Figure 10-4. *Menu5 sample output with radio button menu items*

Figure 10-4 shows the sample output of the Menu5 class, which demonstrates the use of JRadioButtonMenuItem on the menu. In Listing 10-9, you can see most of the code needed to make this happen. The only routine that isn't shown is the one that creates and returns JMenuBar.

[6]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JRadioButtonMenuItem.htn

Listing 10-9. Menu5 Class that Uses Radio Button Menu Items

```
10|class Menu5( java.lang.Runnable ) :
11| def createMenuBar( self ) :
| ...
44| def run( self ) :
45| frame = JFrame(
46| 'Menu5',
47| size = ( 200, 125 ),
48| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
49| )
50| frame.setJMenuBar( self.createMenuBar() )
51| frame.setVisible( 1 )
52| def spam( self, event ) : print 'Menu5.spam()'
53| def eggs( self, event ) : print 'Menu5.eggs()'
54| def bacon( self, event ) : print 'Menu5.bacon()'
55| def about( self, event ) : print 'Menu5.about()'
56| def exit( self, event ) :
57| print 'Menu5.exit()'
58| sys.exit()
```

I think that having the menu-creation process in a method all to itself makes the

remainder of the code more understandable. Listing 10-10 shows the createMenuBar() method from this same class.

Listing 10-10 Menu5 createMenuBar() Method

```
11| def createMenuBar( self ) :
12| menuBar = JMenuBar()
13| fileMenu = JMenu( 'File' )
14| data = [
15| [ 'Spam' , self.spam ],
16| [ 'Eggs' , self.eggs ],
17| [ 'Bacon', self.bacon ]
18| ]
19| bGroup = ButtonGroup()
20| for name, handler in data :
21| rb = JRadioButtonMenuItem(
22| name,
23| actionPerformed = handler,
24| selected = ( name == 'Spam' )
25| )
26| bGroup.add( rb )
27| fileMenu.add( rb )
28| exitItem = fileMenu.add(
29| JMenuItem(
30| 'Exit',
31| actionPerformed = self.exit
32| )
33| )
34| menuBar.add( fileMenu )
35| helpMenu = JMenu( 'Help' )
36| aboutItem = helpMenu.add(
37| JMenuItem(
38| 'About',
39| actionPerformed = self.about 40| )
41| )
42| menuBar.add( helpMenu )
43| return menuBar
```

Notice that this code (see line 24) ensures that one of the radio buttons in the group is selected. It isn’t often that your event handlers (lines 52-58 in Listing

10-9) can actually be performed on a single line. But doing so shortens the listing somewhat.

Using Check Boxes on a Menu

As I'm sure you've already guessed, in addition to radio button menu items, you can also create check box menu items using the JCheckBoxMenuItem class.[7] Figure 10-5 shows some images of the same application using check box menu items.

Figure 10-5. *Menu6 sample output with check box menu items*

Now you can see another advantage of using a separate method to create the menu bar. Listing 10-11 shows only the createMenuBar() method from the Menu6 class because, other than this routine, Menu6.py and Menu5.py are the same.[8]

Listing 10-11. Menu6's createMenuBar() Class

```
10| def createMenuBar( self ) :
11| menuBar = JMenuBar()
12| fileMenu = JMenu( 'File' )
13| data = [
14| [ 'Spam' , self.spam ],
15| [ 'Eggs' , self.eggs ],
16| [ 'Bacon', self.bacon ]
17| ]
18| for name, handler in data :
19| fileMenu.add(
```

```
20| JCheckBoxMenuItem(
21| name,
22| actionPerformed = handler 23| )
24| )
25| exitItem = fileMenu.add(
26| JMenuItem(
27| 'Exit',
28| actionPerformed = self.exit 29| )
30| )
31| menuBar.add( fileMenu )
32| helpMenu = JMenu( 'Help' )
33| aboutItem = helpMenu.add(
34| JMenuItem(
35| 'About',
36| actionPerformed = self.about 37| )
38| )
39| menuBar.add( helpMenu )
40| return menuBar
```

[7]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JCheckBoxMenuItem.html.
[8]Except, of course, for the literal constants that contain the name of the script.

Separating Menu Items

Occasionally, it is convenient to use a horizontal line on a menu to separate groups of items. For example, Figure 10-6 shows how some check box menu items can be separated from one another. I think that you'll agree that the line makes them stand out significantly.

Figure 10-6. *Menu7 sample output showing menu item separation*

Interestingly enough, there are multiple ways that you can add such lines. Listing

10-12 shows three lines from the createMenuBar() method from the Menu7.py file. In it, you see these techniques and how they are used. In the first line, the familiar JMenu.add() method adds an anonymous JSeparator object.[9] This is such a simple and common operation that the JMenu class includes a method to do just this. In the second comment line, you see how the addSeparator() method adds a separator to the end of the specified JMenu. The third line shows that the JMenu has an insertSeparator() method that you can use to insert a separator line at a specified position of an existing menu.

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

Listing 10-12. Menu7 JSeparator() and addSeparator()

```
viewMenu.add( JSeparator() ) # Using JMenu.add() # viewMenu.addSeparator()
# Using addSeparator() # viewMenu.insertSeparator( 1 ) # Using
insertSeparator()
```

Menu Mnemonics and Accelerators

If you look closely at the JMenuItem constructors, you may notice that one includes something called a keyboard *mnemonic*.[10] What is this all about? Well, it lets you specify a key that can be used to select the associated menu item when the menu is visible. To better understand this idea, you'll add a mnemonic to the same menu that you saw earlier. Figure 10-7 shows the Exit menu with an underscore under the letter "x." This indicates that the user can select this entry by pressing the "x" key.

Figure 10-7. *Menu8 menu items with a mnemonic on the Exit menu item*

Figure 10-7 shows what happens to the menu items for which mnemonic values are specified. In each instance, the first instance of the specified letter in the

menu item is underlined.[11] In this case, the “x” in Exit and the “A” in ANSI have menu shortcuts.

The mnemonic keystrokes are active only when the menu is being displayed. So, if you wanted to be able to use the “x” mnemonic to exit the application using the keyboard, you would need to press the F10 key to activate the first menu entry, and then press the “x” key to use the mnemonic to select this menu item.[12]

How do you define a mnemonic for a JRadioButtonMenuItem? Unlike the JMenuItem constructor, no parameter exists for a keyboard mnemonic. This is yet another example where Jython makes life so much easier for you. Instead of creating a JRadioButtonMenuItem and then using the setMnemonic() method inherited from the AbstractButton class, you can simply add a keyword argument to the constructor. Listing 10-13 shows how this is done in Menu8.py.

[10] See http://docs.oracle.com/javase/8/docs/api/javax/swing/JMenuItem.html#JMenuIte
[11]In situations where you want to underline a different letter in the text string, use the setDisplayedMnemonicIndex() method inherited from the AbstractButton class, i.e., http://docs.oracle.com/javase/8/docs/api/javax/swing/AbstractButton.html#setDisplayedMnemonicIndex%28int%29.
[12]Of course, you could select the File menu entry using the mouse and then press the “x” key on the keyboard, but it’s just as easy to use the mouse to select Exit after using it to open the File menu.

Listing 10-13. Using the Mnemonic Keyword Argument on the JRadioButtonMenuItem Constructor

```
25| codeMenu = JMenu( 'Encoding' )
26| data = [
27| [ 'ANSI' , KeyEvent.VK_A ],
28| [ 'UTF-8' , KeyEvent.VK_U ],
29| [ 'UCS-2 Big Endian' , KeyEvent.VK_B ],
30| [ 'UCS-2 Little Endian' , KeyEvent.VK_L ]
31| ]
32| bGroup = ButtonGroup()
33| for suffix, mnemonic in data :
34| name = 'Encoding in ' + suffix
```

```
35| rb = JRadioButtonMenuItem(
36| name,
37| mnemonic = mnemonic,
38| selected = ( suffix == 'ANSI' )
39| )
40| bGroup.add( rb )
41| codeMenu.add( rb )
```

One of the limitations of mnemonics is that the menu entry with a mnemonic has to be visible for the mnemonic key to be recognized. This brings us to the topic of to accelerator keys. The advantage of associating an accelerator key with a menu entry (or item) is that the menu does not have to be visible for the associated event to be initiated. One disadvantage of accelerator keys is that you need to use the menu item setAccelerator() method to associate an accelerator key with a menu item. Unless, of course, you're using a wonderful language like Jython that allows you to use keyword arguments on your constructor calls.

Another possible disadvantage of accelerators is that, unlike mnemonics, there is no obvious indication that an accelerator key is in effect. Listing 10-14 shows a snippet from Menu9.py, which uses a bad practice of associating an unmodified "x" accelerator key with the Exit menu item. This means that when the user presses the "x" key, the program immediately exits. This is unlikely to be an event or action that will be anticipated, or appreciated, by your users.

Listing 10-14. Example of an Unmodified Accelerator

```
18| exitItem = fileMenu.add(
19| JMenuItem(
20| 'Exit',
21| KeyEvent.VK_X,
22| actionPerformed = self.exit,
23| accelerator = KeyStroke.getKeyStroke( 'x' )
24| )
25| )
```

What can and should be done about this? One possibility is to use the modified keys (Alt-X instead of simply X) .Menu10.py tries to do exactly this. Instead of "x" as shown in line 46 of Listing 10-14, Listing 10-15 defines the accelerator as Alt-X (see lines 23-27). Listing 10-15 is from Menu10a.py, which defines Alt-X

as the accelerator key for the Exit menu item.

Listing 10-15. Using Alt-X as an Accelerator

```
42| exitItem = fileMenu.add(
43| JMenuItem(
44| 'Exit',
45| KeyEvent.VK_X,
46| actionPerformed = self.exit,
47| accelerator = KeyStroke.getKeyStroke(
48| 'x',
49| InputEvent.ALT_DOWN_MASK
50| )
51| )
```

What does the menu item look like when you add this kind of accelerator key? Figure 10-8 shows how Swing displays this kind of information. I don't know about you, but I'm pretty impressed by Swing when I see that it automatically adds Alt-X to the menu item to clearly convey the presence of the accelerator key.

Figure 10-8. *Menu10 menu showing Alt-X accelerator key*

Unfortunately, this doesn't work for some reason. When you press Alt-X, nothing happens. Why not? Let's figure out what is wrong. What are the differences between the unmodified KeyStroke in Listing 10-14 (line 23) and the modified KeyStroke in Listing 10-15 (lines 47-50)? Listing 10-14 uses a KeyEvent constant instead a simple character string. Does it matter? Unfortunately, it does. Listing 10-16[13] shows what is returned by the KeyStroke.getKeyStroke() method using four different techniques.

Listing 10-16. Generating Modified KeyStroke Instances

```
wsadmin>from javax.swing import KeyStroke as KS
wsadmin>from java.awt.event import ActionEvent
wsadmin>from java.awt.event import InputEvent
wsadmin>from java.awt.event import KeyEvent
wsadmin>
wsadmin>KS.getKeyStroke( 'x' )
typed x
wsadmin>KS.getKeyStroke( KeyEvent.VK_X, ActionEvent.ALT_MASK )
alt pressed X
wsadmin>KS.getKeyStroke( KeyEvent.VK_X, InputEvent.ALT_MASK ) alt
pressed X
wsadmin>KS.getKeyStroke( 'x', ActionEvent.ALT_MASK )
alt typed x
wsadmin>
wsadmin>a = KS.getKeyStroke( KeyEvent.VK_X, ActionEvent.ALT_MASK )
wsadmin>i = KS.getKeyStroke( KeyEvent.VK_X, InputEvent.ALT_MASK )
wsadmin>a == i
1
wsadmin>
```

[13]Listing 10-16 uses the KS alias for KeyStroke so the lines aren't too long.

What does this mean for your applications? Well, it means that if you try to use the technique from Listing 10-15, line 48, your accelerator keys won't work. It isn't obvious from the user interface that there is a difference, but it's a subtle and important one at that. The last part of Listing 10-16 shows that you can use ActionEvent or InputEvent to identify the modification type—it doesn't matter. The result of the comparison is 1 (true), which means that the values are the same.

Pop-Up Menus

In addition to the menu bars, Swing applications can also use pop-up menus. What is a pop-up menu, exactly? Just as with a menu on the menu bar, when a pop-up menu is activated, its menu items are displayed. Unlike a menu bar menu, a pop-up menu is normally activated using a right-click on the component with which the pop-up menu is associated. Interestingly enough, this approach enables you to define multiple context-sensitive pop-up menus if that makes sense for your application.

Let’s take a quick look at a trivial pop-up menu for a very simple application. Figure 10-9 shows the application output with two labels and two text input fields. If you right-click on either of the input fields, a small pop-up menu is displayed. Should you decide to select one of the values on the menu, that value is placed into the text field over which the right-click occurred.

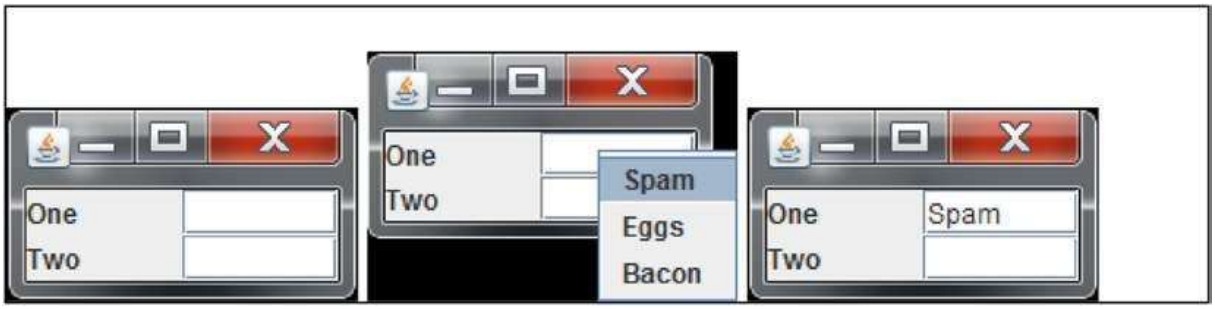


Figure 10-9. *Popup1 sample output*

Listing 10-17 shows some of the Popup1 class methods used to produce the output shown in Figure 10-9.

