSAShelp_02.py

```
11|class WSAShelp( java.lang.Runnable ) :
| ...
28| objs = [
29| ( 'Help' , Help ),
30| ( 'AdminApp' , AdminApp ),
31| ( 'AdminConfig' , AdminConfig ),
32| ( 'AdminControl', AdminControl ),
33| ( 'AdminTask' , AdminTask )
]
tabs = JTabbedPane()
for name, obj in objs :
tabs.addTab(
name,
JScrollPane(
JTextArea(
obj.help().expandtabs(),
20,
90,
font = Font( 'Courier' , Font.PLAIN, 12 )
)
)
)
frame.add( tabs )
...
```

Figure 20-2 shows sample output of the tabs produced by this script. Using this script helps you not only verify the way that it looks and responds, but it's also a good place to stop and think about what you can and should do next.



```
WASX8001I: The AdminTask object enables the execution of available
        admin commands. AdminTask commands operate in two modes:
        the default mode is one which AdminTask communicates with the WebSphere
        server to accomplish its task. A local mode is also available, in which
        no server communication takes place. The local mode of operation is
        invoked by bringing up the scripting client using the command line
        "-conntype NONE" option or setting the
        "com.ibm.ws.scripting.connectiontype=NONE" property in
        wsadmin.properties file.

The number of admin commands varies and depends on your WebSphere
install. Use the following help commands to obtain a list of supported
commands and their parameters:

help -commands                  list all the admin commands
help -commands <pattern>        list admin commands matching with wildcard
                                "pattern"
help -commandGroups             list all the admin command groups
help -commandGroups <pattern>   list admin command groups matching with
                                wildcard "pattern"
```

Figure 20-2. *WSAShelp_02.py sample output*

What can you learn from this script? The first thing is that the text area should be read-only, which is a trivial change. The next is that I wondered how difficult it would be to replace the textArea with a split pane. The top portion could display the general help for the scripting object and the bottom portion could scroll the method help. You’ll see what that looks like before you have to decide if you want to continue down that path or use a different approach.

Adding Split Panes

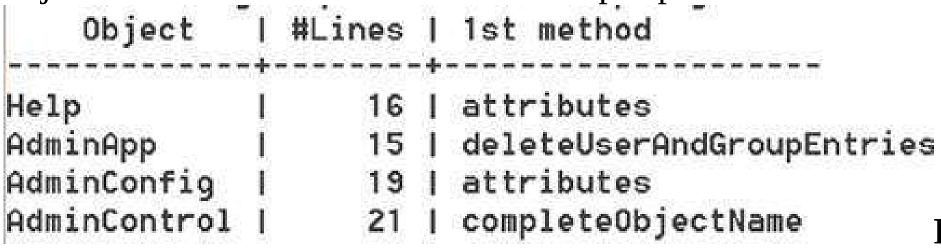
In order to split the scripting object help text, you need to determine where the general description ends and the list of methods begins. To do that, you will use a regular expression (RegExp). This topic is discussed in the WAuJ book in Chapter 7. For all of the scripting objects (except the AdminTask object), you can use a relatively simple RegExp to determine where the method names start in the help text. The AdminTask help is a different beast entirely, so you have to deal with that in a different way. Let’s start by dealing with the four scripting objects for which a relatively simple regular expression pattern works.

Listing 20-3 shows a simple script that demonstrates using a RegExp to determine the length of the description section for the majority of scripting objects.

Listing 20-3. The desc.py Script

```
import re
pat = re.compile( r'^(\w+)(?:\s+.*)$', re.MULTILINE )
objs = [
 ( 'Help' , Help ),
 ( 'AdminApp' , AdminApp ),
 ( 'AdminConfig' , AdminConfig ),
 ( 'AdminControl', AdminControl )
]
print ' Object | #Lines | 1st method'
print '-------------+--------+-------------------'
for name, obj in objs :
 text = obj.help()
 mo = re.search( pat, text )
 desc = text[ :mo.start( 1 ) ].strip().splitlines()
 method = text[ mo.start( 1 ) : mo.end( 1 ) ]
 print '%-12s | %6d | %s' % ( name, len( desc ), method )
```

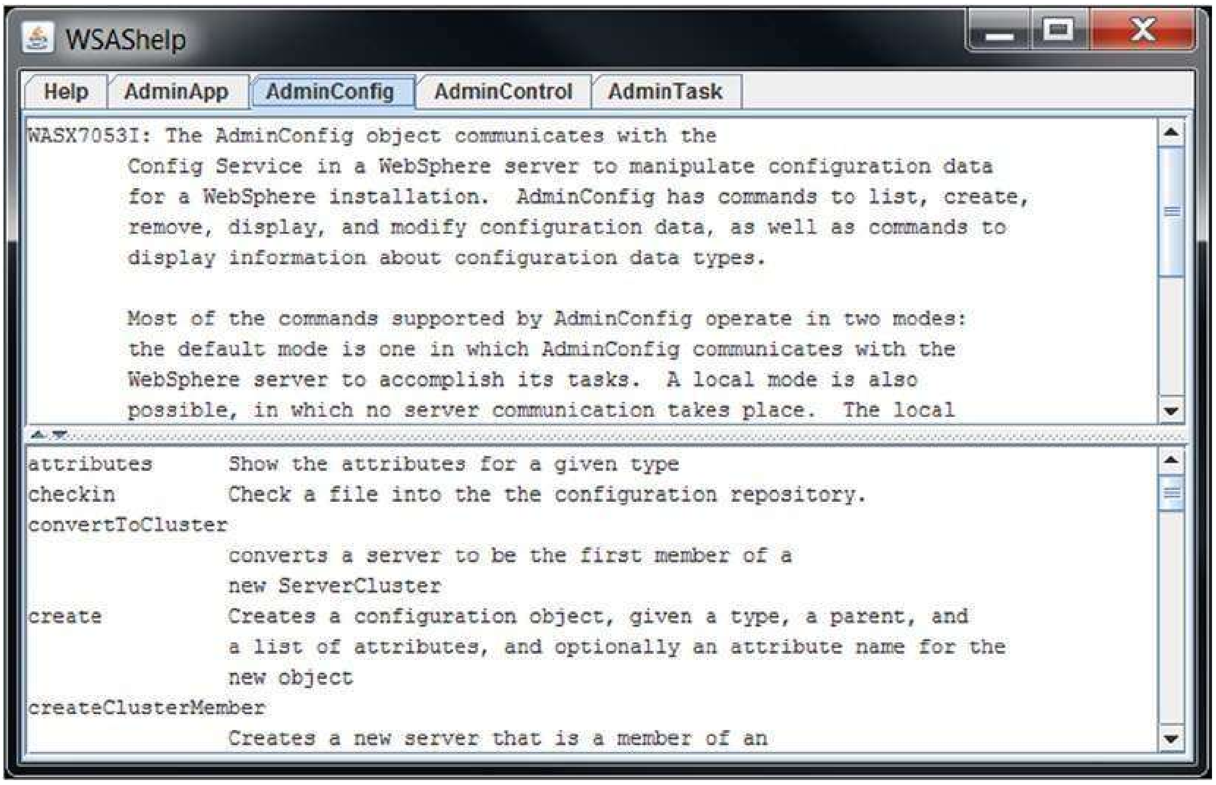
Figure 20-3 shows the output of the desc.py script in Listing 20-3. The WSAShelp_03.py script incorporates this technique to process these scripting objects so that their tabs will contain a split pane.



```
    Object    | #Lines | 1st method
-------------+--------+--------------------
Help         |     16 | attributes
AdminApp     |     15 | deleteUserAndGroupEntries
AdminConfig  |     19 | attributes
AdminControl |     21 | completeObjectName
```

Figure 20-3. *Output generated by desc.py*

Figure 20-4 shows the next version of this WSAShelp script, which uses this technique. In this version of the script, you can see that all of the tabs, except for the AdminTask tab, show a vertical split pane. The scripting object description is above the divider, and the information about the scripting object methods is below. The divider even has one touch expand icons (the little triangles) that can be used to expand or collapse the corresponding top or bottom pane, all with a single click. This is starting to look pretty neat. The complete WSAShelp_03.py script can be found in the code\Chap_20 directory.



```
WASX7053I: The AdminConfig object communicates with the
        Config Service in a WebSphere server to manipulate configuration data
        for a WebSphere installation.  AdminConfig has commands to list, create,
        remove, display, and modify configuration data, as well as commands to
        display information about configuration data types.

        Most of the commands supported by AdminConfig operate in two modes:
        the default mode is one in which AdminConfig communicates with the
        WebSphere server to accomplish its tasks.  A local mode is also
        possible, in which no server communication takes place.  The local
```

```
attributes      Show the attributes for a given type
checkin         Check a file into the the configuration repository.
convertToCluster
                converts a server to be the first member of a
                new ServerCluster
create          Creates a configuration object, given a type, a parent, and
                a list of attributes, and optionally an attribute name for the
                new object
createClusterMember
                Creates a new server that is a member of an
```

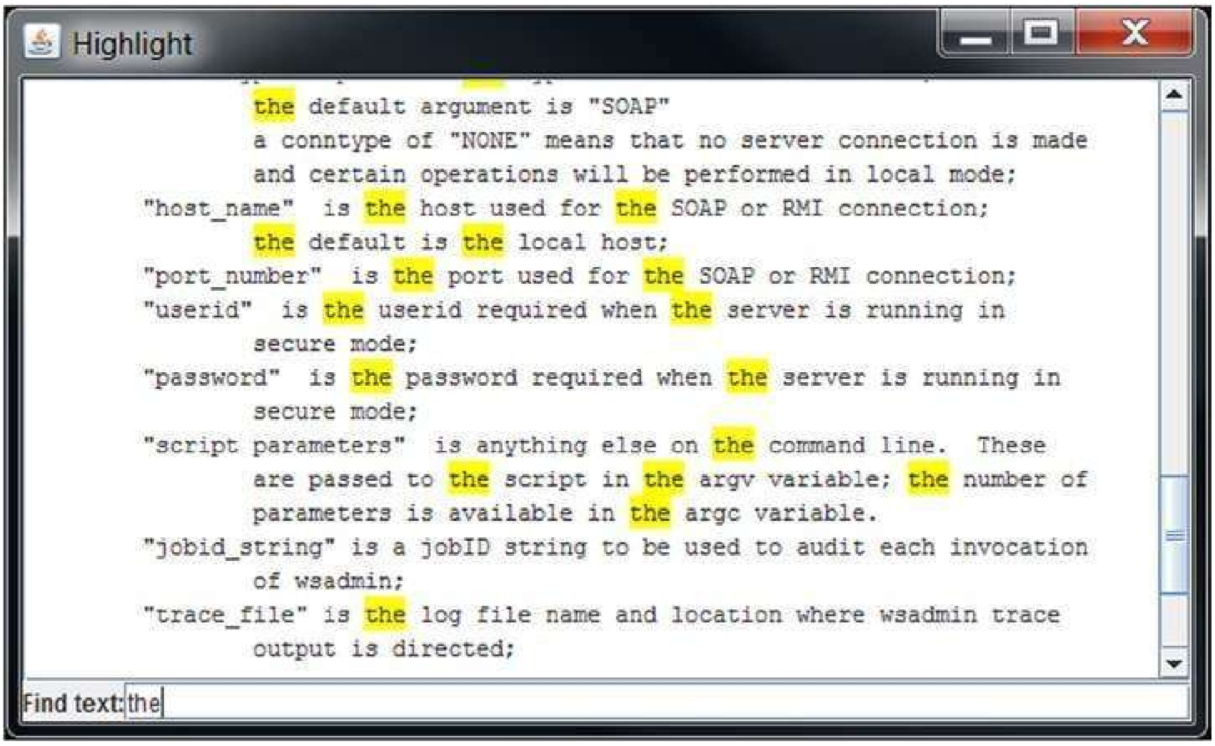
Figure 20-4. *Image from WSAShelp_03.py*

Text Highlighting

Looking at the output of WSAShelp_03.py made me wonder how difficult it would be to add text highlighting to the application. You could use this to locate specific text on a page. Before you change the WSAShelp script, let’s see what is required to do this.

First, you need to realize that the JTextArea class doesn’t provide a way to do this kind of thing. It, along with JTextField and JPasswordField, are subclasses of the abstract JTextComponent class,[3] which allows only a single kind of attribute for the entire component. This means that all of the data in the component must have the same font and color.

What you want to do is use a JTextComponent that allows multiple attributes for the contents. This means that you need to use JEditorPane or JTextPane[4] to hold and display the application’s text.
To decide if you want to add this capability to your WSAShelp script, start simply by creating a simple script that allows you to improve your understanding of what might be required to use this capability well.
Figure 20-5 shows the output of Highlight.py; its complete source is found in the code\Chap_20 directory. The image shows the text of the Help.wsadmin() method, displayed in a read-only JTextPane that is within a ScrollPane. Beneath the text pane is a label and an input field that can be used to enter the highlighted text. The scroll pane above the input field shows an image after the word “the” has been entered,[5] and the scroll pane was scrolled down to display a number of highlighted occurrences of the word.



```
          the default argument is "SOAP"
          a conntype of "NONE" means that no server connection is made
          and certain operations will be performed in local mode;
"host_name"  is the host used for the SOAP or RMI connection;
          the default is the local host;
"port_number"  is the port used for the SOAP or RMI connection;
"userid"  is the userid required when the server is running in
          secure mode;
"password"  is the password required when the server is running in
          secure mode;
"script parameters"  is anything else on the command line.  These
          are passed to the script in the argv variable; the number of
          parameters is available in the argc variable.
"jobid_string" is a jobID string to be used to audit each invocation
          of wsadmin;
"trace_file" is the log file name and location where wsadmin trace
          output is directed;
```

Find text: the

Figure 20-5. *Output of a simple script showing text highlighting*

Listing 20-4 shows part of the Highlight class from the Highlight.py script used to create the images shown in Figure 20-5. The first part of the Highlight class (lines 16-26) contains the center(...) method, which is called in line 51, to position the frame in the center of the screen.

[3] See http://docs.oracle.com/javase/8/docs/api/javax/swing/text/JTextComponent.html.
[4]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JTextPane.html.
[5]Remember that the JTextField ActionListener is called when the user presses Enter.

Lines 32-34 instantiate an instance of the static DefaultHighlightPainter class.[6] This allows the application to specify the color to be used to highlight portions of the JTextPane.
Lines 35-40 instantiate the JTextPane, make it read-only, define the preferred size, initialize the text, set the font, and provides the default highlighter to be used.[7]
Line 41 is needed so that when the application becomes visible, the start of the text pane area is seen. If this statement is commented out or removed, the end of

the text will be shown instead. Line 42 is where the text pane is added to the application frame within a scroll pane.
Lines 43-49 should look familiar to you. This is where the label and text input field are defined and added to a panel, which is then added to the bottom of the application frame. Of particular note is the keyword assignment on line 46, which identifies the ActionListener event handler that is invoked when the user presses Enter while the keyboard focus is on this component.

Listing 20-4. Portions of the Highlight.py Script

```
15|class Highlight( java.lang.Runnable ) :
| ...
27| def run( self ) :
28| frame = JFrame(
29| 'Highlight',
30| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
31| )
32| self.painter = DefaultHighlighter.DefaultHighlightPainter(
33| Color.YELLOW
34| )
35| pane = self.tPane = JTextPane(
| ...
40| )
41| pane.moveCaretPosition( 0 )
42| frame.add( JScrollPane( pane ), 'Center' )
43| info = JPanel( BorderLayout() )
44| info.add( JLabel( 'Find text:' ), 'West' )
45| tf = JTextField(
46| actionPerformed = self.search
47| )
48| info.add( tf, 'Center' )
49| frame.add( info, 'South' )
50| frame.pack()
51| self.center( frame )
52| frame.setVisible( 1 )
53| tf.requestFocusInWindow()
```

Listing 20-5 shows the search(...) method from the Highlight.py script. This is where the actual highlighting works occurs. It begins by retrieving the contents of the input (JTextField) and removing any existing highlights. The advantage of doing it this way is that if the user enters an empty string, pressing the Enter key will remove any existing highlights.

[6]See http://docs.oracle.com/javase/8/docs/api/javax/swing/text/DefaultHighlighter.Def
[7]See http://docs.oracle.com/javase/8/docs/api/javax/swing/text/DefaultHighlighter.htm

Lines 54-70 are where the searching and adding of highlights to the text pane occurs. The string find(...) method, as shown in lines 62 and 70, return the offset or position in the text string of the data for being located. If no match is found, an offset of -1 is returned. So, if a non-negative offset is returned, it identifies the location of the first character of the data string in the text string being searched. The actual work of creating the highlighter occurs in lines 64-68 and identifies the painter to be used to highlight the text in which the user is interested.

Listing 20-5. The search(...) Method from the Highlight.py Script

```
54| def search( self, event ) :
55| data = event.getSource().getText()
56| hiliter = self.tPane.getHighlighter()
57| hiliter.removeAllHighlights()
58| if data :
59| doc = self.tPane.getDocument()
60| text = doc.getText( 0, doc.getLength() )
61| start = 0
62| here = text.find( data, start )
63| while here > -1 :
64| hiliter.addHighlight(
65| here,
66| here + len( data ),
67| self.painter
68| )
69| start = here + len( data )
70| here = text.find( data, start )
```

Adding Text Highlighting to WSAShelp Application
Overall, that appears to be pretty simple and straightforward, right? Is there

anything that you need to consider or be concerned about in order to add this capability to the existing WSAShelp application?

One issue that comes to mind is how the user specifies the text to be highlighted. Do you want to add an input field below the tabbed pane or consider something else? Might you want to add some menu options that allow you to highlight the text?

How should the highlighting work? As you can see in the Highlight.py script, the DefaultHighlighter class instance is associated with the abstract JTextComponent class using the setHighlighter(...) method or using the Jython keyword assignment syntax, as shown in line 45 of Listing 20-4. It isn't clear whether one of these class instances can, or should be, shared among JTextPane instances. This isn't good practice, so I, for one, will avoid doing this. What does this mean for your application? Well, for each of the text pane instances that you want to highlight, you need a highlighter instance.

How should the application deal with highlighted text when the user changes the selected tab? That's going to make things interesting, isn't it?
You can start by adding a ChangeListener to the TabbedPane. What does that take? As is frequently the case, it's much easier than I originally thought it would be. Listing 20-6 shows the trivial changes that you need to make to the WSAShelp_03.py script in order to determine and display the name of the selected tab. The WSAShelp_04.py script includes these minor modifications.
Listing 20-6. Changes Need to Identify Tab Selection from WSAShelp_04.py

```
...
tabs = JTabbedPane( stateChanged = self.tabPicked )
...
def tabPicked( self, event ) :
pane = event.getSource()
print pane.getTitleAt( pane.getSelectedIndex() )
```

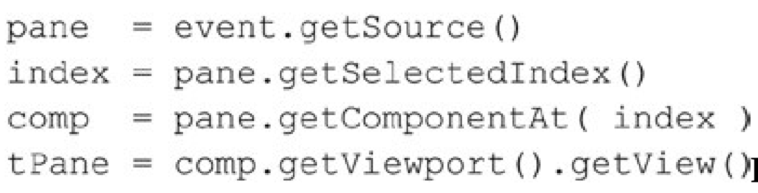
With this ChangeListener method, you can easily access the name of the selected tab. If you add a dictionary to the application class that is indexed by tab name, with each entry being a tab-specific highlighter instance, the change listener method can use this information to highlight the tab-specific text. Let's hold off on this decision for a moment while you learn about some complications.

Tabbed Highlighting Complications

What is the application supposed to do when the user selects a tab that contains a split pane? If you want the text in each pane to be highlighted, that is going to require a separate highlighter for each panel, isn't it? Well, as you saw with the Highlight.py application, you can use the DefaultHighlighter for the text pane. Can't you use this same technique for each of the panes on each of the tabs in the application? If you do, the change listener event handler can process each of the panes on the tab to reset and populate the highlights to be displayed when the tab is selected. This would probably work best if you had a single, shared "find" input field.

During the development and testing of this iteration, it became apparent how quickly things can get complicated. What do I mean? Well, consider this for a few moments. In the change listener, you can easily determine which tab was selected. You can also tell what the tab contains. Unfortunately, it will either be a JScrollPane or a JSplitPane. The JSplitPane will have two parts, each of which will contain a JScrollPane. So far, so good. Now comes some of the "fun" (challenging) part. Given a JScrollPane, you can access the actual JTextPane component by calling the getViewport().getView() methods from the JScrollPane component. I'm sure you can see the circuitous kind of path that needs to be followed to access the JTextPane in which you're interested.

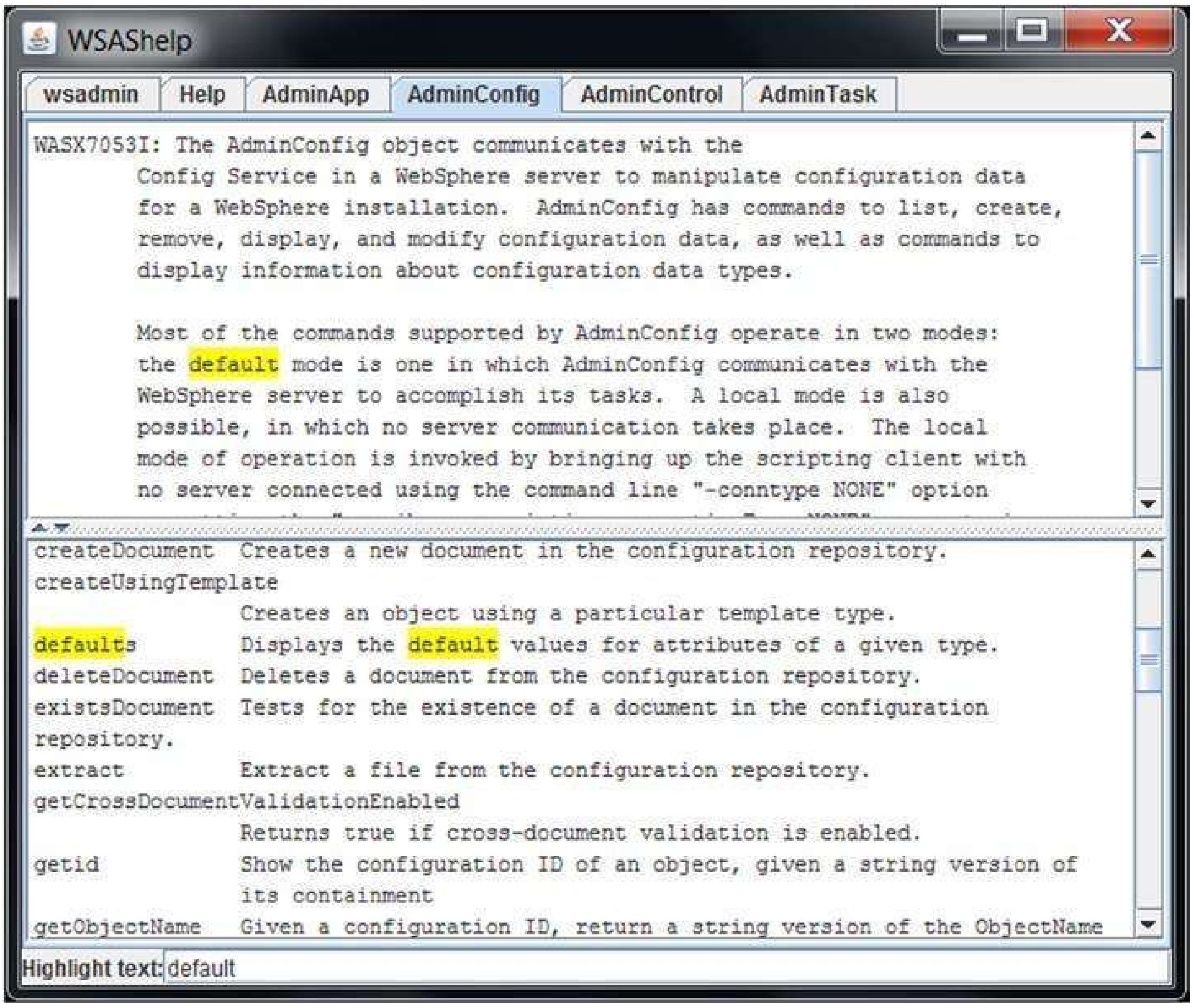
For a change listener method called when the tab contains a JScrollPane and not a JSplitPane, the methods shown in Figure 20-6 need to be called just to get access to the JTextPane contained within the JScrollPane, on the specified tab. To simplify the code as you'll as improve its performance, you'll create a dictionary indexed by the tab name, with references to each of the JTextPanes contained on the specified tab.



```
pane  = event.getSource()
index = pane.getSelectedIndex()
comp  = pane.getComponentAt( index )
tPane = comp.getViewport().getView()
```

Figure 20-6. *Calls needed to find a JTextPane in a JScrollPane*

Based on this observation, changes you're made to produce the WSAShelp_05.py script. Figure 20-7 shows an image from this application after "default" is highlighted on the AdminConfig tabs.



```
WASX7053I: The AdminConfig object communicates with the
        Config Service in a WebSphere server to manipulate configuration data
        for a WebSphere installation.  AdminConfig has commands to list, create,
        remove, display, and modify configuration data, as well as commands to
        display information about configuration data types.

        Most of the commands supported by AdminConfig operate in two modes:
        the default mode is one in which AdminConfig communicates with the
        WebSphere server to accomplish its tasks.  A local mode is also
        possible, in which no server communication takes place.  The local
        mode of operation is invoked by bringing up the scripting client with
        no server connected using the command line "-conntype NONE" option
```

```
createDocument  Creates a new document in the configuration repository.
createUsingTemplate
                Creates an object using a particular template type.
defaults        Displays the default values for attributes of a given type.
deleteDocument  Deletes a document from the configuration repository.
existsDocument  Tests for the existence of a document in the configuration
repository.
extract         Extract a file from the configuration repository.
getCrossDocumentValidationEnabled
                Returns true if cross-document validation is enabled.
getid           Show the configuration ID of an object, given a string version of
                its containment
getObjectName   Given a configuration ID, return a string version of the ObjectName
```

Highlight text: default

Figure 20-7. *WSAShelp_05.py sample output*

To create this iteration of the application required about 50 more lines of code, most of which can be seen in Listing 20-7. The highlight(...) method is called by the change listener event handler (the tabPicked(...) method), which is invoked when the user selects a new tab. Additionally, it is called by the action listener event handler (the lookFor(...) method) when the user presses Enter while the keyboard focus is on the JTextField at the bottom of the window.

Listing 20-7. New and Changed Methods in WSAShelp_05.py

```
def hilight( self, tPane, text ) :
hiliter = tPane.getHighlighter()
hiliter.removeAllHighlights()
if text :
```

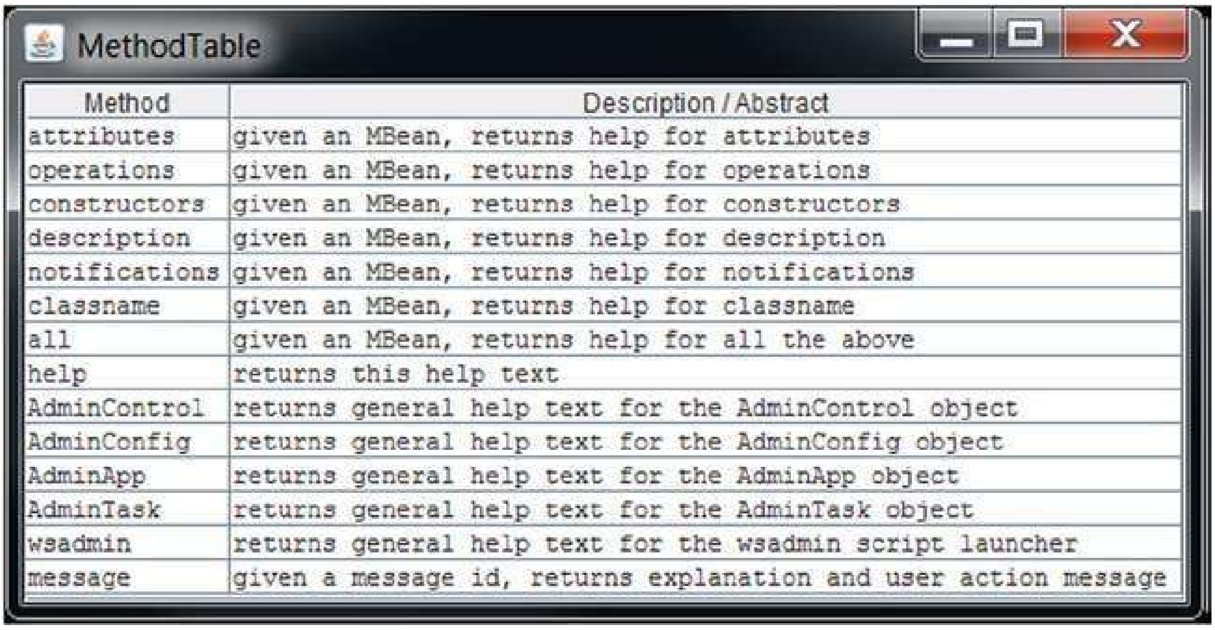
```
doc = tPane.getDocument()
info = doc.getText( 0, doc.getLength() )
start = 0
here = info.find( text, start )
while here > -1 :
hiliter.addHighlight(
here,
here + len( text ),
self.painter
)
start = here + len( text )
here = info.find( text, start )
def lookFor( self, event ) :
text = event.getSource().getText()
index = self.tabs.getSelectedIndex()
name = self.tabs.getTitleAt( index )
for tPane in self.tPanes[ name ] :
self.hilight( tPane, text )
...
def tabPicked( self, event ) :
pane = event.getSource()
index = pane.getSelectedIndex()
name = pane.getTitleAt( index )
try :
for tPane in self.tPanes[ name ] :
self.hilight( tPane, self.textField.getText() )
except :
pass
```

Note You may wonder why you need a try/except clause in the change listener method. it is there because this method is invoked when the first tab is added to the JTabbedPane container. When this occurs, the data structure (the self.tPanes dictionary) hasn’t been initialized, so the specified reference (self.tPanes[name]) is invalid. other than that, the changes are pretty straightforward and make the application more useful, at least in my opinion.

Displaying Methods in a Table

I don’t know about you, but every time that I look at the method (and scripting objects) sections (the bottom split pane sections), I wonder how difficult it would be to display this part in a table within a scroll pane. The big question related to this change is how it affects the highlighting.

Let’s approach these challenges one at a time. You can start by creating a table for the bottom part of the split panes. Then, you can take a look at the cell rendering portion to see if you can easily resolve the highlighting issue.
So how should you go about building a table containing the method name in one column and the method abstract/description in another? Let’s start by using a simple application that does this and only this. Then, you can determine what changes, if any, need to be made before incorporating this into your application. Figure 20-8 shows the method names and descriptions that you’re extracted from the output of calling the Help.help() method.



Method	Description / Abstract
attributes	given an MBean, returns help for attributes
operations	given an MBean, returns help for operations
constructors	given an MBean, returns help for constructors
description	given an MBean, returns help for description
notifications	given an MBean, returns help for notifications
classname	given an MBean, returns help for classname
all	given an MBean, returns help for all the above
help	returns this help text
AdminControl	returns general help text for the AdminControl object
AdminConfig	returns general help text for the AdminConfig object
AdminApp	returns general help text for the AdminApp object
AdminTask	returns general help text for the AdminTask object
wsadmin	returns general help text for the wsadmin script launcher
message	given a message id, returns explanation and user action message

Figure 20-8. *Proof of concept for displaying method info in a table*

Listing 20-8 shows the majority of the statements from the MethodTable1.py proof of concept (PoC) script used to produce the output shown in Figure 20-8. Since it is a PoC script, it’s simple. As you can see, the run(...) method is very simple. It uses the parseMethodHelp(...) method to extract the method names and descriptions from the specified help text in order to populate the table.

Listing 20-8. run(...) Method of MethodTable Class from MethodTable1.py Script

```
10|class MethodTable1( java.lang.Runnable ) :
```

```
...
def run( self ) :
frame = JFrame(
'MethodTable1',
defaultCloseOperation = JFrame.EXIT_ON_CLOSE
)
helpText = Help.help().expandtabs()
headings = [ 'Method', 'Description / Abstract' ]
data = self.parseMethodHelp( helpText )
table = JTable(
data,
headings,
font = Font( 'Courier' , Font.PLAIN, 12 )
)
frame.add( JScrollPane( table ), 'Center' )
frame.pack()
self.center( frame )
frame.setVisible( 1 )
```

The parseMethodHelp(...) method, shown in Listing 20-9, is where all of the real work takes place. It uses a regular expression (RegExp) to locate each method name in the specified input string.
Listing 20-9. parseMethodHelp(...) Method of the MethodTable Class from the MethodTable1.py Script

```
def parseMethodHelp( self, helpText ) :
def fix( text ) :
text = text.replace( '\n', ' ' ).strip()
return re.sub( ' +', ' ', text )
methRE = re.compile( r'^(\w+)(?:\s+.*)$', re.MULTILINE )
result = []
mo = methRE.search( helpText )
name = None
while mo :
start, finish = mo.span( 1 )
if name :
result.append(
[
name,
53| fix( helpText[ prev : start ] )
54| ]
55| )
56| name = helpText[ start : finish ]
57| prev = finish + 1
58| mo = methRE.search( helpText, finish )
59| if name :
60| result.append( [ name, fix( helpText[ prev: ] ) ] )
61| return result
```

The remainder of the text up to the next method name is considered the associated description. The result of this method is an array, with one row for each method, and two columns. The first column of the table contains the method name and the second column the entire description string. In order to do this, the fix routine, shown in lines 40-42 of Listing 20-9, replaces the newline characters ('\n') with a space. Then, all of the leading, trailing, and multiple adjacent spaces are removed.

Highlighting Text Within the Table

One of the nice things about using a JTextPane component to hold the text to be displayed is that you can use the DefaultHighlighter as you'll as the DefaultHighlightPainter class instances to locate and highlight text of interest to the user. Unfortunately, these classes aren't available on the contents of the JTable instance. So, what can you do? The default renderer for the table cells uses a JLabel instance to format the cell contents.

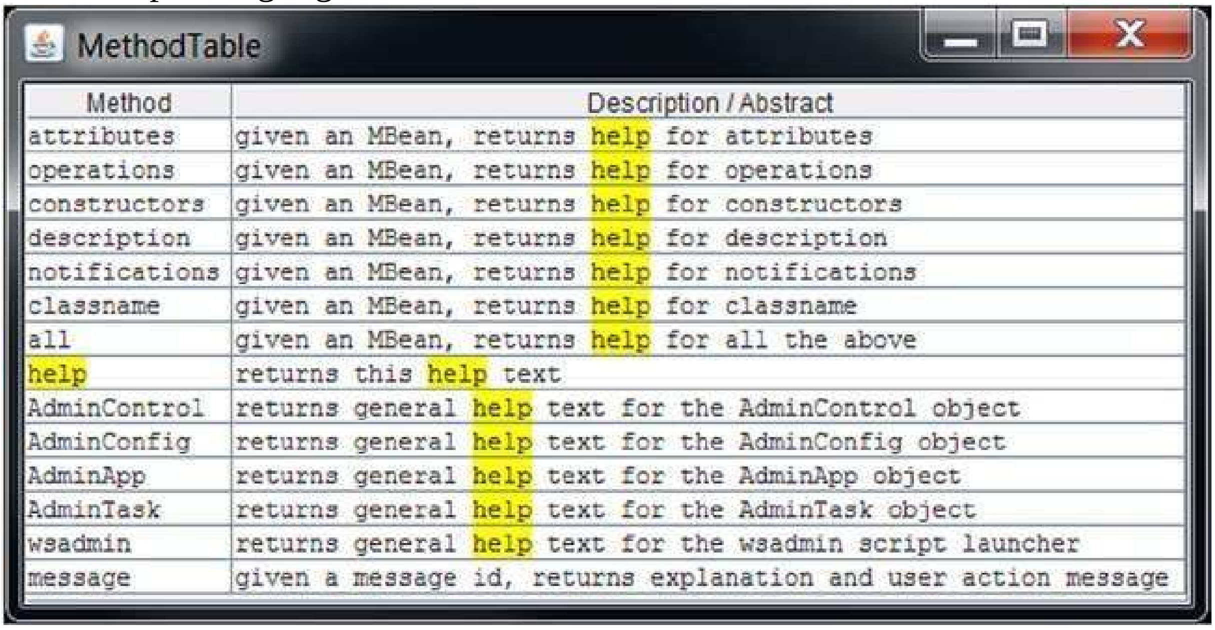
The neat thing about this is that a JLabel can use HTML tags to format the cell contents. This means that all you have to do to highlight a portion of a cell is surround it with the appropriate HTML tags. To do that, make the changes shown in Listing 20-10.

Listing 20-10. Changes to the Highlight Table Text from MethodTable2.py

```
10|class MethodTable2( java.lang.Runnable ) : | ...
22| def run( self ) :
| ...
29| data = self.parseMethodHelp( helpText )
30| for r in range( len( data ) ) :
```

```
for c in range( len( data[ r ] ) ) :
data[ r ][ c ] = self.hiliteText(
data[ r ][ c ],
'help'
)
table = JTable(
data,
headings,
font = Font( 'Courier' , Font.PLAIN, 12 )
)
| ...
def hiliteText( self, text, findWord ) :
return '<html>' + text.replace(
findWord,
'<font bgcolor=yellow>%s</font>' % findWord
)
```

Figure 20-9 shows the sample output generated by this script. By default, the word “help” is highlighted.



Method	Description / Abstract
attributes	given an MBean, returns help for attributes
operations	given an MBean, returns help for operations
constructors	given an MBean, returns help for constructors
description	given an MBean, returns help for description
notifications	given an MBean, returns help for notifications
classname	given an MBean, returns help for classname
all	given an MBean, returns help for all the above
help	returns this help text
AdminControl	returns general help text for the AdminControl object
AdminConfig	returns general help text for the AdminConfig object
AdminApp	returns general help text for the AdminApp object
AdminTask	returns general help text for the AdminTask object
wsadmin	returns general help text for the wsadmin script launcher
message	given a message id, returns explanation and user action message

Figure 20-9. *PoC for displaying highlighted table text*

Using Tables in the Help Application

Are there any problems with adding this to the help application? If you take a few moments to test the MethodTable scripts, you are likely to see a noticeable delay during the application startup. Unfortunately, this is the kind of thing that users find particularly annoying. As discussed earlier, whenever you encounter an operation that could take a long time, it should execute on a separate thread. Since the operation in question involves the creation of the attribute and description table, it should be executed on a separate thread.

These choices might cause you to ask yourself some questions. For example, what should you display on the bottom part of the split pane until the table thread completes? An easy and reasonable approach is to display a simple message indicating that results will be available shortly. So, you’ll just use a JLabel instance with a message.

You might also wonder which attributes the tables should use. You don’t want the user to attempt to make any changes to the table data, so the table cells should be read-only. Additionally, you know that the data type for each cell should be a java.lang.String. Add a table model class (methodTableModel) that descends from the DefaultTableModel class.
Listing 20-11. The methodTableModel Class from WSAShelp_06.py

```
class methodTableModel( DefaultTableModel ) :
def __init__( self, data, headings ) :
DefaultTableModel.__init__( self, data, headings )
def isCellEditable( self, row, col ) :
return 0
def getColumnClass( self, col ) :
return String
```

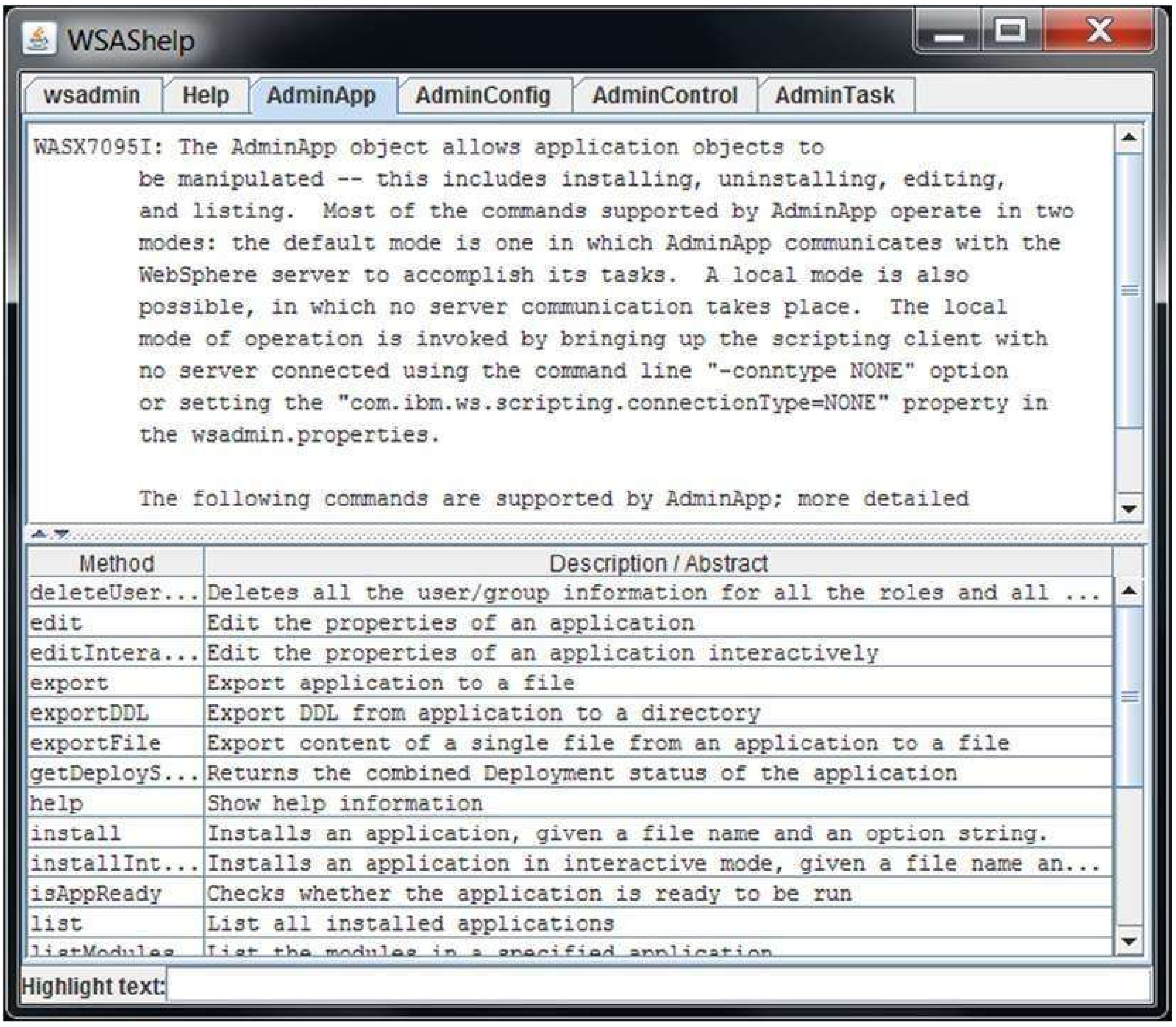
Listing 20-11 shows just how simple this kind of descendent class can be. Unfortunately, you also need to add code to adjust the table column widths. Without it, the table will give each column as close to 50% of the table width as it can. With the information that you want to display, this isn’t a great distribution of the available space. So, WSAShelp_06.py also includes the setColumnWidths(...) method shown in Listing 20-12 to try to adjust the column widths.

Listing 20-12. The setColumnWidths(...) Method from WSAShelp_06.py

```
def setColumnWidths( self, table ) :
header = table.getTableHeader()
tcm = table.getColumnModel() # Table Column Model
data = table.getModel() # To access table data
margin = tcm.getColumnMargin() # gap betyouen columns
rows = data.getRowCount() # Number of rows
```

```
cols = tcm.getColumnCount() # Number of cols
for i in range( cols ) : # For col 0..N
col = tcm.getColumn( i ) # TableColumn: col i
idx = col.getModelIndex() # model index: col i
render = col.getHeaderRenderer()# header renderer
if render :
comp = render.getTableCellRendererComponent(
table,
col.getHeaderValue(),
0,
0,
-1,
i
)
cWidth = comp.getPreferredSize().width
else :
cWidth = -1
for row in range( rows ) :
val = str( data.getValueAt( row, idx ) )
r = table.getCellRenderer( row, i )
comp = r.getTableCellRendererComponent(
table,
val, # formatted value
0, # not selected
0, # not in focus
row, # row num
i # col num
)
cWidth = max(
cWidth,
comp.getPreferredSize().width
)
if cWidth > 0 :
col.setPreferredWidth( cWidth + margin )
```

Figure 20-10 shows a sample image from the WSAShelp_06.py script. As you can see, some of the tabs can’t display complete information that is available to both columns. When this occurs, the contents have ellipses (. . .) appended to the truncated values.



```
WASX7095I: The AdminApp object allows application objects to
        be manipulated -- this includes installing, uninstalling, editing,
        and listing.  Most of the commands supported by AdminApp operate in two
        modes: the default mode is one in which AdminApp communicates with the
        WebSphere server to accomplish its tasks.  A local mode is also
        possible, in which no server communication takes place.  The local
        mode of operation is invoked by bringing up the scripting client with
        no server connected using the command line "-conntype NONE" option
        or setting the "com.ibm.ws.scripting.connectionType=NONE" property in
        the wsadmin.properties.

        The following commands are supported by AdminApp; more detailed
```

Method	Description / Abstract
deleteUser...	Deletes all the user/group information for all the roles and all ...
edit	Edit the properties of an application
editIntera...	Edit the properties of an application interactively
export	Export application to a file
exportDDL	Export DDL from application to a directory
exportFile	Export content of a single file from an application to a file
getDeployS...	Returns the combined Deployment status of the application
help	Show help information
install	Installs an application, given a file name and an option string.
installInt...	Installs an application in interactive mode, given a file name an...
isAppReady	Checks whether the application is ready to be run
list	List all installed applications
listModules	List the modules in a specified application

Figure 20-10. *Sample output from WSAShelp_06.py*

Grab and drag the separator bar between the column headings and see what happens when you change the column widths. Is that responsive enough for you? I don’t notice any delay at all and am quite pleased.

Fixing the Table Appearance

Many people would find the ellipses a distraction and an annoyance. How do you fix this? The first thing to do is to remember how the cell data is represented. In Chapter 12, you learned about cell renderers. Fortunately, all of the table cells contain simple strings. So you only need to provide a renderer for this data type.

The challenge, though, is how you deal with the cell data that doesn’t fit in the available space. First, you need to be able to figure out:

- How much space is available
- How you want to display the data
- How you split the data to fit in the available space

The program uses a monospace font, which makes sense when you’re displaying the help text that is normally seen in an interactive wsadmin session.
Another choice that makes life a little easier is to use a JTable to display the method information. How does it make things easier? If you take another look back at Chapter 12, you’ll see that the default cell renderer for a JTable will use a JLabel to display a cell. You can then use HTML to format the cell contents. This allows you to specify <br> (the HTML line break tag) where you want the break to occur. Unfortunately, if you are going to use multiple lines of text to display the cell text, you also have top deal with the row height to allow multiple lines.
An additional challenge involves how you intend to split any long method names. Blanks aren’t allowed as part of a method name, so you have to decide if you want to split the method name between arbitrary letters or use a little additional effort and recognize that method names use a camelcase[8] naming convention. Personally, I prefer to split the method names between the camelcase words. You need to use a JLabel method, specifically the getFontMetrics(...) method, to determine how much of the data will fit in the available width.
Listings 20-13 and 20-14 show the table cell renderer class from WSAShelp_07.py. Unfortunately, it doesn’t fit on one page so has to be shown in two pieces.

Listing 20-13. The methRenderer Class from WSAShelp_07.py Part 1

```
class methRenderer( DefaultTableCellRenderer ) :
def __init__( self ) :
self.fm = JLabel().getFontMetrics( monoFont )
self.widths = [ 0, 0 ]
self.hiText = "
def getTableCellRendererComponent(
self,
table, # JTable - table containing value
value, # Object - value being rendered
isSelected, # boolean - Is value selected?
hasFocus, # boolean - Does this cell have focus?
row, # int - Row # ( 0 .. N-1 )
39| col # int - Col # ( 0 .. N-1 )
40| ) :
41| def camelWords( name ) :
42| prev, result = 0, []
43| for i in range( len( name ) ) :
44| ch = name[ i ]
45| if ch == ch.upper() or ch == '_' : 46| result.append( name[ prev:i ] ) 47| prev =
i
48| result.append( name[ prev: ] )
49| return result
50| DTCR = DefaultTableCellRenderer
51| comp = DTCR.getTableCellRendererComponent( 52| self, table, value,
isSelected, hasFocus, row, col 53| )
54| pWidth = self.widths[ col ] # Preferred column width
```

[8]See http://en.wikipedia.org/wiki/CamelCase.

The first portion of the class, shown in Listing 20-13, shows the class constructor in lines 28-31 as well as the initial portion of the getTableCellRendererComponent(...) method. Note that this method includes the local camelWords(...) method on lines 41-49. It is only used in the next segment of code (on line 62) where the contents of column 0 are being processed.

It takes advantage of the similarity of splitting method names and the method abstract text. If the text doesn't fit into the available width, the data is reformatted using HTML to use multiple lines. Since the renderer is the class that uses HTML to format the data, it is also the right place to add HTML to highlight any text that the user requests.

If you look closely at the renderer class in Listing 20-14, you might notice a couple of methods that are unique to this renderer (they don't exist in the base class). These are the setHiText(...), and setWidths(...) methods on lines 92-95. These were added to provide the class instance with the information that it needs to format the text to fit in the available width, as well as add the appropriate highlighting, as needed. Is this a perfect answer? Probably not, but it is certainly good enough, especially for this application.

Listing 20-14. The methRenderer Class from WSAShelp_07.py Part 2

```
if pWidth : # Has it been set?
hiHTML = '<font bgcolor=yellow>%s</font>'
pWidth -= 3
if self.fm.stringWidth( value ) > pWidth :
if col :
pad, words = ' ', value.split( ' ' )
else :
pad, words = ", camelWords( value )
result, curr = '<html>', "
for word in words :
width = self.fm.stringWidth(
curr + pad + word
)
if width > pWidth :
result += curr + '<br>'
curr = "
if curr :
curr += pad + word
else :
curr = word
result += curr
if self.hiText :
if result.count( self.hiText ) > 0 :
result = result.replace(
self.hiText,
hiHTML % self.hiText
)
comp.setText( result )
else :
if self.hiText :
if value.count( self.hiText ) > 0 :
value = value.replace(
self.hiText,
hiHTML % self.hiText
)
comp.setText( '<html>' + value )
return comp
def setHiText( self, text ) :
self.hiText = text
def setWidths( self, width0, width1 ) :
self.widths = [ width0, width1 ]
```

How and when would the setWidths(...) method need to be called? Think about it for a few moments. When does the table need to be displayed? The first time would be when the tab on which a table exists is selected. Another time would be when the application frame was resized (maximized). If you take a look at the

WSAShelp_07.py script, and search for calls to this routine, you will find calls to this routine in these two event handlers.

Take a few moments to test this script and see what happens when you resize the frame. Personally, I like the way it responds to the resize requests; I hope that you agree.

Selecting Table Cells

Now that you have the scripting object methods in a nice little table at the bottom of each split pane, what’s next? You can now progress to one of the goals that I had for this application. I wanted to be able to select the tab for a specific scripting object, scroll down to show the name of the method in which I was interested, and select the method. Wouldn’t it be neat if selecting the method in the table showed the help for the selected method? What is needed to do this? Listing 20-15 shows the class that was added to provide this capability to the WSAShelp_08.py script.

Listing 20-15. ListSelectionListener from WSAShelp_08.py

```
class cellSelector( ListSelectionListener ) :
def __init__( self, table, WASobj ) :
self.table = table
self.WASobj = WASobj
self.objName = WASobj.help()[ :40 ].split( ' ' )[ 2 ]
def valueChanged( self, event ) :
if not event.getValueIsAdjusting() :
table = self.table
row = table.getSelectedRow()
if row > -1 :
method = table.getModel().getValueAt( row, 0 )
title = '%s.help( "%s" )' % (
self.objName,
method
)
text = self.WASobj.help( method )
text = JTextArea(
text,
20,
80,
font = monoFont
51| )
52| dialog = JDialog(
53| None, # owner 54| title, # title 55| 1, # modal = true 56| layout =
BorderLayout(),
57| locationRelativeTo = None 58| )
59| dialog.add(
60| JScrollPane(
61| text
62| ),
63| BorderLayout.CENTER
64| )
65| dialog.pack()
66| dialog.setVisible( 1 )
```

The class constructor may require a bit of explanation though. The reason for providing the table as an argument to the constructor will be explained shortly. The reason for having the WASobj (AdminApp, AdminConfig, and so on), specified makes things for the valueChanged(...) method much simpler, because given the scripting object, and the name of the method in question, all the routine needs to do is invoke the help(...) method, as see on line 45. This leaves you with the expression on line 34. What is it doing?

Listing 20-16 shows how the constructor expression works. Basically, it depends on the fact that the first line of the help text for each of these scripting objects uses the same format. The message number is followed by a sentence that starts with “The”, followed by the name of the scripting object. The expression on line 34 of Listing 20-15 simply uses this fact to save the name of the specified scripting object.

Listing 20-16. Extracting an Object Name from its Help Text

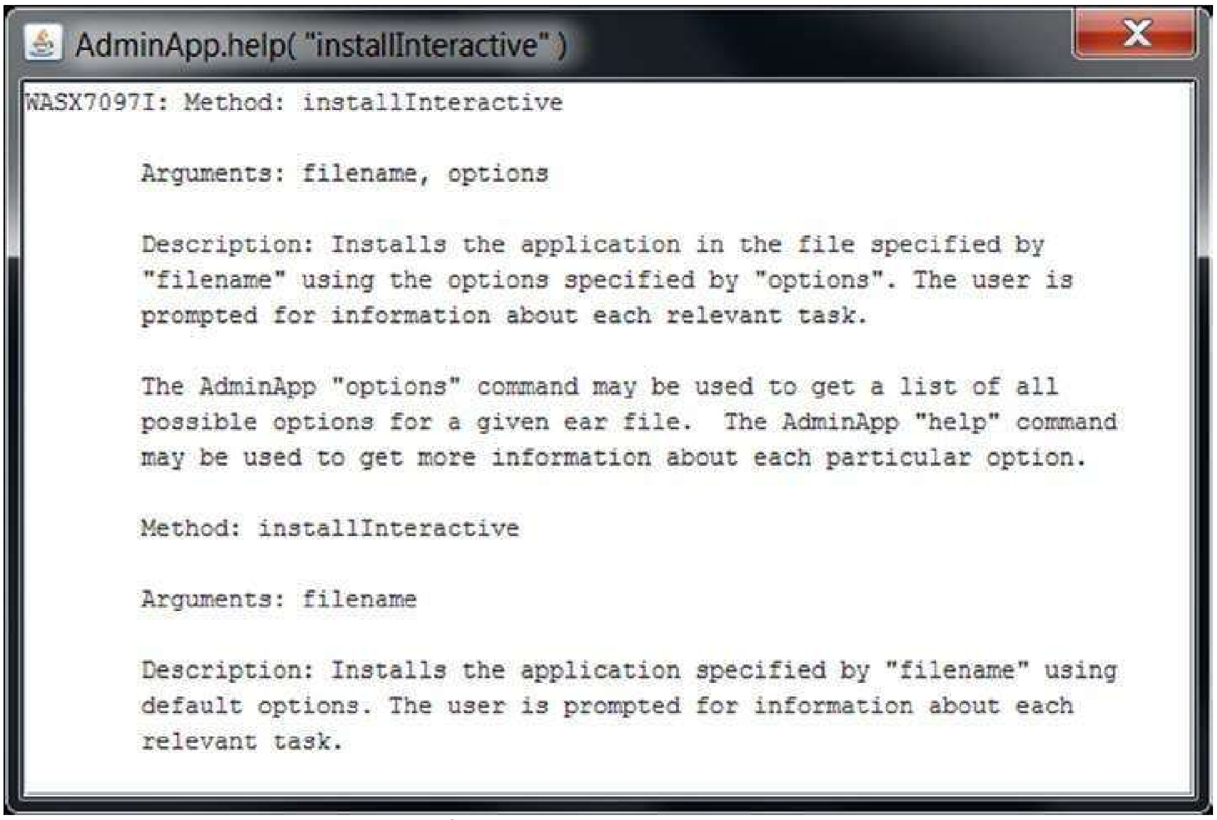
```
wsadmin>for obj in [ AdminApp, AdminConfig, AdminControl, Help ] :
wsadmin> print obj.help()[ :40 ]
wsadmin>
WASX7095I: The AdminApp object allows ap
WASX7053I: The AdminConfig object commun
WASX7027I: The AdminControl object enabl
WASX7028I: The Help object has two purpo
wsadmin>
```

Instances of the cellSelector listener class shown in Listing 20-15 are added when the table is created in the tableTask instance, specifically in the done(...) method. Listing 20-17 shows how the selection listener is added to table using its selection model. The fact that the selection listener instance is added to the selection model, and not the table itself, is the main reason that the table instance is passed on the cellSelector constructor. The code in the valueChanged(...) event handler method, as shown in Listing 20-15, uses the table to identify the user-selected method, so that the appropriate help text can be displayed.

Listing 20-17. Adding Selection Listener Instance from WSAShelp_08.py

```
table.getSelectionModel().addListSelectionListener(
cellSelector(
table,
self.WASobj
)
)
```

What do these changes allow you to do? Figure 20-11 shows a sample dialog box that is displayed when some of the scripting object table rows are selected. Note how the title of the dialog box identifies the scripting object and method for which help is being displayed.



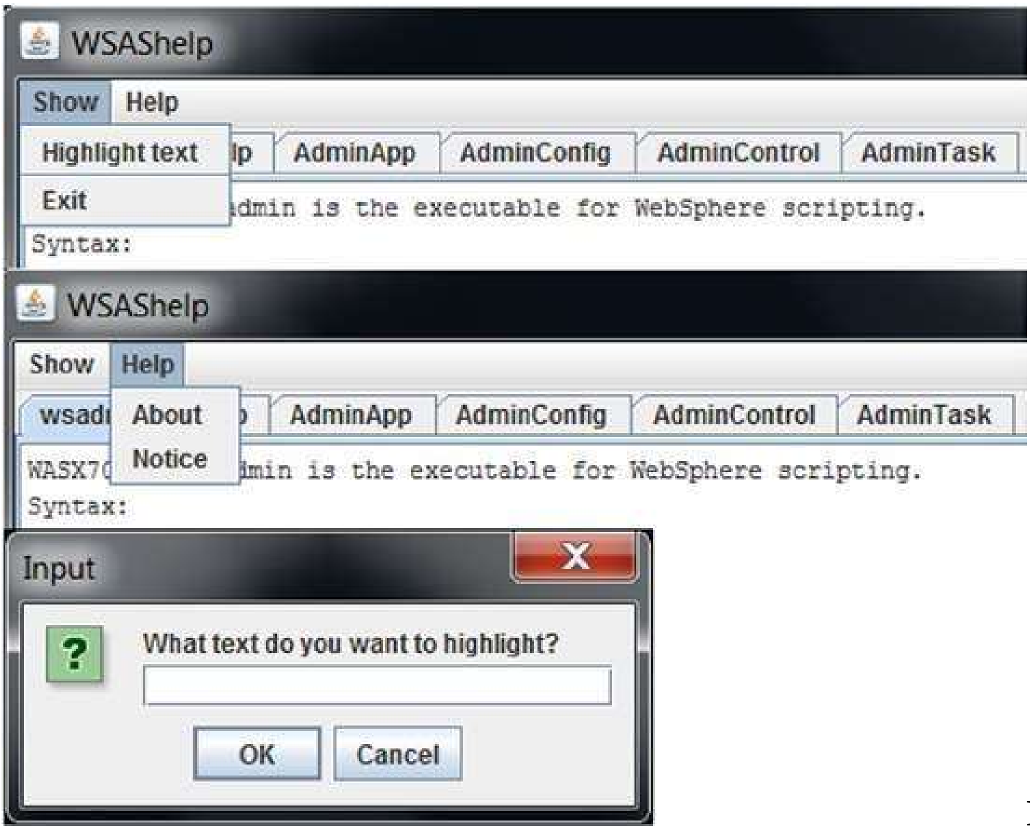
```
WASX7097I: Method: installInteractive

        Arguments: filename, options

        Description: Installs the application in the file specified by
        "filename" using the options specified by "options". The user is
        prompted for information about each relevant task.

        The AdminApp "options" command may be used to get a list of all
        possible options for a given ear file.  The AdminApp "help" command
        may be used to get more information about each particular option.

        Method: installInteractive

        Arguments: filename

        Description: Installs the application specified by "filename" using
        default options. The user is prompted for information about each
        relevant task.
```

Figure 20-11. *Sample image from WSAShelp_08.py script*

Note to make things simpler for the application, modal dialog boxes are used. additionally, the text is displayed in a scroll pane, just in case it's larger than the available space.

Adding a Menu

One of the things that I wasn't too happy about with the previous scripts was how the "Highlight text" label and input field were always showing on the bottom of the frame. Another alternative is to use menu items for this and other kinds of actions. Let's start simple and replace the label and input field with a "Show" menu that allows the user to specify text to be highlighted. You can also provide some information about the script with some "Help" menu items. What does this do to the application? Figure 20-12 has some images from the WSAShelp_09.py script, showing the initial look of the menu and the associated drop-down menus. Additionally, you see an image showing the input dialog box that is displayed when the user selects the Show ➤ Highlight Text menu item.



admin is the executable for WebSphere scripting.
Syntax:

WASX7C admin is the executable for WebSphere scripting.
Syntax:

What text do you want to highlight?

Figure 20-12. *Sample images from WSAShelp_09.py script*
The complete application can be found in the code\Chap_20\WSAShelp_09.py file and requires only about 80 additional lines of code to add this non-trivial functionality.

AdminTask.help('-commands')

In the WAuJ book two different scripts were used to format all of the help text for the wsadmin scripting objects. That's because the AdminTask scripting object is significantly different from the other scripting objects. In fact it provides a framework for WebSphere developers to add scripting capabilities by adding methods to the AdminTask framework, instead of forcing changes to be made to the existing objects.

When I began to consider adding the ability of displaying AdminTask help information to the WSAShelp scripts, I first considered adding some kind of listener to the AdminTask pane that would allow the user to select either the "help -commands", or "help -commandGroups" portions of that pane. Unfortunately, this would not be an obvious solution to the problem. How would you convey this information to the user? Maybe by using some kind of special highlighting, possibly even using something like the convention used by browsers to indicate that text is, in fact, a link to additional information. I quickly discarded this approach after taking another look at the menu items that had just been added to the WSAShelp_09.py script. This appeared to be a much more obvious solution to the problem.

Unfortunately, trying to display information about the available AdminTask commands brings challenges of its own. For example, depending on the version of the WebSphere Application Server product being used, there can be a huge list of available AdminTask commands from which to choose. The following sections address some of the questions that come to mind.

How Do We Find and Identify Existing Commands?

The obvious answer to this question is to take a look at the output produced by the AdminTask.help('-commands') method call. How much text is generated by this call? As mentioned earlier, it depends. To give you an idea, you can run a quick test. Listing 20-18 shows that for a WebSphere version 7.0, network deployment installation, almost 1,200 lines of text are generated.

Listing 20-18. Lines of Text from the AdminTask.help('-commands') Output

```
wsadmin>print len( AdminTask.help( '-commands' ).splitlines() ) 1190
wsadmin>
```

WebSphere version 8.0 has more than 1,250 lines and WebSphere version 8.5 has more than 1,400. The time to process this output is not something that you would want to do in a simple event handler. That would be likely to cause a noticeable and unacceptable delay in the application. As you’ve read, you should consider doing this processing on a separate (SwingWorker) thread.

Showing the User that Something Is Happening

Since the processing of the AdminTask.help('-commands') command output may cause a non-trivial delay, you need to consider how you communicate this fact to the user. One thing that came to mind was that you can initialize the Show ➤ AdminTask ➤ -commands menu item as disabled, at least until the processing is complete and the data exists in a usable form.

If you proceed with this decision, the SwingWorker class that you’re going to use has to know the menu item that needs to be enabled once processing completes. Additionally, it needs to know how to make the results of its processing available to the application.

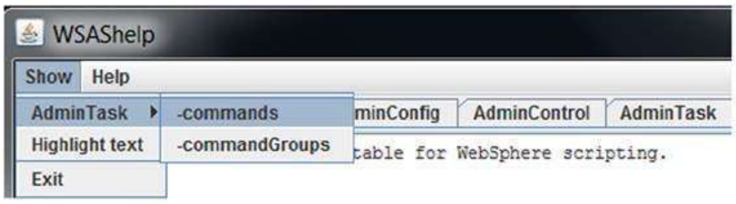
It seems pretty obvious that the result of the processing of this text should be an array or list of AdminTask command names. Don’t you think?
Alright, what’s it going to take to have a separate SwingWorker thread to process the output of the AdminTask.help('-commands') call? Listing 20-19 shows that it doesn’t take much effort. The doInBackground(...) method makes the call and processes each line of the resulting text. It can use a simple regular expression to identify the method names and each one that is found is added to the list. Then, all the done(...) method needs to do is enable the specified menu item.

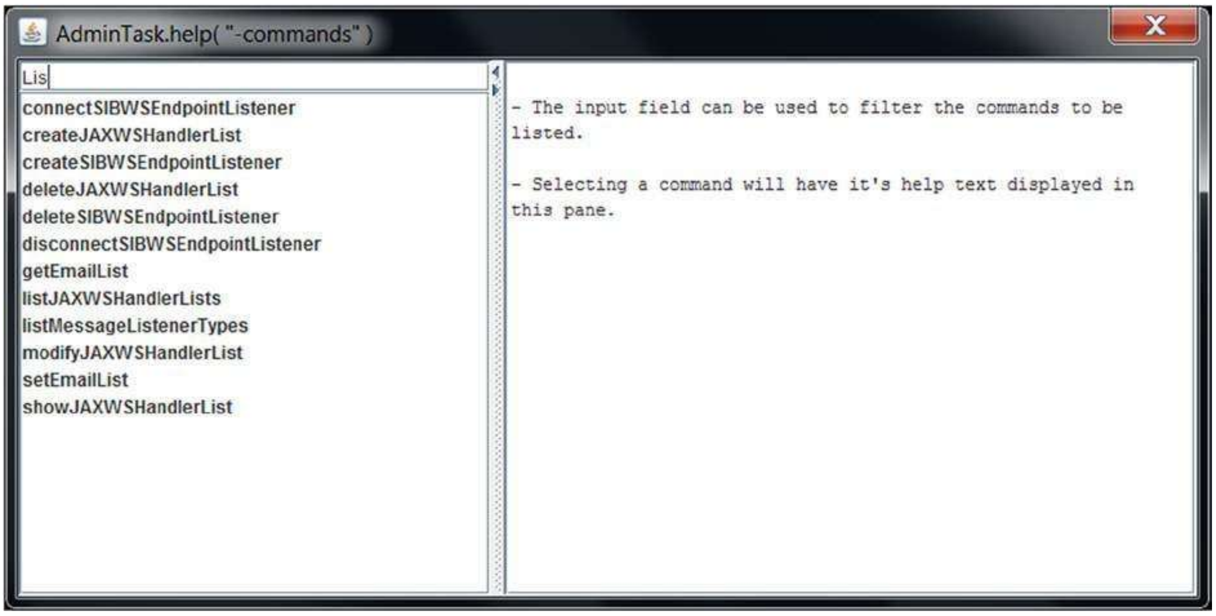
Listing 20-19. ATcommandTask Class from WSAShelp_10.py

```
class ATcommandTask( SwingWorker ) :
def __init__( self, List, menuItem ) :
SwingWorker.__init__( self )
self.commands = List
self.menuItem = menuItem
def doInBackground( self ) :
data = AdminTask.help( '-commands' ).splitlines()
for line in data[ 1: ] :
mo = re.match( '([a-zA-Z_2]+) -', line )
if mo :
self.commands.append( mo.group( 1 ) )
def done( self ) :
self.menuItem.setEnabled( 1 )
```

Now that you have this long list of commands or methods, how do you make things easy for your users? Scrolling through a list of 1,000 plus names is not exactly user friendly.
Do you remember what you did back in Chapter 15, with the javadocInfo script? You added an input field that allowed the user to filter the list of items to be displayed.[9] You can do the same thing here. If I’m looking for the AdminTask methods that have something to do with a particular topic, it can be somewhat daunting to look through 1,000 plus command names. By adding a filtering mechanism, you can greatly diminish the number of commands through which the user has to look.
Figure 20-13 shows some sample images from the WSAShelp_10.py script. The

first shows what happens to the application when the “Show” menu item has been selected. The second shows what happens when the AdminTask menu item has been selected. The third shows the initial view of the dialog box. One thing that you should notice is the size of the scrollbar thumb, which gives you an indication of the size of the list. Finally, you see what happens to the number of commands that contain “Lis”. I think that you’ll agree that this technique vastly improves the user experience when dealing with this huge list of AdminTask commands.





```
Lis
connectSIBWSEndpointListener
createJAXWSHandlerList
createSIBWSEndpointListener
deleteJAXWSHandlerList
deleteSIBWSEndpointListener
disconnectSIBWSEndpointListener
getEmailList
listJAXWSHandlerLists
listMessageListenerTypes
modifyJAXWSHandlerList
setEmailList
showJAXWSHandlerList
```

```
- The input field can be used to filter the commands to be
listed.

- Selecting a command will have it's help text displayed in
this pane.
```

Figure 20-13. *Sample images from WSAShelp_10.py script*

[9]The filtering was first added to javadocInfo_06.py, which is discussed in the section entitled “Filtering the List.”

One thing that you may notice about the WSAShelp_10.py script output is what happens when you select Show ➤ AdminTask ➤ -commandGroups menu item. A placeholder dialog box is displayed, indicating that work remains to be done.