Listing 10-17. Popup1 actionPerformed() and run() Methods

```
11|class Popup1( java.lang.Runnable ) :
12| def actionPerformed( self, event ) :
13| self.target.setText( event.getActionCommand() )
14| def run( self ) :
15| frame = JFrame(
16| 'Popup1',
17| layout = GridLayout( 0, 2 ),
18| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
19| )
20| frame.add( JLabel( 'One' ) )
21| frame.add(
22| JTextField(
23| 5,
24| mousePressed = self.PUcheck,
25| mouseReleased = self.PUcheck
26| )
27| )
28| self.PU = self.PUmenu()
29| frame.add( JLabel( 'Two' ) )
30| frame.add(
31| JTextField(
```

```
32| 5,
33| mousePressed = self.PUcheck,
34| mouseReleased = self.PUcheck
35| )
36| )
37| frame.pack()
38| frame.setVisible( 1 )
```

Listing 10-18 shows the remainder of the Popup1 class methods found in the Popup1.py script.

Listing 10-18. Popup1 PUmenu() and PUcheck() Methods

```
39| def PUmenu( self ) :
40| def MenuItem( text ) :
41| return JMenuItem(
42| text,
43| actionPerformed = self.actionPerformed
44| )
45| popup = JPopupMenu()
46| popup.add( MenuItem( 'Spam' ) )
47| popup.add( MenuItem( 'Eggs' ) )
48| popup.add( MenuItem( 'Bacon' ) ) 49| return popup
50| def PUcheck( self, event ) :
51| if event.isPopupTrigger() : 52| self.target = event.getSource() 53|
self.PU.show(
54| event.getComponent(), 55| event.getX(),
56| event.getY()
57| )
```

One of the reasons for including multiple input fields was to demonstrate that the event used to display the popup can also be used to determine which component to associate with the event (see line 52 in Listing 10-18). In order for the pop-up menu to have access to this information, it needs to be saved somewhere in the application. Jython makes this really easy, by allowing you to dynamically create object attributes as they are needed. You can see that the only other occurrence of the self.target object attribute is found on line 13 of Listing 10-17 in the ActionListener actionPerformed() method. Table 10-1 describes the various parts of the Popup1 class.

Table 10-1. *Popup1 Class, Explained*

Lines	Description
12-13	The actionPerformed() method is invoked when a pop-up menu item is selected (line 43).
14-38	Popup1 run() method, which: · Creates the application frame (lines 15-19). · Creates the labels and text fields (lines 20-36). Note the use of the mousePressed and mouseReleased keyword assignments. · Creates the pop-up menu (line 28). · Resizes and shows the frame (lines 37-38).
39-49	PUmenu() method that's used to create the JPopupMenu and MenuItems.
50-57	PUcheck() method that's invoked by mouse-listener routines to verify that a pop-up trigger occurred. If a trigger did occur, it saves the associated input field and then shows the pop-up menu based on the cursor's location.

Summary

This chapter is all about menus and menu items—more specifically, it's about JMenus and JMenuItems. It is important to remember the difference. The former are AWT classes and the latter are Swing classes. You saw how easy it is to create menu bars, pop-up menus, and the event handlers that are associated with these menu items. In the next chapter, you take a look at really useful structure—JTree—which is great for displaying hierarchical relationships between information.

Chapter 11

Using JTree to Show the Forest: Hierarchical Relationships of Components

This chapter is all about building and using trees to display a hierarchical relationship between elements. Trees allow you to convey a specific relationship between different pieces of information. For example, this kind of structure is great for showing something like an organizational chart. By the time you're through with this chapter, you should not only be able to use trees in your own

applications, but you will also more aware of the capabilities and limitations of this useful structure.

Displaying the Servers in a WebSphere Application Server Cell

One of the really useful Swing structures is JTree,[1] which lets you show a hierarchical relationship of components. A simple WebSphere Application Server cell will have a number of nodes, with each node having a number of servers. This hierarchy is well suited for a tree structure. Figure 11-1 shows a sequence of images created by the Tree1.py script. Here's a description of each image, from top-left to bottom-right:

• This image only shows the cell name and the associated node names within the cell. The horizontal scroll bar shows that the tree is within a scroll bar, and that the available horizontal space isn't sufficient to display the widest node name.

• This image shows what the tree looks like with the first node expanded. The highlighting is an indication that a tree node is selected.
• This image shows what happens when the second node is expanded.
• The final image shows the application after it has been resized and has multiple nodes selected.

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

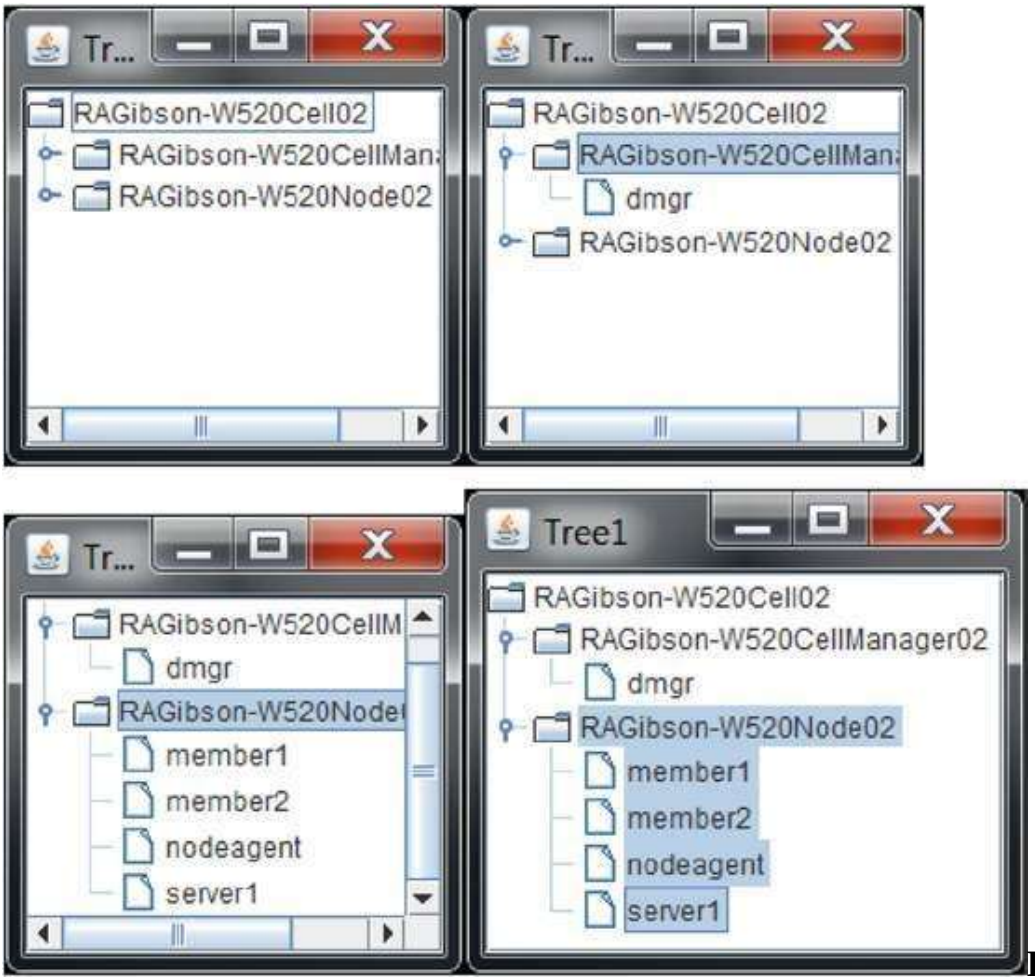


Figure 11-1. *Tree1 sample output*

The resized window is large enough to display the complete tree, so no scroll bars are necessary. Trees display information vertically, with each line or row displaying a single piece of information, or data. Each of these data items is called a *node*, and the node at the top of the tree is called the *root node*.[2] A node may or may not have additional nodes beneath it. If it does, it is called a *branch node*, if it doesn't, it's called a *leaf node*. A branch node is also called the parent of the nodes beneath it, and they are referred to as children, or child nodes. You may have noticed that there is a bit of a terminology collision[3] here.

Figure 11-1 shows the name of the WebSphere cell as the root node, and the names of the WebSphere nodes as children of the root node of the tree. Subsequent images in this figure show how individual nodes can be selected, collapsed, or expanded using the mouse or keyboard.[4]

The second image in the figure shows an expanded branch node. In this case, the

first child node beneath the root identifies the WebSphere deployment manager node that has one child (leaf) node, identified as dmgr. In this application, each leaf node represents an application server and shows the name of each server. It's important to remember that server names must be unique within the same node, but can occur multiple times within the cell.
This hierarchical representation of the names of the WebSphere cell, nodes, and servers should match your conceptual understanding of the relationship between these WebSphere configuration objects.
As Figure 11-1 illustrates, it is very likely that the JTree instance will be in a JScrollPane to allow scroll bars to displayed as needed.

[2] To minimize the chance of confusion, I'm going to try to be consistent and refer to Swing tree nodes as nodes, and WebSphere Application Server nodes as WebSphere nodes.
[3]This topic also came up in Chapter 10.
[4]Tree nodes can be expanded or collapsed by positioning the entry using a mouse or cursor keys and pressing Enter.

Listing 11-1 shows the Tree1 class from the Tree1.py script used to generate the output shown in Figure 11-1.[5]
Listing 11-1. Tree1 Class Using the JTree Class

```
8|class Tree1( java.lang.Runnable ) :
9| def cellTree( self ) :
10| cell = AdminConfig.list( 'Cell' )
11| root = DefaultMutableTreeNode( self.getName( cell ) ) 12| for node in
AdminConfig.list( 'Node' ).splitlines() : 13| here = DefaultMutableTreeNode(
14| self.getName( node )
15| )
16| servers = AdminConfig.list( 'Server', node ) 17| for server in
servers.splitlines() :
18| leaf = DefaultMutableTreeNode(
19| self.getName( server )
20| )
21| here.add( leaf )
22| root.add( here )
23| return JTree( root )
24| def getName( self, configId ) :
25| return AdminConfig.showAttribute( configId, 'name' ) 26| def run( self ) :
```

```
27| frame = JFrame(
28| 'Tree1',
29| size = ( 200, 200 ),
30| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 31| )
32| frame.add( JScrollPane( self.cellTree() ) ) 33| frame.setVisible( 1 )
```

Table 11-1 provides a short description of the code used to build and display the hierarchical tree structure. From the code in Listing 11-1, the description in Table 11-1, and the output in Figure 11-1, it should be clear that the JTree class is easy to use and quite powerful. With no additional code, you can select one or more items as well as expand or collapse individual nodes.[6]

Table 11-1. *Tree1 Listing Explained*

Lines	Description
9–23	cellTree() method that creates, populates, and returns a JTree instance representing the names of the WebSphere cell, nodes, and application servers. Each node in the tree is an instance of DefaultMutableTreeNode and may have zero or more child nodes (aka children).
24–25	getName() method that uses the AdminConfig.showAttribute() scripting object method call to obtain and return the name of the specified configuration object.
26–33	Tree1 run() method that creates the application frame, populates it, and displays it on the Swing event dispatch thread.

[5] Remember that AdminConfig is a wsadmin scripting object, and therefore is only available when wsadmin is used to execute the Jython script.

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

JTree Attributes and Methods

You can see, by looking at the cellTree() method in Listing 11-1, that it is really easy to create a JTree. You can also see, from Figure 11-1, that the default settings or attributes might not match your expectations. For example, you might not want to allow the users to select multiple entries. How do you limit the users to selecting zero or one item at a time? You'll see how to do this shortly. Before you do that, though, it is important to see how the tree structure is separated

from the data being represented.

The TreeSelectionModel Class

One scenario that occurs frequently in the Swing class hierarchy is having one class to display a structure, and having a related but separate class that holds the data. This is also the case for trees. The class used by a tree to hold its data is an implementation of the TreeModel interface.[7] The only implementation of this class used by this book is the DefaultTreeModel class.[8]

In addition to the data model, the JTree class also has a TreeSelectionModel associated with it. Unless, of course, you don't want to let the users select any of the tree nodes. If this is the case, it should be set to None (or null in Java terms).

What, exactly, does this mean? Does it mean that you can't expand or collapse the tree nodes? No, it doesn't. It just means that no node can be selected. Figure 11-2 shows images similar to those Figure 11-1, but this time none of the nodes can be selected.

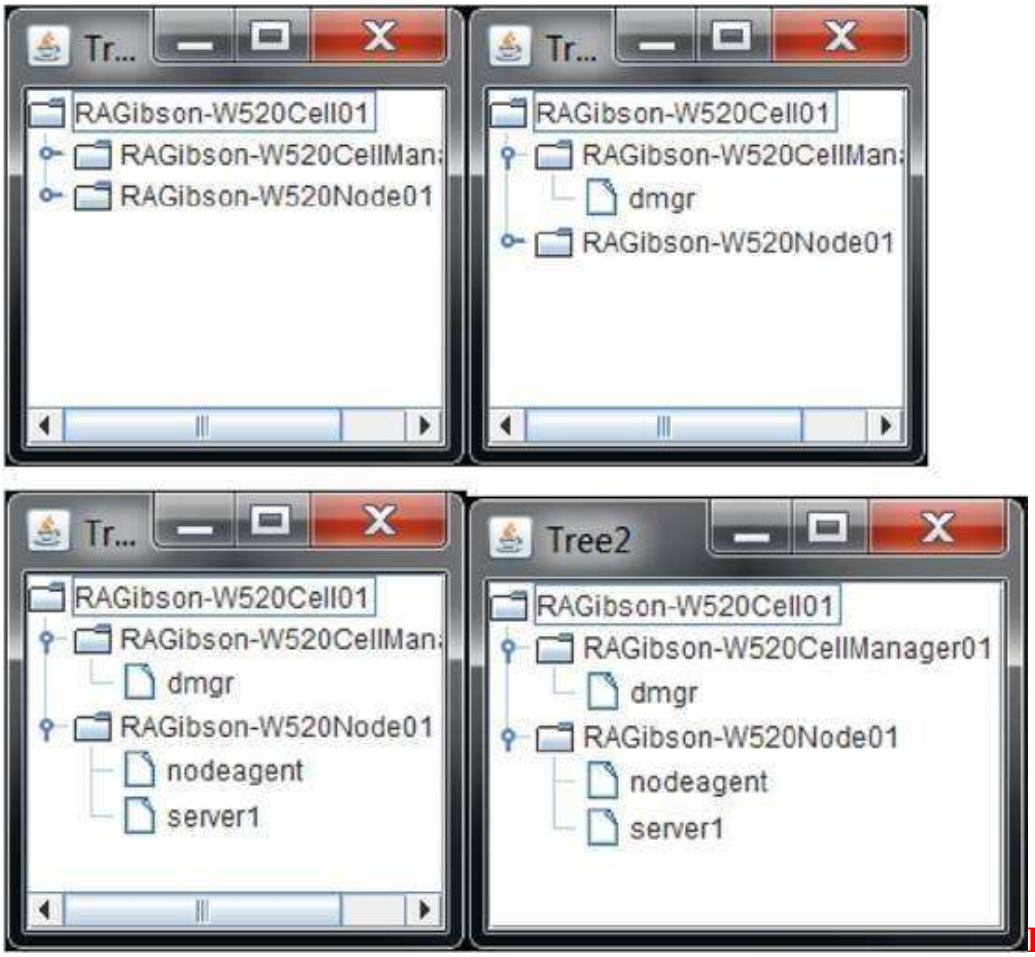


Figure 11-2. *Tree2 sample output*

[7]See http://docs.oracle.com/javase/8/docs/api/javax/swing/tree/TreeModel.html.

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

What do you need to do to set this kind of limitation? Well, it's really quite simple. Listing 11-2 contains the run() method from Tree2.py, which shows that you only need to change lines 32-34, which correspond to line 32 in Listing 11-1.

Listing 11-2. Tree2 run() Method

```
26| def run( self ) :
27| frame = JFrame(
28| 'Tree2',
29| size = ( 200, 200 ),
```

```
30| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
31| )
32| tree = self.cellTree()
33| tree.setSelectionModel( None )
34| frame.add( JScrollPane( tree ) )
35| frame.setVisible( 1 )
```

Let’s get back to the question raised earlier. How do you limit the selection to zero or one tree node? The answer is that you need to change the default selection mode using the setSelectionMode() method. What kinds of values does this method allow? The value must correspond to the TreeSelectionModel constants shown in Table 11-2.[9]

Table 11-2. *TreeSelectionModel Constants*

SINGLE_TREE_SELECTION	Allows a maximum of one tree node to be selected.
CONTIGUOUS_TREE_SELECTION	Allows a maximum of one continuous adjacent group of nodes to be selected.
DISCONTIGUOUS_TREE_SELECTION	Allows an unlimited number of nodes to be selected, with no restriction that they be adjacent.

This table may make you think about lists and how you can limit the selection of list items.[10] In fact, there is a direct correlation between the selection settings allowed by JList instances and JTree instances. In the example scripts in Chapter 9, you used the JList selectionMode keyword argument to limit the JList instances to a single item.

You might wonder if you can do a similar thing with JTree instances. Listing 11-3 shows how you can use the classInfo function, which you learned about in Chapter 4, to display the JTree class hierarchy. It also shows all of the attributes that contain the string "selectionmode". Unlike the corresponding JList class hierarchy, the JTree class does not have a selectionMode attribute. So the answer to the question is no, you can’t do it quite that easily.

[9]See http://docs.oracle.com/javase/8/docs/api/javax/swing/tree/TreeSelectionModel.ht

[10]See the section in Chapter 9 entitled “Limiting the Selectable Items.”

Listing 11-3 JTree class hierarchy showing selectmode attributes

```
wsadmin>from javax.swing import JTree
wsadmin>
wsadmin>classInfo( JTree, attr = 'selectionmode' ) javax.swing.JTree

selectionModel
| 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.swing.Scrollable
| javax.accessibility.Accessible
wsadmin>
```

So, how can you do it? Well, Listing 11-4 contains the run() method from Tree3.py. The statement used to limit the tree selection to a single node is performed in lines 34-36, which correspond to line 33 in Listing 11-2. From this you can see that in order to change the selection mode you need to obtain the selection model used by this tree (via a call to the getSelectionModel() method as shown in line 34). You then call the setSelectionMode() method of this model to make the change.

Listing 11-4. Tree3 run() Method, Limiting Node Selection

```
27| def run( self ) :
28| frame = JFrame(
29| 'Tree3',
30| size = ( 200, 200 ),
31| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
32| )
33| tree = self.cellTree()
34| tree.getSelectionModel().setSelectionMode(
35| TreeSelectionModel.SINGLE_TREE_SELECTION
36| )
```

```
37| frame.add( JScrollPane( tree ) )
38| frame.setVisible( 1 )
```

Note it is possible to use the Jython-provided attributes to minimize the explicit calls to the specified methods, but that is likely to cause confusion. Code to do this would look something like:
tree.selectionModel.selectionMode =
TreeSelectionModel.SINGLE_TREE_SELECTION
You will have to decide if using the attribute names like this instead of the getter and setter methods is worth the potential confusion that it might cause.
TreeSelectionListener

There are times when your applications will need to know when the user makes a tree node selection. For example, when the user selects a tree node it is likely that they want information about this node displayed immediately. You will need the application to display information about the selected WebSphere node as soon as the user selects the corresponding tree node.

To do this, you need to use the TreeSelectionListener interface.[11] Just like the ListSelectionListener,[12] which is described in Chapter 9, this interface has a single valueChanged() method. Can you use the valueChanged keyword argument on the JTree constructor call to identify a TreeSelectionListener method for the JTree instance? The simple quick answer to this question is yes. Figure 11-3 shows one possible use for this kind of listener.

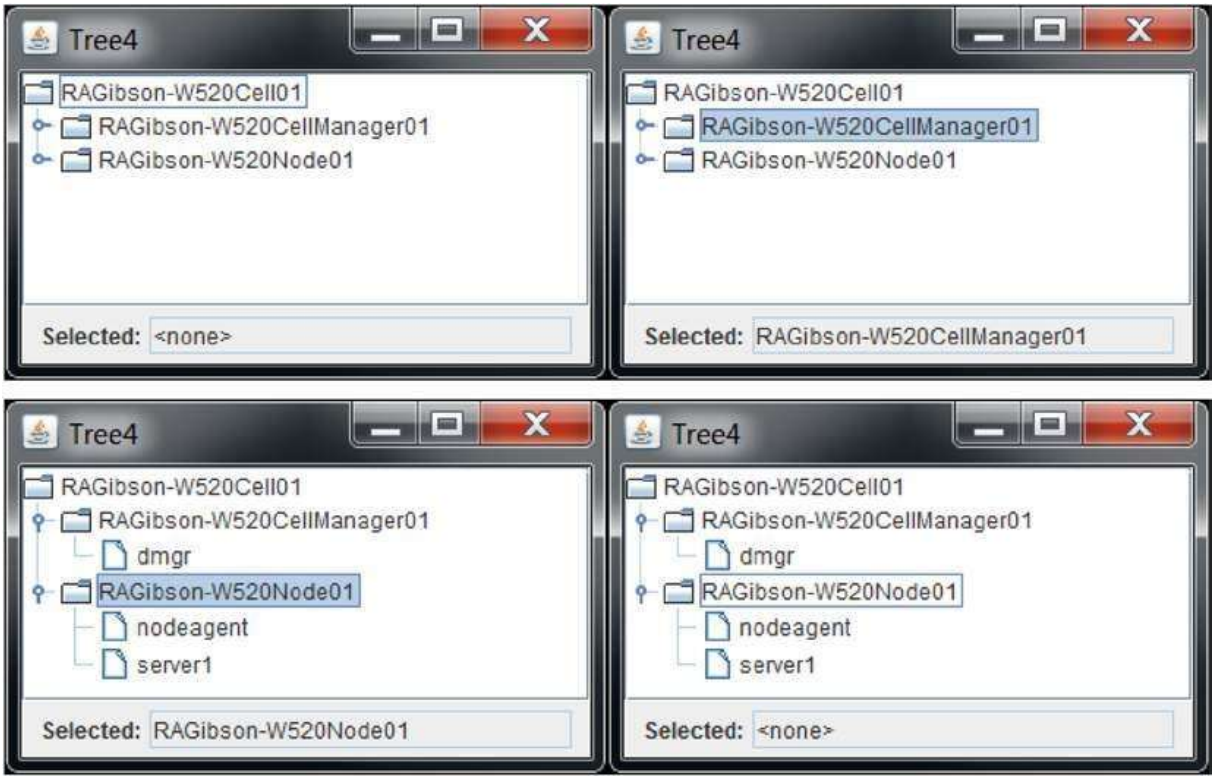


Figure 11-3. *Tree4 sample output*

In Figure 11-3, you can see how the text field is updated by the tree selection listener event handler to reflect the data from the selected node; it also shows "<none>" if no node has been selected. This will prove to be very useful, as you'll see when you build on this application. Listings 11-5 and 11-6 show the Tree4 class that this application uses.

[11]See http://docs.oracle.com/javase/8/docs/api/javax/swing/event/TreeSelectionListener
[12]See http://docs.oracle.com/javase/8/docs/api/javax/swing/event/ListSelectionListener.

Listing 11-5. Tree4 Class Using a TreeSelectionListener

```
13|class Tree4( java.lang.Runnable ) :
14| def cellTree( self ) :
15| cell = AdminConfig.list( 'Cell' )
16| root = DefaultMutableTreeNode( self.getName( cell ) )
17| for node in AdminConfig.list( 'Node' ).splitlines() :
18| here = DefaultMutableTreeNode(
19| self.getName( node )
```

```
20| )
21| servers = AdminConfig.list( 'Server', node )
22| for server in servers.splitlines() :
23| leaf = DefaultMutableTreeNode(
24| self.getName( server )
25| )
26| here.add( leaf )
27| root.add( here )
28| return JTree( root, valueChanged = self.select )
29| def getName( self, configId ) :
30| return AdminConfig.showAttribute( configId, 'name' )
```

Listing 11-6 shows the rest of the Tree4 class, including the run() method and the TreeSelectionListener event handler. The select() method is the event handler method for this application. This routine is identified as the listener for the JTree instance by using the valueChanged keyword assignment on line 28 in Listing 11-5.

The run() method creates and populates the application frame. Lines 38-42 of Listing 11-6 show how easy it is to create the tree, limit the number of items that can be selected, and position it all on the application frame. This is accomplished in just three statements.

Setting up the rest of the frame is also as easy. Lines 43-52 show how four statements can create a label and a read-only text field. These are placed in a panel, which is then positioned on the bottom part of the frame.
This is a wonderful example of how easy it is to create an interesting and useful application using Swing components in a creative manner.

Listing 11-6. Tree4 Class Using a TreeSelectionListener, Continued

```
31| def run( self ) :
32| frame = JFrame(
33| 'Tree4',
34| size = ( 320, 200 ),
35| layout = BorderLayout(),
36| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
37| )
38| tree = self.cellTree()
```

```
39| tree.getSelectionModel().setSelectionMode(
40| TreeSelectionModel.SINGLE_TREE_SELECTION
41| )
42| frame.add( JScrollPane( tree ), BorderLayout.CENTER )
43| panel = JPanel()
44| panel.add( JLabel( 'Selected:' ) )
45| self.msg = panel.add(
46| JTextField(
47| '<none>', # Initial value 48| 20, # Field width (columns) 49| editable = 0 #
Disable editing 50| )
51| )
52| frame.add( panel, BorderLayout.SOUTH )
53| frame.setVisible( 1 )
54| def select( self, event ) :
55| tree = event.getSource()
56| if tree.getSelectionCount() :
57| node = str( tree.getLastSelectedPathComponent() ) 58| else :
59| node = '<none>'
60| self.msg.setText( node )
```

Table 11-3 describes the important changes from the previous tree examples and the Tree4 class shown in Listings 11-4 and 11-5. I think it's impressive how easily you can use the Swing classes to build this neat little application.[13]

Table 11-3. *Tree4 Listing Explained*

Lines	Description
28	The only real difference is where you specify the valueChanged keyword on the JTree constructor to specify the TreeSelectionListener event handler routine to be called.
35	To make the presentation more aesthetically appealing, the frame uses the Border Layout Manager.
42	You position the JScrollPane instance containing the JTree in the center of the application frame.
43–52	You create a Panel container to hold the label and the disabled text field on the bottom of the application frame.