AdminTask.help(‘-commandGroups’)

Now you get to figure out how to deal with the “-commandGroups” menu item. When you were working with the “-commands” menu item, it made sense to generate a list of the available AdminTask commands. With this selection, that makes less sense. Since each commandGroup is supposed to represent a group of similar commands, doesn’t it make more sense to display this as a tree, with each node of the tree representing one of the command groups? That way, to see the commands within a group, you would only have to expand the group by selecting (double-clicking) on a group name.

Like the ATcommandTask class seen in Listing 20-19 it seems reasonable to expect that you’re going to need another SwingWorker descendent class to process the AdminTask.help('-commandGroups') output to build the application dialog box.

The first thing that this class will need to do is to extract the commandGroup names from the AdminTask.help('-commandGroups') output. To figure out how to do this, first take a look at what this output looks like. Listing 20-20 shows the first few lines of output produced by this command.

Listing 20-20. The AdminTask.help('-commandGroups') Output wsadmin>print AdminTask.help('-commandGroups') WASX8005I: Available admin command groups:

```
AdminAgentNode - Admin Agent Managed Node related tasks
AdminAgentSecurityCommands - Commands used to configure security ...
AdminReports - Admin configuration reports
AdministrativeJobs - This command group contains all the job mana...
AppManagementCommands - Application management commands.
AuditAuthorizationCommands - Audit Authorization Table Commands ...
```

This format allows for easy identification and processing of the command group names. When a group name is identified (AdminAgentNode), then a subsequent AdminTask.help(...) method call can be used to identify the associated commands, if any, that exist within the group. For example, the output seen in Listing 20-21 shows the help text for two different command groups, only one of which (AdminReports) has associated commands. The format used by these command group help text messages make processing of them easy as well.

Listing 20-21. AdminTask.help(...) Using commandGroup Names

```
wsadmin>print AdminTask.help( 'AdminAgentNode' ) WASX8007I: Detailed
help for command group: AdminAgentNode
Description: Admin Agent Managed Node related tasks
Commands:
wsadmin>print AdminTask.help( 'AdminReports' ) WASX8007I: Detailed help
for command group: AdminReports

Description: Admin configuration reports Commands:
reportConfigInconsistencies - Checks the configuation repository ...
reportConfiguredPorts - Generates a report of the ports configure...

wsadmin>
```

Listing 20-22 shows the SwingWorker descendent class used to process the AdminTask.help('-commandGroups') text, to identify the names of the command groups and the associated AdminTask commands. This class is from the WSAShelp_11.py script. It may be interesting to note that the constructor requires only one parameter, that is, the variable used to identify the menu item to be enabled when processing is complete.

Listing 20-22. ATgroupsTask Class from WSAShelp_11.py

```
class ATgroupsTask( SwingWorker ) :
def __init__( self, menuItem ) :
SwingWorker.__init__( self )
self.menuItem = menuItem
def doInBackground( self ) :
try :
data = AdminTask.help(
'-commandGroups'
).expandtabs().splitlines()
self.root = DefaultMutableTreeNode(
'command groups'
)
empty = []
for line in data[ 1: ] :
mo = re.match( '([a-zA-Z ]+) -', line )
if mo :
groupName = mo.group( 1 )
```

```
group = None
text = AdminTask.help( groupName )
cmds = text.find( 'Commands:' )
if cmds > 0 :
for line in text[cmds+9:].splitlines() :
mo = re.match('([a-zA-Z_2]+) -', line)
if mo :
if not group :
group = DefaultMutableTreeNode(
groupName
)
group.add(
DefaultMutableTreeNode(
mo.group( 1 )
)
)
if group :
self.root.add( group )
else :
empty.append( groupName )
except :
print '\nError: %s\nvalue: %s' % sys.exc_info()[ :2 ]
def done( self ) :
self.menuItem.setEnabled( 1 )
```

Notice that this class constructor didn't include a parameter for the result like you saw in the ATcommandTask class, in Listing 20-19. This was done simply to show another approach. Are there problems with this approach? To answer that question, let's take a look at the code used to instantiate this class and access the resulting data (a DefaultMutableTreeNode[10]) for the root node. Listing 20-23 shows the other places in the code that instantiate and execute the ATgroupsTask, as well as retrieve the result of its processing.

Listing 20-23. Use of ATgroupsTask class in WSAShelp_11.py

```
...
self.ATgroupsTree = None
self.groups = ATgroupsTask( groupMI )
self.groups.execute()
...
def showCmdGroups( self, event ) :
```

```
...
left = JPanel( layout = BorderLayout() )
left.add(
JScrollPane(
JTree(
self.groups.root,
rootVisible = 0,
valueChanged = self.pickATgroup
)
),
BorderLayout.CENTER
)
...
```

■**Note** it’s important to note that in order to access the result of the ATgroupsTask processing, you need to have a variable for referencing the object. that is why WSAShelp_11.py includes the self.groups variable.

It might not be immediately obvious that the showCmdGroups(...) method uses the self.groups variable to reference the root node for the tree. The event listener routine is called when the Show ➤ AdminTask ➤ -commandGroups menu item is selected.

What does this mean as far as the JTree instance is concerned? To understand this, you should consider when and how often this routine is called. Each time the user invokes this routine, a new JTree instance is created. For a tiny application such as this, it might not make much of a difference, but it is better to understand the implications of your decisions as far as potential performance impact is concerned.

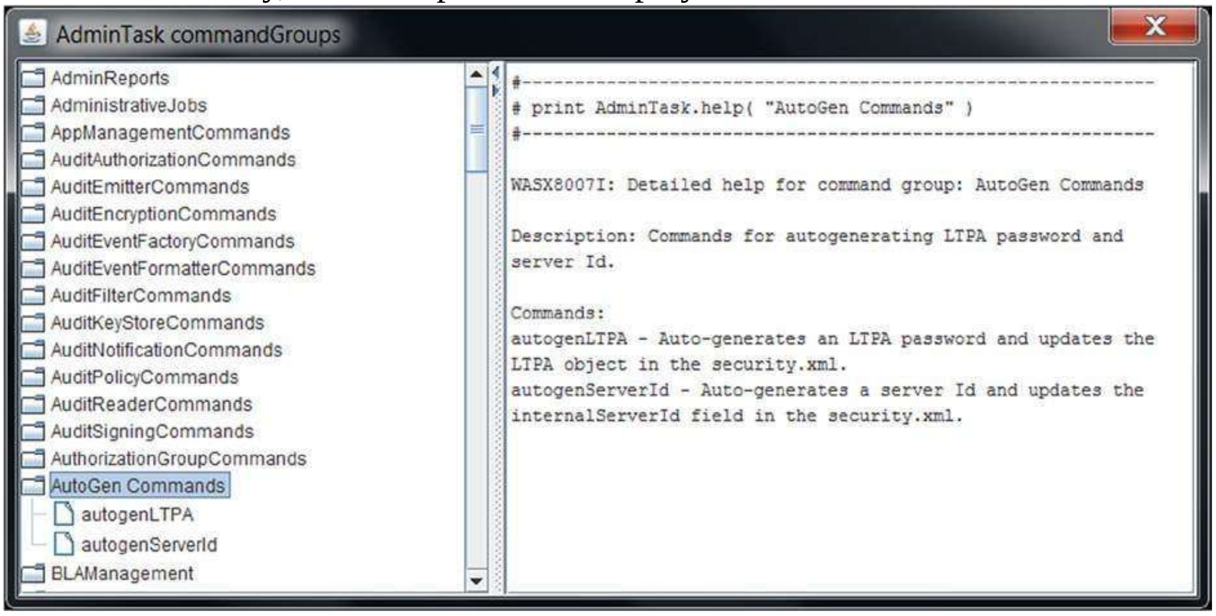
[10]See http://docs.oracle.com/javase/8/docs/api/javax/swing/tree/DefaultMutableTreeNo

Another ATgroupsTask Implementation

If you want to have the JTree populated by the ATgroupsTask process, what do you need to do? First, you have to realize what is required in order for the ATgroupsTask instance to modify the data of a JTree parameter passed to it. The JTree documentation[11] shows that no method exists to replace the tree root (no setRoot(...) method exists). Why is this?

You have to remember that the JTree class, like many other Swing classes (the JTable), has the data managed by an associated model. In the case of the JTree class, the data exists in, and is maintained by, the associated TreeModel[12] instance, which in all likelihood would be a DefaultTreeModel[13] object instance. The DefaultTreeModel class is where the setRoot(...) method resides. So, what do you need to change in the ATgroupsTask? Not much. You need to add a parameter for the constructor to hold the JTree object to be updated. It then adds a statement in the done(...) method that gets the tree model instance and uses its setRoot(...) method to replace the tree data. Another encouraging thing about this approach is that it also simplifies the event handler, instead of creating a new JTree.

Figure 20-14 shows the output after selecting the commandGroups menu item and then double-clicking the AutoGen Commands entry. Note that if you only select a table entry, it isn't expanded to display the child nodes.



```
#------------------------------------------------------------
# print AdminTask.help( "AutoGen Commands" )
#------------------------------------------------------------

WASX8007I: Detailed help for command group: AutoGen Commands

Description: Commands for autogenerating LTPA password and
server Id.

Commands:
autogenLTPA - Auto-generates an LTPA password and updates the
LTPA object in the security.xml.
autogenServerId - Auto-generates a server Id and updates the
internalServerId field in the security.xml.
```

Figure 20-14. *Sample image from WSAShelp_12.py script*

[11] See http://docs.oracle.com/javase/8/docs/api/javax/swing/JTree.html.
[12]See http://docs.oracle.com/javase/8/docs/api/javax/swing/tree/TreeModel.html.
[13]See http://docs.oracle.com/javase/8/docs/api/javax/swing/tree/DefaultTreeModel.htm

Step It Up: Displaying the "Steps" Help Text

I was going to finish this application, as well as the chapter, with the previous iteration of the script. But I demonstrated it to a friend who asked me some very good questions. What about the “steps” that some of the AdminTask commands have? Is there any way to view the help for these steps?

I told him that I would think about it and see what I could do. To begin, I had to see how many of the AdminTask commands had steps, as well as to see what it would take to display this information with a minimum impact to the existing script.

It didn’t take much effort to write a script to locate and display the names of the AdminTask commands that had any steps. It also displayed a little detail about the information that was processed. For example, when I ran this script (steps.py) in a WebSphere V 7 environment, it showed that 42 of the 1,162 commands have one or more steps (3.61%). That’s why I had forgotten all about this aspect of some AdminTask commands—only a small number have the “Steps” section populated.

How Should You Do It?

Realizing that steps exist doesn’t explain how to display the help text for a step. To understand that, you need only take a look at the AdminTask help text, as shown in Listing 20-24.

Listing 20-24. The Part of AdminTask.help() Text About Steps

```
help commandName stepName display detailed information for the specified
step belonging to the specified command
```

This answers the question, “how do you get the help text for a command step?” But it doesn’t answer the more challenging question, “how do you modify the WSAShelp_12.py script to allow the user to see this help text?”

Should You Add a Menu?

One thought is to add a menu to the command window that would allow the users to get the help for a specific step. When the command help text is being processed, if no steps exist, then the menu could be disabled or you could decide not to add the menu. Each step could be listed on the menu, thereby allowing the users to select the step for which help could be displayed in a modal dialog box. The good news is that you know how to do these things and you’ve seen them done. However, it’s not clear whether this is the best approach available. Let’s see what other options exist before you make a decision.

Should You Add Another Split Pane?

Another alternative that comes to mind to detect when steps exist and then use a split pane with the general command help description at the top, and a table with one row for each of the steps at the bottom. Again, this is something that you know how to do since this technique is shown on the middle tabs of the application. The bad news is that it would seem to be a significant amount of work and code for very infrequent use. Since less than 5% of the AdminTask commands have steps, it is not clear that this approach is worth the time and effort to implement.

Can You Make Parts of the Text Pane Selectable?

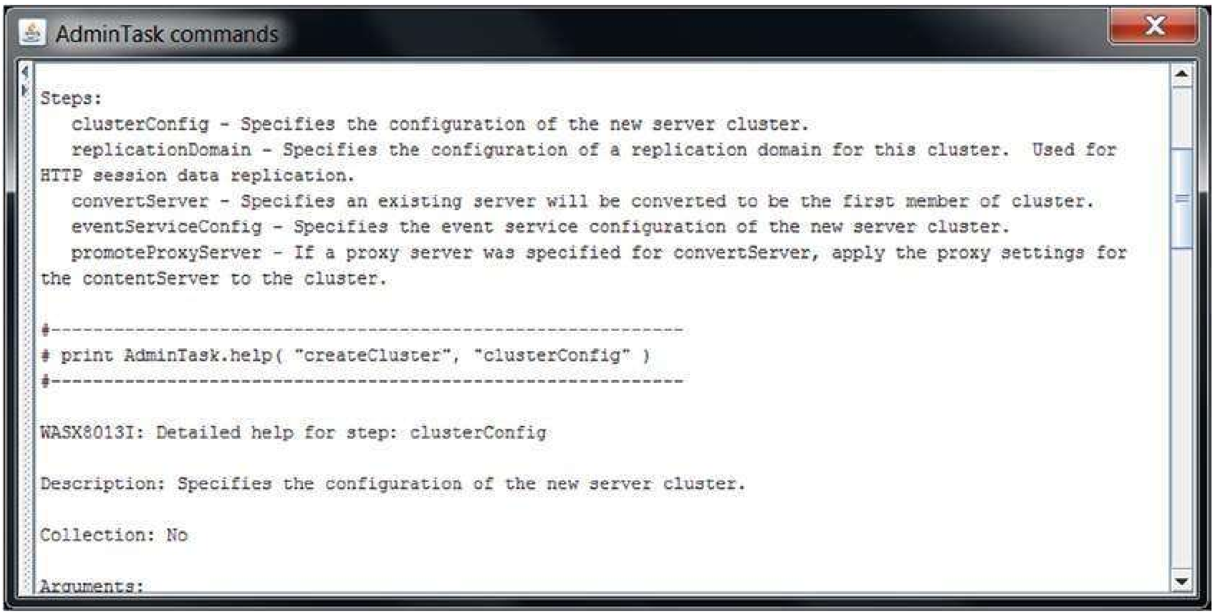
Another alternative is to allow the users to click on any of the visible step names. This is a little different and might be interesting to implement. Unfortunately, since you haven't done this yet, it's not clear how much time and effort, not to mention code, is required to do this.

And Now for Something Completely Different...

One of the things that got my attention while I was thinking about how best to approach this task was the fact that all of the previous options require extra effort on the part of the user to decide which of the steps they want to investigate. If they want to view multiple steps, multiple actions are required (select a menu item, look at the modal dialog box, close it, and repeat as necessary). Is there an easier way?

That's when I wondered about the possibility of displaying the help text for any and all steps in the same scrollable text pane containing the help text for the command. Another advantage to this approach is the fact that it requires about a dozen lines of code to be added in two places (the pickATcmd(...) and the pickATgroup(...) methods).

This iteration of the script is available in the code\Chap_20 directory, and is named WSAShelp_13.py. Figure 20-15 shows the "Steps" section of the help text for the createCluster command, and the beginning of the help text for the first of the associated steps.



```
Steps:
   clusterConfig - Specifies the configuration of the new server cluster.
   replicationDomain - Specifies the configuration of a replication domain for this cluster.  Used for
HTTP session data replication.
   convertServer - Specifies an existing server will be converted to be the first member of cluster.
   eventServiceConfig - Specifies the event service configuration of the new server cluster.
   promoteProxyServer - If a proxy server was specified for convertServer, apply the proxy settings for
the contentServer to the cluster.

#------------------------------------------------------------
# print AdminTask.help( "createCluster", "clusterConfig" )
#------------------------------------------------------------

WASX8013I: Detailed help for step: clusterConfig

Description: Specifies the configuration of the new server cluster.

Collection: No

Arguments:
```

Figure 20-15. *The help text for the clusterConfig step*

One of the important things about this iteration is that the first approach that came to mind wasn't the easiest or best approach for this particular situation. In fact, I considered multiple approaches and started to investigate what changes would be required for each approach before the "simple" answer came to me. Sometimes, it's not the first idea that comes to mind that ends up being the best approach.

Summary

All in all, this is now a pretty decent example of the iterative development of a non-trivial graphical application that uses a large variety of Swing capabilities. Even though it isn't perfect, it's a good tutorial on one way that Jython Swing applications can be developed. It is also a good example of how you might want to decide which approach is the most appropriate for you and your users' needs. Don't be afraid to stop and think about the options and which ones are going to make the application most useful for your users. It's also good to remember that you can do a quick proof of concept for potential enhancements for your application.

In the next chapter, you'll look at a different application and use an iterative improvement to add functionality as you improve the application. This application will be used to display details related to the security configuration report about a WebSphere environment.

Chapter 21

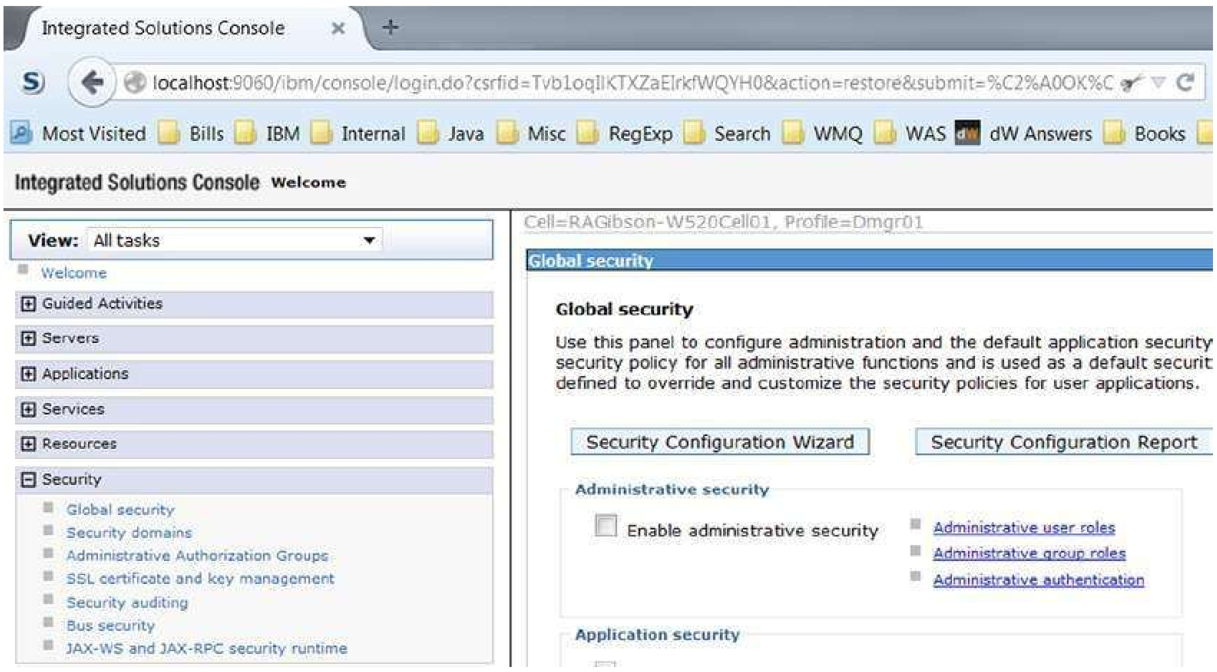
A Security Configuration Report Application

Not too long ago, someone asked me how difficult it would be to generate a security configuration report, similar to what is available from the WebSphere administration console. This chapter explores the creation of such an application to display the security report in a user-friendly way, all the while using the techniques that you've learned so far. This will allow the users to interact with the information in a way that provides a better understanding of the security-related configuration information for a WebSphere Application Server.

Generating the Administration Console Report

To get started, you are going to generate a static HTML security report for the environment. Then you will learn how, using iterative steps, to create an application that uses a GUI to display this security information in a user friendly way. To begin, an administrator must use the administration console to generate the report. This requires the administrator to be logged into the administration console using an appropriate security role. To generate the report, follow these steps (see Figure 21-1):

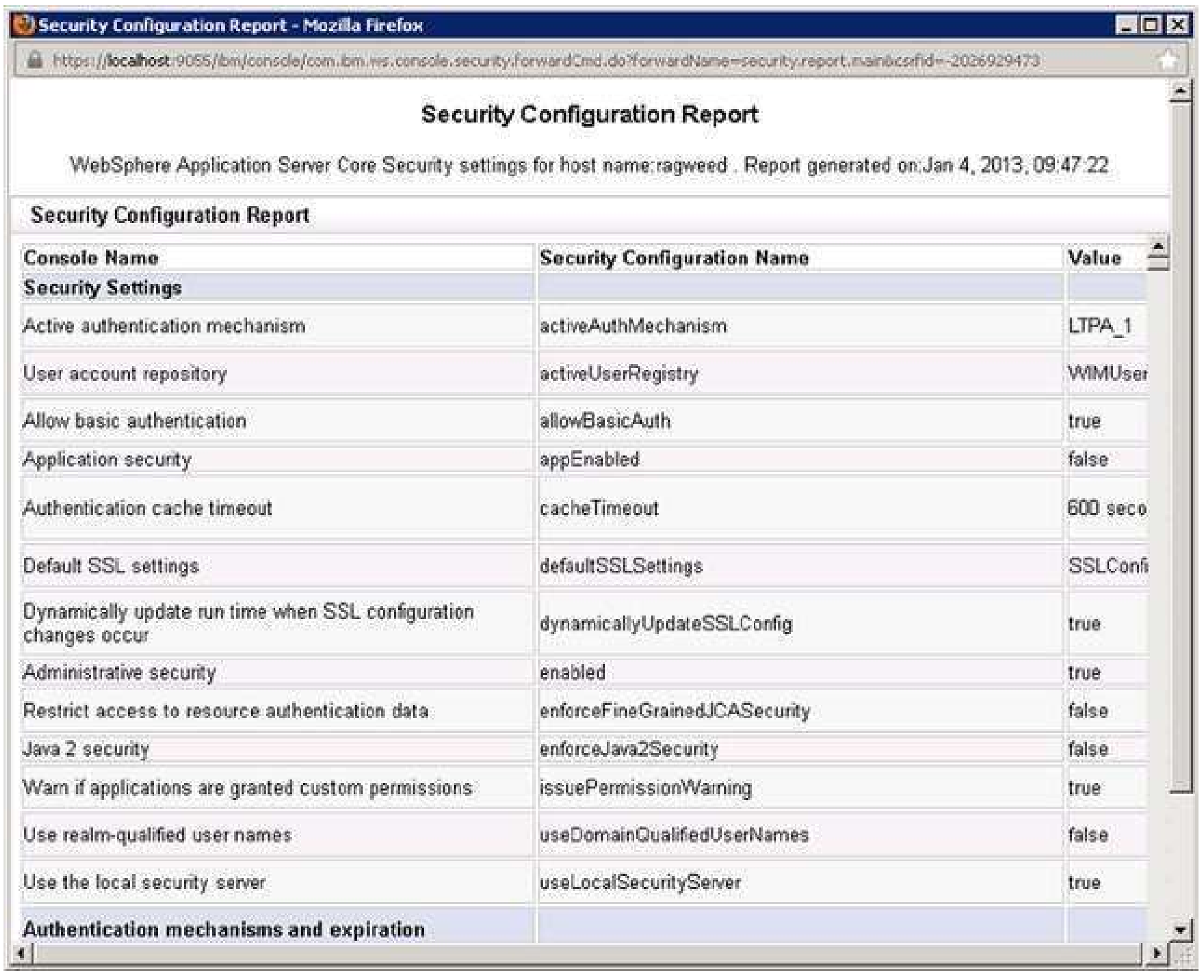
1. Select/expand the Security section on the left frame.
2. Select the Global Security link.
3. Click on the Security Configuration Report button.


Integrated Solutions Console
localhost:9060/ibm/console/login.do?csrfid=Tvb1oqIlKTXZaElrkfWQYH0&action=restore&submit=%C2%A0OK%C
Most Visited
Bills
IBM
Internal
Java
Misc
RegExp
Search
WMQ
WAS
dW Answers
Books
Integrated Solutions Console Welcome
View: All tasks
Welcome
Guided Activities
Servers
Applications
Services
Resources
Security
Global security
Security domains
Administrative Authorization Groups
SSL certificate and key management
Security auditing
Bus security
JAX-WS and JAX-RPC security runtime
Cell=RAGibson-W520Cell01, Profile=Dmgr01
Global security
Global security
Security Configuration Wizard
Security Configuration Report
Administrative security
Enable administrative security
Administrative user roles
Administrative group roles
Administrative authentication
Application security

Use this panel to configure administration and the default application security
security policy for all administrative functions and is used as a default securit
defined to override and customize the security policies for user applications.

Figure 21-1. *Generating a security configuration report*

This causes a browser page to be displayed containing the security report. Figure 21-2 shows the top of one such configuration report browser page.


Security Configuration Report - Mozilla Firefox

https://localhost:9055/ibm/console/com.ibm.ws.console.security.forwardCmd.do?forwardName=security.report.main&csrfid=-2026929473

Security Configuration Report

WebSphere Application Server Core Security settings for host name:ragweed . Report generated on:Jan 4, 2013, 09:47:22

Security Configuration Report

Console Name	Security Configuration Name	Value
Security Settings		
Active authentication mechanism	activeAuthMechanism	LTPA_1
User account repository	activeUserRegistry	WIMUser
Allow basic authentication	allowBasicAuth	true
Application security	appEnabled	false
Authentication cache timeout	cacheTimeout	600 seco
Default SSL settings	defaultSSLSettings	SSLConfi
Dynamically update run time when SSL configuration changes occur	dynamicallyUpdateSSLConfig	true
Administrative security	enabled	true
Restrict access to resource authentication data	enforceFineGrainedJCASecurity	false
Java 2 security	enforceJava2Security	false
Warn if applications are granted custom permissions	issuePermissionWarning	true
Use realm-qualified user names	useDomainQualifiedUserNames	false
Use the local security server	useLocalSecurityServer	true
Authentication mechanisms and expiration		

Figure 21-2. *Sample Security Configuration Report output*

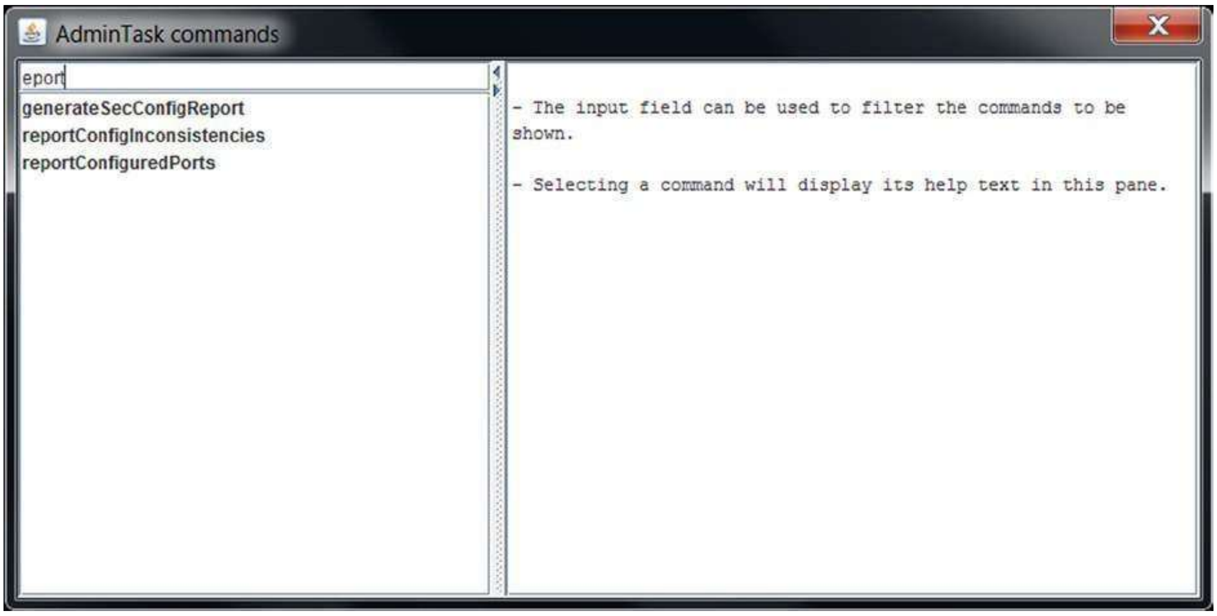
Do you see any problem with this? My biggest complaint about this report is that it is very static. About the only thing that you can do with it is to use the text-searching capability of the browser to locate specific text. These steps were tested on three different versions of the WebSphere Application Server product and are the same for versions 7.0, 8.0, and 8.5, which is good. It makes it easier to describe it here. The biggest challenge that I encountered in verifying these steps in this version is the fact that I had to see which, if any, of the deployment managers were active, start those that weren't currently active, log into the administration console, and check the steps.

The next challenge was consuming the report data. The browser page produced when the report button is selected is, in a word, huge. The good news, however, is that it is in a table format, with each section having its own header row shaded in a different color. The report is so big because there are about two dozen sections and some of the sections are quite large. In fact, some of the sections have more than 300 rows of information. In case you are interested, I've included the script that I used to determine this information—reportSectionSize.py—in the code\Chap_21 directory.

The Scripting Report Method

Since you're interested in accessing this information using an wsadmin script, you need to determine whether there is a command or scripting object method that allows you access to this information. You can even use the WSAShelp.py script from the previous chapter to locate it.

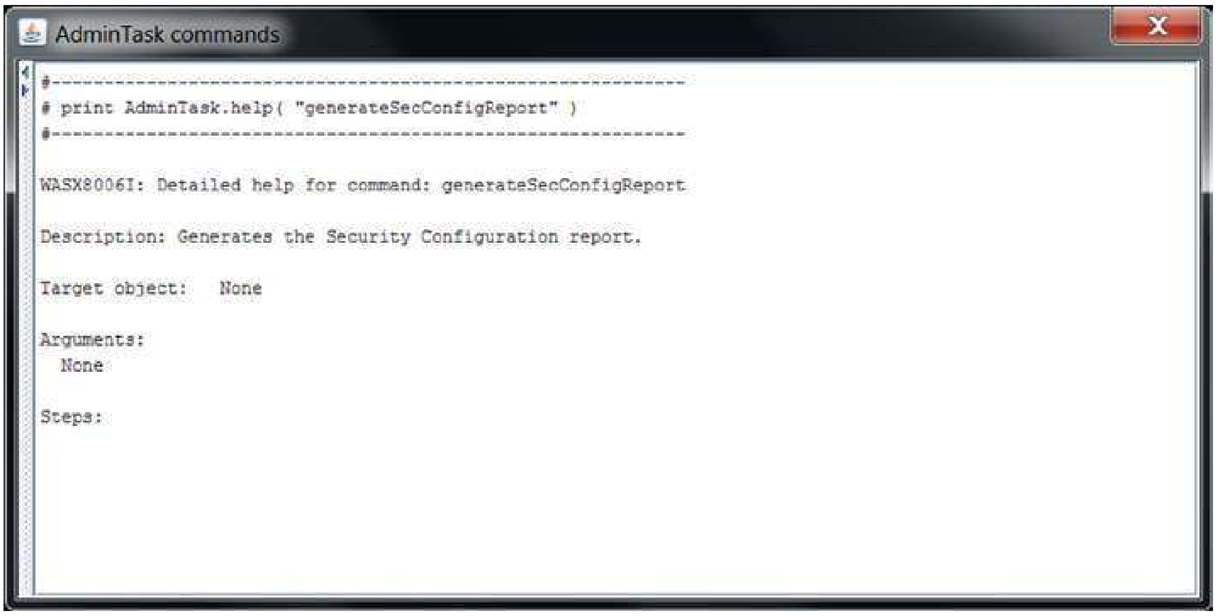
Choose the Show ➤ Highlight Text menu and move through the various tabs. You'll see that there isn't a method with a name or abstract that contains the word "report." Use the Show ➤ AdminTask ➤ -commands filtering mechanism to verify that the only commands that include "eport" (the "r" is omitted to avoid case-sensitive returns) are those shown in Figure 21-3.



```
generateSecConfigReport
reportConfigInconsistencies
reportConfiguredPorts
```

```
- The input field can be used to filter the commands to be
shown.

- Selecting a command will display its help text in this pane.
```

Figure 21-3. *AdminTask commands containing "eport"*
Select the first option—generateSecConfigReport—and click on the top one-touch expansion icon on the divider. You'll see an image similar to Figure 21-4.



AdminTask commands

```
#-------------------------------------------------------------
# print AdminTask.help( "generateSecConfigReport" )
#-------------------------------------------------------------

WASX8006I: Detailed help for command: generateSecConfigReport

Description: Generates the Security Configuration report.

Target object:   None

Arguments:
  None

Steps:
```

Figure 21-4. *AdminTask commands containing "eport"*

The unfortunate thing about this help text is that it doesn't tell you anything about the format of the report that is produced. To do that, you need to execute the command and take a look at the output. Fortunately, this is really easy to do with the wsadmin command. Because of the number of lines generated, I suggest that you use the wsadmin -c command-line option to execute a single command and have the output redirected to a file. Listing 21-1 shows an example of this wsadmin command.[1]

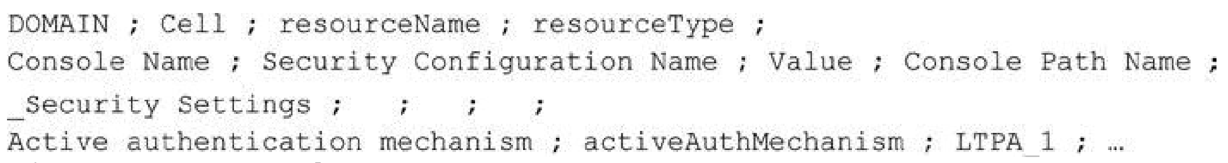
Listing 21-1. Sending the generateSecConfigReport to a File

```
wsadmin -conntype none -lang jython -c \
"print AdminTask.generateSecConfigReport()" >SecCfgRpt.txt
```

It doesn't take too much effort to understand the format of the data produced by this AdminTask command. The first few lines are shown in Figure 21-5. There are four columns of data on each line, and semicolons (;) delimit the column data. The second line of text corresponds to the column headings on the table (the rows containing Console Name and Security Configuration Name).

[1] The -lang jython command-line option is needed only if you haven't changed the default scripting language in the appropriat wsadmin.properties file. The -conntype none option tells wsadmin to execute in local mode, so no connection is attempted to a possibly inactive application server. It is important to remember that the command should appear on a single line. The backslash\isn't actually

part of the command and should not be entered.



```
DOMAIN ; Cell ; resourceName ; resourceType ;
Console Name ; Security Configuration Name ; Value ; Console Path Name ;
_Security Settings ;   ;   ;   ;
Active authentication mechanism ; activeAuthMechanism ; LTPA_1 ; …
```

Figure 21-5. *Example security report output*

Each section heading row has a leading underscore (_) in front of the section heading (such as _Security Settings). The rows that follow contain the values for each row in the table.[2]

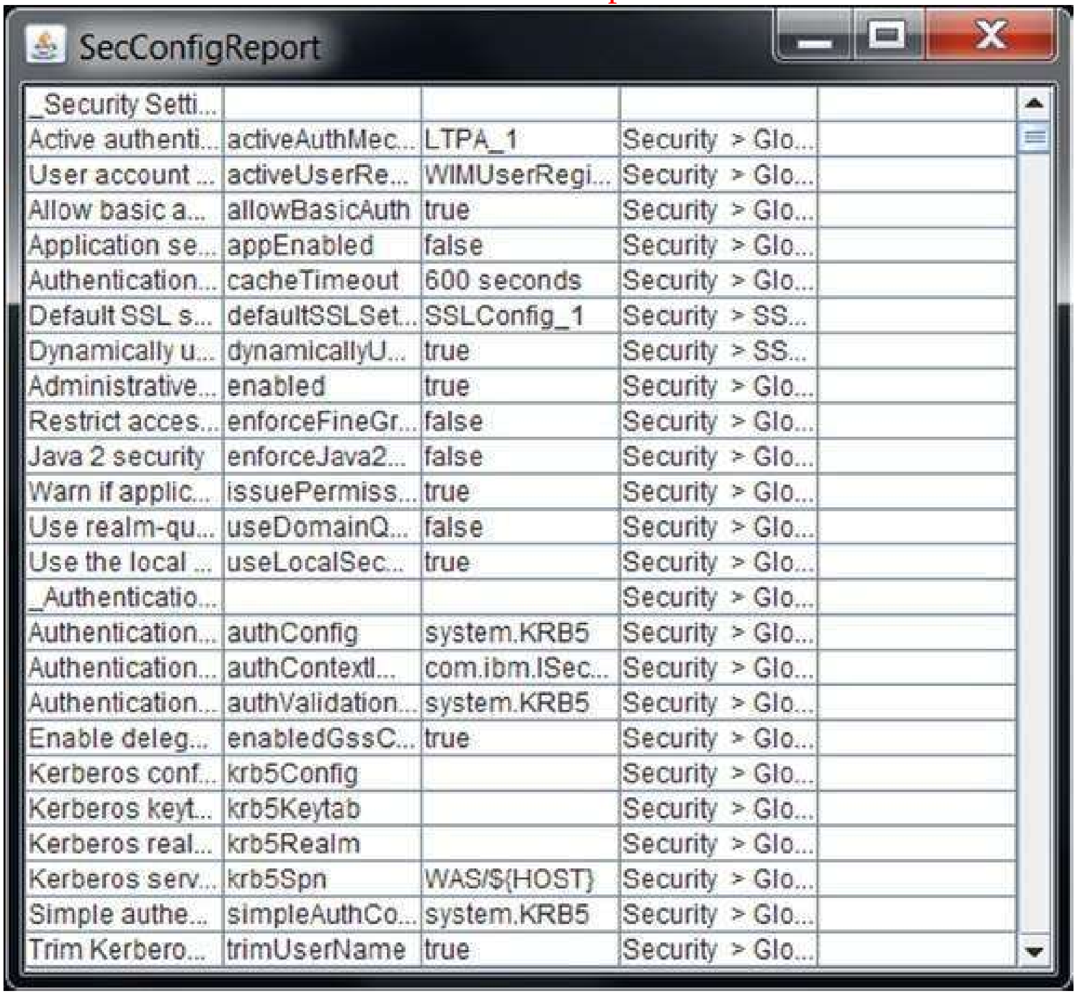
First Attempt

Let's see what it takes to process this information using the data format that was just discussed, and see how it looks. Listing 21-2 shows the first (arguably quick and dirty) attempt at doing this.

Listing 21-2. SecConfigReport_01.py—Quick and Dirty Attempt

```
class SecConfigReport_01( java.lang.Runnable ) :
def run( self ) :
frame = JFrame(
'SecConfigReport_01',
size = ( 300, 300 ),
locationRelativeTo = None,
defaultCloseOperation = JFrame.EXIT_ON_CLOSE
)
data = []
text = AdminTask.generateSecConfigReport()
for line in text.splitlines()[ 2: ] :
data.append(
[ info.strip() for info in line.split( ';' ) ]
)
frame.add(
JScrollPane(
JTable( data, ';;;;'.split( ';' ) )
)
)
frame.pack()
frame.setVisible( 1 )
```

Figure 21-6 shows the output that is produced when this script is executed. The good news is that you don’t have to invest too much time or effort into this application. The bad news is that this lack of effort shows. [2]In fact, this information is how the reportSectionSize.py script processes the data to determine the number and sizes of the report sections.



SecConfigReport

_Security Setti...				
Active authenti...	activeAuthMec...	LTPA_1	Security > Glo...	
User account ...	activeUserRe...	WIMUserRegi...	Security > Glo...	
Allow basic a...	allowBasicAuth	true	Security > Glo...	
Application se...	appEnabled	false	Security > Glo...	
Authentication...	cacheTimeout	600 seconds	Security > Glo...	
Default SSL s...	defaultSSLSet...	SSLConfig_1	Security > SS...	
Dynamically u...	dynamicallyU...	true	Security > SS...	
Administrative...	enabled	true	Security > Glo...	
Restrict acces...	enforceFineGr...	false	Security > Glo...	
Java 2 security	enforceJava2...	false	Security > Glo...	
Warn if applic...	issuePermiss...	true	Security > Glo...	
Use realm-qu...	useDomainQ...	false	Security > Glo...	
Use the local ...	useLocalSec...	true	Security > Glo...	
_Authenticatio...			Security > Glo...	
Authentication...	authConfig	system.KRB5	Security > Glo...	
Authentication...	authContextI...	com.ibm.ISec...	Security > Glo...	
Authentication...	authValidation...	system.KRB5	Security > Glo...	
Enable deleg...	enabledGssC...	true	Security > Glo...	
Kerberos conf...	krb5Config		Security > Glo...	
Kerberos keyt...	krb5Keytab		Security > Glo...	
Kerberos real...	krb5Realm		Security > Glo...	
Kerberos serv...	krb5Spn	WAS/${HOST}	Security > Glo...	
Simple authe...	simpleAuthCo...	system.KRB5	Security > Glo...	
Trim Kerbero...	trimUserName	true	Security > Glo...	

Figure 21-6. *Output of SecConfigReport_01.py*

From this output, you can see that each cell in the last column is empty. That’s a little strange, isn’t it? Based on this information, I took another, closer look at the text returned by the AdminTask command and found that every line ends with a semicolon, followed by a space (that’s “;”). So, you can take this into account and remove this extraneous delimiting data in your subsequent attempts.

The other thing that this first attempt shows is that the time needed to process the text, although it wasn’t insignificant, wasn’t exactly terrible, at least on the machine that I was using. You will need to decide if the data processing should be performed on a separate thread, and if so, what the application should display while this processing is occurring.

Second Attempt, Ignoring the Last Delimiter

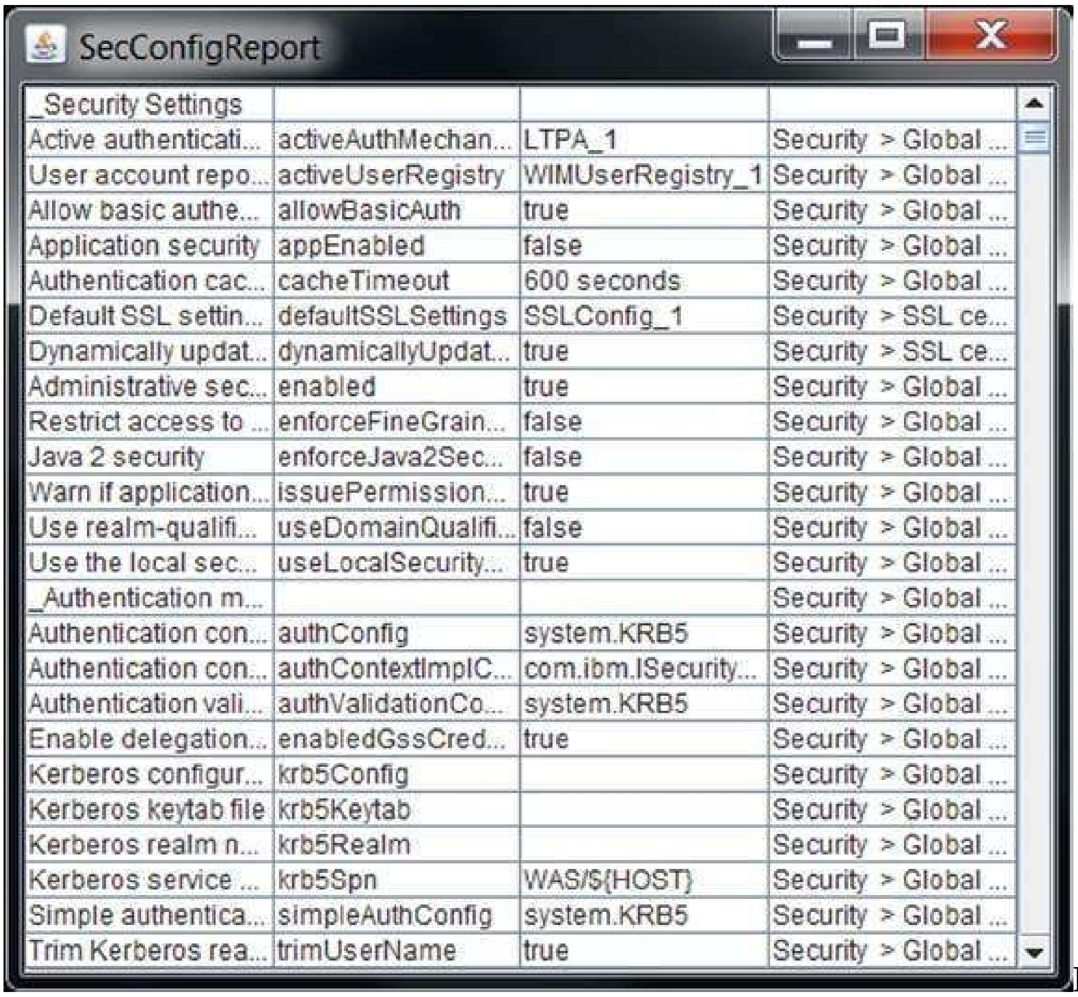
This next attempt focuses on fixing those immediately obvious shortcomings. This requires two changes to the code. Listing 21-3 shows these modifications.
Listing 21-3. SecConfigReport_02.py—Modified Lines

```
| ...
17| for line in text.splitlines()[ 2: ] :
18| data.append(
19| [
20| info.strip() for info in
21| line[ :-2 ].split( ';' )
22| ]
23| )
24| frame.add(
25| JScrollPane(
26| JTable( data, ';;;'.split( ';' ) ) 27| )
28| )
| ...
```

1. Instead of splitting each line of text using the semicolon delimiter, you simply need to ignore the last two characters of each line. To understand this change, compare line 19 in Listing 21-2 with lines 19-22 in Listing 21-3. The only reason that this list is shown on multiple lines in Listing 21-3 is to fit better within the book’s margins.

2. Since you are removing and ignoring the last delimiter from each line, you also need to remove one of the semicolons in the second parameter of the JTable instantiation. This is so the empty header array has the same number of columns as the data array parameter.

Figure 21-7 shows the resulting output of this application. Notice the empty last column that you saw in the first attempt is now gone.


SecConfigReport

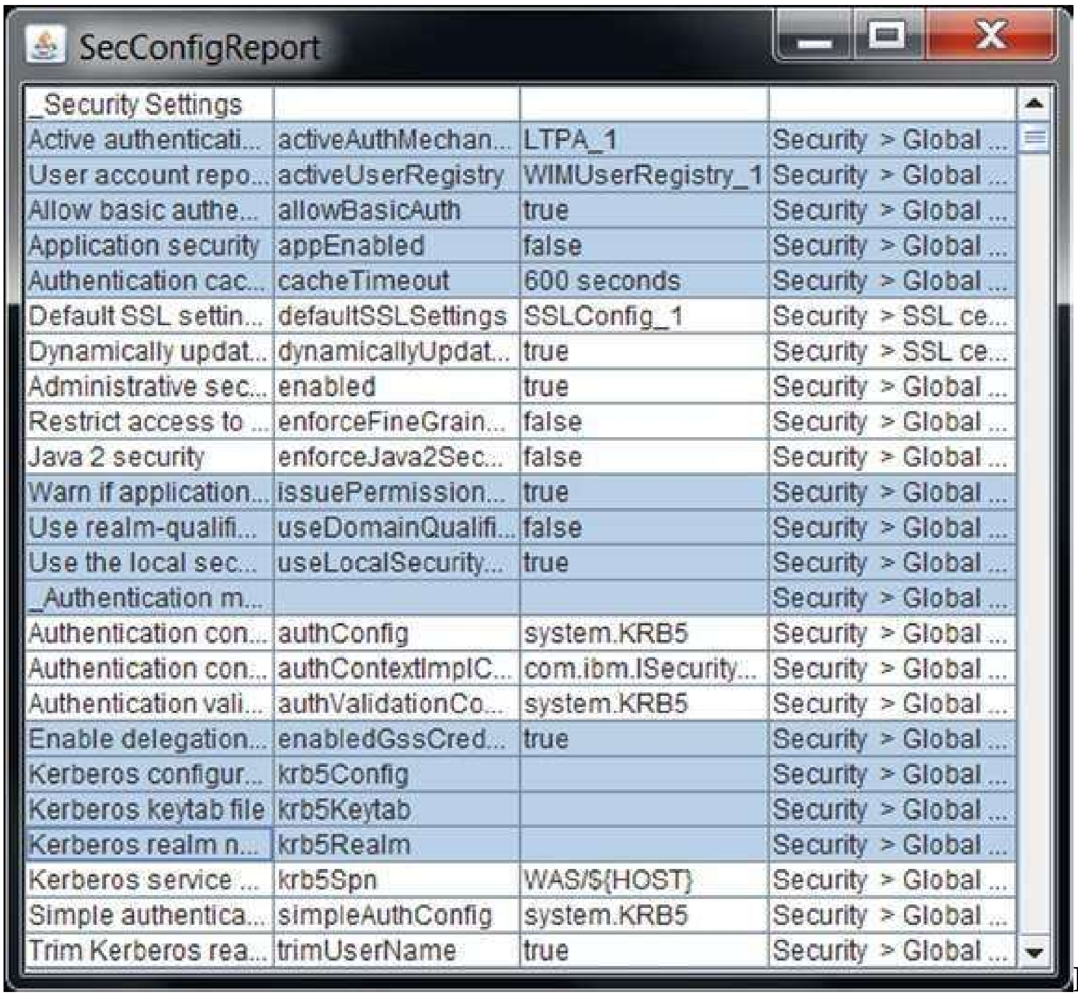
_Security Settings			
Active authenticati...	activeAuthMechan...	LTPA_1	Security > Global ...
User account repo...	activeUserRegistry	WIMUserRegistry_1	Security > Global ...
Allow basic authe...	allowBasicAuth	true	Security > Global ...
Application security	appEnabled	false	Security > Global ...
Authentication cac...	cacheTimeout	600 seconds	Security > Global ...
Default SSL settin...	defaultSSLSettings	SSLConfig_1	Security > SSL ce...
Dynamically updat...	dynamicallyUpdat...	true	Security > SSL ce...
Administrative sec...	enabled	true	Security > Global ...
Restrict access to ...	enforceFineGrain...	false	Security > Global ...
Java 2 security	enforceJava2Sec...	false	Security > Global ...
Warn if application...	issuePermission...	true	Security > Global ...
Use realm-qualifi...	useDomainQualifi...	false	Security > Global ...
Use the local sec...	useLocalSecurity...	true	Security > Global ...
_Authentication m...			Security > Global ...
Authentication con...	authConfig	system.KRB5	Security > Global ...
Authentication con...	authContextImplC...	com.ibm.ISecurity...	Security > Global ...
Authentication vali...	authValidationCo...	system.KRB5	Security > Global ...
Enable delegation...	enabledGssCred...	true	Security > Global ...
Kerberos configur...	krb5Config		Security > Global ...
Kerberos keytab file	krb5Keytab		Security > Global ...
Kerberos realm n...	krb5Realm		Security > Global ...
Kerberos service ...	krb5Spn	WAS/${HOST}	Security > Global ...
Simple authentica...	simpleAuthConfig	system.KRB5	Security > Global ...
Trim Kerberos rea...	trimUserName	true	Security > Global ...

Figure 21-7. *Initial output of SecConfigReport_02.py*

Adding a Table Model and Cell Renderer

If you test the application, it shouldn't take long to figure out that, as discussed in Chapter 12, the default values used by the JTable class may not be best fit for this application. For example, is there any reason that you would want the user to be able to modify any of the table cells? Not if you want to create an application that can display security configuration settings.

Another default setting that doesn't work well here is the fact that the user can select multiple non-contiguous rows. Figure 21-8 shows this multiple selection issue at work.



SecConfigReport

_Security Settings			
Active authenticati...	activeAuthMechan...	LTPA_1	Security > Global ...
User account repo...	activeUserRegistry	WIMUserRegistry_1	Security > Global ...
Allow basic authe...	allowBasicAuth	true	Security > Global ...
Application security	appEnabled	false	Security > Global ...
Authentication cac...	cacheTimeout	600 seconds	Security > Global ...
Default SSL settin...	defaultSSLSettings	SSLConfig_1	Security > SSL ce...
Dynamically updat...	dynamicallyUpdat...	true	Security > SSL ce...
Administrative sec...	enabled	true	Security > Global ...
Restrict access to ...	enforceFineGrain...	false	Security > Global ...
Java 2 security	enforceJava2Sec...	false	Security > Global ...
Warn if application...	issuePermission...	true	Security > Global ...
Use realm-qualifi...	useDomainQualifi...	false	Security > Global ...
Use the local sec...	useLocalSecurity...	true	Security > Global ...
_Authentication m...			Security > Global ...
Authentication con...	authConfig	system.KRB5	Security > Global ...
Authentication con...	authContextImplC...	com.ibm.ISecurity...	Security > Global ...
Authentication vali...	authValidationCo...	system.KRB5	Security > Global ...
Enable delegation...	enabledGssCred...	true	Security > Global ...
Kerberos configur...	krb5Config		Security > Global ...
Kerberos keytab file	krb5Keytab		Security > Global ...
Kerberos realm n...	krb5Realm		Security > Global ...
Kerberos service ...	krb5Spn	WAS/${HOST}	Security > Global ...
Simple authentica...	simpleAuthConfig	system.KRB5	Security > Global ...
Trim Kerberos rea...	trimUserName	true	Security > Global ...

Figure 21-8. *Output of SecConfigReport_02.py showing selection issue*

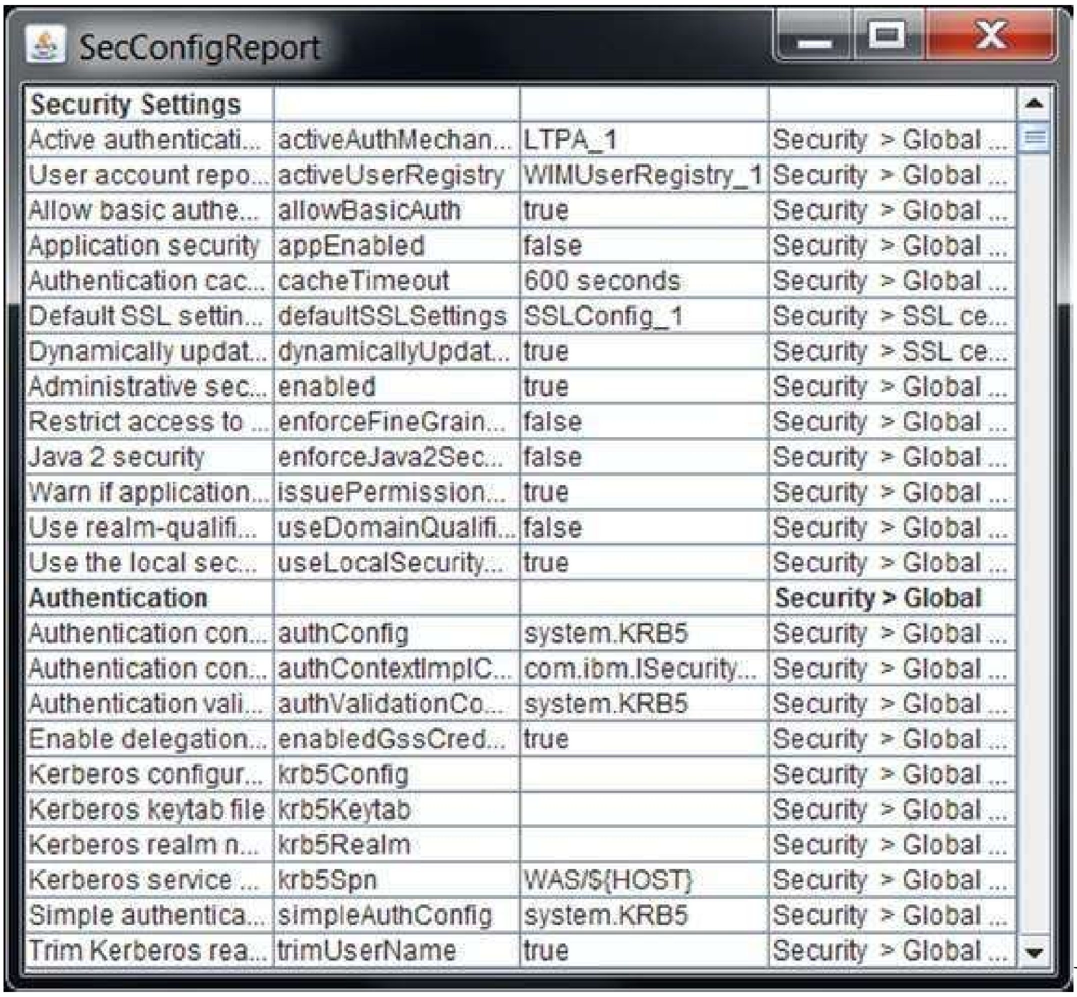
It's also not great that the sections are hard to distinguish from the rest of the data. Maybe you could add a cell renderer to make these rows easier to find and see. Listing 21-4 shows the two classes that were added to this script for the table model and cell renderer instances.

Listing 21-4. SecConfigReport_03.py—TableModel and Cell Renderer

```
class reportTableModel( DefaultTableModel ) :
def __init__( self, data, headings ) :
DefaultTableModel.__init__( self, data, headings )
def isCellEditable( self, row, col ) :
return 0
def getColumnClass( self, col ) :
return String
```

```
class reportRenderer( DefaultTableCellRenderer ) :
def getTableCellRendererComponent(
self,
table, # JTable - table containing value
value, # Object - value being rendered
isSelected, # boolean - Is value selected?
hasFocus, # boolean - Does this cell have focus?
row, # int - Row # ( 0 .. N-1 )
col # int - Col # ( 0 .. N-1 )
) :
DTCR = DefaultTableCellRenderer
comp = DTCR.getTableCellRendererComponent(
self, table, value, isSelected, hasFocus, row, col
)
if value :
if table.getValueAt( row, 0 ).startswith( '_' ) :
if col == 0 :
value = value[ 1: ]
comp.setText( '<html><b>%s</b>' % value )
return comp
```

Figure 21-9 shows the effects of these changes. This is a real improvement. I’m not sure if displaying the section row data in bold is good enough, but it does make it easier to locate these rows, doesn’t it?



Security Settings			
Active authenticati...	activeAuthMechan...	LTPA_1	Security > Global ...
User account repo...	activeUserRegistry	WIMUserRegistry_1	Security > Global ...
Allow basic authe...	allowBasicAuth	true	Security > Global ...
Application security	appEnabled	false	Security > Global ...
Authentication cac...	cacheTimeout	600 seconds	Security > Global ...
Default SSL settin...	defaultSSLSettings	SSLConfig_1	Security > SSL ce...
Dynamically updat...	dynamicallyUpdat...	true	Security > SSL ce...
Administrative sec...	enabled	true	Security > Global ...
Restrict access to ...	enforceFineGrain...	false	Security > Global ...
Java 2 security	enforceJava2Sec...	false	Security > Global ...
Warn if application...	issuePermission...	true	Security > Global ...
Use realm-qualifi...	useDomainQualifi...	false	Security > Global ...
Use the local sec...	useLocalSecurity...	true	Security > Global ...
Authentication			**Security > Global**
Authentication con...	authConfig	system.KRB5	Security > Global ...
Authentication con...	authContextImplC...	com.ibm.ISecurity...	Security > Global ...
Authentication vali...	authValidationCo...	system.KRB5	Security > Global ...
Enable delegation...	enabledGssCred...	true	Security > Global ...
Kerberos configur...	krb5Config		Security > Global ...
Kerberos keytab file	krb5Keytab		Security > Global ...
Kerberos realm n...	krb5Realm		Security > Global ...
Kerberos service ...	krb5Spn	WAS/${HOST}	Security > Global ...
Simple authentica...	simpleAuthConfig	system.KRB5	Security > Global ...
Trim Kerberos rea...	trimUserName	true	Security > Global ...

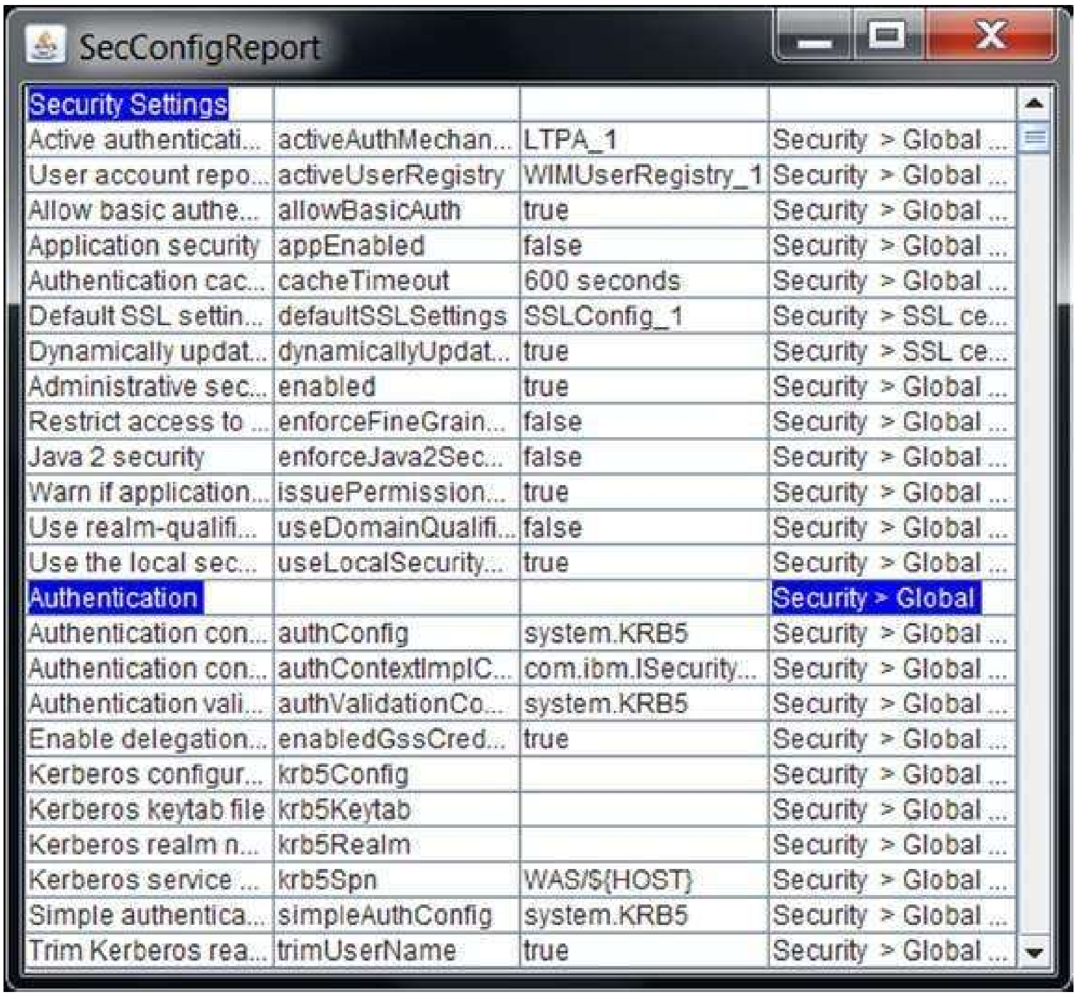
Using Color Instead of a Bold Font

Wait a minute, since you're using HTML to make the section rows bold, why don't you just use HTML to add color instead? Listing 21-5 shows the changes made to SecConfigReport_04.py to do just this.

Listing 21-5. SecConfigReport_04.py—HTML Coloring

```
html = '<html><font bgcolor=blue color=white>%s</font>'
if value :
if table.getValueAt( row, 0 ).startswith( '_' ) :
if col == 0 :
value = value[ 1: ]
comp.setText( html % value )
```

This should solve the problem, right? Well, not quite. Figure 21-10 shows that for empty cells the HTML coloring has no effect.


SecConfigReport

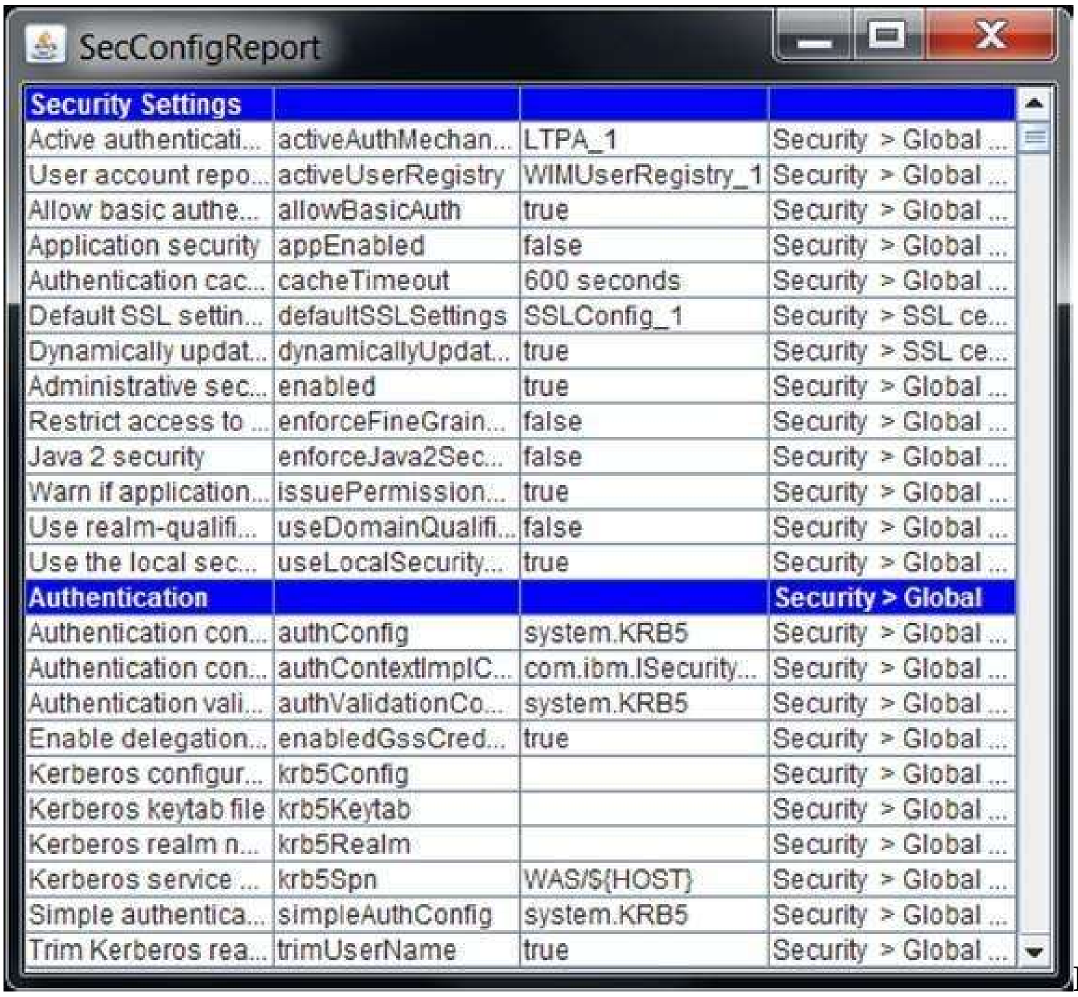
Security Settings			
Active authenticati...	activeAuthMechan...	LTPA_1	Security > Global ...
User account repo...	activeUserRegistry	WIMUserRegistry_1	Security > Global ...
Allow basic authe...	allowBasicAuth	true	Security > Global ...
Application security	appEnabled	false	Security > Global ...
Authentication cac...	cacheTimeout	600 seconds	Security > Global ...
Default SSL settin...	defaultSSLSettings	SSLConfig_1	Security > SSL ce...
Dynamically updat...	dynamicallyUpdat...	true	Security > SSL ce...
Administrative sec...	enabled	true	Security > Global ...
Restrict access to ...	enforceFineGrain...	false	Security > Global ...
Java 2 security	enforceJava2Sec...	false	Security > Global ...
Warn if application...	issuePermission...	true	Security > Global ...
Use realm-qualifi...	useDomainQualifi...	false	Security > Global ...
Use the local sec...	useLocalSecurity...	true	Security > Global ...
Authentication			Security > Global
Authentication con...	authConfig	system.KRB5	Security > Global ...
Authentication con...	authContextImplC...	com.ibm.ISecurity...	Security > Global ...
Authentication vali...	authValidationCo...	system.KRB5	Security > Global ...
Enable delegation...	enabledGssCred...	true	Security > Global ...
Kerberos configur...	krb5Config		Security > Global ...
Kerberos keytab file	krb5Keytab		Security > Global ...
Kerberos realm n...	krb5Realm		Security > Global ...
Kerberos service ...	krb5Spn	WAS/${HOST}	Security > Global ...
Simple authentica...	simpleAuthConfig	system.KRB5	Security > Global ...
Trim Kerberos rea...	trimUserName	true	Security > Global ...

You can solve this problem by remembering that the JLabel component that's used by the renderer also has color properties. Don't forget that the component is reused for every cell of the same type, so if you are going to change the cell color for section row cells, you also have to do so for every other cell in the table. Listing 21-6 shows the cell renderer class from SecConfigReport_05.py.

Listing 21-6. SecConfigReport_05.py—Revised Cell Renderer

```
class reportRenderer( DefaultTableCellRenderer ) :
def __init__( self ) :
self.bg = self.fg = None
def getTableCellRendererComponent(
self,
table, # JTable - table containing value
value, # Object - value being rendered
isSelected, # boolean - Is value selected?
hasFocus, # boolean - Does this cell have focus?
row, # int - Row # ( 0 .. N-1 )
col # int - Col # ( 0 .. N-1 )
) :
DTCR = DefaultTableCellRenderer
comp = DTCR.getTableCellRendererComponent(
self, table, value, isSelected, hasFocus, row, col
)
if self.bg == self.fg :
self.bg = comp.getBackground()
self.fg = comp.getForeground()
if value :
if table.getValueAt( row, 0 ).startswith( '_' ) :
comp.setBackground( Color.blue )
comp.setForeground( Color.white )
if col == 0 :
value = value[ 1: ]
comp.setText( '<html><b>%s</b>' % value )
else :
comp.setBackground( self.bg )
comp.setForeground( self.fg )
return comp
```

What does this do for the application output? Figure 21-11 shows that this significantly improves the application’s output.