54–60 The select() method is the TreeSelectionListener valueChanged event handler. This method uses the event.getSource() method to obtain a reference to the associated tree, and if a node has been selected, it uses the getLastSelectedPathComponent() method call to identify that node.

JTree Manipulation

Some applications provide a way for the users to make changes to the tree being displayed. For example, you might want your application to provide a way for the user to change the tree node to reflect new information. This can be as simple as allowing the displayed values to be modified, or it may be appropriate to add and remove specific nodes.

[13]The dataType of the value returned by the getLastSelectedPathComponent() method call is an object, so it needs to be converted to a string before this value can be used to update the JTextField value.

.Let’s break this example into pieces, and not just because the listing, even without comments and blank lines, is too big. To start, let’s see what the application looks like when the DynamicTree.py script is executed. Figure 11-4 shows three images of this application before any changes are made.[14]

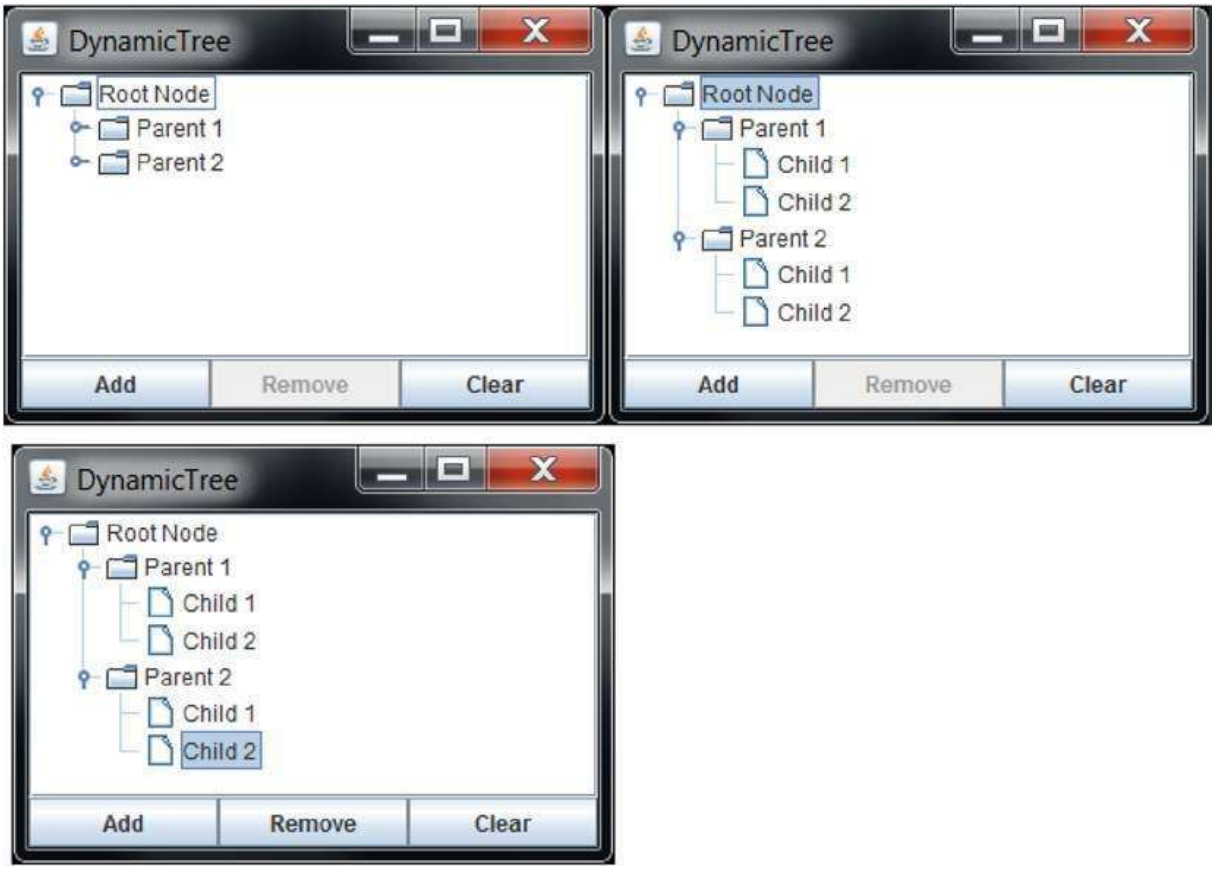


Figure 11-4. *DynamicTree sample output*

The first part of the DynamicTree class, shown in Listing 11-7, provides enough information to understand the layout of the application shown in Figure 11-4. The top part of the application window has a scroll pane containing the initial tree. Beneath this, there is a row of buttons. The run() method, shown in lines 59-77, creates and populates the frame, as you have seen before. For now, don't worry about the constructor (the __init__() method) or the getSuffix() method. You'll see how they are used shortly.

[14]This example is based on the DynamicTreeDemo from the Java Swing Tutorial, which can be found at http://docs.oracle.com/javase/tutorial/uiswing/components/tree.html#dynamic.

Listing 11-7. DynamicTree Class (Part 1 of 5)

```
53|class DynamicTree( java.lang.Runnable ) :
54| def __init__( self ) :
55| self.nodeSuffix = 0
56| def getSuffix( self ) :
```

```
57| self.nodeSuffix += 1
58| return self.nodeSuffix
59| def run( self ) :
60| frame = JFrame(
61| 'DynamicTree',
62| layout = BorderLayout(),
63| locationRelativeTo = None,
64| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
65| )
66| self.tree = self.makeTree() # Keep references handy
67| self.model = self.tree.getModel()
68| frame.add(
69| JScrollPane(
70| self.tree,
71| preferredSize = Dimension( 300, 150 )
72| ),
73| BorderLayout.CENTER
74| )
75| frame.add( self.buttonRow(), BorderLayout.SOUTH )
76| frame.pack()
77| frame.setVisible( 1 )
```

The only new thing in this run() method is line 63, which you might not have seen. What does this keyword argument do? It corresponds with the setLocationRelativeTo() method from the JFrame class that is inherited from the java.awt.Window class.[15] By initializing this value to None, which corresponds to the Java null, the application window is positioned in the center of the screen.[16] Pretty neat, eh?

The buttonRow() method, called in line 75 of Listing 11-7, can be seen on lines 78-94 in Listing 11-8. There really shouldn't be anything too surprising in this routine, which uses a GridLayout instance to position the buttons being created in one horizontal row and assigns the actionPerformed event handler for each button using a keyword argument (line 90).

Listing 11-8. DynamicTree Class (Part 2 of 5, buttonRow() Method)

```
78| def buttonRow( self ) :
79| buttonPanel = JPanel( GridLayout( 0, 3 ) )
```

```
80| data = [
81| [ 'Add' , self.addEvent ],
82| [ 'Remove', self.delEvent ],
83| [ 'Clear' , self.clsEvent ]
84| ]
85| self.buttons = {}
86| for name, handler in data :
87| self.buttons[ name ] = buttonPanel.add ( 88| JButton(
89| name,
90| actionPerformed = handler, 91| enabled = name != 'Remove' 92| )
93| )
94| return buttonPanel
```

[15]See http://docs.oracle.com/javase/8/docs/api/java/awt/Window.html#setLocationRela
[16]Actually, the top-left corner of the application is positioned on the center of the screen. This should be “close enough” for now.

Listing 11-9 shows the code for the three button event handlers that are referenced by the data array in lines 80-84 of Listing 11-8. For now, I’ll just say that an event handler method is called when the button is pressed. I’ll defer a discussion of these specific event handlers until a little later. For now, it makes sense to provide them here because of their references in the previous listing.

Listing 11-9. DynamicTree Class (Part 3 of 5, Button Event Handlers)

```
95| def addEvent( self, event ) :
96| sPath = self.tree.getSelectionModel().getSelectionPath()
97| if sPath : # Use selected node
98| parent = sPath.getLastPathComponent()
99| else : # Nothing selected, use root 100| parent = self.model.getRoot()
101| kids = parent.getChildCount()
102| child = DefaultMutableTreeNode(
103| 'New node %d' % self.getSuffix()
104| )
105| self.model.insertNodeInto( child, parent, kids ) 106|
self.tree.scrollPathToVisible(
107| TreePath( child.getPath() )
108| )
```

```
109| def delEvent( self, event ) :
110| currentSelection = self.tree.getSelectionPath() 111| if currentSelection :
112| currentNode = currentSelection.getLastPathComponent() 113| if
currentNode.getParent() :
114| self.model.removeNodeFromParent( currentNode ) 115| return
116| def clsEvent( self, event ) :
117| self.model.getRoot().removeAllChildren()
118| self.model.reload()
```

The DefaultTreeModel Class

Listing 11-10 shows the makeTree() method called in line 66 of Listing 11-7. It uses instances of the DefaultMutableTreeNode class, which you saw previously. What's new in this method is how the root node of the tree is passed as an argument to the DefaultTreeModel class constructor (line 126).

Listing 11-10. DynamicTree Class (Part 4 of 5, the makeTree() Method)

```
119| def makeTree( self ) :
120| root = DefaultMutableTreeNode( 'Root Node' )
121| for name in 'Parent 1,Parent 2'.split( ',' ) :
122| here = DefaultMutableTreeNode( name )
123| for child in 'Child 1,Child 2'.split( ',' ) :
124| here.add( DefaultMutableTreeNode( child ) )
125| root.add( here )
126| model = DefaultTreeModel(
127| root,
128| treeModelListener = myTreeModelListener()
129| )
130| tree = JTree(
131| model,
132| editable = 1,
133| showsRootHandles = 1,
134| valueChanged = self.select
135| )
136| tree.getSelectionModel().setSelectionMode(
137| TreeSelectionModel.SINGLE_TREE_SELECTION
138| )
139| return tree
```

Don’t forget that the JTree instance doesn’t actually contain the data being displayed. It only provides a view of the data, which like most non-trivial Swing components, is provided by a component data model. In the case of the JTree class, it’s almost always an instance of the DefaultTreeModel class.[17]

What does the tree model provide? Well, a lot of methods are provided by this class. Unfortunately, I don’t have the time or space to investigate and describe how and when they are used. At this point, I will only take a look at a few of them, specifically the ones related to adding and removing TreeModelListeners. In fact, that’s exactly what is done in line 128. The DefaultTreeModel instance is then passed to the JTree constructor in line 131. At the same time, other keyword arguments make the tree modifiable (line 132) and make the root node handle visible (line 133). Note that the valueChanged keyword assignment argument (line 134) is used to identify the TreeSelectionListener event handler for this tree instance.

Listing 11-11 shows the TreeSelectionListener event handler method for this DynamicTree class application. It is only a little different from the one you saw earlier in this chapter. This example determines if a removable node has been selected. Only non-root nodes can be selected, so you want the Remove button to be enabled only if the user has selected a non-root node. To do this, the code determines if a node has been selected, and if so, how far down the tree this selected node is. If no node has been selected, the count will be zero. The root node, by definition, has a depth of 1. This information is used in line 148 to determine if the Remove button should be enabled.

[17]See http://docs.oracle.com/javase/8/docs/api/javax/swing/tree/DefaultTreeModel.html

Listing 11-11. DynamicTree Class (Part 5 of 5, the select() Method)

```
140| def select( self, event ) :
141| tree = event.getSource() # Get access to tree
142| count = tree.getSelectionCount()
143| sPath = tree.getSelectionModel().getSelectionPath()
144| if sPath : # How deep is the pick?
145| depth = sPath.getPathCount()
146| else : # Nothing selected
147| depth = 0
148| self.buttons[ 'Remove' ].setEnabled( count and depth > 1 )
```

The TreeModelListener Interface

Now you finally get to the topic mentioned in Listing 11-10 on line 128—the TreeModelListener interface.[18] As you have seen, listeners are used when certain events occur. In this case, the methods in this listener are called when tree nodes are added, changed, or removed. In addition, another method is called when the tree structure changes. It is important to note the difference between this listener and the TreeSelectionListener discussed earlier.

By creating a class based on this listener interface, you can monitor changes that occur in the tree. The question, though, is how? Well, the TreeModelListener class is set up as a main class and isn't nested inside the application class. This demonstrates how you can use the TreeModelEvent[19] provided with each of the TreeModelListener methods in order to detect any changes that have occurred.

Listing 11-12 shows the myTreeModelListener class, which includes some common, shared methods that locate specific parts of the tree model. They identify the parent or the specific node of interest.

Listing 11-12. The myTreeModelListener Class

```
19|class myTreeModelListener( TreeModelListener ) :
20| def getNode( self, event ) :
21| try :
22| parent = self.getParent( event )
23| node = parent.getChildAt(
24| event.getChildIndices()[ 0 ]
25| )
26| except :
27| node = event.getSource().getRoot()
28| return node
29| def getParent( self, event ) :
30| try :
31| path = event.getTreePath().getPath()
32| parent = path[ 0 ] # Start with root node 33| for node in path[ 1: ] : # Get
parent of changed node 34| parent = parent.getChildAt(
35| parent.getIndex( node )
36| )
37| except :
38| parent = None
```

```
39| return parent
40| def treeNodesChanged( self, event ) :
41| node = self.getNode( event )
42| print ' treeNodesChanged():', node.getUserObject() 43| def
treeNodesInserted( self, event ) :
44| node = self.getNode( event )
45| print 'treeNodesInserted():', node.getUserObject() 46| def
treeNodesRemoved( self, event ) :
47| print ' treeNodesRemoved(): child %d under "%s"' % ( 48|
event.getChildIndices()[ 0 ],
49| self.getParent( event )
50| )
51| def treeStructureChanged( self, event ) :
52| print 'treeStructureChanged():'
```

[18]See http://docs.oracle.com/javase/8/docs/api/javax/swing/event/TreeModelListener.ht
[19]See http://docs.oracle.com/javase/8/docs/api/javax/swing/event/TreeModelEvent.html

One question that people sometimes ask is, “How do I edit a node?” The sequence of images in Figure 11-5 demonstrates this process. You begin with the tree expanded and the root node selected. The users indicate that they want to edit the node by triple-clicking the node (three left clicks in a short time interval).[20] The node text is temporarily replaced with a text field containing the selected node text. This allows users to easily replace the text when they start typing. Should they want to cancel the edit operation, they can simply press Escape. To complete the edit, thereby replacing the original node value with the new one, users simply need to press Enter. This is the point at which the TreeModelListener treeNodesChanged() method is invoked.

[20]It’s interesting to note that the specified node is expanded or collapsed as part of the normal double-click process. That’s why the tree is collapsed as part of this edit process.

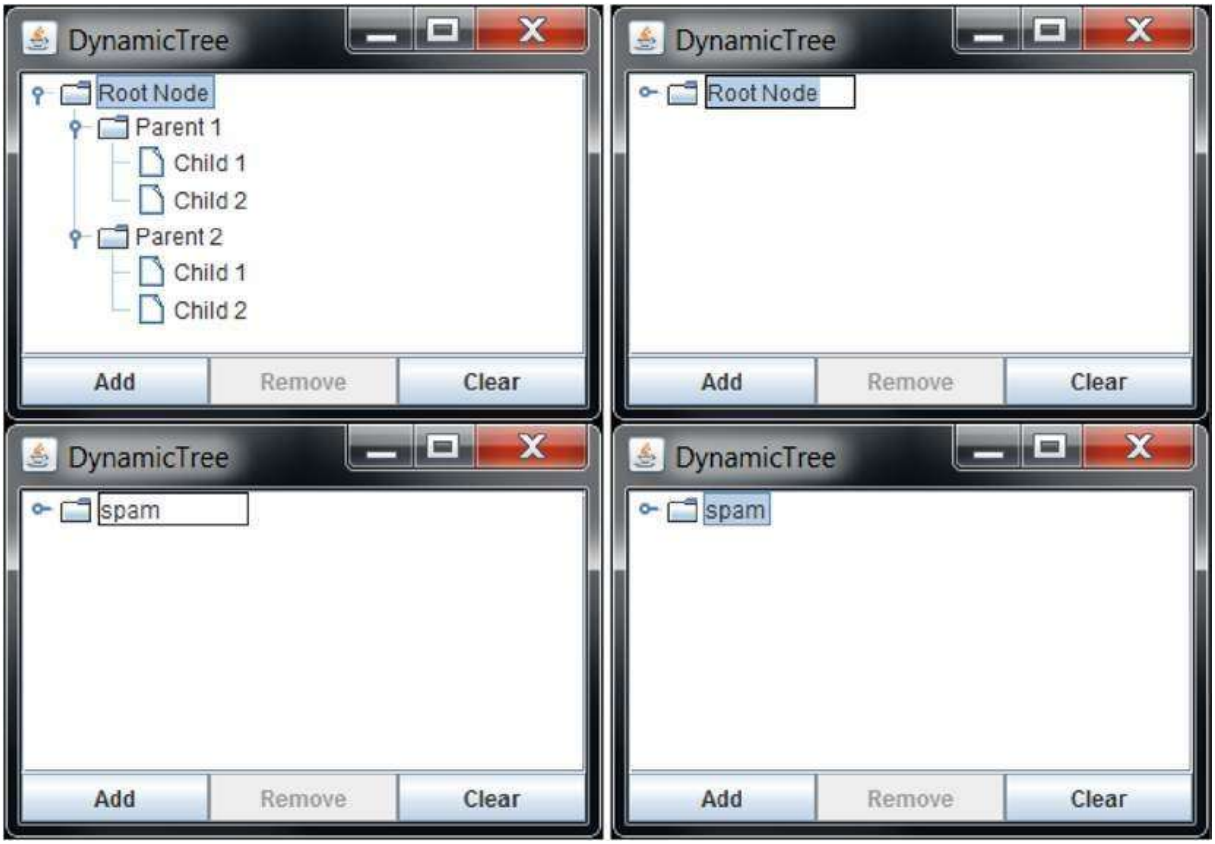


Figure 11-5. *Editing tree nodes*

How in the world would someone know this? Where is this documented? Well, if you think about it, there has to be some editor associated with the tree in order for you to modify node values. A DefaultTreeCellEditor[21] is one of the many things provided by the JTree class. This editor determines how the change is triggered, as well as the kind of edit input field to be displayed.

In addition to a default tree cell editor, the JTree constructor will also provide DefaultTreeCellRenderer[22] to determine how the tree nodes will be displayed. The fact that these support classes are part of the Swing hierarchy makes the lives of Swing application developers significantly easier.

Creating your own replacements for either of these classes is well beyond the scope of this book. However, the topic of cell renderers and editors is discussed in Chapter 12, which covers tables.

Summary

This chapter was all about creating, displaying, and manipulating trees. One of the most important points to remember about the JTree class is that it does not actually contain the data; it simply displays the hierarchical data in a form similar to an outline. The next chapter discusses the JTable class used to display data in a tabular format.

[21]See http://docs.oracle.com/javase/8/docs/api/javax/swing/tree/DefaultTreeCellEditor.l
[22]See http://docs.oracle.com/javase/8/docs/api/javax/swing/tree/DefaultTreeCellRender

Chapter 12

Motion to Take from the Table: Building Tables

One of the most useful structures in graphical applications is the table format, when information is displayed in an organized arrangement of rows and columns. In fact, you have seen lots of examples of tables throughout this book. As a side note, it is interesting how often words have multiple meanings and uses. This is especially true in the technology industry. It can also be the case with natural languages such as English. If you look for the definition of the

word “table,” you’ll find that it is often used as a noun and occasionally as a transitive verb (i.e., to place on or take off of an agenda). It is this second form that was the source for the title of this chapter,[1] just to be different. A table is such a common idea that most people would be hard pressed to remember when the concept was first described to them. This chapter is all about how to build, display, control, manage, and manipulate information in tables.

Tables Can Be Really Easy

We use tables of information all the time. In fact, it was while I was working with a simple wsadmin Jython script that listed port numbers and EndPoint names for my WebSphere Application Server environment that I started thinking about displaying the information in a table. The name of the application was ListPorts.py and Chapter 22 is all about the iterative development of a Swing application that does this.

Over time, I’ve written many different versions of that ListPorts.py script file. So

many that it's hard to keep track of them all. I wrote a script to locate the various instances of this script on my system and then created a table of the file date, size, and locations. Using this information, I produced a script that displays this information in table form, as shown in Figure 12-1.

[1]See *Robert's Rules of Order,* Article VI, Section 35.

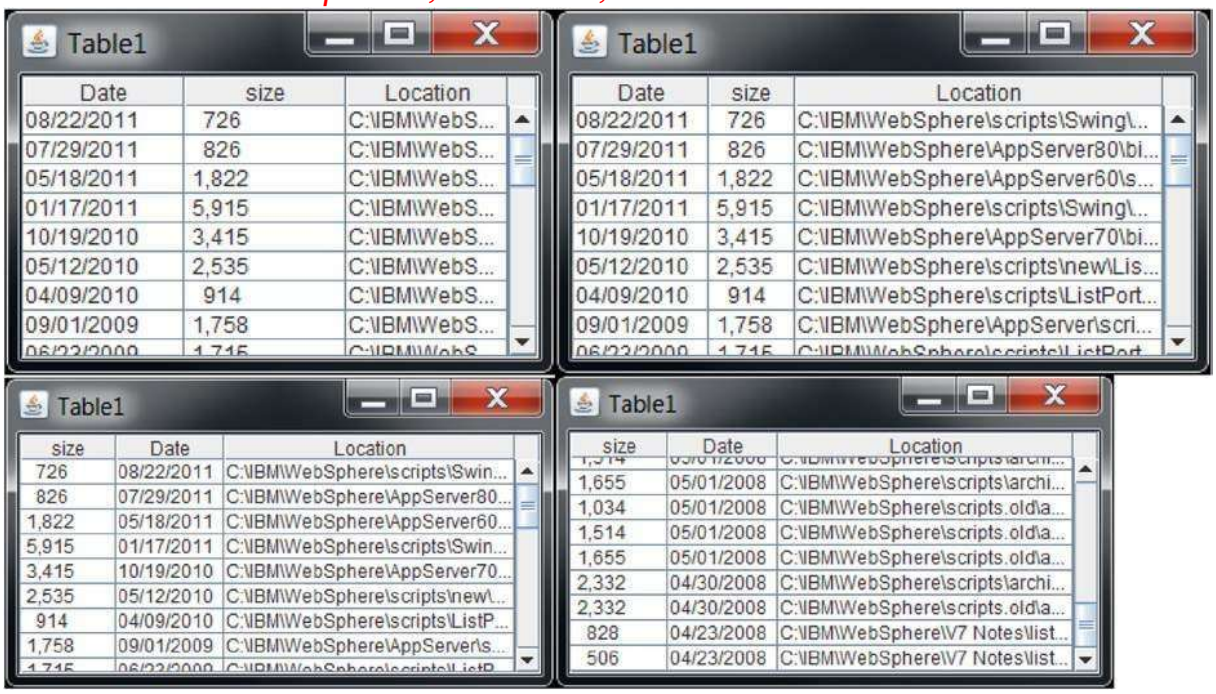


Figure 12-1. *Table1.py output images*

There are multiple images in this figure to show that the table columns can be resized and moved around easily. To resize a column, you simply drag the vertical bar between two column headings one way or the other. To reorder the columns, you drag a column heading (such as "Date") to the desired position. The other columns are reordered accordingly. All of these capabilities are provided by the JTable class.

It's also interesting to note how the right-most column, which isn't wide enough to display the complete directory path, uses an ellipse (. . .) to indicate that the data has been truncated. Just as in Listing 12-1, which shows that lines 10-39 have been left out.[2] The last image shows that the column headings stay in place, even when the data scrolls down to display other rows in the table.

Listing 12-1. Excerpts from Table1.py

7|class Table1(java.lang.Runnable) :
8| def __init__(self) :
9| info = r'''
|...
40|04/23/2008| 506|C:\IBM\WebSphere\V7 Notes\listports.py
41|'''
42| self.data = [# list comprehension
43| line.split('|') # each row is an array
44| for line in info.splitlines() # each line is a row
45| if line # ignore blank lines
46|]

[2] Of course, the Java Swing table does this automatically for you, whereas I had to do this manually in Listing 12-1. 47| def run(self) :
48| frame = JFrame(
49| 'Table1',
50| size = (300, 200),
51| locationRelativeTo = None,
52| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 53|)
54| headings = 'Date,size,Location'.split(',') 55| frame.add(
56| JScrollPane(
57| JTable(self.data, headings) 58|)
59|)
60| frame.setVisible(1)

How much code is required to provide all of this functionality? Almost none, as you can see by looking at the Table1 class in Listing 12-1. Line 57 contains the JTable[3] constructor.[4] In fact, no additional code was required to provide all of the functionality just described. It is all provided by the Java Swing class library.

Defaults Can Be Harmful to your . . . Mental Health

There's a great deal that can be said for simplicity. In other words, defaults can sometimes drive you crazy. Just remember though—you get what you pay for. By that, I mean it is extremely unlikely that you will be able to have a really useful application that uses all of the Swing class defaults.

Let's take another look at the simple table created by the Table1.py script. Did you notice that all of the columns in the first image are the same width?

Unfortunately, the data found in each column isn't all the same width. But the default settings allocate each column the same amount of column space and each column is expected to contain the same type of data (i.e., a string).

You'll investigate column manipulation a little later in this chapter (in the section entitled "Column Manipulation"). For now, don't worry about it. First, I describe other aspects of tables.

Picky, Picky, Picky. . . Selecting Parts of a Table

What parts of the table can be selected? As you can see in Figure 12-2, multiple groups of rows can be selected. This figure also shows that cell values can be edited. When a cell is edited,[5] the cell highlighting is distinctly different and the cell has a more pronounced border. One of the challenges with the defaults is that every cell within the table is considered a string, so (almost) any kind of character data can be entered. For example, while editing the selected cell in Figure 12-2, you could easily enter any text you want. The caveat is that some characters will cause the edit mode to terminate (such as the Tab and Enter keys).

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

[4]As you have seen before, it is a really good idea to place a potentially large component, such as a JTable instance, in a JScrollPane, as shown in lines 56-58.

[5]Either by double-clicking on the cell or positioning the focus on a cell using cursor control keys and pressing the spacebar.