Security Settings			
Active authenticati...	activeAuthMechan...	LTPA_1	Security > Global ...
User account repo...	activeUserRegistry	WIMUserRegistry_1	Security > Global ...
Allow basic authe...	allowBasicAuth	true	Security > Global ...
Application security	appEnabled	false	Security > Global ...
Authentication cac...	cacheTimeout	600 seconds	Security > Global ...
Default SSL settin...	defaultSSLSettings	SSLConfig_1	Security > SSL ce...
Dynamically updat...	dynamicallyUpdat...	true	Security > SSL ce...
Administrative sec...	enabled	true	Security > Global ...
Restrict access to ...	enforceFineGrain...	false	Security > Global ...
Java 2 security	enforceJava2Sec...	false	Security > Global ...
Warn if application...	issuePermission...	true	Security > Global ...
Use realm-qualifi...	useDomainQualifi...	false	Security > Global ...
Use the local sec...	useLocalSecurity...	true	Security > Global ...
Authentication			Security > Global
Authentication con...	authConfig	system.KRB5	Security > Global ...
Authentication con...	authContextImplC...	com.ibm.ISecurity...	Security > Global ...
Authentication vali...	authValidationCo...	system.KRB5	Security > Global ...
Enable delegation...	enabledGssCred...	true	Security > Global ...
Kerberos configur...	krb5Config		Security > Global ...
Kerberos keytab file	krb5Keytab		Security > Global ...
Kerberos realm n...	krb5Realm		Security > Global ...
Kerberos service ...	krb5Spn	WAS/${HOST}	Security > Global ...
Simple authentica...	simpleAuthConfig	system.KRB5	Security > Global ...
Trim Kerberos rea...	trimUserName	true	Security > Global ...

Figure 21-11. *Output of SecConfigReport_05.py*

Adjusting Column Widths

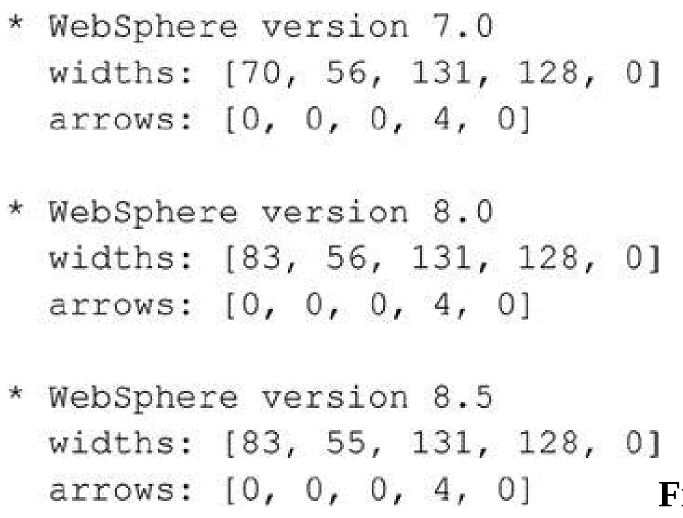
If you take another look at any of the preceding figures that show the output from any of the scripts, you should notice that even though all of the tables are within scroll panes, only vertical scrollbars are visible. Why do you think that is? Taking a closer look at the tables should give you a hint. The width of the table is fixed, and by default, each column is allocated the same amount of horizontal space. When a value can't be displayed in the available column space, the value is truncated and ellipses appear.

Unfortunately, the table doesn't have any column headings, so you can't adjust the width of individual columns. You can only maximize the application and hope that the screen on which the application is being displayed is wide enough to allow all of the available information. This isn’t a good choice for developers. What you need to do is figure how to deal with this and make some changes to the application so that the users can see all of the information that exists.

In Chapter 20, you learned that you can use the renderer to deal with cells where the data was too wide for the available space. Let’s take a closer look at the data that exists in each column to see if you can do the same kind of thing with this application. You can write a quick script to tell you about the maximum number of characters in each column. Note that the last column has a number of entries that contain the arrow, or greater than, symbol >. While you’re finding the width of each column, you can also count the maximum number of arrows in each column. Listing 21-7 shows the processReport routine from the columnInfo1.py script; you can find it in the code\Chap_21 directory. **Listing 21-7.** The processReport Routine from the columnInfo1.py Script

```
def columnInfo1() :
widths = [ 0 ] * 5
arrows = [ 0 ] * 5
report = AdminTask.generateSecConfigReport()
for line in report.splitlines()[ 2: ] :
col = 0
for cell in line.split( ';' ) :
widths[ col ] = max(
len( cell.strip() ),
widths[ col ]
)
arrows[ col ] = max(
cell.count( '>' ),
arrows[ col ]
)
col += 1
print ' widths:', widths
print ' arrows:', arrows
```

Figure 21-12 shows the results of using this script on the three most current versions of WebSphere Application Server.



```
* WebSphere version 7.0
  widths: [70, 56, 131, 128, 0]
  arrows: [0, 0, 0, 4, 0]

* WebSphere version 8.0
  widths: [83, 56, 131, 128, 0]
  arrows: [0, 0, 0, 4, 0]

* WebSphere version 8.5
  widths: [83, 55, 131, 128, 0]
  arrows: [0, 0, 0, 4, 0]
```

Figure 21-12. *Output of the columnInfo1.py script*

If you are going to have the cell renderer process the cell data and possibly use HTML to display the cell contents on multiple lines, any routine that determines the preferred column width should use the same kind of processing to more accurately determine the widest line in the column cells. Looking at the results in Figure 21-12, you'll want to take a closer look at the contents of third and fourth columns to reduce the width requirements for these columns.

Column Widths and Row Heights

The next iteration of the script, SecConfigReport_06.py, adds a routine to process the data to determine the maximum width of each column. It also sets the height of each row based on the space required to display the data in the available column width. Listings 21-8 and 21-9 show the setColumnWidths() method.

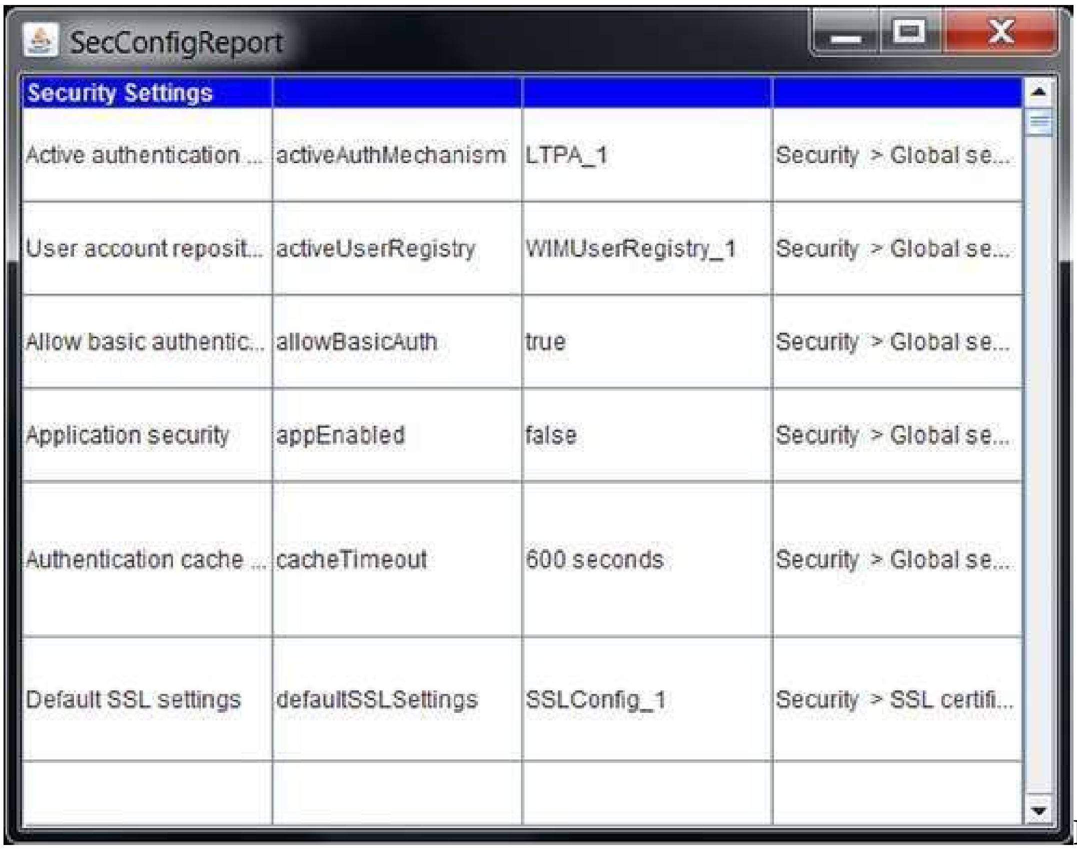
```
def setColumnWidths( self, table ) :
tcm = table.getColumnModel() # Table Column Model
model = table.getModel() # access the table data
margin = tcm.getColumnMargin() # gap between columns
rows = model.getRowCount() # How many rows exist?
cols = tcm.getColumnCount() # How many columns exist?
labels = [
JLabel( font = plainFont ),
JLabel( font = boldFont )
]
print ' Now Min Pre Max'
print '---------------+---+---+---+---'
```

```
100| metrics = [ fmPlain, fmBold ]
101| tWidth = 0 # Table width
102| section = 0 # is this row a section?
103| sections = 0 # Number of sections
104| for i in range( cols ) : # i == column index
105| col = tcm.getColumn( i )
106| idx = col.getModelIndex()
107| cWidth = 0 # Initial column width
108| for row in range( rows ) :
109| v0 = model.getValueAt( row, 0 )
110| if v0.startswith( '_' ) :
111| section = 1
112| sections += 1
113| else :
114| section = 0
115| comp = labels[ section ]
116| fm = metrics[ section ] # FontMetric
117| r = table.getCellRenderer( row, i )
118| v = model.getValueAt( row, idx )
119| if v.startswith( '_' ) :
120| v = v[ 1: ]
121| comp.setText( v )
122| cWidth = max(
123| cWidth,
124| comp.getPreferredSize().width
125| )
```

One thing that might catch your eye when you look at Listing 21-8 is the code on lines 111 and 112. What’s the difference between the section and sections variables? The former is used to indicate when the current row is a section header. So its value will be 0 or 1 to indicate this. If you search for occurrences of this variable, you will see where it is used as an index to both the labels and metrics arrays (lines 115 and 116). The sections variable is used to count the total number of sections processed in the report.

```
126| if cWidth > 0 :
127| col.setMinWidth( 128 + margin )
128| col.setPreferredWidth( 128 + margin ) 129| col.setMaxWidth( cWidth +
margin )
130| print 'Col: %d widths |%3d|%3d|%3d|%d' % ( 131| i,
col.getWidth(),
col.getMinWidth(),
col.getPreferredWidth(),
col.getMaxWidth()
)
tWidth += col.getPreferredWidth()
print '---------------+---+---+---+---'
h0 = table.getRowHeight()
print 'rowHeight:', h0
sections /= cols
print '#Sections:', sections
for row in range( rows ) :
lines = 1
for i in range( cols ) :
col = tcm.getColumn( i )
pre = col.getPreferredWidth()
idx = col.getModelIndex()
val = model.getValueAt( row, idx )
if not i :
section = val.startswith( '_' )
fm = metrics[ section ] # FontMetric
lines = max(
lines, int(
round( fm.stringWidth( val ) / pre ) + 1
)
)
table.setRowHeight( row, lines * h0 )
table.setPreferredScrollableViewportSize(
Dimension(
tWidth,
sections * table.getRowHeight()
)
)
```

Figure 21-13 shows the output of SecConfigReport_06.py. Comparing this image with the one in Figure 21-11, you can see that the most significant difference is the heights of the individual rows.



Security Settings			
Active authentication ...	activeAuthMechanism	LTPA_1	Security > Global se...
User account reposit...	activeUserRegistry	WIMUserRegistry_1	Security > Global se...
Allow basic authentic...	allowBasicAuth	true	Security > Global se...
Application security	appEnabled	false	Security > Global se...
Authentication cache ...	cacheTimeout	600 seconds	Security > Global se...
Default SSL settings	defaultSSLSettings	SSLConfig_1	Security > SSL certifi...

Figure 21-13. *Output of SecConfigReport_06.py*

Adding a Frame Resize Listener

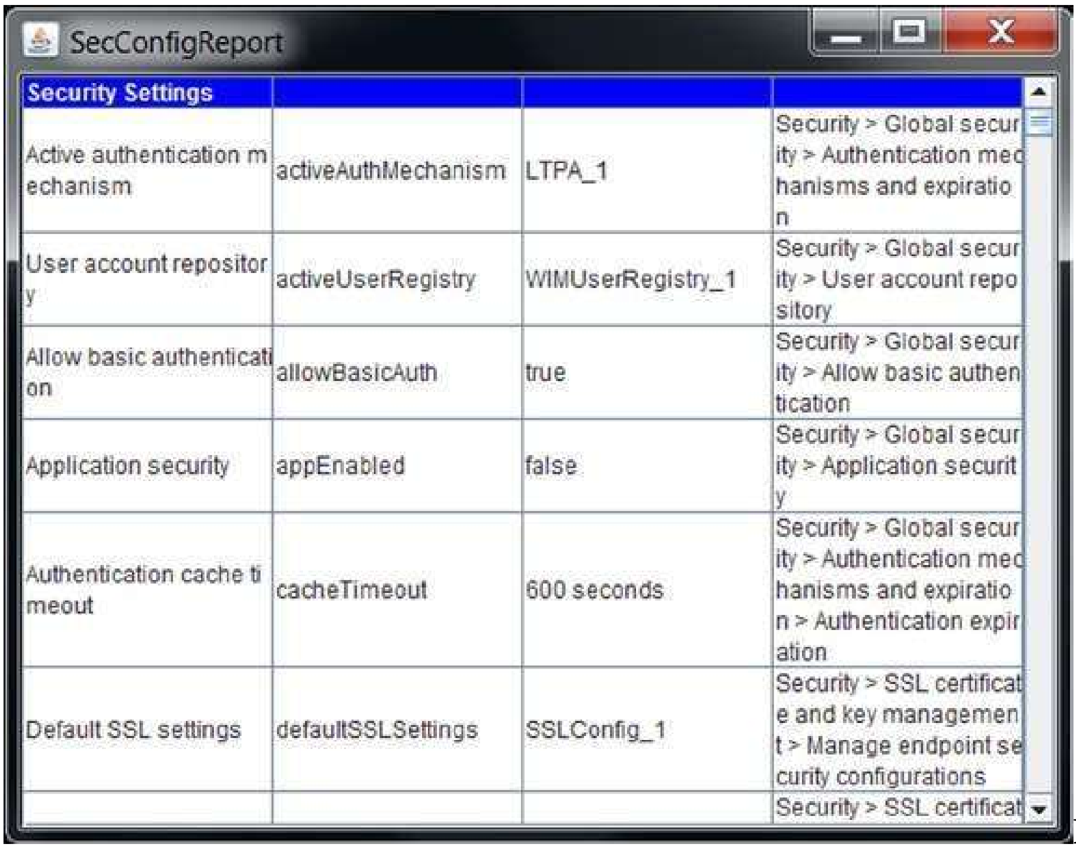
The next iteration of the script adds code to display the cell data (the width and height). Additionally, a new event handler routine determines how the information is displayed when the frame is resized. Listing 21-10 contains the source for this event handler.

Listing 21-10. The frameResized Method from SecConfigReport_07.py

```
def frameResized( self, ce ) :
try :
table = self.table
model = table.getModel() # Access the table data
width = table.getParent().getExtentSize().getWidth()
pWidth = int( width ) >> 2
tcm = table.getColumnModel() # Table Column Model
margin = tcm.getColumnMargin() # gap between columns
```

```
cols = tcm.getColumnCount()
for c in range( cols ) :
col = tcm.getColumn( c )
w = min( col.getMaxWidth, pWidth )
col.setWidth( w )
col.setPreferredWidth( w )
height = table.getRowHeight()
for row in range( model.getRowCount() ) :
table.setRowHeight( row, height )
table.repaint()
except :
print '\nError: %s\nvalue: %s' % sys.exc_info()[ :2 ]
```

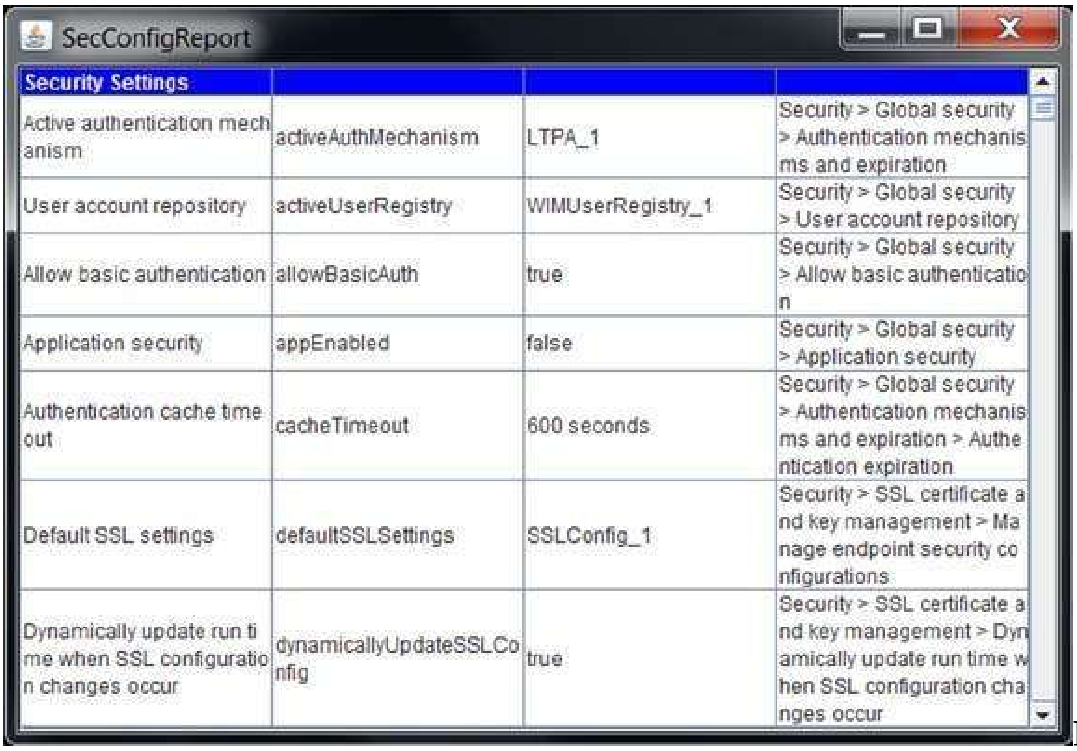
What does this mean as far as the application output is concerned? Figure 21-14 shows the initial, default rendering of the table and Figure 21-15 shows how the text changes when the frame, and therefore the table, are widened.



SecConfigReport

Security Settings			
Active authentication mechanism	activeAuthMechanism	LTPA_1	Security > Global security > Authentication mechanisms and expiration
User account repository	activeUserRegistry	WIMUserRegistry_1	Security > Global security > User account repository
Allow basic authentication	allowBasicAuth	true	Security > Global security > Allow basic authentication
Application security	appEnabled	false	Security > Global security > Application security
Authentication cache timeout	cacheTimeout	600 seconds	Security > Global security > Authentication mechanisms and expiration > Authentication expiration
Default SSL settings	defaultSSLSettings	SSLConfig_1	Security > SSL certificate and key management > Manage endpoint security configurations
			Security > SSL certificat

Figure 21-14. *Narrow output of SecConfigReport_07.py*


SecConfigReport

Security Settings			
Active authentication mechanism	activeAuthMechanism	LTPA_1	Security > Global security > Authentication mechanisms and expiration
User account repository	activeUserRegistry	WIMUserRegistry_1	Security > Global security > User account repository
Allow basic authentication	allowBasicAuth	true	Security > Global security > Allow basic authentication
Application security	appEnabled	false	Security > Global security > Application security
Authentication cache timeout	cacheTimeout	600 seconds	Security > Global security > Authentication mechanisms and expiration > Authentication expiration
Default SSL settings	defaultSSLSettings	SSLConfig_1	Security > SSL certificate and key management > Manage endpoint security configurations
Dynamically update run time when SSL configuration changes occur	dynamicallyUpdateSSLConfig	true	Security > SSL certificate and key management > Dynamically update run time when SSL configuration changes occur

Figure 21-15. *Wider output of SecConfigReport_07.py*

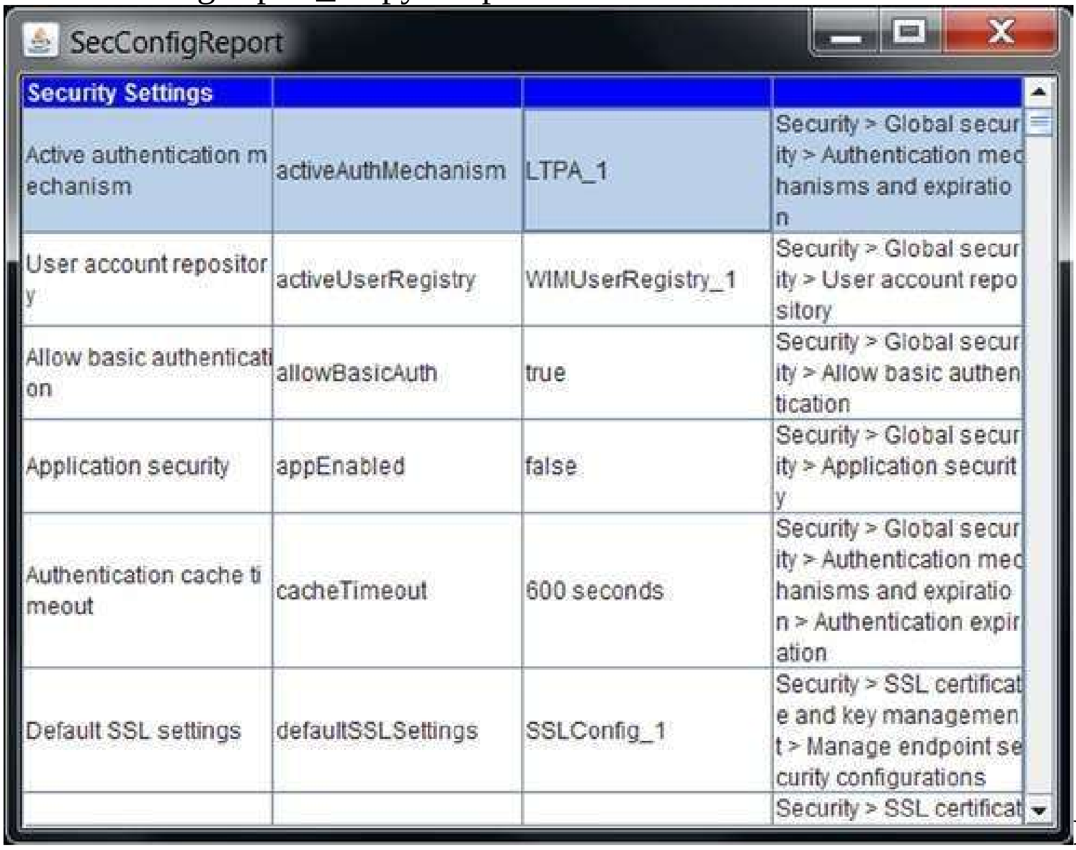
Notice how the cell contents are wrapped (especially in the first and last columns). Figure 21-15 shows what happens to the cell contents when the frame is widened. It is important that you notice how the row heights are adjusted based on how much vertical space is required to display the text.

Fixing the Row Selection Colors

Unfortunately there is a problem with this iteration of the script. If you test it by selecting a row, you'll witness the problem. There is no visual indication that a row has been selected. Fortunately, the changes required to fix this issue are localized to the table cell renderer code. The most significant changes in the reportRenderer class are these:

• The constructor initializes some arrays to hold the background colors for both selected and unselected rows.
• The getTableCellRendererComponent(...) method now:
• Saves the background and foreground colors for selected and unselected rows when they are first encountered.
• Decides which background and foreground color to use based on whether a row is selected and whether it is one of the header rows.

Figure 21-16 shows the sample output of the modified code. It can be found in the SecConfigReport_08.py script file.



Security Settings			
Active authentication mechanism	activeAuthMechanism	LTPA_1	Security > Global security > Authentication mechanisms and expiration
User account repository	activeUserRegistry	WIMUserRegistry_1	Security > Global security > User account repository
Allow basic authentication	allowBasicAuth	true	Security > Global security > Allow basic authentication
Application security	appEnabled	false	Security > Global security > Application security
Authentication cache timeout	cacheTimeout	600 seconds	Security > Global security > Authentication mechanisms and expiration > Authentication expiration
Default SSL settings	defaultSSLSettings	SSLConfig_1	Security > SSL certificate and key management > Manage endpoint security configurations
			Security > SSL certificat

Figure 21-16. *Output of SecConfigReport_08.py*

Which Rows Are Visible?

The next iteration of the script, shown in SecConfigReport_09.py, adds a descendent of the ChangeListener class to monitor changes in the viewport (which is a kind of JViewport[3]), which is the parent of the JTable containing the application information.[4] When the user views a different part of the table, the change listener event handler is called and it prints some information about which rows of the table are visible. Listing 21-11 shows the rowFinder class from this iteration of the script.

Listing 21-11. The rowFinder Class from SecConfigReport_09.py

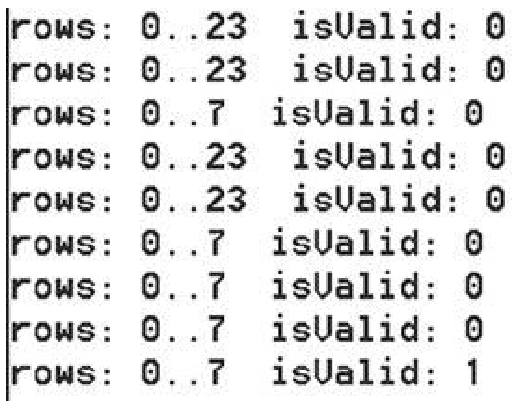
```
class rowFinder( ChangeListener ) :
def __init__( self, table ) :
self.table = table
```

```
def stateChanged( self, ce ) :
vPort = ce.getSource()
table = self.table
rect = vPort.getViewRect()
first = table.rowAtPoint(
Point( 0, rect.y )
)
last = table.rowAtPoint(
Point( 0, rect.y + rect.height - 1 )
)
print 'rows: %d..%d isValid: %d' % (
first,
last,
vPort.isValid()
)
```

I was hoping that the result of calling the isValid() method could be used to determine which rows are visible. Unfortunately, the result returned from this method wasn't quite what I expected. When it returns true, you can easily determine which rows are visible using code similar to Listing 21-11. However, the first and last rows that are visible include partial rows. So you can't use the result of this method to easily determine which complete table rows are visible. To do that requires additional computation.

If you look again at the output in Figure 21-17, you'll see that the routine is invoked multiple times. For most of these the result of the isValid() method is false. Eventually it will return a value of true, which is when you can determine which rows are partially visible. If you look back at Figure 21-16, you can see that rows 0 through 6 are fully visible, but only a little bit of row 7 is shown.

[3] See http://docs.oracle.com/javase/7/docs/api/javax/swing/JViewport.html.
[4]You can display the value returned by vPort.getClass(),which tells you that the variable refers to an instance of javax.swing. JViewport.



```
rows: 0..23  isValid: 0
rows: 0..23  isValid: 0
rows: 0..7  isValid: 0
rows: 0..23  isValid: 0
rows: 0..23  isValid: 0
rows: 0..7  isValid: 0
rows: 0..7  isValid: 0
rows: 0..7  isValid: 0
rows: 0..7  isValid: 1
```

Figure 21-17. *Sample output of the rowFinder stateChanged(...) method*

Table Alignment in the Viewport

The previous iteration shows that sometimes you try something and realize that it isn't going to work out as well as you had hoped. That's one of the really wonderful things about software. Unlike something like woodworking, software is infinitely malleable. You can change it and then change it right back if you don't like the change! In this next iteration, SecConfigReport_10.py, you'll remove the rowFinder class and add the upDownAction class shown in Listing 21-12.

Listing 21-12. Part 1 of the upDownAction Class from SecConfigReport_10.py

```
25|class upDownAction( AbstractAction ) :
26| def __init__( self, table, keyName ) :
27| self.table = table
28| ks = KeyStroke.getKeyStroke( keyName )
29| self.up = keyName.find( 'UP' ) > -1
30| self.action = action = table.getInputMap(
31| JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT
32| ).get( ks )
33| self.original = table.getActionMap().get( action )
34| self.table.getActionMap().put( action, self )
| ...
181| keys = 'UP,DOWN,PAGE_UP,PAGE_DOWN,ctrl END'.split( ',' )
182| for key in keys :
183| upDownAction( table, key )
```

Don't be confused by the indentation in Listing 21-12. The last three lines (181-

183) are not part of the upDownAction class constructor. They are, in fact, from the run(...) method of the SecConfigReport class. However, it is much easier to include them in this listing than it is to create one just for those three lines. Additionally, since these lines are the only reference to this class, it makes sense to show them here to allow for a more complete understanding of how the class is instantiated.

What does this class do? The hint can be found in lines 181-183. It is here that the class is used. Note how the class constructor uses the table instance and the name of a keystroke. In lines 26-34, you can see how the class constructor uses the table to determine the action currently associated with the specified keystroke. This action is saved in an instance variable (called self.original) and replaces it with the action being constructed.

Listing 21-13 shows the remainder of the upDownAction class, specifically the actionPerformed(...) method, which is invoked when a key-related event is generated. When the user presses any of these keys, the newly constructed Action instance is invoked. It uses the original keystroke action, and then adjusts the viewport to better align the table within the available space.

Without this alignment, the top and/or bottom of the viewport may be somewhere in the middle of the row. So, the top of the viewport could be bisecting the row, and the bottom of the viewport might be showing a similar fraction of the row at the bottom. The role of the upDownAction class is to adjust the viewport to align with the bottom of the last visible row when the keystroke used is moving down. When the direction is up, the alignment of the viewport will be with the top row visible row.

Why is this necessary? The simple answer is that when the table rows have variable heights, the default movement actions don’t align the table rows nicely in the viewport. So, this is all related to the fact that the information you want to display doesn’t fit well in the available horizontal space.

Listing 21-13. Part 2 of the upDownAction Class from SecConfigReport_10.py

```
35| def actionPerformed( self, actionEvent ) :
36| table = self.table
37| self.original.actionPerformed( actionEvent )
38| if self.action == 'selectLastRow' :
```

```
self.original.actionPerformed( actionEvent )
vPort = table.getParent()
rect = vPort.getViewRect()
row = table.getSelectedRow()
if row > -1 :
cRect = table.getCellRect( row, 0, 1 )
rBot = rect.y + rect.height # Bottom of viewPort
cBot = cRect.y + cRect.height # Bottom of cell
if rect.y <= cRect.y and rBot >= cBot :
return
if self.up :
first = table.rowAtPoint( Point( 0, rect.y ) )
cell = table.getCellRect( first, 0, 1 )
diff = rect.y - cell.y
else :
if row > -1 :
cell = table.getCellRect( row, 0, 1 )
else :
last = table.rowAtPoint(
Point(
0,
rect.y + rect.height - 1
)
)
cell = table.getCellRect( last, 0, 1 )
bot = rect.y + rect.height
end = cell.y + cell.height
diff = end - bot
point = vPort.getViewPosition()
vPort.setViewPosition(
Point( point.x, point.y + diff )
)
```

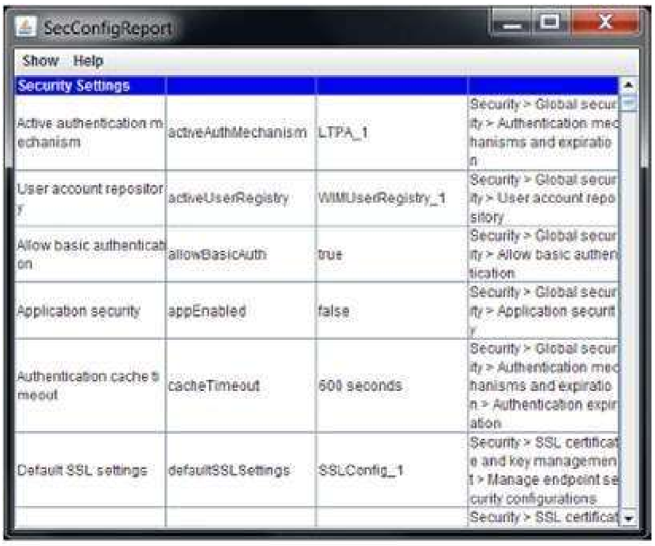
Table Row Filtering

In this next iteration, called SecConfigReport_11.py, you add menu items and row filtering. What’s row filtering? Well, you’ve already seen that the application can identify which rows are section heading rows.

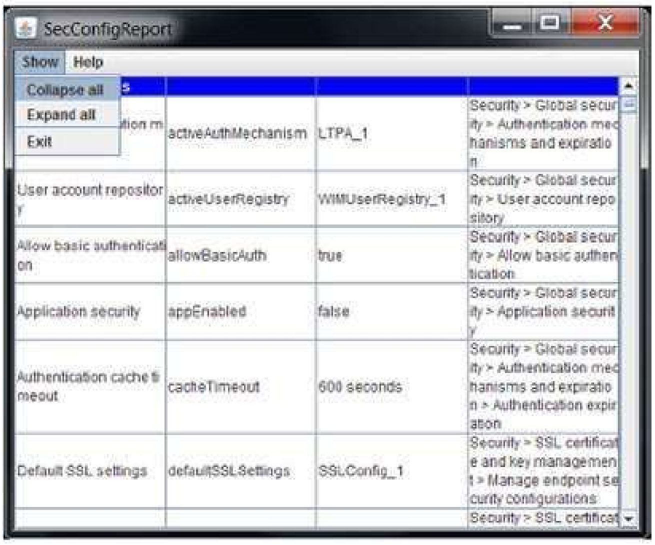
When I was first thinking about this application, I wondered whether there was any easy way to make only specific sections of the report visible. I thought that maybe I could have each section on a CardLayout[5] (see Chapter 5) or on a JTabbedPane.[6] The problem with using a tabbed pane is that there are too many sections. An application with two dozen tabs wouldn't look very good, would it? If you used a CardLayout, you would need some way to specify which section should be displayed.

That's when I remembered row filtering. Row filtering enables your application to decide which table rows to display. What does that mean for the output? Figure 21-18 shows some sample output of this iteration of the application. The first image shows the initial application display. Here you can see the new Show and Help menu items. The second image shows that the Show menu has three sub-menu items, called Collapse All, Expand All, and Exit. The last image shows the result of selecting Collapse All—the row filtering hides the non-section heading table rows.

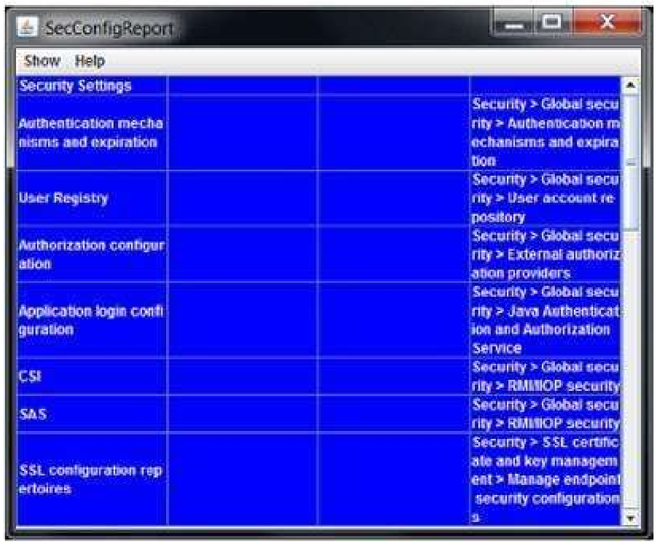
[5]See http://docs.oracle.com/javase/7/docs/api/java/awt/CardLayout.html. [6]See http://docs.oracle.com/javase/7/docs/api/javax/swing/JTabbedPane.html.



Security Settings			
Active authentication mechanism	activeAuthMechanism	LTPA_1	Security > Global security > Authentication mechanisms and expiration
User account repository	activeUserRegistry	WIMUserRegistry_1	Security > Global security > User account repository
Allow basic authentication	allowBasicAuth	true	Security > Global security > Allow basic authentication
Application security	appEnabled	false	Security > Global security > Application security
Authentication cache timeout	cacheTimeout	600 seconds	Security > Global security > Authentication mechanisms and expiration > Authentication expiration
Default SSL settings	defaultSSLSettings	SSLConfig_1	Security > SSL certificate and key management > Manage endpoint security configurations
			Security > SSL certificat



s			
tion m	activeAuthMechanism	LTPA_1	Security > Global security > Authentication mechanisms and expiration
User account repository	activeUserRegistry	WIMUserRegistry_1	Security > Global security > User account repository
Allow basic authentication	allowBasicAuth	true	Security > Global security > Allow basic authentication
Application security	appEnabled	false	Security > Global security > Application security
Authentication cache timeout	cacheTimeout	600 seconds	Security > Global security > Authentication mechanisms and expiration > Authentication expiration
Default SSL settings	defaultSSLSettings	SSLConfig_1	Security > SSL certificate and key management > Manage endpoint security configurations
			Security > SSL certificat



Security Settings			
Authentication mechanisms and expiration			Security > Global security > Authentication mechanisms and expiration
User Registry			Security > Global security > User account repository
Authorization configuration			Security > Global security > External authorization providers
Application login configuration			Security > Global security > Java Authentication and Authorization Service
CSI			Security > Global security > RMI/IIOP security
SAS			Security > Global security > RMI/IIOP security
SSL configuration repertoires			Security > SSL certificate and key management > Manage endpoint security configurations

Figure 21-18. *Output of SecConfigReport_11.py*

Listing 21-14 shows the changes to SecConfigReport_11.py to make this happen. One of the interesting things is how simple the descendent of the abstract RowFilter class[7] is. In fact, as you can see in lines 98-100, the class needs only three lines. However, in order to do this, you need to add two methods to the table model class called reportTable Model—specifically the setVisible() and isVisible() methods shown in lines 109-112. These methods use a new class attribute—the self.visible array—to initialize the constructor on line 104. This array has a Boolean value indicating whether the corresponding row is visible.

Listing 21-14. Row Filtering Changes in SecConfigReport_11.py

```
98|class sectionFilter( RowFilter ) :
99| def include( self, entry ) :
100| return entry.getModel().isVisible( entry.getIdentifier() ) 101|class
reportTableModel( DefaultTableModel ) :
102| def __init__( self, data, headings ) :
103| DefaultTableModel.__init__( self, data, headings ) 104| self.visible = [ 1 ] *
len( data )
| ...
109| def isVisible( self, row ) :
110| return self.visible[ row ]
111| def setVisible( self, row, trueFalse ) :
112| self.visible[ row ] = trueFalse
| ...
189| def collapse( self, event ) :
190| table = self.table
191| model = table.getModel()
192| for row in range( model.getRowCount() ) :
193| model.setVisible(
194| row,
195| model.getValueAt( row, 0 ).startswith( '_' ) 196| )
197| table.getRowSorter().setRowFilter(
198| sectionFilter()
199| )
| ...
202| def expand( self, event ) :
```

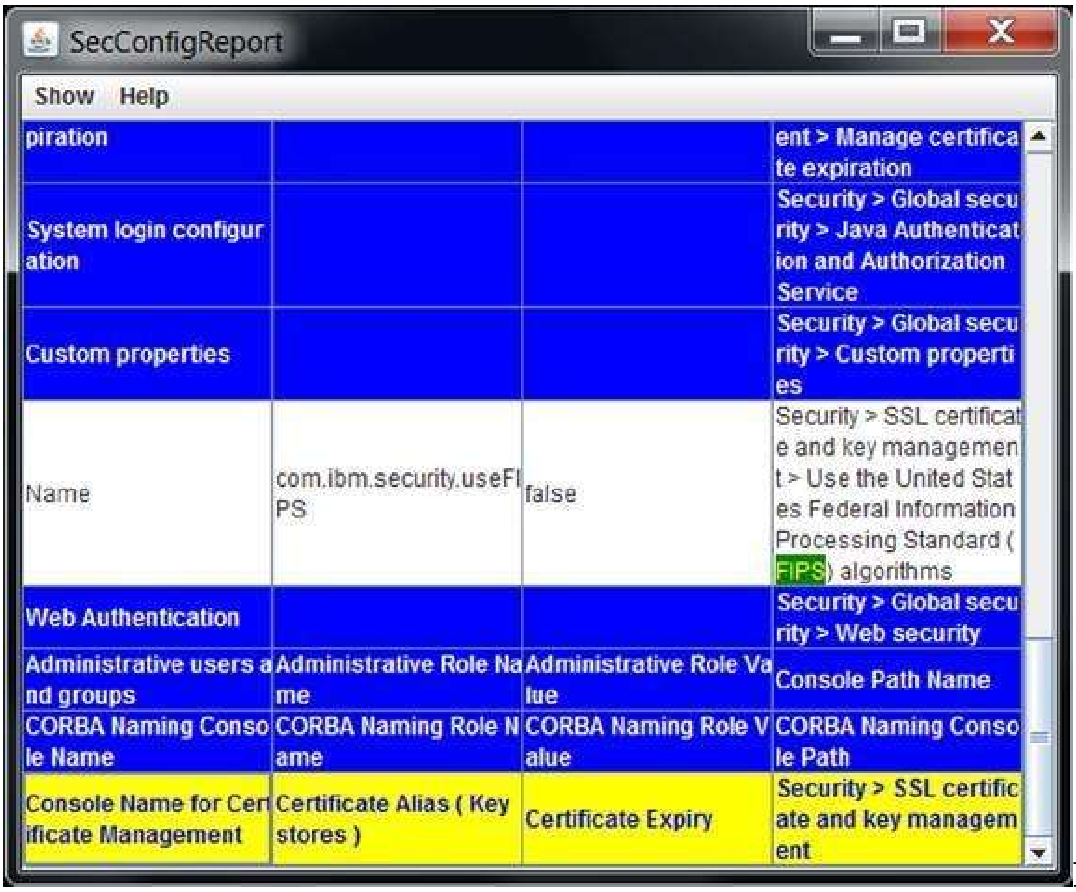
```
table = self.table
model = table.getModel()
for row in range( model.getRowCount() ) :
model.setVisible( row, 1 )
table.getRowSorter().setRowFilter(
sectionFilter()
)
```

The initial implementations of the event handler routines invoked by the menu items collapse the sections by setting the non-section rows visible entry to 0 (false) and expand the sections by setting every row’s visible entry to 1 (true).

[7]See http://docs.oracle.com/javase/7/docs/api/javax/swing/RowFilter.html.

Finding Text

The next iteration, called SecConfigReport_12.py, adds another menu item so the users can specify text to be found and highlighted. This works in conjunction with row filtering so that the user can collapse all rows, find the text of interest, and make those rows visible as well. Figure 21-19 shows the sample output of this new feature.



piration			ent > Manage certificate expiration
System login configuration			Security > Global security > Java Authentication and Authorization Service
Custom properties			Security > Global security > Custom properties
Name	com.ibm.security.useFIPS	false	Security > SSL certificate and key management > Use the United States Federal Information Processing Standard (FIPS) algorithms
Web Authentication			Security > Global security > Web security
Administrative users and groups	Administrative Role Name	Administrative Role Value	Console Path Name
CORBA Naming Console Name	CORBA Naming Role Name	CORBA Naming Role Value	CORBA Naming Console Path
Console Name for Certificate Management	Certificate Alias (Key stores)	Certificate Expiry	Security > SSL certificate and key management

Figure 21-19. *Output of SecConfigReport_12.py*

The output shown in Figure 21-19 was produced using the following steps:
1. Choose Show ➤ Collapse All to hide all of the non-section heading rows.
2. Choose Show ➤ Find. Type FIPS in the input field and press the Enter key.
3. Press Ctrl-End to reposition the view to the bottom of the table.[8]

Listings 21-15 and 21-16 show the majority of the code changes required to add this capability to the script. One choice that made this change particularly easy was to add the findText attribute to the reportTableModel class. With this attribute and some getter and setter methods (getFindText(...) and setFindText(...)), the row filter class (sectionFilter) can easily determine what, if any, text needs to be located in the row to ensure that it will be displayed.
Listing 21-15. Part 1 of the Changes Made to SecConfigReport_12.py

[8]Note that a different color scheme indicates when a section row is selected (as you can see on the last row of the table).

```
45|hilightHTML = '<font bgcolor="green" color="yellow">%s</font>' |...
99|class sectionFilter( RowFilter ) :
100| def include( self, entry ) :
101| model = entry.getModel()
102| result = model.isVisible( entry.getIdentifier() ) 103| findText =
model.getFindText()
104| if findText :
105| for col in range( entry.getValueCount() ) : 106| if entry.getStringValue( col
).find( findText ) > 107| result = 1
108| break
109| return result
110|class reportTableModel( DefaultTableModel ) :
111| def __init__( self, data, headings ) :
| ...
114| self.findText = None
| ...
121| def getFindText( self ) :
122| return self.findText
123| def setFindText( self, text ) :
124| self.findText = text
```

The table cell renderer, shown in Listing 21-16 can use the same getter method to see if the text needs to be matched and highlighted in the cell. The hilightHTML variable shown on line 189 refers to a global format string that is used to highlight the specified text using a simple HTML font tag. The variable assignment is on line 45 in Listing 21-15.

Listing 21-16. Part 2 of the Changes Made to SecConfigReport_12.py

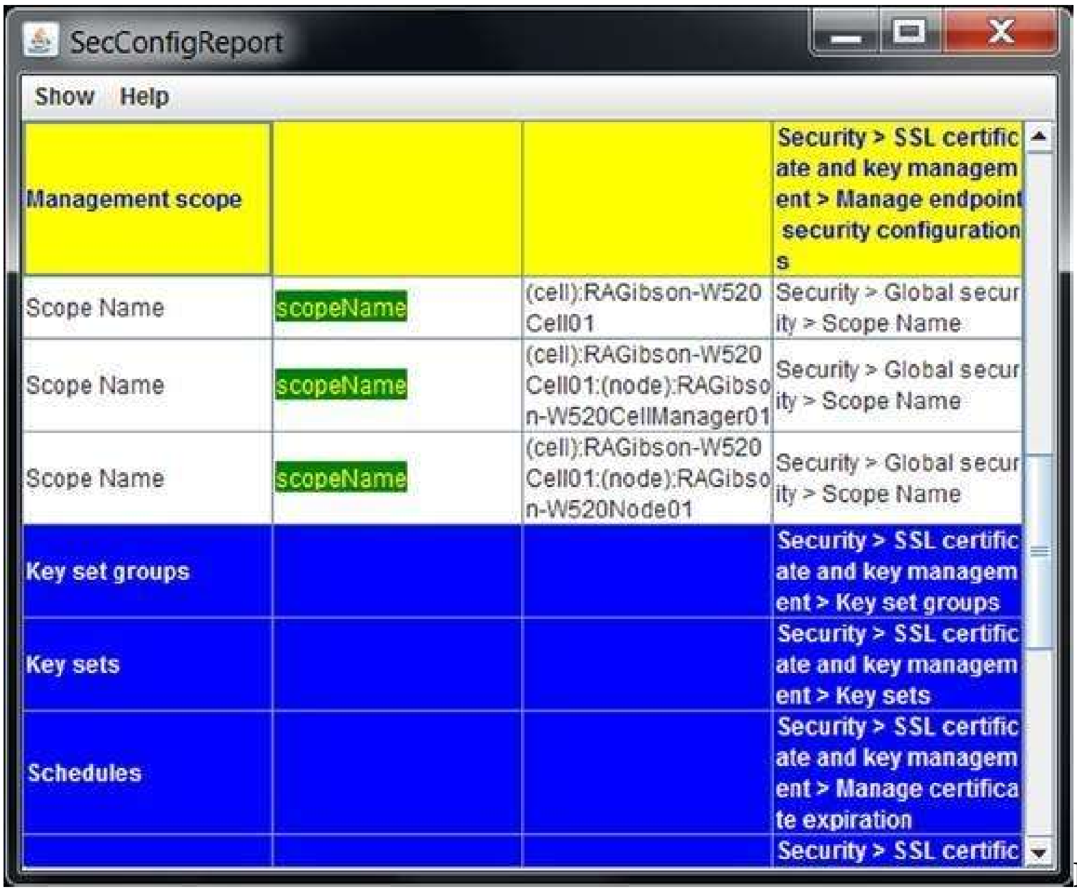
```
127|class reportRenderer( DefaultTableCellRenderer ) : | ...
181| result = result.replace(
182| '>', '&gt;'
183| ).replace( '\n', '<br>' )
184| findText = table.getModel().getFindText()
185| if findText :
186| if result.find( findText ) > -1 :
187| result = result.replace(
188| findText,
189| hilightHTML % findText
)
value = '<html>' + result.replace( ' ', '&nbsp;' ) |...
class SecConfigReport_12( java.lang.Runnable ) : |...
def Find( self, event ) :
result = JOptionPane.showInputDialog(
self.frame, # parentComponent
'Text to be found:' # message text
)
self.table.getModel().setFindText( result )
self.table.getRowSorter().setRowFilter(
sectionFilter()
)
```

This code simply locates instances of the user-specified text and replaces that text with the HTML that highlights the text for the user.

Section Visibility

If you use this iteration for a short time, you will find that being able to collapse and expand all sections is nice, but not quite good enough. Wouldn’t it be great if you could show or hide individual sections simply by double-clicking on the section row?

Figure 21-20 shows a sample image from the latest iteration of the application, SecConfigReport_13.py. In it, you can see that all sections have been collapsed, and the view has been scrolled about half way down the table until the section named Management Scope is visible at the top of the viewport. The image shows the output after finding the rows containing scopeName; three additional rows are now visible.



Management scope			Security > SSL certificate and key management > Manage endpoint security configurations
Scope Name	scopeName	(cell):RAGibson-W520Cell01	Security > Global security > Scope Name
Scope Name	scopeName	(cell):RAGibson-W520Cell01:(node):RAGibson-W520CellManager01	Security > Global security > Scope Name
Scope Name	scopeName	(cell):RAGibson-W520Cell01:(node):RAGibson-W520Node01	Security > Global security > Scope Name
Key set groups			Security > SSL certificate and key management > Key set groups
Key sets			Security > SSL certificate and key management > Key sets
Schedules			Security > SSL certificate and key management > Manage certificate expiration
			Security > SSL certific

Figure 21-20. *Output of SecConfigReport_13.py*

Listings 21-16 through 21-20 show the code changes that provide this capability. Most of the changes are to the reportTableModel class in order to add attributes and methods to identify the sections in the table.
Let's start by taking a look at the modified sectionFilter class, which is shown in Listing 21-17. The only changes made here are on lines 101 through 103. The role of this method is to return true (1) when the row specified by the entry argument should be visible. In previous versions of this class, the row would be visible only if the whole section was visible or if the user specified some text that exists in the current row.

Listing 21-17. Modified sectionFilter Class from SecConfigReport_13.py

```
98|class sectionFilter( RowFilter ) :
99| def include( self, entry ) :
100| model = entry.getModel()
101| row = entry.getIdentifier()
```

```
102| section = model.getSectionNumber( row )
103| result = model.isRowVisible( row ) or
| model.isSectionVisible( section )
104| findText = model.getFindText()
105| if findText :
106| for col in range( entry.getValueCount() ) :
107| if entry.getStringValue( col ).find( findText ) > -1 : 108| result = 1
109| break
110| return result
```

Changes to the include(...) method (see line 103) assume that one section is visible. You can see this by collapsing all of the sections (choose Show ➤ Collapse All) and then double-clicking on one of the visible section headings.

Listing 21-18. The reportTableModel Class from SecConfigReport_13.py

```
111|class reportTableModel( DefaultTableModel ) :
112| def __init__( self, data, headings ) :
113| DefaultTableModel.__init__( self, data, headings )
114| L = len( data )
115| self.visible = [ 1 ] * L
116| self.sectionNumber = [ 0 ] * L
117| self.sections = 0
118| section = -1
119| for i in range( L ) :
120| row = data[ i ]
121| if row[ 0 ].startswith( '_' ) :
122| self.sections += 1
123| section += 1
124| self.sectionNumber[ i ] = section
125| self.sectionVisible = [ 1 ] * self.sections
126| self.findText = None
127| def getColumnClass( self, col ) :
128| return String
129| def getFindText( self ) :
130| return self.findText
131| def getSectionCount( self ) :
132| return self.sections
133| def getSectionNumber( self, row ) :
134| try :
```

```
135| result = self.sectionNumber[ row ] 136| except :
137| result = -1
138| return result
139| def isCellEditable( self, row, col ) :
140| return 0
141| def isSectionVisible( self, sectionNum ) : 142| try :
143| result = self.sectionVisible[ sectionNum ] 144| except :
145| result = 0
146| return result
147| def isRowVisible( self, row ) :
148| return self.visible[ row ]
149| def setFindText( self, text ) :
150| self.findText = text
151| def setSectionVisible( self, sectionNum, trueFalse ) : 152|
self.sectionVisible[ sectionNum ] = trueFalse 153| def setRowVisible( self, row,
trueFalse ) : 154| self.visible[ row ] = trueFalse
```

The changes to the sectionFilter class are not the only ones you need to make; you also have to make some significant changes to the reportTableModel class. This revised class is shown in Listing 21-18. You should be able to see that most of the changes to this class deal with identifying sections and being able to determine the section number for each row in the data.

You also need to add a mouse listener event handler in the application class (SecConfigReport) to detect when the user double-clicks on the table. The mouseClicked event handler method shown in Listing 21-19 is added to each row of the table. It is important to note, however, that it only changes the visibility of a section by checking that the row that was double-clicked is a section heading (see line 241).

Listing 21-19. The clicker(...) Method in SecConfigReport_13.py

```
217|class SecConfigReport_13( java.lang.Runnable ) :
| ...
234| def clicker( self, event ) :
235| if event.getClickCount() == 2 :
236| table = event.getSource()
237| model = table.getModel()
238| view = row = table.getSelectedRow()
```

```
if view > -1 :
row = table.convertRowIndexToModel( view )
if model.getValueAt( row, 0 ).startswith( '_' ) :
sNum = model.getSectionNumber( row )
model.setSectionVisible(
sNum,
not model.isSectionVisible( sNum )
)
else :
sNum = -1
else :
sNum = -1
table.getRowSorter().setRowFilter(
sectionFilter()
)
```

The other significant changes are made to the Find(...) method so that it takes the table model section attributes and methods into account. The modified Find(...) method is shown in Listing 21-20.

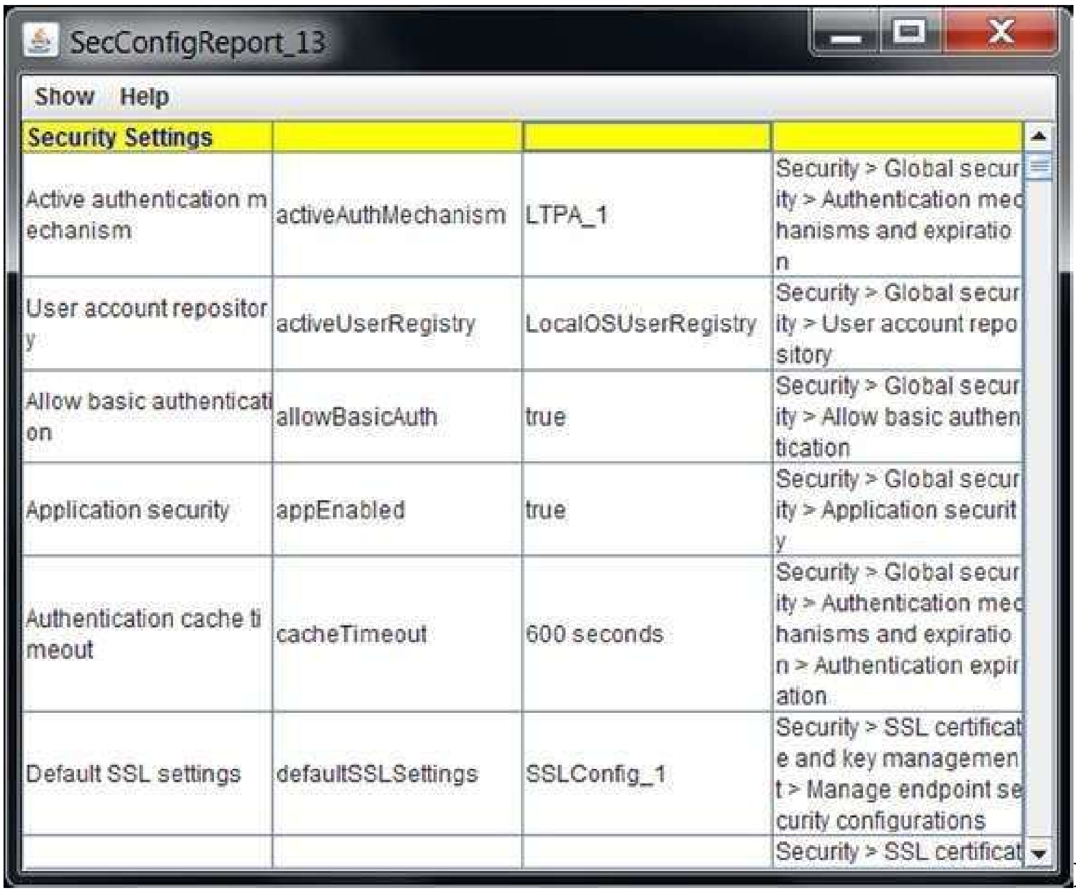
Listing 21-20. Modified Find(...) Method from SecConfigReport_13.py

```
def Find( self, event ) :
table = self.table
cols = table.getColumnModel().getColumnCount()
model = table.getModel()
result = JOptionPane.showInputDialog(
self.frame, # parentComponent
'Text to be found:' # message text
)
model.setFindText( result )
for row in range( model.getRowCount() ) :
visible = model.getValueAt(
row,
0
).startswith( '_' )
if result and not visible :
for col in range( cols ) :
val = model.getValueAt( row, col )
if val.find( result ) > -1 :
visible = 1
```

```
break
model.setRowVisible( row, visible )
table.getRowSorter().setRowFilter(
sectionFilter()
)
```

Does It Work?

One thing that you should have learned by this point is that you always need to test your applications to verify that they act as you expect. To see if the double-click event handler is doing its job, you can use the Show ➤ Collapse All menu selections, and then double-click the visible section heading rows to see if the associated rows become visible. This test appears to validate the expectations. What else can you test? Does it work correctly when you double-click a section heading row after expanding the rows? See Figure 21-21.



Security Settings			
Active authentication mechanism	activeAuthMechanism	LTPA_1	Security > Global security > Authentication mechanisms and expiration
User account repository	activeUserRegistry	LocalOSUserRegistry	Security > Global security > User account repository
Allow basic authentication	allowBasicAuth	true	Security > Global security > Allow basic authentication
Application security	appEnabled	true	Security > Global security > Application security
Authentication cache timeout	cacheTimeout	600 seconds	Security > Global security > Authentication mechanisms and expiration > Authentication expiration
Default SSL settings	defaultSSLSettings	SSLConfig_1	Security > SSL certificate and key management > Manage endpoint security configurations
			Security > SSL certificat

Figure 21-21. *Testing SecConfigReport_13.py*

Unfortunately, no, it doesn't. What's going on? Why doesn't it work correctly? Take another look at the include(...) method in the sectionFilter class in Listing 21-17. Under what conditions will a row be visible? Line 103 shows that in order for a row to be hidden, both isRowVisible(...) and isSectionVisible(...) must return false.

What happens to the mouseClicked event handler method when a section heading row is the target of the event? Which of these model properties is affected? Only the sectionVisible property is affected, so the visible property of each row in the section is unaffected. That's the problem right there. To fix this, the event handler has to change the visible property for each row in the section.

Listing 21-21 shows the modified clicker(...) method event handler. Note how the target row is verified as a section header row. Its new visibility is determined (line 243), set (line 244), and then used to assign the visibility of each row within the section (lines 245 through 249).

Listing 21-21. Modified clicker() Method from SecConfigReport_14.py

```
234| def clicker( self, event ) :
235| if event.getClickCount() == 2 :
236| table = event.getSource()
237| model = table.getModel()
238| view = row = table.getSelectedRow()
239| if view > -1 :
240| row = table.convertRowIndexToModel( view )
241| if model.getValueAt( row, 0 ).startswith( '_' ) :
242| sNum = model.getSectionNumber( row )
243| vis = not model.isSectionVisible( sNum )
244| model.setSectionVisible( sNum, vis )
245| for row in range( row + 1, model.getRowCount() ) : 246| if
model.getSectionNumber( row ) == sNum : 247| model.setRowVisible( row, vis
) 248| else :
249| break
250| else :
251| sNum = -1
252| else :
253| sNum = -1
254| table.getRowSorter().setRowFilter(
sectionFilter()
)
```

Please keep in mind that the event handler code should be done quickly. You don’t want the routine to delay the update. If you test this version of the application, you’ll see that the event handler doesn’t take too long to complete.

Progress Indicator

In relation to the possibility of a delay in the event handler, I wondered also about the delay that was occurring when the script begins to execute. A bit of testing shows that there is a non-trivial amount of time required to execute the AdminTask.generateSecConfigReport() method. See for yourself. Start an interactive wsadmin session[9] and, when the command prompt is displayed, execute the generateSecConfigReport() method.[10] Take note of how long it takes to complete.

Based on this improved understanding of what was occurring, I tried a few things to display some kind of progress indicator to show the user that something was happening. Unfortunately, I didn’t have much luck. My attempt to use a SwingWorker task to perform the AdminTask method call and display an indeterminate progress bar in a dialog box didn’t work for some reason. I decided to use a different approach.

Listing 21-22 shows the changes made to this next iteration of the script. As you can see in lines 66-70, a trivial SwingWorker class performs the call to the AdminTask method on a background thread and makes the result available via a simple getter method.

Listing 21-22. Modified run(...) Method from SecConfigReport_15.py

```
66|class reportTask( SwingWorker ) :
67| def doInBackground( self ) :
68| self.results = AdminTask.generateSecConfigReport( | ).splitlines()[ 2: ]
69| def getResults( self ) :
70| return self.results
|...
```

[9]wsadmin -conntype none -lang jython [10]text =

AdminTask.generateSecConfigReport()

```
229|class SecConfigReport_15( java.lang.Runnable ) :
|...
364| def run( self ) :
365| try :
366| task = reportTask()
367| task.execute()
368| chars = r'-\|/'
369| char = 0
370| while not task.isDone() :
371| print '\b%s\b' % chars[ char ],
372| sleep( 0.25 )
373| char = ( char + 1 ) % len( chars )
374| print '\b \b',
375| info = task.getResults()
376| except :
377| print '\nError: %s\nvalue: %s' % sys.exc_info()[ :2 ]
378| sys.exit()
|...
```

Lines 366 and 367 show how this task instance is created and executed. You need to save a reference to the task instance object in order to call its getResults() method once the task is complete.[11] Lines 368-373 display a trivial character-oriented indicator to show that the AdminTask method call has not yet completed. Once the task is complete, its results are retrieved, and the application can get on with its purpose.

Is this application perfect? Absolutely not. As indicated earlier, it is merely a proof of concept (PoC). If you wanted to enhance it and use it in a more permanent arrangement, you could base those decisions on what you can see and do with this script. You could also use it to test possible enhancements or improvements.

Summary

The purpose of this chapter is twofold. First it is intended to show how easily you can turn a report into an interactive graphical user application. It is also intended to show that by taking small steps you can more easily verify that each iteration of the script works as it should. If it isn't, you can more easily determine the source of the problem and correct it.wsadmin

[11]Note the use of the isDone() method on line 370, which is used to determination when the background task is complete. **Chapter 22**

WASports: A WebSphere Port Application

One of the first topics that grabbed my imagination when I started thinking about graphical wsadmin scripts was the possibility of displaying and managing all of the TCP/IP port numbers being used by a WebSphere Application Server cell. Trying to use the administration console to view the port numbers being used by all of the servers in a cell can be frustrating and tedious. I wanted an iterative graphical application that can be used to quickly and easily understand which application servers exist in the cell, and use a tree structure to show the hierarchical relationship between these servers. Additionally I wanted to be able to show the port numbers being used by each of these servers. This chapter shows how to build this application using the same type of iterative approach used previously.

Using the Administration Console

What does it take to use the Administration console to view the port numbers currently being used by a managed application server? The following steps are needed to view and change just one TCP/IP port number for an existing application server using the administrative console.

1. If the Deployment Manager is active, skip to Step 3.
2. Use the startManager command to start the Deployment Manager.

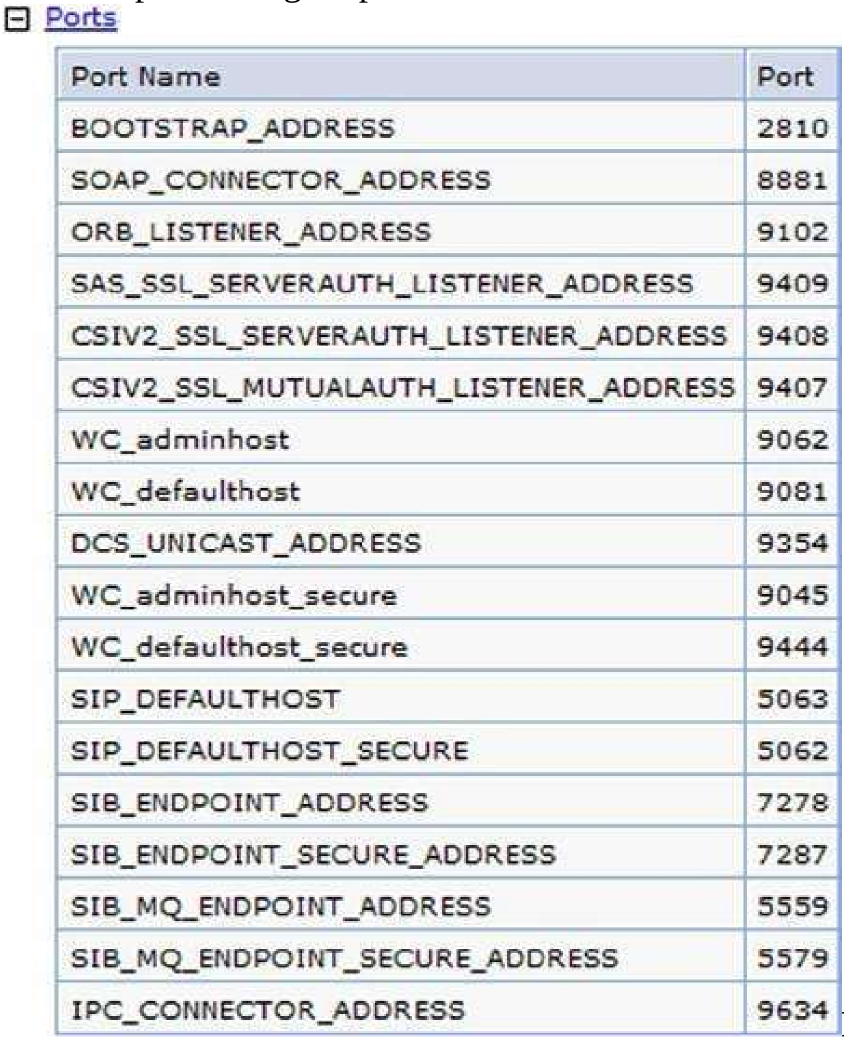
3. Use a browser to access the Administration console associated with the Deployment Manager. Enter the appropriate username and password to view the desired information.

4. Select and expand the Servers section in the left frame.
5. Select and expand the Server Types section.
6. Select the WebSphere Application Servers link.
7. Select the appropriate server link (such as server1).
8. Expand the Ports section under the Communications heading.

9. Use the Details button or the Ports link to show a table of the named endpoints and the associated port number values used by the specified application server.

10. Select the desired endpoint name to be modified, such as BOOTSTRAP_ADDRESS.
11. Modify the Port input field to identify the new port number value.
12. Select the Apply or OK button.
13. Select the Save link to update the master configuration.

14. After all the required changes have been made, stop and restart the server to use the modified port numbers.
Figure 22-1 shows a simple example of a table of the port numbers being used by a single application server. This corresponds to the kind of table that is the result of performing Step 8.