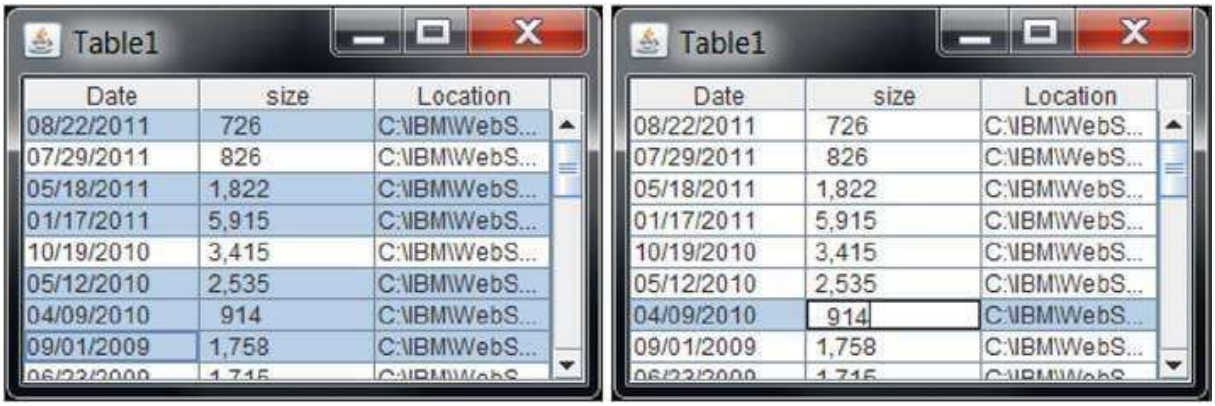


Figure 12-2. *Table1 output: row selection and editing*

This provides you with a quick overview about some of the aspects related to Swing tables. The rest of the chapter discusses tailoring them to act in ways that make sense.

Row, Row, Row Your . . . Table? Working with Rows

If you play with the Table1.py application, you'll see that when you click on any cell in the table, the whole row is selected. In Figure 12-2, you can see that multiple groups of rows can, by default, be selected.[6] How can you change the row selection behavior? That's easy; the JTable class has a row selectionMode property that uses the same constants used when you were working with lists and trees. Table 12-1 describes the possible selection values.

Table 12-1. *Selection Mode Constants*

Selection Mode Constant	**Description**
SINGLE_SELECTION	Zero or one row can be selected.
SINGLE_INTERVAL_SELECTION	One contiguous group of rows can be selected.
MULTIPLE_INTERVAL_SELECTION	The default mode allows multiple groups of rows to be selected.

You can see, by looking at the JTable class Java documentation, that there is a setter but not a getter for the selectionMode property.[7] Don't be confused by the fact that there is a getter and setter for the selectionModel property; this is not what we're talking about. The fact that the class provides only a setter method means that you can use the setter (table.setSelectionMode()) method or you can specify the selectionMode keyword argument when the table is constructed, as shown in Listing 12-2, which is from Table2.py.

[6] Press and hold the Ctrl key as you use the mouse to select or deselect an individual row. If you hold the Shift key while making a selection, you can select multiple contiguous rows.

[7] I looked all over, but have yet to find any explanation for the lack of getter for this table property.

Listing 12-2. Specifying the selectionMode Using a Keyword Argument

```
6|from javax.swing import ListSelectionModel as LSM | ...
56| frame.add(
57| JScrollPane(
58| JTable(
59| self.data,
```

```
60| headings,
61| selectionMode = LSM.SINGLE_SELECTION 62| )
63| )
64| )
```

Does the presence of the selectionMode attribute in the JTable class mean that you can access its value (the operation normally performed by a getter method)? No, it doesn’t. If you try to do that, you’ll get an exception like the one shown in Listing 12-3.

Listing 12-3. Write-Only Attributes

```
wsadmin>from javax.swing import JTable
wsadmin>
wsadmin>data = [ [ '1.1', '1.2' ], [ '2.1', '2.2' ] ] wsadmin>head = [ 'Uno', 'Dos' ]
wsadmin>
wsadmin>table = JTable( data, head )
wsadmin>
wsadmin>table.selectionMode
WASX7015E: Exception running command: "table.selectionMode";

exception information:
com.ibm.bsf.BSFException: exception from Jython:

Traceback (innermost last):
File "<input>", line 1, in ?
AttributeError: write-only attr: selectionMode

wsadmin>
```

What does that mean for your applications? Basically, it means that if your application needs to display and modify this attribute, it needs to provide some way to maintain the property value as well as monitor any changes made to it.

Selecting Columns

The JTable class has columnSelectionAllowed and rowSelectionAllowed attributes. Interestingly enough, the former doesn’t appear on the Javadoc page of the JTable class, but it does exist, as you can see in Listing 12-4.

Listing 12-4. JTable selectionAllowed Attributes

```
wsadmin>from javax.swing import JTable
wsadmin>
wsadmin>classInfo( JTable, attr = 'SelectionAllowed' ) javax.swing.JTable

columnSelectionAllowed, rowSelectionAllowed | 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.swing.event.TableModelListener
| | java.util.EventListener
| javax.swing.Scrollable
| javax.swing.event.TableColumnModelListener | | java.util.EventListener
| javax.swing.event.ListSelectionListener
| | java.util.EventListener
| javax.swing.event.CellEditorListener
| | java.util.EventListener
| javax.accessibility.Accessible
| javax.swing.event.RowSorterListener
| | java.util.EventListener
wsadmin>
```

Selecting Individual Cells

If you look at the JTable documentation, you should be able to find the cellSelectionEnabled attribute. Notice that it has been obsolete since version 1.3. How do you allow your table cells to be selected? Well, a cell can be selectable only when both rowSelectionAllowed and columnSelectionAllowed are true.

Interestingly enough, both a getter and a setter are provided for the cellSelectionEnabled attribute. The setter is used to assign the specified (Boolean) value to the rowSelectionAllowed and columnSelectionAllowed attributes, and the getter returns true only if both attributes are true. Table 12-2 shows the values returned by the rowSelectionAllowed, columnSelectionAllowed, and cellSelectionEnabled getter methods after the statement or method call listed in column 1 of the table has been executed.

Table 12-2. *Row, Column, and Cell Selection Values*
Statement Executed Row Col Cell JTable() 1 0 0 setColumnSelectionAllowed(1) 1 1 1 setRowSelectionAllowed(0) 0 1 0 setRowSelectionAllowed(1) 1 1 1 setCellSelectionEnabled(0) 0 0 0 setCellSelectionEnabled(1) 1 1 1

One question that might come to mind, especially if you have seen the TableSelectionDemo application[8] from the Java Swing Tutorial website, is "Aren't the Row, Column, and Cell selection attributes affected by the selection mode?" The answer is no, they aren't. The developer of that application has added this association, but this is not something that the JTable class does. So, the values in Table 12-2 are the same, regardless of the table selectionMode setting. However, this does point out how developers can choose to enhance classes to provide functionality for their applications.

I Am the Very Model of a Modern Major General: Table Models[9]

One of the many things that you haven't read about yet about JTable instances is the fact that the displayed information isn't stored in the JTable. The table actually provides a view to the data. The JTable class isn't the first complex Swing class that you've seen that does this. In Chapter 11, you learned how the JTree class used a data model to hold the information and the class used to display it.

What are the advantages to separating it like this? For one, it allows the view (the JTable class) to do things like determine the column order to be displayed and to rearrange it, if necessary. In case you are interested, you can disable the movement of columns. Listing 12-5 shows how you can use the JTable getTableHeader() method to access the JTableHeader[10] instance for this table. You then call the setReorderingAllowed() method with a value of 0 (false) to indicate that column reordering is not allowed.

Listing 12-5. Disabling Column Reordering
```
table.getTableHeader().setReorderingAllowed( 0 )
```
Types of Table Models

What kinds of table models exist in the Swing class hierarchy for your applications to use? There is an AbstractTableModel[11] class as well as a DefaultTableModel.[12] I'll let you guess which class is used as the default should

one not be provided for your JTable constructor call.

[8] See http://docs.oracle.com/javase/tutorial/uiswing/examples/components/TableSelecti components/TableSelectionDemo.java.
[9]With thanks and homage to Gilbert and Sullivan for their marvelous works.
[10]See http://docs.oracle.com/javase/8/docs/api/javax/swing/table/JTableHeader.html.
[11]See http://docs.oracle.com/javase/8/docs/api/javax/swing/table/AbstractTableModel.h
[12]See http://docs.oracle.com/javase/8/docs/api/javax/swing/table/DefaultTableModel.ht

Looking at the DefaultTableModel class documentation, you'll find more than two dozen methods, as well as another dozen inherited from the AbstractTableModel base class. What do these methods allow you to do with your table data? Well, from the user interface perspective, you can do things like add, move, and remove rows of data. One of the most important methods, however, is the isCellEditable() method, which is used by the JTable class to determine whether users are allowed to modify data in the specified cell. Listing 12-6 shows just how easily this can be used.

Listing 12-6. Creating a Read-Only TableModel Class

```
9|class roTM( DefaultTableModel ) :
10| def __init__( self, data, headings ) :
11| DefaultTableModel.__init__( self, data, headings ) 12| def isCellEditable(
self, row, col ) :
13| return 0
14|class Table3( java.lang.Runnable ) :

|...
62| frame.add(
63| JScrollPane(
64| JTable(
65| roTM( self.data, headings ),
66| selectionMode = LSM.SINGLE_SELECTION
67| )
68| )
```

69|)

The interesting point about the table model is how it is used, automatically, by the JTable class to determine what information is to be displayed and how. How does the JTable class determine how the data is displayed? It calls the tableModel getColumnClass(...) method, inherited from the base AbstractTableModel class, which you can and should override.

Wait a minute, what does this mean? Previously, you learned that the default data type for each cell is a string. Really, it's an *object* that's represented as a string by default. You can and should provide your own table model getColumnClass(...) instance to identify the appropriate data type for each column. Why a column? One simplifying implementation choice made by the Swing designers was to guarantee that all data in a JTable column would be of a single type. This shouldn't be too much of a restriction though since you can choose to identify the data type of the column as an object and figure out how to display and manipulate the actual data.

Listing 12-7 shows a relatively simple table model descendent class that only allows the values in column one (the second column)[13] to be modified (every other column is read-only). It also includes the getColumnClass(...) method, which identifies the appropriate data type for each column.

Listing 12-7. My Table Model Class

13|class myTM(DefaultTableModel) :
14| def __init__(self, data, headings) :
15| info = []
16| df = DateFormat.getDateInstance(DateFormat.SHORT) 17| for date, size, path in data :
18| info.append(
19| [
20| df.parse(date),
21| Integer(size.strip().replace(',', '')), 22| String(path)
23|]
24|)
25| DefaultTableModel.__init__(self, info, headings) 26| def getColumnClass(self, col) :
27| return [Date, Integer, String][col]

```
28| def isCellEditable( self, row, col ) :
29| return col == 1
```

[13]Remember that Jython and Java both use zero origin array indexing.

Figure 12-3, shows the resulting output of this application (the source for which is in Table4.py). Is the way the information is displayed in the table what you expected? Perhaps, perhaps not. Did you notice how the middle column is now right-aligned, whereas all of the others are left-aligned? Why do you think that is?

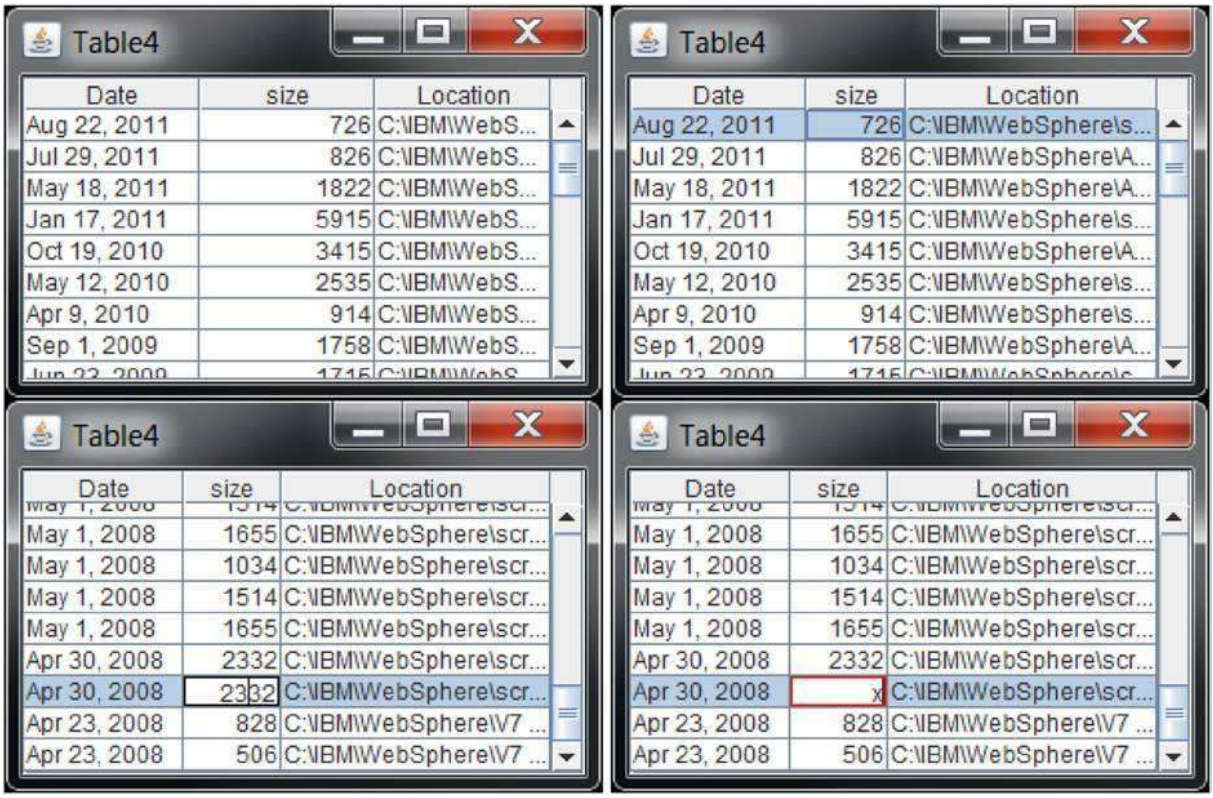


Figure 12-3. *Table4 output*

Cell Renderers

Another important aspect of the complex JTable class is the fact that each data type has a renderer instance that determines how the information should be presented to the user. Unless you provide one, an instance of the DefaultTableCellRenderer[14] class will be used. It is this renderer instance that displays the Integer values in the middle column using right justification. What

kind of cell renderers are provided by Swing? Table 12-3 provides this information.