⊟ Ports

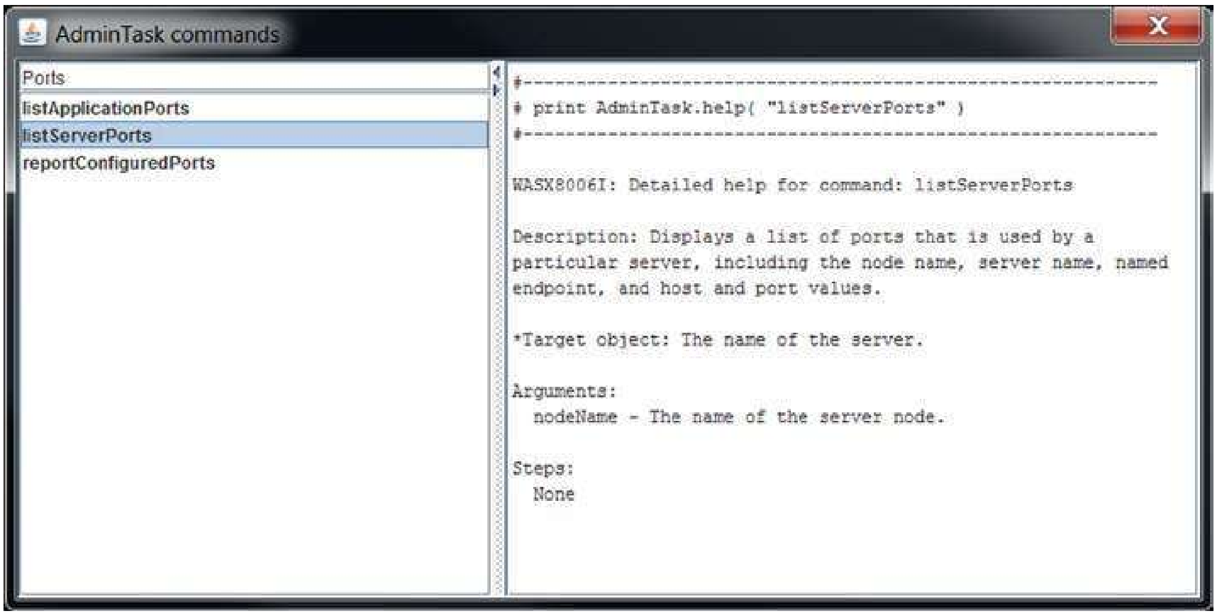
Port Name	Port
BOOTSTRAP_ADDRESS	2810
SOAP_CONNECTOR_ADDRESS	8881
ORB_LISTENER_ADDRESS	9102
SAS_SSL_SERVERAUTH_LISTENER_ADDRESS	9409
CSIV2_SSL_SERVERAUTH_LISTENER_ADDRESS	9408
CSIV2_SSL_MUTUALAUTH_LISTENER_ADDRESS	9407
WC_adminhost	9062
WC_defaulthost	9081
DCS_UNICAST_ADDRESS	9354
WC_adminhost_secure	9045
WC_defaulthost_secure	9444
SIP_DEFAULTHOST	5063
SIP_DEFAULTHOST_SECURE	5062
SIB_ENDPOINT_ADDRESS	7278
SIB_ENDPOINT_SECURE_ADDRESS	7287
SIB_MQ_ENDPOINT_ADDRESS	5559
SIB_MQ_ENDPOINT_SECURE_ADDRESS	5579
IPC_CONNECTOR_ADDRESS	9634

Figure 22-1. *Sample list of application server port settings*

If you view and modify the port numbers being used by the Deployment Manager or the node agents, you need to start by selecting and expanding the System Administration section instead of performing Steps 4 and 5. All this is used to show the number of steps needed to view and modify the port numbers being used by an application server using the administration console.

The AdminTask.listServerPorts( ) Method

Using the WAShelp.py script from Chapter 20, you can search for any scripting object methods that identify the ports being used by a server. Figure 22-2 shows what happens when you search for AdminTask methods containing the word “Ports” anywhere in the method name.



```
#---------------------------------------------------------------
# print AdminTask.help( "listServerPorts" )
#---------------------------------------------------------------

WASX8006I: Detailed help for command: listServerPorts

Description: Displays a list of ports that is used by a
particular server, including the node name, server name, named
endpoint, and host and port values.

*Target object: The name of the server.

Arguments:
  nodeName - The name of the server node.

Steps:
  None
```

Figure 22-2. *AdminTask methods with “Ports” in the name*

The listServerPorts(...) method looks really promising. Let’s see what it produces for the dmgr server.[1] The result is a string list identifying the port numbers configured for the specified application server. Unfortunately, the result of calling this method isn’t as useful as you might hope it would be. Listing 22-1 shows some sample output from this method. Imagine the processing required to use this information. Few people would think the processing required to parse this output worth the effort. This is especially true since the information is more easily available using other techniques.

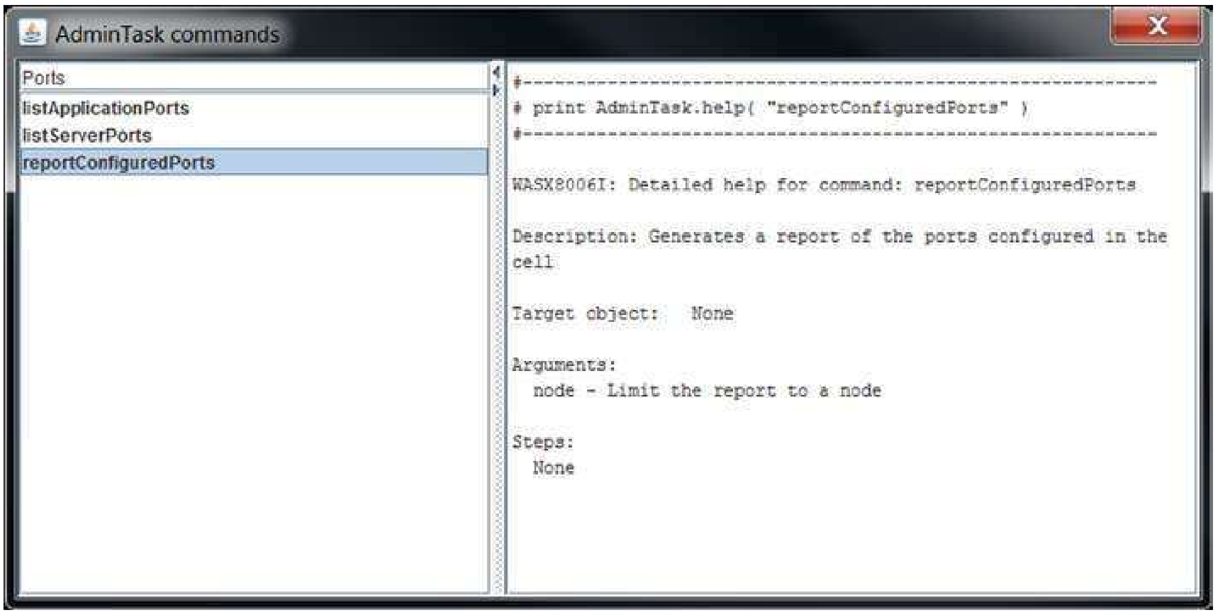
Listing 22-1. Sample AdminTask.listServerPorts(...) Output

```
wsadmin>print AdminTask.listServerPorts( 'dmgr' )
[[IPC_CONNECTOR_ADDRESS [[[host ${LOCALHOST_NAME}] [node
RAGibson
-W520CellManager01] [server dmgr] [port 9632] ]]] ]
[[CSIV2_SSL_SERVERAUTH_LISTENER_ADDRESS [[[host RAGibson-
W520.ral
eigh.ibm.com] [node RAGibson-W520CellManager01] [server dmgr] [po
rt 9403] ]]] ]
[[WC_adminhost [[[host *] [node RAGibson-W520CellManager01] [serv
er dmgr] [port 9060] ]]] ]
[[DataPowerMgr_inbound_secure [[[host *] [node RAGibson-W520CellM
anager01] [server dmgr] [port 5555] ]]] ]
[[DCS_UNICAST_ADDRESS [[[host *] [node RAGibson-
W520CellManager01
] [server dmgr] [port 9352] ]]] ]
[[BOOTSTRAP_ADDRESS [[[host RAGibson-W520.raleigh.ibm.com] [node
RAGibson-W520CellManager01] [server dmgr] [port 9809] ]]] ]
[[SAS_SSL_SERVERAUTH_LISTENER_ADDRESS [[[host RAGibson-
W520.ralei
gh.ibm.com] [node RAGibson-W520CellManager01] [server dmgr] [port

9401] ]]] ]
[[SOAP_CONNECTOR_ADDRESS [[[host RAGibson-W520.raleigh.ibm.com]
[ node RAGibson-W520CellManager01] [server dmgr] [port 8879] ]]] ]
[[CELL_DISCOVERY_ADDRESS [[[host RAGibson-W520.raleigh.ibm.com] [
node RAGibson-W520CellManager01] [server dmgr] [port 7277] ]]] ]
[[ORB_LISTENER_ADDRESS [[[host RAGibson-W520.raleigh.ibm.com] [no
de RAGibson-W520CellManager01] [server dmgr] [port 9100] ]]] ]
[[CSIV2_SSL_MUTUALAUTH_LISTENER_ADDRESS [[[host RAGibson-
W520.ral eigh.ibm.com] [node RAGibson-W520CellManager01] [server dmgr]
[po rt 9402] ]]] ]
[[WC_adminhost_secure [[[host *] [node RAGibson-W520CellManager01 ]
[server dmgr] [port 9043] ]]] ]
wsadmin>
```

[1]Note that if the server name is not unique, a qualifying -nodename parameter must be specified.

The AdminTask.reportConfiguredPorts( ) Method

The other interesting AdminTask method listed by the WAShelp script is the reportConfiguredPorts(...) method. Figure 22-3 shows the help text for this method.



```
#---------------------------------------------------------------
# print AdminTask.help( "reportConfiguredPorts" )
#---------------------------------------------------------------

WASX8006I: Detailed help for command: reportConfiguredPorts

Description: Generates a report of the ports configured in the
cell

Target object:   None

Arguments:
  node - Limit the report to a node

Steps:
  None
```

Figure 22-3. *Help text for AdminTask.reportConfiguredPorts(...) method*

The description of this method is quite promising. The real question though is what kind of output is generated by this method? Listing 22-2 shows the initial output generated by a call to the AdminTask.reportConfiguredPorts(...) method.

Listing 22-2. Initial Output of reportConfiguredPorts(...) Method Call

```
wsadmin>print AdminTask.reportConfiguredPorts() Ports configured in cell
RAGibson-W520Cell01

Node RAGibson-W520CellManager01 / Server dmgr
RAGibson-W520.raleigh.ibm.com:7277 CELL_DISCOVERY_ADDRESS
RAGibson-W520.raleigh.ibm.com:9809 BOOTSTRAP_ADDRESS
${LOCALHOST_NAME}:9632 IPC_CONNECTOR_ADDRESS
RAGibson-W520.raleigh.ibm.com:8879 SOAP_CONNECTOR_ADDRESS
RAGibson-W520.raleigh.ibm.com:9100 ORB_LISTENER_ADDRESS
RAGibson-W520.raleigh.ibm.com:9401 SAS_SSL_SERVERAUTH_LIST...
RAGibson-W520.raleigh.ibm.com:9402 CSIV2_SSL_MUTUALAUTH_LI...
RAGibson-W520.raleigh.ibm.com:9403 CSIV2_SSL_SERVERAUTH_LI...
*:9060 WC_adminhost
*:9043 WC_adminhost_secure
```

```
*:9352 DCS_UNICAST_ADDRESS
*:5555 DataPowerMgr_inbound_secure

...
```

This certainly seems to be much easier to read and understand from the human perspective. However, some challenges remain if you intend to process this kind of result in your scripts.

Using AdminConfig Methods

In Chapter 11, you saw how easy it was to produce a tree hierarchy representing the cell.[2] Let's see what it takes to produce information about the nodes, servers, and their configured ports using a non-graphical script. Listing 22-3 shows most of the main routine needed to do just that.[3]

Listing 22-3. ListPorts Routine from the ListPorts.py Script[4]

```
def ListPorts() :
gAV = getAttributeValue # For line shortening purposes
names = {}
nodes = 0
for node in AdminConfig.list( 'Node' ).splitlines() :
nodes += 1
names[ 'nodeName' ] = gAV( node, 'name' )
names[ 'profName' ] = profileName( node )
SEs = AdminConfig.list( 'ServerEntry', node )
servers = 0
for se in SEs.splitlines() :
servers += 1
names[ 'servName' ] = gAV( se, 'serverName' )
names[ 'hosts' ] = ', '.join( getHostnames( se ) )
print formatString % names
data = []
NEPs = AdminConfig.list( 'NamedEndPoint', se )
for nep in NEPs.splitlines() :
name = gAV( nep, 'endPointName' )
epId = gAV( nep, 'endPoint' )
port = gAV( epId, 'port' )
data.append( ( port, name ) )
data.sort( lambda a, b : cmp( a[ 1 ], b[ 1 ] ) )
for port, name in data :
```

```
print '%5d | %s' % ( port, name )
print
```

[2] For example, see the code\Chap_11\Tree4.py sample script.
[3]The complete script can be found in the code\Chap_22\ListPorts.py file.
[4]Remember that many of the scripts listed in the book are written to fit within the available horizontal space.

What does the output of this script look like? Listing 22-4 shows the initial portion of the output generated when this script was executed using a WebSphere Application Server V 7.0 environment on my local machine.
Listing 22-4. Sample Output of the ListPorts.py Script

```
Profile name: Dmgr01
Host name(s): RAGibson-W520.raleigh.ibm.com Node name: RAGibson-
W520CellManager01

Server name: dmgr

Port | EndPoint Name
------+-------------
9809 | BOOTSTRAP_ADDRESS
7277 | CELL_DISCOVERY_ADDRESS
9402 | CSIV2_SSL_MUTUALAUTH_LISTENER_ADDRESS
9403 | CSIV2_SSL_SERVERAUTH_LISTENER_ADDRESS
9352 | DCS_UNICAST_ADDRESS
5555 | DataPowerMgr_inbound_secure
9632 | IPC_CONNECTOR_ADDRESS
9100 | ORB_LISTENER_ADDRESS
9401 | SAS_SSL_SERVERAUTH_LISTENER_ADDRESS
8879 | SOAP_CONNECTOR_ADDRESS
9060 | WC_adminhost
9043 | WC_adminhost_secure
```

The output produced by this simple script has some nice properties including the ease of reading and understanding the information. It also corresponds closely with the way in which information is displayed on the Administration console, as shown in Figure 22-1.

Step 0: Creating a WASports Application

Let’s start by creating a simple empty frame application. I choose to call this step 0 because using an existing script template is so trivial. The code in Listing 22-5 should be very familiar to you by now, so I won’t bother to elaborate. The only part that might not be familiar to you is in lines 18 and 19, where a JDesktopPane instance is created and used as the frame content pane. This is discussed in more detail in Chapter 19.

Listing 22-5. WASports Class from the WASports_00.py Script

```
6|class WASports_00( java.lang.Runnable ) :
7| def run( self ) :
8| screenSize = Toolkit.getDefaultToolkit().getScreenSize()
9| w = screenSize.width >> 1 # Use 1/2 screen width 10| h = screenSize.height
>> 1 # and 1/2 screen height 11| x = ( screenSize.width - w ) >> 1 # Top left
corner 12| y = ( screenSize.height - h ) >> 1
13| frame = self.frame = JFrame(
14| 'WASports_00',
15| bounds = ( x, y, w, h ),
16| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 17| )
18| desktop = JDesktopPane()
19| frame.setContentPane( desktop )
20| frame.setVisible( 1 )
```

Step 1: Adding an Empty Internal Frame

Next you need to add a little code to create an empty internal frame to the application desktop. Listing 22-6 shows how this is done in the next iteration of the WASports script (in WASports_01.py).

Listing 22-6. Defining and Using an InternalFrame Class

```
9|class InternalFrame( JInternalFrame ) :
10| def __init__( self, title, size, location, closable = 0 ) : 11|
JInternalFrame.__init__(
12| self,
13| title,
14| resizable = 1,
15| closable = closable,
16| maximizable = 1,
17| iconifiable = 1,
18| size = size
19| )
20| self.setLocation( location )
21| self.setVisible( 1 )
22|class WASports_01( java.lang.Runnable ) : 23| def run( self ) :

| ...
34| desktop = JDesktopPane()
35| internal = InternalFrame(
36| 'InternalFrame',
37| size = Dimension( w >> 1, h >> 1 ),
38| location = Point( 5, 5 )
39| )
40| desktop.add( internal )
41| frame.setContentPane( desktop )
42| frame.setVisible( 1 )
```

Step 2: Adding an Empty JSplitPane to the Internal Frame

Next you need to add an empty split pane to the internal frame. Listing 22-7 shows how easily this can be done. The code from this listing is from the WASports_02.py script file. The statement in line 30 is used to position the divider in the middle of the internal frame.

Listing 22-7. JSplitPane Code Added to WASports_02.py

```
11|class InternalFrame( JInternalFrame ) :
12| def __init__( self, title, size, location, closable = 0 ) : | ...
22| self.setLocation( location )
23| pane = self.add(
24| JSplitPane(
25| JSplitPane.HORIZONTAL_SPLIT,
26| JLabel( 'Left' ),
27| JLabel( 'Right' )
28| )
29| )
30| pane.setDividerLocation( size.width >> 1 )
```

```
31| self.setVisible( 1 )
```

The JLabel instances in this example are simple placeholders for other kinds of Swing components.

Step 3: Adding a Cell Hierarchy Tree to the JSplitPane

Listing 22-8 shows that very little new code is needed in the WASports_03.py script file in order to create a cell hierarchy tree on the left portion of the split pane. The cellTree(...) method (lines 40-54) contains code very similar to Listing 22-3.

Listing 22-8. JTree Code Added to WASports_03.py

```
15|class InternalFrame( JInternalFrame ) :
16| def __init__( self, title, size, location, closable = 0 ) : | ...
26| self.setLocation( location )
27| tree = self.cellTree()
28| tree.getSelectionModel().setSelectionMode(
29| TreeSelectionModel.SINGLE_TREE_SELECTION
30| )
31| pane = self.add(
32| JSplitPane(
33| JSplitPane.HORIZONTAL_SPLIT,
34| JScrollPane( tree ),
35| JLabel( 'Right' )
36| )
37| )
38| pane.setDividerLocation( size.width >> 1 )
39| self.setVisible( 1 )
40| def cellTree( self ) :
41| cell = AdminConfig.list( 'Cell' )
42| root = DefaultMutableTreeNode( self.getName( cell ) )
43| for node in AdminConfig.list( 'Node' ).splitlines() :
44| here = DefaultMutableTreeNode(
45| self.getName( node )
46| )
47| servers = AdminConfig.list( 'Server', node )
48| for server in servers.splitlines() :
```

```
leaf = DefaultMutableTreeNode(
self.getName( server )
)
here.add( leaf )
root.add( here )
return JTree( root )
def getName( self, configId ) :
return AdminConfig.showAttribute( configId, 'name' )
```

It’s probably about time to see some sample output of this iteration of the WASports script. Figure 22-4 shows that the WASports application is starting to take shape. The application frame and internal frame can be moved, resized, and so on, as the user desires. It also limits the number of tree nodes that may be selected at one time.

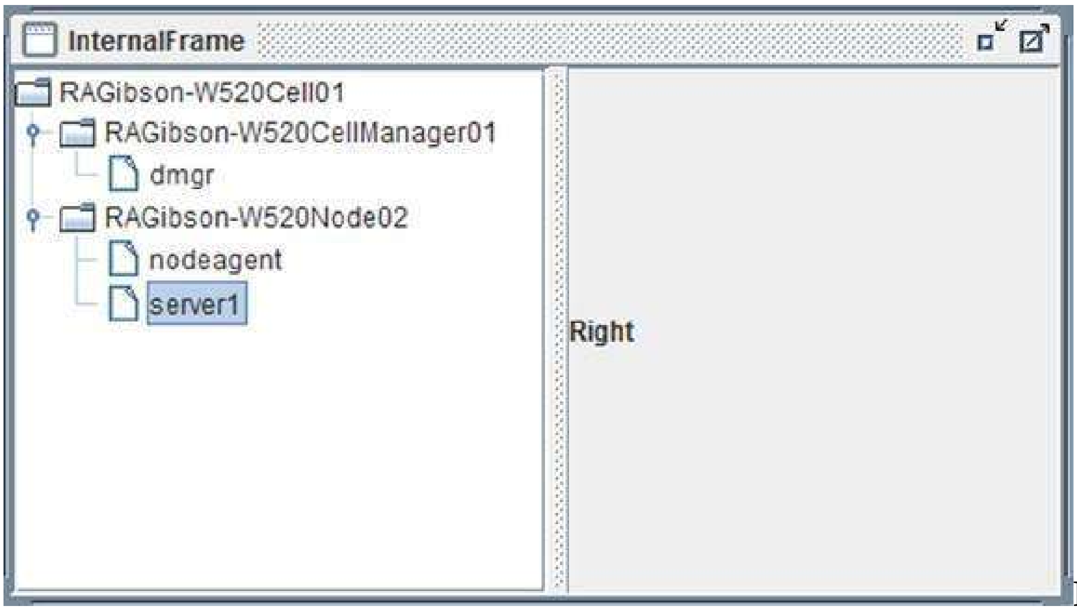


Figure 22-4. *Sample output of the WASports_03.py script*

Step 4: Updating the Right Pane

The next step is to update the other side of the split pane based on the user selections of the tree nodes. This requires a bit more work. In fact, if you simply look at the “lines of code” (ignoring comments and blank lines), this version of the WASports script is about half as large as the previous one. However, it still isn’t too much of a difference because it only requires about 125 lines of code to provide this kind of functionality. What “big changes” are needed to do this?

Listing 22-9 shows the cellTSL class that is used by the script to react to tree selection events.

Listing 22-9. The cellTSL Class from WASports_04.py[5]

```
class cellTSL( TreeSelectionListener ) :
def __init__( self, tree, pane, name2cfgId ) :
self.tree = tree
self.pane = pane
self.name2cfgId = name2cfgId
def valueChanged( self, tse ) :
format = (
'<html>&nbsp;&nbsp;node:&nbsp;%s<br/>' +
'isLeaf:&nbsp;%s<br/>parent:&nbsp;%s'
)
pane = self.pane
node = self.tree.getLastSelectedPathComponent()
if node :
text = format % (
node,
[ 'No', 'Yes' ][ node.isLeaf() ],
node.getParent()
)
if node.isLeaf() :
key = ( str( node.getParent() ), str( node ) )
else :
key = node.toString()
if self.name2cfgId.has_key( key ) :
text += '<br/><br/>%s' % self.name2cfgId[ key ]
else :
text += '<br/><br/> key missing: %s' % key
else :
text = 'Nothing selected'
pane.setText( text )
```

[5]Where TSL is an acronym for TreeSelectionListener.

When a cellTSL object is instantiated, the caller must provide references to:
• The JTree instance being monitored
• The pane being updated
• A dictionary containing information to be used to update the pane

The first two parameters are likely to be obvious, but the last is less likely to be so. This dictionary is used in lines 38 and 39 simply to demonstrate how the selection can be used to retrieve the associated configuration ID from the dictionary. For this example, this configuration ID is part of the information that is displayed in the right pane when a tree node is selected.

Listing 22-10. Tree and Split Pane Creation Code from WASports_04.py

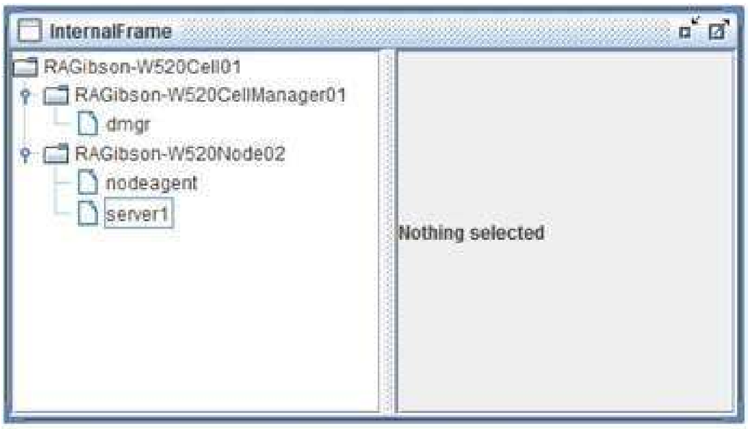
```
...
self.setLocation( location )
tree, self.name2cfgId = self.cellTree()
tree.getSelectionModel().setSelectionMode(
TreeSelectionModel.SINGLE_TREE_SELECTION
)
self.status = JLabel( 'Right' )
tree.addTreeSelectionListener(
cellTSL(
tree,
self.status,
self.name2cfgId
)
)
pane = self.add(
JSplitPane(
JSplitPane.HORIZONTAL_SPLIT,
JScrollPane( tree ),
JScrollPane( self.status )
)
)
```

Listing 22-10 shows the modification to the WASports_04.py script to instantiate the TreeSelectionListener for the cell hierarchy tree. It's important to note that the constructor call requires a dictionary as the third argument. Listing 22-11 shows the modifications that were made to the cellTree() method in order to build and return this dictionary. It is important to note that the dictionary index is simple for non-leaf tree nodes (the root and all of the nodes corresponding to WebSphere node names). It is only the leaf nodes that use a tuple as an index. Why is this? Because the leaf nodes correspond to the individual application servers in the cell, the names of which do not have to be unique. The server names only have to be unique within a node. The dictionary entries for the individual servers use a tuple composed of the node name and the server name (see line 93).

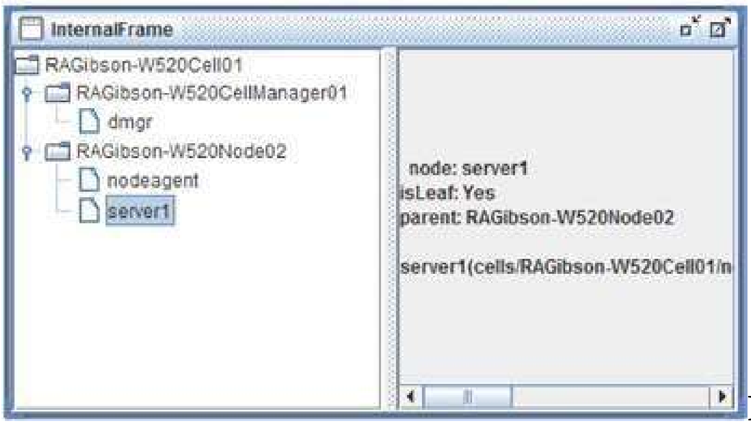
Listing 22-11. Modified cellTree() Method

```
78| def cellTree( self ) :
79| cell = AdminConfig.list( 'Cell' )
80| cellName = self.getName( cell )
81| root = DefaultMutableTreeNode( cellName )
82| result = { cellName : cell }
83| for node in AdminConfig.list( 'Node' ).splitlines() :
84| nodeName = self.getName( node )
85| here = DefaultMutableTreeNode(
86| nodeName
87| )
88| result[ nodeName ] = node
89| servers = AdminConfig.list( 'Server', node )
90| for server in servers.splitlines() :
91| name = self.getName( server )
92| leaf = DefaultMutableTreeNode( name )
93| result[ ( nodeName, name ) ] = server
94| here.add( leaf )
95| root.add( here )
96| return JTree( root ), result
```

What does this mean for the application? Figure 22-5 shows how the right pane is updated based on user selections on the tree in the left pane. The second image shows the result of selecting a server. Note how the bottom line of the right pane is the configuration ID for the selected server.



Nothing selected

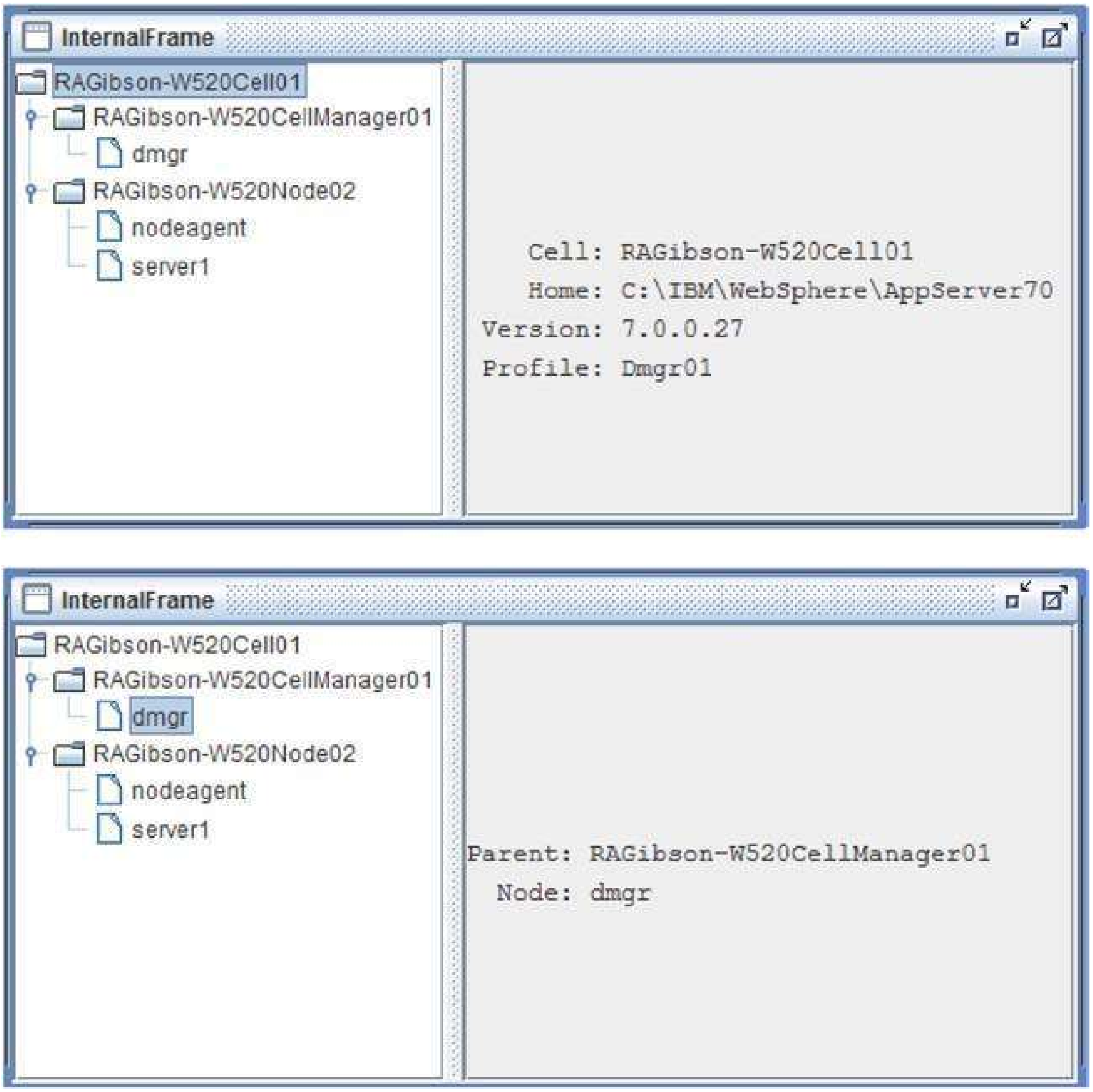


```
 node: server1
isLeaf: Yes
parent: RAGibson-W520Node02

server1(cells/RAGibson-W520Cell01/n
```

Figure 22-5. *Sample output of the WASports_04.py script*

Step 5: Displaying Cell and Node Details

A reasonable next step is to see what you can do to improve the information displayed in the right panel when the cell and node (non-leaf) tree items are selected. Figure 22-6 has some sample output from the next iteration of this script. It shows the different kinds of information displayed depending on the various types of tree nodes that can be selected. The first shows what is displayed when the cell node is selected, the second shows the slightly different information displayed when a WebSphere node is selected, and the last shows a minimum amount of information when a server entry is selected.



```
RAGibson-W520Cell01
  RAGibson-W520CellManager01
    dmgr
  RAGibson-W520Node02
    nodeagent
    server1
```

```
   Cell: RAGibson-W520Cell01
   Home: C:\IBM\WebSphere\AppServer70
Version: 7.0.0.27
Profile: Dmgr01
```

```
RAGibson-W520Cell01
  RAGibson-W520CellManager01
    dmgr
  RAGibson-W520Node02
    nodeagent
    server1
```

```
Parent: RAGibson-W520CellManager01
  Node: dmgr
```

Figure 22-6. *Sample output of the WASports_05.py script*

Utility Routines

At first glance, it might appear simple to determine the values being displayed in the images in Figure 22-6. Unfortunately, it takes a bit more effort and code than you might imagine. For example, Listing 22-12 includes the utility routines needed to obtain some of this information.

Listing 22-12. Part 1 of the WASports_05.py Utility Routines

```
39|def findScopedTypes( Type, value, scope = None, attr = None ) :
```

```
if not attr :
attr = 'name'
return [
x for x in AdminConfig.list( Type, scope ).splitlines()
if getAttributeValue( x, attr ) == value
]
def getAttributeValue( cfgId, attr ) :
return AdminConfig.showAttribute( cfgId, attr )
def getIPaddresses( hostnames ) :
result = []
for hostname in hostnames :
try :
addr = gethostbyname( hostname )
if addr not in result :
result.extend( addr.split( ',' ) )
except :
pass
return result
def getHostnames( nodeName, serverName ) :
exclude = [
'*',
'localhost',
'${LOCALHOST_NAME}',
'232.133.104.73',
'ff01::1'
]
result = []
node = findScopedTypes( 'Node', nodeName )[ 0 ]
server = findScopedTypes(
'ServerEntry',
serverName,
node,
'serverName'
)[ 0 ]
NEPs = AdminConfig.list( 'NamedEndPoint', server )
for nep in NEPs.splitlines() :
epId = getAttributeValue( nep, 'endPoint' )
host = getAttributeValue( epId, 'host' )
if host not in exclude :
result.append( host )
exclude.append( host )
return result
```

Table 22-1 describes each of the routines found in Listing 22-12. It is interesting to see how much of these routines use WebSphere specific objects and attribute values.

Table 22-1. *Part 1 of the WASports_05.py Utility Routines, Explained*

Lines	**Description**
39–45	The findScopedTypes(...) routine is used to return a list of configuration IDs for configuration objects of the specified Type that have a particular attribute value.
46–47	The getAttributeValue(...) routine is simply a technique used to shorten lines that would normally call the Adminconfig.showAttribute(...) method.
48–57	The getIPaddresses(...) routine returns a list IP addresses for the specified hostname.
58–81	The getHostnames(...) routine returns a list of unique hostnames referenced by the endPoint configuration objects on the specified server. **Note:** Since server names are only guaranteed to be unique within a node, the node name also needs to be provided.

Listing 22-13 contains some additional utility routines, all of which have names beginning with “WAS” to indicate how specific they are to WebSphere Application Server product information. In fact, most of them use the WebSphere product variables in order to determine the value to be returned.

Listing 22-13. Part 2 of the WASports_05.py Utility Routines

```
82|def WASversion( id ) :
83| if AdminConfig.getObjectType( id ) == 'Cell' :
84| nodeName = System.getProperty( 'local.node' )
85| else :
86| nodeName = getAttributeValue( id, 'name' )
87| return AdminTask.getNodeBaseProductVersion(
88| '[-nodeName %s]' % nodeName
89| )
90|def WASprofileName( id ) :
91| result = WASvarLookup( id, 'USER_INSTALL_ROOT' )
92| if result :
93| result = result.split( os.sep )[ -1 ]
```

```
94| return result
95|def WAShome( id ) :
96| return WASvarLookup( id, 'WAS_INSTALL_ROOT' )
97|def WASvarLookup( id, name ) :
98| VSE = AdminConfig.list( 'VariableSubstitutionEntry', id )
99| for var in VSE.splitlines () :
100| if getAttributeValue( var, 'symbolicName' ) == name : 101| result =
getAttributeValue( var, 'value' ) 102| break
103| else :
104| result = None
105| return result
```

Table 22-2 provides a description of each of these routines.