Table 12-3. *Data Type-Specific Cell Renderers* **Data Type** Boolean Number Double Float Date Icon Object

Renderer Description Displayed as a check box. Displayed as a right-justified label. Like Number, but the value is provided by a NumberFormat instance, which is locale-specific. Same as Double. Displayed as a left-justified label formatted by a DateFormat instance using the SHORT variation. Displayed as a centered label. Displayed as a left-justified string in a label field.

Figure 12-4 shows some application images using various data types. For this application, the default isCellEditable(...) method, that is, the one provided by DefaultTableModel, returns true for every cell. The first image shows the initial column widths, the next image shows how a simple date value can be entered, and the last two images show how a value of 1234567890 is displayed by the integer and float renderers. Notice how commas are present in the Float column, but not in the Integer column.

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

Figure 12-4. *Table5 output: data type rendering*

Custom Cell Renderers

I have a question for you to consider: what do you think about representing a Boolean value as a check box? This might make sense, at least as long as the data is editable. It is also very likely that Boolean values will be editable.

While working on this chapter, I wondered if the selection model had any impact on the row, column, and cell selection values. So I wrote a little Swing application to answer this question. The results of this test are displayed in Table 12-2.

Three versions of the application can be found in the TableSelection#.py script files, where # is 1, 2, or 3. The first version uses the simple expedient of not providing a TableModel, so all of the table cells default to being an object, the default representation of which is a left-justified string value.

A reasonable improvement or iteration is to provide a simple table model that identifies each cell as being of type Boolean, which results in the values being displayed using a check box centered in the cell. That's exactly what the other versions of the application do. This is the point at which I wondered what it would take to provide my own renderer, one that displays the Boolean value as a 0 or a 1. That's what the third version of the application does. Since a simple table model class already exists to identify each cell as a Boolean, and since a default renderer for type Boolean already exists (see Table 12-3), you have to figure out how to replace this default Boolean renderer with one of your own, specifically one that displays each value as a digit, centered in the cell. Figure 12-5 shows images from all three verisons of this application.

Figure 12-5. *TableSelection output with default and custom renderer*

The first application, TableSelection1.py, simply creates each JTable instance using the defaults. The second image, generated by TableSelection2.py, uses a table model that identifies each cell as being of type Boolean. The last image, generated by TableSelection3.py, replaces the default renderer of Boolean values with the one shown in Listing 12-8 (line 81).

Listing 12-8: Custom Renderer for Boolean Values

```
| 17|class boolRenderer( DefaultTableCellRenderer ) :                    |
| 18|    def __init__( self ) :                                           |
| 19|        self.result = JLabel(                                        |
| 20|            horizontalAlignment = SwingConstants.CENTER             |
| 21|        )                                                            |
| 22|    def getTableCellRendererComponent(                               |
| 23|        self,                                                        |
| 24|        table,               # JTable  - table containing value     |
| 25|        value,               # Object  - value being rendered       |
| 26|        isSelected,          # boolean - Is value selected?         |
| 27|        hasFocus,            # boolean - Does this cell have focus?|
| 28|        rowIndex,            # int     - Row # (0..N)               |
| 29|        vColIndex            # int     - Col # (0..N)               |
| 30|    ) :                                                              |
| 31|        self.result.setText( value.toString() )                      |
| 32|        return self.result                                           |
|...| ...                                                                 |
| 81|        bTable.setDefaultRenderer( Boolean, boolRenderer() )         |
|...| ...                                                                 |
```

A Few Cautions. . .

As you can see in Listing 12-8 as well as in the Javadoc for the DefaultTableCellRenderer class and the TableCellRenderer interface[15] on which it is based, only one method class exists. When providing a custom TableCellRenderer, it is important that you realize the potential performance impacts that can occur because of how the getTableCellRendererComponent(...) method is used. So, take a few moments to read the “Implementation Note” on the DefaultTableCellRenderer documentation page.

One of the important things that a custom TableCellRenderer should do is reuse a component instance, instead of instantiating a new one on each call to the getTableCellRendererComponent(...) method. Listing 12-8 shows how this class creates a common JLabel component instance in the constructor and reuses it on each use (lines 31 and 32).

It’s interesting to note that if, for some reason, you don’t like the column-centric technique for choosing the cell renderer, it is possible to create a table class based on JTable, which provides a getCellRenderer(...) method that uses a different technique. However, as you might imagine, the difficulties associated with a drastic approach such as this is not something to be underestimated. Although possible, it should rarely be considered a viable option.

[15]See http://docs.oracle.com/javase/8/docs/api/javax/swing/table/TableCellRenderer.htn

Which Cell Renderer to Use?

For tables that contain a variety of data types, it is quite possible that multiple cell renderer instances exist. When this occurs, it might not be clear how the table determines which one to use. How is this determined? Well, if a renderer has been defined for the specified table column, it takes precedence. If not, the renderer for the data type for that column will be used. So, if you want to use a custom renderer, you have to decide if you want all cells of the same type to be rendered in the same fashion or if you only want the data in a specific column to use this custom renderer. This should help you decide which technique to use to specify the custom renderer.

What would be a good example of this kind of decision? Consider a situation whereby your table contains a number of columns containing floating-point values. You might want some of these rendered using one format (such as a percentage, with a specific number of decimal places) and another column as a completely different format (such as in currency).

You just saw how to set up a cell renderer for a specific data type. How do you set up a column-specific one? First, you need to access the TableColumn[16] instance for the specific table column in question. Fortunately, this is easily done by using the getColumn(...) method of the TableColumnModel[17] instance that is used to hold the column information for the specific table instance. This TableColumn class includes a setCellRenderer(...) method, which allows you to specify the renderer instance for this particular table column. Listing 12-9 shows how easy this is. Please note how the col variable identifies the column number for which this renderer instance is being specified.

Listing 12-9. Defining a Column-Specific Cell Renderer

```
| t = JTable( ... )                                                    |
| t.getColumnModel().getColumn( col ).setCellRenderer( myRenderer() )  |
```

Don’t JLabel Me

The custom renderer used in Listing 12-8 used an instance of the JLabel class. Can some other component class be used? Certainly! In fact, it appears to be quite reasonable to use something like the JFormattedTextField class that you first saw in Chapter 7.

Figure 12-6 shows some sample images of the first attempt to use a

JFormattedTextField as the component returned for the custom renderer of the last column of the table. This column, which contains values of type Double, has exactly two digits after the decimal point. Notice how the editor displays the cell data.

[16]See http://docs.oracle.com/javase/8/docs/api/javax/swing/table/TableColumn.html.
[17]See http://docs.oracle.com/javase/8/docs/api/javax/swing/table/TableColumnModel.ht

Figure 12-6. *Table6a output with a custom renderer*

Unfortunately, it isn't quite right, is it? For example, when the first row is selected, the last cell isn't highlighted the same way that the rest of the row is. Additionally, the numeric values, although they are displayed with exactly two decimal places, are left aligned in the cell. Nor does the cell have a border when it is selected, as the other cells in the row do. Listing 12-10 shows the first attempt at a renderer, which uses a JFormattedTextField instance to display the values.

Listing 12-10. Table6a.py: First Attempt at Custom Renderer

```
18|class myRenderer( DefaultTableCellRenderer ) :
19| def __init__( self ) :
20| nf = NumberFormat.getInstance()
21| nf.setMinimumFractionDigits( 2 )
```

```
22| nf.setMaximumFractionDigits( 2 )
23| self.result = JFormattedTextField( nf )
24| def getTableCellRendererComponent(
25| self,
26| table, # JTable - table containing value
27| value, # Object - value being rendered
28| isSelected, # boolean - Is value selected?
29| hasFocus, # boolean - Does this cell have focus?
30| rowIndex, # int - Row # (0..N)
31| vColIndex # int - Col # (0..N)
32| ) :
33| self.result.setValue( value )
34| return self.result
|...
74| model = myTM( self.data, headings )
75| table = JTable(
76| model,
77| selectionMode = ListSelectionModel.SINGLE_SELECTION 78| )
79| table.getColumnModel().getColumn(
80| model.getColumnCount() - 1 # i.e., last column 81| ).setCellRenderer(
82| myRenderer()
83| )
```

Let's start by fixing the alignment problem. A little investigation into the JFormattedTextField Javadoc[18] shows that it inherits the horizontal alignment property from the JTextField class.[19] This allows you to add a horizontalAlignment keyword argument to the JFormattedTextField constructor call. Listing 12-11 shows this minor modification.

Listing 12-11. Table6b.py with Horizontal Alignment

```
19|class myRenderer( DefaultTableCellRenderer ) :
20| def __init__( self ) :
21| nf = NumberFormat.getInstance()
22| nf.setMinimumFractionDigits( 2 )
23| nf.setMaximumFractionDigits( 2 )
24| self.result = JFormattedTextField(
25| nf,
26| horizontalAlignment = JTextField.RIGHT
```

27|)

What does this do to the application's output? Figure 12-7 shows how this change affects the cells' appearance. This does improve things a little bit, but additional challenges remain.

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

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

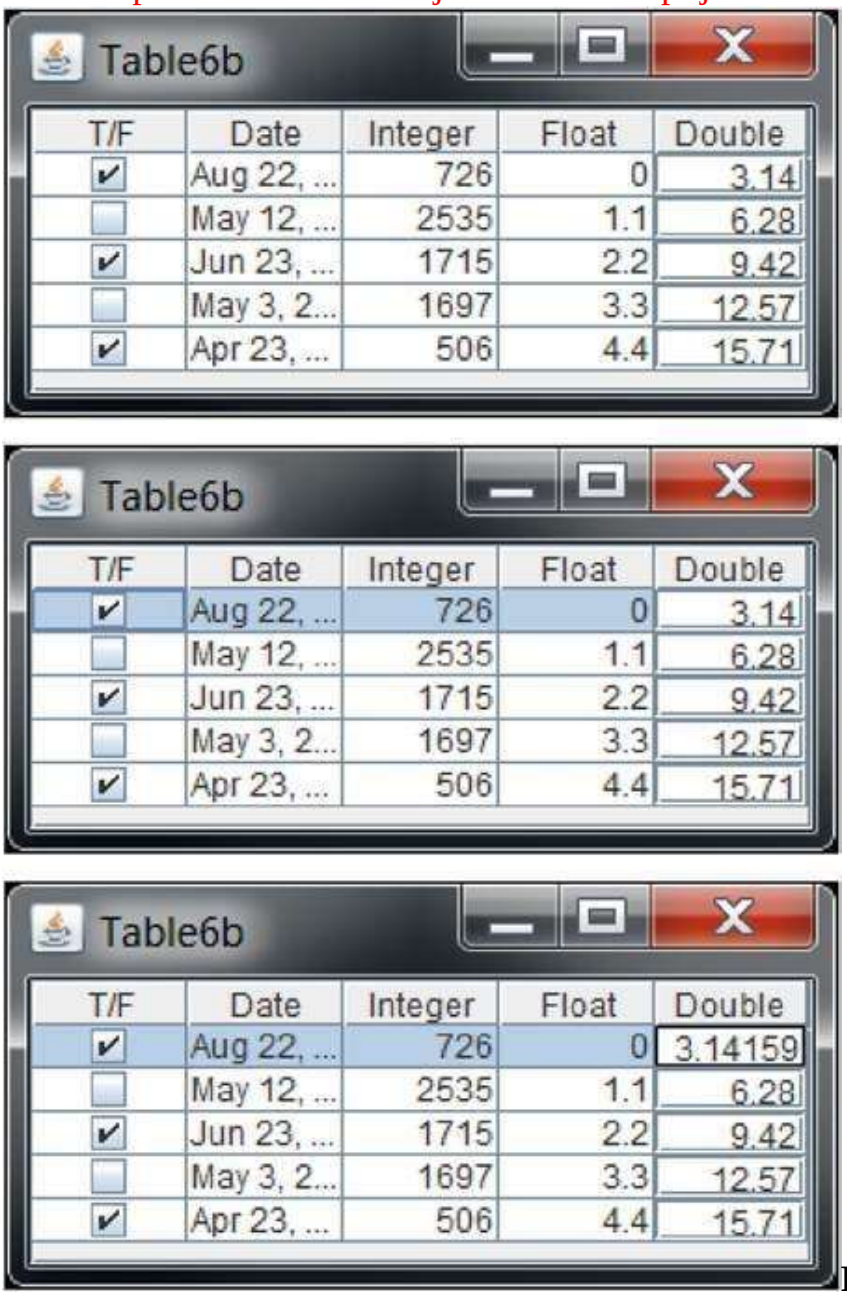


Figure 12-7.