Table 22-2. *Part 2 of the WASports_05.py Utility Routines, Explained*

Lines	**Description**
82–89	The WASversion(...) routine is called when a tree node that corresponds to the cell or a WebSphere node is selected. A string is returned that identifies the product version of the cell or WebSphere node.
90–94	The WASprofileName(...) routine determines and returns the name of the WebSphere profile associated with the specified configuration ID.
95–96	The WAShome(...) routine determines and returns the WebSphere installation directory.
97–105	The WASvarLookup(...) routine determines the value of the specified WebSphere environment variable.

New Classes for WASports_05

Listing 22-14 shows the revised and simplified cellTSL class that includes the actions to be taken when tree selection events occur. The most significant change is related to the last parameter specified on the constructor, which has changed from a dictionary to a cellInfo object.

Listing 22-14. The WASports_05.py Revised cellTSL Class

```
106|class cellTSL( TreeSelectionListener ) :
107| def __init__( self, tree, pane, data ) :
108| self.tree = tree
109| self.pane = pane
110| self.data = data
111| def valueChanged( self, tse ) :
```

```
pane = self.pane
node = self.tree.getLastSelectedPathComponent()
if node :
if node.isLeaf() :
text = leafFormatString % (
node.getParent(),
node
)
else :
text = self.data.getInfoValue( node.toString() )
else :
text = '<html><br/><b>Nothing selected<b/>'
pane.setText( text )
```

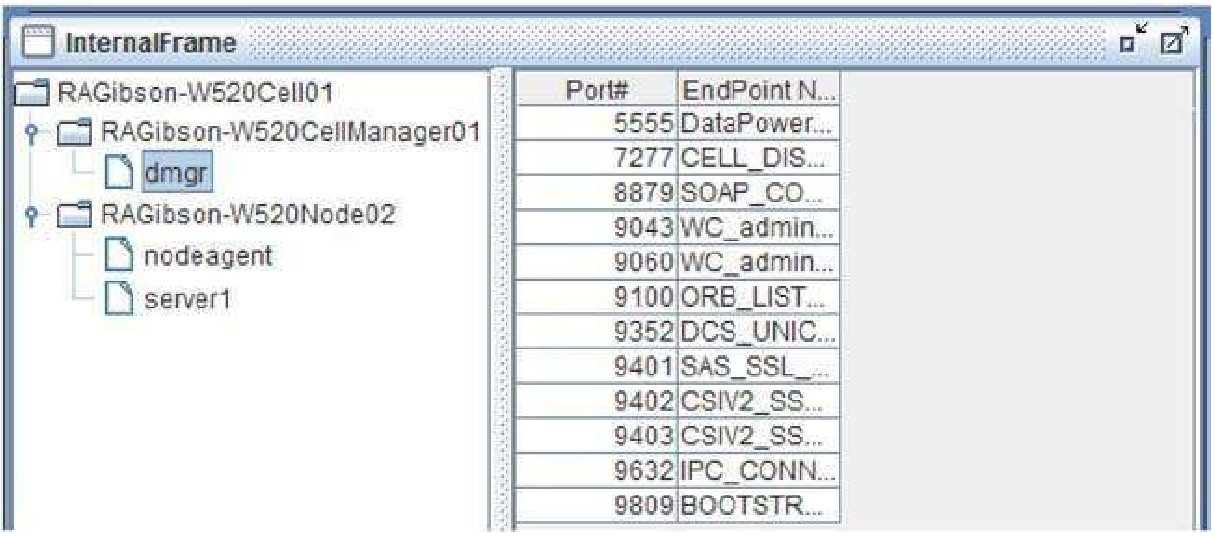
Listing 22-15 shows the cellInfo class that is instantiated by the cellTree method (not shown) and that’s used by the TreeSelectionListener class shown in Listing 22-14.

Listing 22-15. The WASports_05.py Inner cellInfo Class

```
class cellInfo :
def __init__( self ) :
self.names = {} # Dict[ name ] -> configId
self.info = {} # Dict[ name ] -> node info
def getNames( self ) :
return self.names
def setNames( self, names ) :
self.names = names
def addInfoValue( self, index, value ) :
self.info[ index ] = '<html>' + (
value.replace( '&', '&amp;' ).replace( '<',
'&lt;' ).replace( '>', '&gt;' ).replace( ' ',
'&nbsp;' ).replace( '\n', '<br/>' )
)
def getInfoValue( self, index ) :
return self.info[ index ]
```

Step 6: Displaying Server Port Number Information

The next logical step is to add code to display information about the selected server in the right side of the split pane. Let’s take a quick look at the result of these changes. Figure 22-7 shows the initial attempt of displaying the table of port numbers and their associated named endpoints. Even though it isn’t perfect, it does show that you’re going in the right direction.


InternalFrame

```
RAGibson-W520Cell01
  RAGibson-W520CellManager01
    dmgr
  RAGibson-W520Node02
    nodeagent
    server1
```

Port#	EndPoint N...
5555	DataPower...
7277	CELL_DIS...
8879	SOAP_CO...
9043	WC_admin...
9060	WC_admin...
9100	ORB_LIST...
9352	DCS_UNIC...
9401	SAS_SSL_...
9402	CSIV2_SS...
9403	CSIV2_SS...
9632	IPC_CONN...
9809	BOOTSTR...

Figure 22-7. *Sample output of the WASports_06.py script*

To add this functionality to this iteration of the script, start by recognizing the possible value of the firstNamedConfigType(...) routine, shown in Listing 22-16. It can be used to simplify calls elsewhere in the script. It will return the first configuration ID matching the specified values or None.

Listing 22-16. New Utility Routine for WASports_06.py

```
53|def firstNamedConfigType(
54| Type, value, scope = None, attr = None
55|) :
56| items = findScopedTypes( Type, value, scope, attr )
57| if len( items ) :
58| result = items[ 0 ]
59| else :
60| result = None
61| return result
```

Listing 22-17 shows the modified TreeSelectionListener class this is now used to display the appropriate information about the selected tree item, including the JTable instance that is returned by the call to the getPortTable(...) method that starts on line 135.

Listing 22-17. Modified cellTSL (TreeSelectionListener) Class

```
class cellTSL( TreeSelectionListener ) :
def __init__( self, tree, pane, data ) :
self.tree = tree
self.pane = pane # Reference to splitpane
self.data = data
def valueChanged( self, tse ) :
pane = self.pane
loc = pane.getDividerLocation()
node = self.tree.getLastSelectedPathComponent()
if node :
if node.isLeaf() :
pane.setRightComponent (
JScrollPane(
self.data.getPortTable(
(
str( node.getParent() ),
str( node )
)
)
)
)
else :
pane.setRightComponent (
JScrollPane(
JLabel(
self.data.getInfoValue( str( node ) ),
font = MONOFONT
)
)
)
else :
pane.setRightComponent (
JScrollPane(
JLabel(
'<html><br/><b>Nothing selected<b/>',
font = MONOFONT
)
)
160| )
161| pane.setDividerLocation( loc )
```

A new PortLookupTask class, shown in Listing 22-18, was created to perform the necessary calls to the AdminConfig scripting object in order to create a JTable instance of the port numbers and associated named endpoints (shown on lines 267- 270). When the task finishes, the data object, which is an instance of the cellInfo class, is populated with the appropriate server port table data.

Listing 22-18. Part 1 of the New PortLookupTask Class

```
246|class PortLookupTask( SwingWorker ) :
247| lock = threading.Lock()
248| def __init__(
249| self,
250| nodeName,
251| serverName,
252| data
253| ) :
254| self.nodeName = nodeName
255| self.servName = serverName
256| self.data = data
257| SwingWorker.__init__( self )
258| def doInBackground( self ) :
259| self.lock.acquire()
260| try :
261| pDict = self.getPorts( self.nodeName, self.servName )
262| ports = pDict.keys()
263| ports.sort( lambda x,y: cmp( int( x ), int( y ) ) )
264| result = []
265| for port in ports :
266| result.append( [ port, pDict[ port ] ] )
267| table = JTable(
268| PortTableModel( result ),
269| autoResizeMode = JTable.AUTO_RESIZE_OFF
270| )
271| table.getTableHeader().setReorderingAllowed( 0 )
272| self.data.addPortTable(
273| ( self.nodeName, self.servName ),
table
)
except :
print '\nError: %s\nvalue: %s' % sys.exc_info()[ :2 ]
self.lock.release()
```

The remainder of this class is shown in Listing 22-19. Remember that the doInBackground(...) method will call the done(...) method when it is complete. In this case, this method doesn't have to do any additional processing so it only contains the pass statement. The method is left as a placeholder to remind you to consider if additional actions need to be performed.

Listing 22-19. Part 2 of the New PortLookupTask Class

```
def done( self ) :
pass
def getPorts( self, nodeName, serverName ) :
scope = firstNamedConfigType( 'Node', nodeName )
serverEntry = firstNamedConfigType(
'ServerEntry',
serverName,
scope,
'serverName'
)
result = {}
if serverEntry :
nEPs = AdminConfig.list(
'NamedEndPoint',
serverEntry
)
for namedEndPoint in nEPs.splitlines() :
Name = getAttributeValue(
namedEndPoint,
'endPointName'
)
epId = getAttributeValue(
namedEndPoint,
'endPoint'
)
port = getAttributeValue( epId, 'port' )
```

```
result[ port ] = Name
return result
```

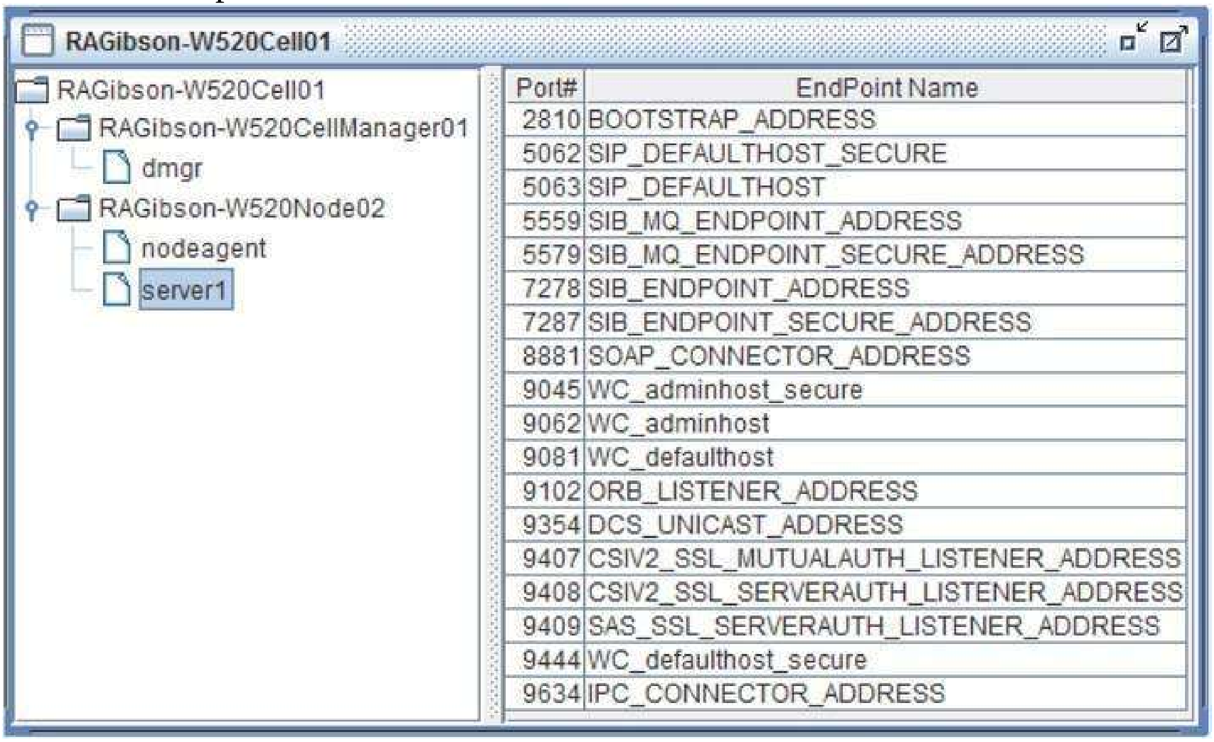
Listing 22-20 shows the PortTableModel class used to hold the data for each server in the cell. This table model class isn’t quite complete, but it is a reasonable start. It identifies the data type for each of the two columns, it identifies only the first column (column 0) as editable, and it includes range checking on the user-supplied port number values.

Listing 22-20. New PortTableModel Class for Holding Port Table Data

```
class PortTableModel( DefaultTableModel ) :
headings = 'Port#,EndPoint Name'.split( ',' )
def __init__( self, data ) :
for row in range( len( data ) ) :
data[ row ] = [
int( data[ row ][ 0 ] ), data[ row ][ 1 ]
]
DefaultTableModel.__init__( self, data, self.headings )
def getColumnClass( self, col ) :
if col == 0 :
return Integer
else :
return String
def isCellEditable( self, row, col ) :
return col == 0
def setValueAt( self, value, row, col ) :
if 0 <= value <= 65535 :
index = ( self, row )
DefaultTableModel.setValueAt( self, value, row, col )
else :
DefaultTableModel.setValueAt(
self,
self.getValueAt( row, col ),
row,
col
)
self.fireTableCellUpdated( row, col )
```

Step 7: Computing Table Column Widths

Next, you'll see that by adding a small amount of code, you can improve the appearance of the application. Figure 22-8 shows the result of adding a routine to determine the preferred widths of the table columns.


RAGibson-W520Cell01

RAGibson-W520Cell01
- RAGibson-W520CellManager01
  - dmgr
- RAGibson-W520Node02
  - nodeagent
  - server1

Port#	EndPoint Name
2810	BOOTSTRAP_ADDRESS
5062	SIP_DEFAULTHOST_SECURE
5063	SIP_DEFAULTHOST
5559	SIB_MQ_ENDPOINT_ADDRESS
5579	SIB_MQ_ENDPOINT_SECURE_ADDRESS
7278	SIB_ENDPOINT_ADDRESS
7287	SIB_ENDPOINT_SECURE_ADDRESS
8881	SOAP_CONNECTOR_ADDRESS
9045	WC_adminhost_secure
9062	WC_adminhost
9081	WC_defaulthost
9102	ORB_LISTENER_ADDRESS
9354	DCS_UNICAST_ADDRESS
9407	CSIV2_SSL_MUTUALAUTH_LISTENER_ADDRESS
9408	CSIV2_SSL_SERVERAUTH_LISTENER_ADDRESS
9409	SAS_SSL_SERVERAUTH_LISTENER_ADDRESS
9444	WC_defaulthost_secure
9634	IPC_CONNECTOR_ADDRESS

Figure 22-8. *Sample output of the WASports_07.py script*

Listing 22-21 shows the setColumnWidths(...) method that was added to the PortLookupTask class. It is brief because it knows about the table contents. For example, rather than looking for the widest value in column 1, the preferred width for this column is determined using the largest allowed value (65535) for this column.

Listing 22-21. The setColumnWidths(...) Method Added to WASports_07.py

```
def setColumnWidths( self, table ) :
tcm = table.getColumnModel() # Table Column Model
data = table.getModel() # To access table data
margin = tcm.getColumnMargin() # gap between columns
render = table.getCellRenderer( 0, 0 )
comp = render.getTableCellRendererComponent(
table, # table being processed
'65535', # max port number
0, # not selected
0, # not in focus
0, # row num
0 # col num
)
cWidth = comp.getPreferredSize().width
col = tcm.getColumn( 0 )
col.setPreferredWidth( cWidth + margin )
cWidth = -1
for row in range( data.getRowCount() ) :
render = table.getCellRenderer( row, 1 )
comp = render.getTableCellRendererComponent(
table,
data.getValueAt( row, 1 ), # cell value
0, # not selected
0, # not in focus
row, # row num
1 # col num
)
cWidth = max(
cWidth,
comp.getPreferredSize().width
)
col = tcm.getColumn( 1 )
col.setPreferredWidth( cWidth + margin )
```

Step 8: Adding Menu Items

What do the changes in this iteration do to the application? Figure 22-9 shows the results of these changes. You can see the menu items that were added.

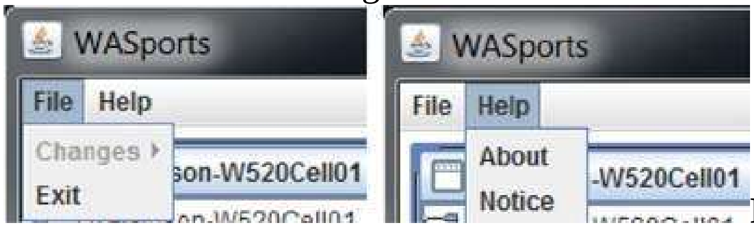


Figure 22-9. *Sample menu items from the WASports_08.py script*

Listing 22-22 shows the new MenuBar(...) method that was added to the WASports class to create the menu and specify the corresponding event-handling routines for each menu item. One thing that is important to notice is the fact that the Changes menu entry is initialized as disabled (as you can see from the first image in Figure 22-9). This iteration of the script doesn't include code to enable this menu entry, so you won't be able to see the Save and Discard menu items until a future iteration of the script.

Listing 22-22. New MenuBar(...) Method in the WASports Class from WASports_08.py

```
def MenuBar( self ) :
menu = JMenuBar()
self.ChangesMI = JMenu( 'Changes', enabled = 0 )
self.ChangesMI.add(
JMenuItem(
'Save',
actionPerformed = self.save
)
)
self.ChangesMI.add(
JMenuItem(
'Discard',
actionPerformed = self.discard
)
)
jmFile = JMenu( 'File' )
jmFile.add( self.ChangesMI )
jmFile.add(
JMenuItem(
'Exit',
actionPerformed = self.Exit
)
)
menu.add( jmFile )
jmHelp = JMenu( 'Help' )
jmHelp.add(
JMenuItem(
'About',
actionPerformed = self.about
)
)
```

```
jmHelp.add(
JMenuItem(
'Notice',
actionPerformed = self.notice )
)
menu.add( jmHelp )
return menu
```

Listing 22-23 shows the initial implementations of the event handler methods for the new menu items. One important thing to note is the reference by the about(...) method to the aboutTask.getResult() method. A new AboutTask class was also added to this iteration of the script in order to process the script docstring[6] and create an HTML string that displays nicely. The reason a separate thread task was used is to allow the application to perform the string conversion processing off the main thread, without causing the entire application to pause.

Listing 22-23. The New Event Handlers Added to WASports_08.py

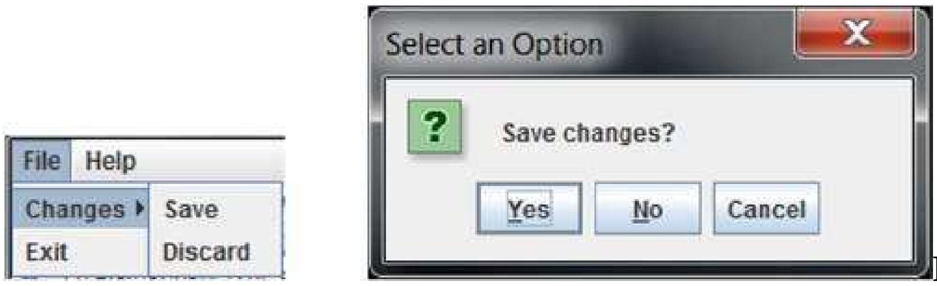
```
def about( self, e ) :
JOptionPane.showMessageDialog(
self.frame,
JLabel(
self.aboutTask.getResult(),
font = MONOFONT
),
'About',
JOptionPane.PLAIN_MESSAGE
)
def notice( self, e ) :
JOptionPane.showMessageDialog(
self.frame,
Disclaimer,
'Notice',
JOptionPane.WARNING_MESSAGE
)
def save( self, e ) :
print 'save() - Not yet implemented'
def discard( self, e ) :
AdminConfig.reset()
```

```
def Exit( self, e ) :
sys.exit()
```

[6]See https://www.python.org/dev/peps/pep-0257/.

Step 9: Implementing Save and Discard

Now that some menu items are in place, it is reasonable to implement the event handlers and associated code to allow these menu items to be used. Figure 22-10 shows that when a port number value changes, the Changes menu item is enabled, which allows the user to save or discard the changes. Additionally, a dialog box is displayed when the user tries to exit the application without saving or tries to discard their changes.



Select an Option

Save changes?

Figure 22-10. *Sample menu items from the WASports_08.py script*

Listing 22-24 shows the appCleanup(...) routine that displays a confirm dialog box when the user tries to exit the application when unsaved changes exist. If the user doesn’t want to discard the changes, control is returned to the caller, which is responsible for resuming operation.

Listing 22-24. The appCleanup() Routine Checks for Unsaved Changes from WASports_09.py

```
def appCleanup( app ) :
if AdminConfig.hasChanges() :
answer = JOptionPane.showConfirmDialog(
app, 'Save changes?'
)
if answer == JOptionPane.YES_OPTION :
AdminConfig.save()
elif answer in [
JOptionPane.CLOSED_OPTION,
JOptionPane.CANCEL_OPTION
] :
return
else :
AdminConfig.reset()
print '\nConfiguration changes discarded.'
System.gc() # call Java Garbage Collector
time.sleep( 0.5 ) # Slight delay for garbage pickup
sys.exit( 0 )
```

Listing 22-25 shows the new SaveTask and DiscardTask classes, which are used to perform the potentially long-running AdminConfig.save() or AdminConfig.reset() as a background task. In addition, the DiscardTask is responsible for updating the port number values in the tables on which the changes were made. **Listing 22-25.** The SaveTask and DiscardTask Classes from WASports_09.py

```
class SaveTask( SwingWorker ) :
def __init__( self, cellData ) :
self.cellData = cellData
def doInBackground( self ) :
try :
original = self.cellData.clearOriginals()
AdminConfig.save()
except :
print '\nError: %s\nvalue: %s' % sys.exc_info()[ :2 ]
class DiscardTask( SwingWorker ) :
def __init__( self, cellData ) :
self.cellData = cellData
def doInBackground( self ) :
try :
original = self.cellData.getOriginal()
tables = []
for index in original.keys() :
table, row = index
table.getModel().resetPortValue(
row,
original[ index ]
332| )
333| if table not in tables :
334| tables.append( table )
335| for table in tables :
336| table.repaint()
337| AdminConfig.reset()
338| except :
339| print '\nError: %s\nvalue: %s' % sys.exc_info()[ :2 ]
```

Listing 22-26 required changes in order to perform the actual AdminConfig.modify(...) method (see line 470), which uses the AdminConfig scripting object to change the specified port number for the indicated named endpoint on the user-selected application server. It is also responsible for calling the cellInfo addOriginal(...) method that's used to save the original value should the Discard menu entry be called.

Listing 22-26. Part 1 of the Modified PortTableModel Class from WASports_09.py

```
444|class PortTableModel( DefaultTableModel ) :
445| headings = 'Port#,EndPoint Name'.split( ',' )
446| def __init__( self, data ) :
447| self.table = None
448| self.nodeName = None
449| self.serverName = None
450| self.epIdDict = None
451| self.app = None
452| for row in range( len( data ) ) :
453| data[ row ][ 0 ] = int( data[ row ][ 0 ] )
454| DefaultTableModel.__init__( self, data, self.headings ) 455| def
getColumnClass( self, col ) :
456| if col == 0 :
457| return Integer
458| else :
459| return String
460| def isCellEditable( self, row, col ) :
461| return col == 0
462| def resetPortValue( self, row, value ) :
463| DefaultTableModel.setValueAt( self, value, row, 0 ) 464|
self.fireTableCellUpdated( row, 0 )
```

Listing 22-27 shows the remainder of the PortTableModel class. You are encouraged to search in the script file to see which of these methods is referenced by the rest of the script.

Listing 22-27. Part 2 of the Modified PortTableModel Class from WASports_09.py

```
def setValueAt( self, value, row, col ) :
prev = self.getValueAt( row, col )
if 0 <= value <= 65535 :
name = self.getValueAt( row, 1 )
epId = self.epIdDict[ name ]
AdminConfig.modify( epId, [ [ 'port', value ] ] )
self.app.ChangesMI.setEnabled( 1 )
DefaultTableModel.setValueAt( self, value, row, col )
self.app.cellData.addOriginal( self.table, row, prev )
else :
DefaultTableModel.setValueAt(
self,
prev,
row,
col
)
self.fireTableCellUpdated( row, col )
def getContext( self ) :
return self.table, self.nodeName, self.serverName
def setContext(
self, table, nodeName, serverName, epIdDict, app
) :
self.table = table
self.nodeName = nodeName
self.serverName = serverName
self.epIdDict = epIdDict
self.app = app
```

Listing 22-28 shows the changes that need to be made to the cellInfo class to allow the original port number values to be saved when the user modifies a value.

Listing 22-28. The Modified (Inner) cellInfo Class

```
class WASports_09( java.lang.Runnable ) :
class cellInfo :
...
def addOriginal( self, table, row, value ) :
self.lock.acquire()
index = ( table, row )
if not self.before.has_key( index ) :
self.before[ index ] = value
self.lock.release()
def getOriginal( self ) :
self.lock.acquire()
result = self.before
self.lock.release()
return result
def clearOriginals( self ) :
self.lock.acquire()
self.before = {}
self.lock.release()
```

Listing 22-29 defines a WindowAdapter descendent for the application. The event handler method in this class is invoked when the user clicks on the application close icon (the [X] in the upper-right corner of the application).

Listing 22-29. The WindowAdapter Class Handles the windowClosed Events from WASports_09.py

```
class windowAdapter( WindowAdapter ) :
def windowClosed( self, e ) :
frame = e.getWindow()
appCleanup( frame )
frame.setVisible( 1 ) # User chose cancel or close
```

Step 10: Implementing the Export Functionality

Now you get to some new and interesting[7] stuff that is unrelated to Swing, but directly related to non-trivial applications that need to be developed. Take a few moments to consider how you might implement an Export capability. It is important to realize that one of the most important questions you should ask is, “What data format should be used?” A number of potential alternatives may come to mind, some being more useful than others.

When I first started thinking about this topic, I considered using XML files to hold the information. This also makes a lot of sense when dealing with WebSphere Application Server configuration files because so many of them use the XML format.

My initial investigations into using the simplest XML module (xml.dom.minidom) didn’t fare well. I tried using a trivial example to test the validity of this as far as the needs of this program were concerned. Listing 22-30 shows a simple interactive wsadmin session and the inability of the xml.dom.minidom module to perform the simplest of XML parsing.

[7]Remember that old curse, “may you live in interesting times?” See http://en.wikipedia.org/wiki/May_you_live_in_ interesting_times.

Listing 22-30. Testing the xml.dom.minidom

```
wsadmin>from xml.dom.minidom import parse
wsadmin>
wsadmin>try :
wsadmin> dom = parse( '70DM.xml' )
wsadmin>except :
wsadmin> print '\nError: %s\nvalue: %s' % sys.exc_info()[ :2 ] wsadmin>
Failed to get environment, environ will be empty: ...

Error: exceptions.AttributeError value: feed
wsadmin>
```

Does this mean that you can’t use XML to represent the data that the application needs to save (export) or load (import)? Not at all. The xml.dom.minidom module is part of the optional libraries that are provided by wsadmin scripts.[8] Just because they are present doesn’t mean that you have to use them.

What other XML modules and libraries exist? It is important, especially at times like these, to remember that other libraries exist as part of Java J2EE.[9] There are a number of free online resources that explain how to work with XML in Java.[10] I spent some time reading the J2EE tutorial,[11] specifically Chapter 2, “Understanding XML.” There is also a number of very good publications that discuss XML processing with Java. Unfortunately, the Jython version of those would be a whole separate book, so I'll provide some examples that correspond very closely to the kind of Java examples available elsewhere.

Using the Document Object Model API

The kinds of XML processes discussed in this section use the Document Object Model (DOM) Application Programming Interface (API) and the Simple API for XML (SAX). Of these two, the DOM is much simpler and easier to write and therefore to read and understand. Let's start with it.

Listing 22-31 shows the important part of a simple script that demonstrates how to read an XML file and then process the data structure that is created when the input file is successfully parsed. It is so simple to read and parse an XML file that it can be performed in a single statement (see lines 22-25). The traverse routine on lines 6-19 uses recursion to traverse the data structure that's produced.

Listing 22-31. Simple DOM Routines from xmlDOM.py

```
6|def traverse( node, indent = 0 ) :
7| prefix = '%*s' % ( indent, '' )
8| while node :
9| if node.getNodeType() == Node.ELEMENT_NODE : 10| print '%s<%s>' % (
prefix, node.getNodeName() ) 11| traverse( node.getFirstChild(), indent + 2 ) 12|
print '%s</%s>' % ( prefix, node.getNodeName() ) 13| elif node.getNodeType()
== Node.TEXT_NODE : 14| value = node.getNodeValue()
15| if value :
16| value = value.strip()
17| if value :
18| print '%s"%s"' % ( prefix, value ) 19| node = node.getNextSibling()
20|def dom( filename ):
21| try :
22| doc = DocumentBuilderFactory.newInstance( 23|
).newDocumentBuilder().parse(
24| File( filename )
25| )
26| root = doc.getDocumentElement()
27| root.normalize()
28| traverse( root )
```

```
except :
print '\nError: %s\nvalue: %s' % sys.exc_info()[ :2 ]
```

[8] For example, %WAS_HOME%\optionalLibraries\jython\Lib\xml\dom\minidom.py.

[9]*I admit that I tend to use the J2EE Reference because WebSphere is an Application Server.*

[10]See http://docs.oracle.com/javaee/1.4/tutorial/information/faq.html.

[11]See http://docs.oracle.com/javaee/1.4/tutorial/doc/index.html or http://docs.oracle.com/javaee/1.4/ tutorial/doc/J2EETutorial.pdf.

This example demonstrates one of the greatest drawbacks of the DOM, and that is the fact that parsing an XML file using the DOM API requires the entire contents of the file to be loaded into memory. For the current application, this might not be too much of a restriction if you are working with the information pertaining to a single WebSphere Application Server cell. However, some cells can be quite large, so this might be a limiting factor. Additionally, I can imagine the application being expanded to work with multiple cells, which could significantly increase the memory requirements of the application.

Using the Simple API for XML (SAX)

Just how simple is the Simple API for XML? It's really not too bad at all. It is possible to have a single statement that parses an XML file using the SAX API, but I'm not going to be that mean to myself or others.[12] Listing 22-32 shows a trivial routine from the xmlSAX.py script file that shows how to instantiate a SAX parser and use it to parse a userspecified XML input file.

Note one of the import differences between these two examples is that this script validates the XMl during the processing. So, in addition to the XMl input, a Document type Definition (DtD) file defines the tags that can exist in a valid XMl file, how they should be used, and their relationship to one another. Unfortunately, details describing and explaining DtDs are beyond the scope of this book.

[12]Just because it is possible to write a one-line statement to perform this task does *not* mean that you should. **Listing 22-32.** The readFileSAX Routine from xmlSAX.py

```
def readFileSAX( filename ) :
try :
FIS = FileInputStream( File( filename ) )
ISR = InputStreamReader( FIS, ENCODING )
src = InputSource( ISR, encoding = ENCODING )
factory = SAXParserFactory.newInstance()
factory.setValidating( 1 )
parser = factory.newSAXParser()
parser.parse( src, SAXhandler() )
except :
print '\nError: %s\nvalue: %s' % sys.exc_info()[ :2 ]
```

Listings 22-33 and 22-34 show the SAXhandler class from the xmlSAX.py file. It is important to note that the SAXhandler methods are called by the SAX parser while the input file is being processed. This allows the script using this class to transform the input data into whatever data structure the developer chooses. Using the DOM technique results in a completely processed XML file being represented in memory using the Document Object Model. If this data structure isn’t appropriate for your application needs, you need to transform the DOM to a more appropriate data structure for your specific needs.

Listing 22-33. Part 1 of the SAXhandler class from xmlSAX.py

```
class SAXhandler( DefaultHandler ) :
def __init__( self ) :
self.chars = "
self.prefix = "
self.width = 0
def indent( self ) :
self.width += 2
self.prefix = '%*s' % ( self.width, " )
def dedent( self ) :
self.width -= 2
self.prefix = '%*s' % ( self.width, " )
def startElement( self, uri, localName, name, attributes ) :
if self.chars :
print '%s"%s"' % ( self.prefix, self.chars )
self.chars = "
attr = [
(
attributes.getQName( i ),
attributes.getValue( i )
)
for i in range( attributes.getLength() )
]
if attr :
print '%s<%s %s>' % (
self.prefix,
name,
', '.join(
[
'%s="%s"' % ( n, v ) for n, v in attr
]
)
)
else :
print '%s<%s>' % ( self.prefix, name )
self.indent()
```

Listing 22-34 shows the remainder of the SAXhandler class.

Listing 22-34. Part 2 of the SAXhandler Class from xmlSAX.py

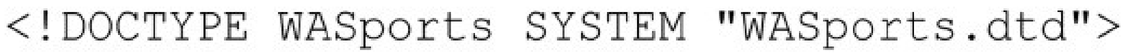
```
def endElement( self, uri, localName, name ) :
if self.chars :
print '%s"%s"' % ( self.prefix, self.chars )
self.chars = "
self.dedent()
print '%s</%s>' % ( self.prefix, name )
def characters( self, ch, start, length ) :
value = str( String( ch, start, length ) ).strip()
if value :
self.chars += value
def warning( self, e ) :
print 'Warning:', e.getMessage()
def error( self, e ) :
print 'Error:', e.getMessage()
def fatalError( self, e ) :
print 'Fatal error:', e.getMessage()
```

The SAX parser calls only some of the methods in the SAXhandler class directly. Specifically, the startElement(...), endElement(...), and characters(...) methods are called during normal parsing of an XML document. The warning(...), error(...), and fatalError(...) methods are called when unexpected input is encountered.

It might not be obvious why the chars attribute is used as a sort of holding buffer. Why not just display the data when the characters(...) method is called? The answer is that there is no guarantee that all of the data between a start and end tag will be processed at one time. Character sequences can be processed in multiple segments. This class buffers the character data until the next start or end tag is encountered. That is why the startElement(...) and endElement(...) methods begin by looking for buffered character data.

The Document Type Definition (DTD)

One of the useful features of the XML parsers is the support provided to validate the input being processed against the description of valid input. That is the role of the Document Type Definition (DTD). It identifies the valid tags and their order, and identifies which tags are allowed to have attributes. Given an appropriate DTD, the parser factory can have validation enabled by calling the setValidating(...) method, as shown in Listing 22-32, line 67. Then, the XML file should identify how the validation should be checked. One way to do this is to include a DOCTYPE definition that identifies whether the DTD is contained in the XML file or in an external file, as shown in Figure 22-11.



```
<!DOCTYPE WASports SYSTEM "WASports.dtd">
```

Figure 22-11. *DOCTYPE definition identifying an external DTD file*

Initial WASports DTD

Using the information contained in the cellInfo class, you can describe the format of an XML file using the DTD shown in Listing 22-35.[13] If you aren’t familiar with the syntax of the DTD file, information is available in a number of places on the Internet[14] and in publications about XML.

Listing 22-35. Initial WASports.dtd

```
<!ELEMENT WASports (cell)>
<!ATTLIST WASports version CDATA #REQUIRED>
<!ELEMENT cell (name,WAShome,WASversion,profile,node+)>
```