FormattedTextField with horizontal alignment fix

You can use the fact that the DefaultTableCellRenderer already does a lot of the work for you to resolve another issue. With a little bit of code, you should be able to take advantage of this fact. Listing 12-12 shows one way to use the DefaultTableCellRenderer class to solve some of the custom rendering issues.
Listing 12-12. Table6c.py: One Solution to the Custom Rendering Issues

```
19|class myRenderer( DefaultTableCellRenderer ) :
20| def __init__( self ) :
21| nf = NumberFormat.getInstance()
22| nf.setMinimumFractionDigits( 2 )
23| nf.setMaximumFractionDigits( 2 )
24| self.result = JFormattedTextField(
25| nf,
26| border = None,
27| horizontalAlignment = JTextField.RIGHT
28| )
29| self.DTCR = DefaultTableCellRenderer()
30| def getTableCellRendererComponent(
31| self,
32| table, # JTable - table containing value
33| value, # Object - value being rendered
34| isSelected, # boolean - Is value selected?
35| hasFocus, # boolean - Does this cell have focus?
36| row, # int - Row # (0..N)
37| col # int - Col # (0..N)
38| ) :
39| comp = self.DTCR.getTableCellRendererComponent(
40| table, value, isSelected, hasFocus, row, col
41| )
42| result = self.result
43| result.setForeground( comp.getForeground() )
44| result.setBackground( comp.getBackground() )
45| result.setBorder( comp.getBorder() )
46| result.setValue( value )
47| return result
```

What does this mean as far as this output is concerned? Well, as you can see in

Figure 12-8 when a row is selected, all of the cells in the row have the same color scheme. The second image shows that when a cell in the last column is selected, a slightly darker border is visible. And finally, the last image shows that when an invalid value is specified, this fact is highlighted in a very obvious fashion. So it appears that using the default cell renderer is a viable solution for some of these display issues.

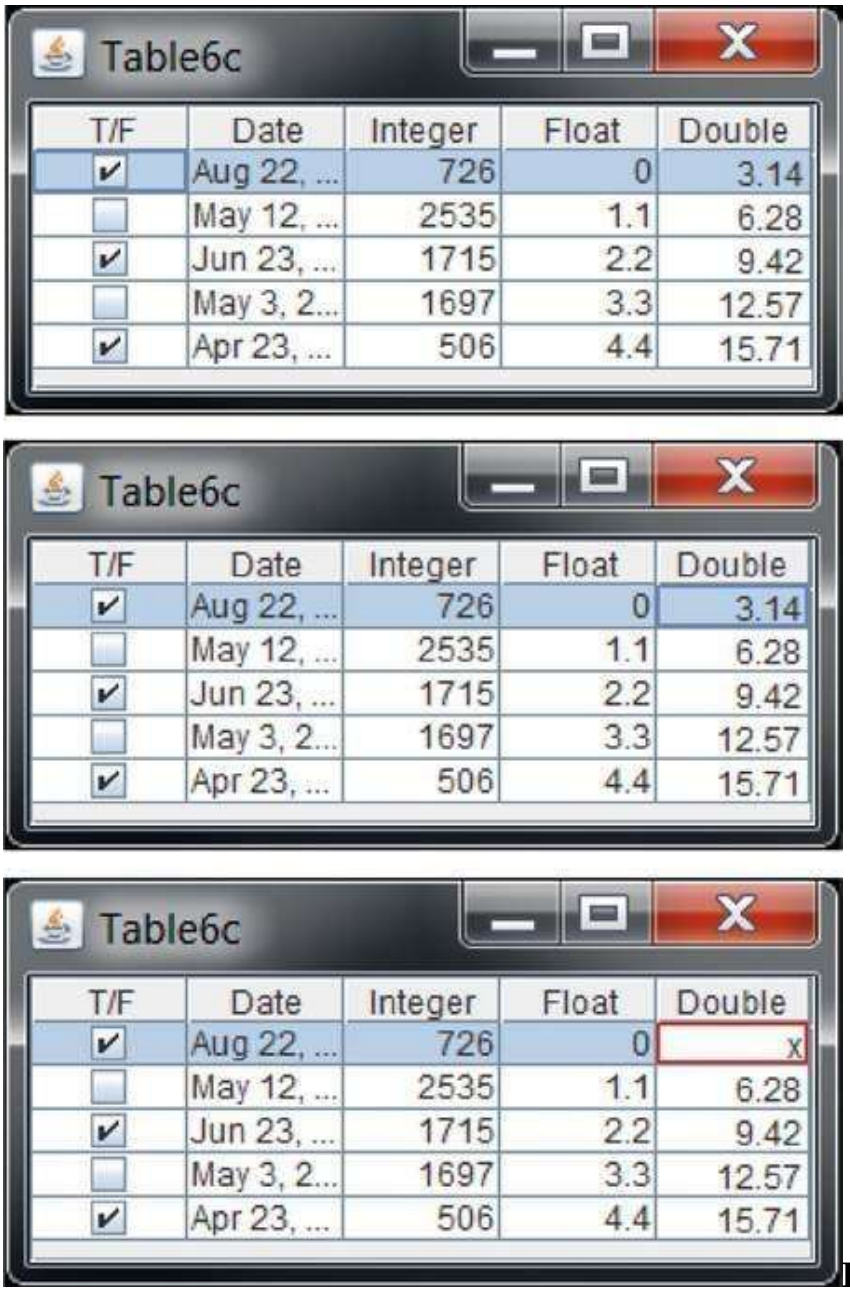


Figure 12-8. *Table6c output showing expected results*

Using Cell Editors

Another important part of displaying information in a tabular form is the fact that you'll often want to allow the user to modify the information. This is where a cell editor comes into play. For this purpose, the Swing hierarchy includes a CellEditor,[20] a TableCellEditor[21] interface, and a DefaultCellEditor[22] class. Which of these are important for providing your own cell editor? Well, the base CellEditor interface provides a group of methods that are rarely replaced or overridden. So, it is best if you just leave them alone.

On the other hand, the TableCellEditor interface provides a method called getTableCellEditorComponent(...) that enables developers to provide or specify an editor component with a table.

[20] See http://docs.oracle.com/javase/8/docs/api/javax/swing/CellEditor.html.
[21]See http://docs.oracle.com/javase/8/docs/api/javax/swing/table/TableCellEditor.html.
[22]See http://docs.oracle.com/javase/8/docs/api/javax/swing/DefaultCellEditor.html.

Boolean Cell Editors

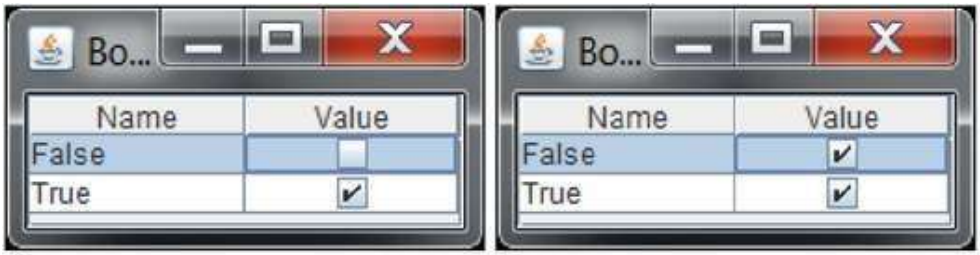
You've already seen some of the simple, default, cell editors. Consider for a moment what you are doing when you have a table display Boolean values with a check box. Every time you toggle the check box, you are in fact editing the value in that cell. That's why, if you have a table model class in your application and your table includes Boolean values, the setValueAt(...) method must save the value as a Boolean, and not as the Integer that is provided by the setValueAt(...) method. Listing 12-13 shows the simple table model class that was used to verify this fact. Notice how the setValueAt(...) method, in lines 27-29, displays the value and its data type before using the Boolean constructor to save the appropriate kind of value.

Listing 12-13. BoolEdit.py Table Model Class

```
13|class tm( DefaultTableModel ) :
14| def __init__( self ) :
15| head = 'Name,Value'.split( ',' )
16| self.data = [
```

```
17| [ 'False', Boolean( 0 ) ],
18| [ 'True' , Boolean( 1 ) ]
19| ]
20| DefaultTableModel.__init__( self, self.data, head )
21| def getColumnClass( self, col ) :
22| return [ String, Boolean ][ col ]
23| def isCellEditable( self, row, col ) :
24| return col == 1
25| def getValueAt( self, row, col ) :
26| return self.data[ row ][ col ]
27| def setValueAt( self, value, row, col ) :
28| print 'tm.setValueAt():', value, type( value )
29| self.data[ row ][ col ] = Boolean( value )
```

Figure 12-9 shows a couple of images of the application, as well as the output that is generated by the setValueAt(...) method. Note how the value that is provided to the method is a Python integer. This is why you need to convert it to a Boolean before saving it in the data array instance.




```
Press <Enter> to terminate the application:
tm.setValueAt(): 1 org.python.core.PyInteger
tm.setValueAt(): 0 org.python.core.PyInteger
tm.setValueAt(): 1 org.python.core.PyInteger
```

Figure 12-9. *BoolEdit sample output*

Did you notice that you didn’t even have to provide a custom or specialized editor for this example? In Table 12-3, you can see how Swing provides a renderer for Boolean types. Since the possible values are very limited, you don’t need to worry about an editor for Boolean types.

Numeric Cell Editors

What about the numeric types? What do the default renderers and editors for these types provide? Figure 12-10 shows some images from the sample NumbEdit.py application that show the minimum values for each type in the top rows and the maximum values for each type in the bottom rows.

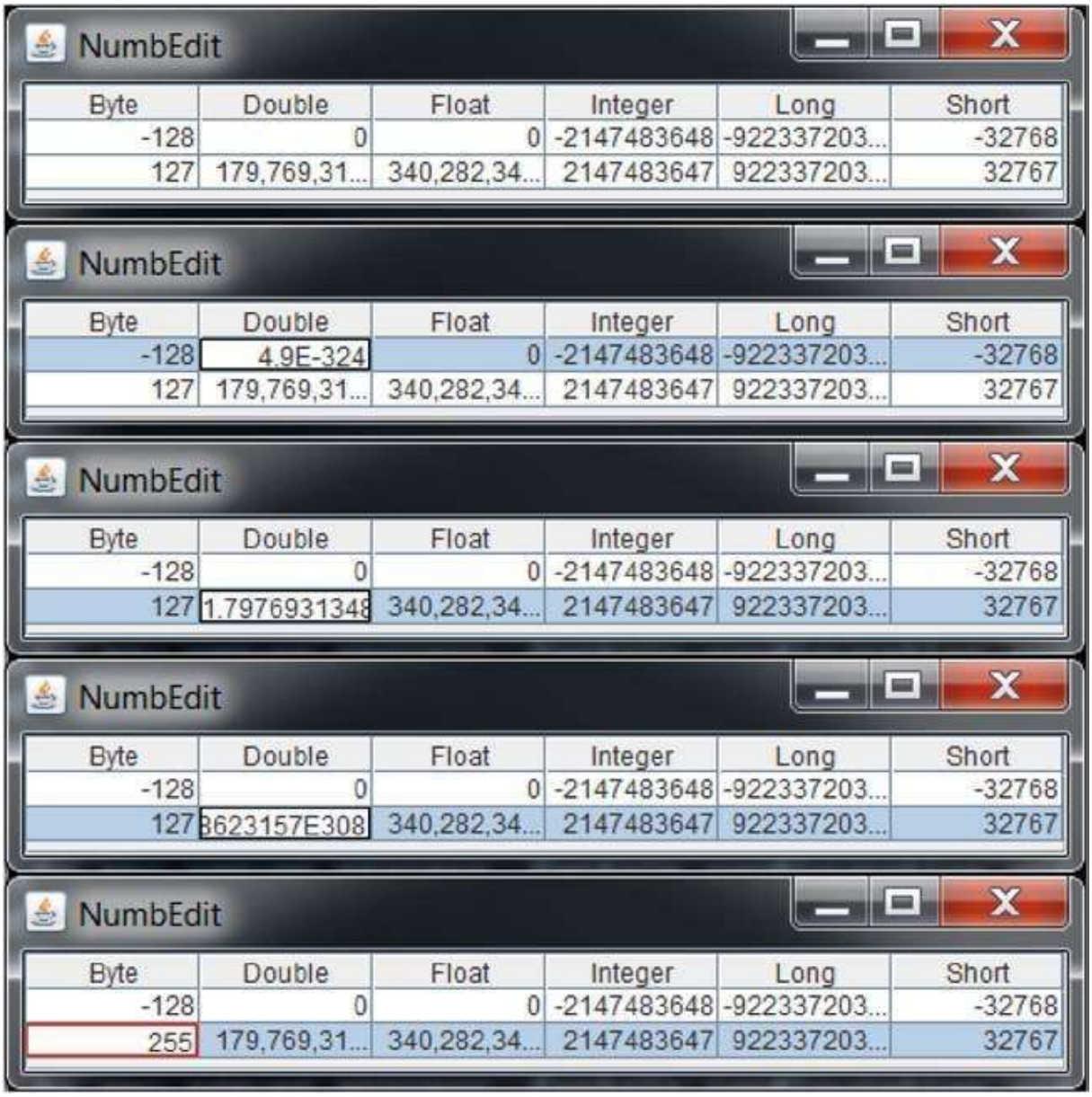


Figure 12-10. *NumbEdit.py output: default numeric renderers and editors*

At first glance, it may seem that the minimum Double and Float values should be something other than zero. The fact that they are non-zero, just very small, as is evident when the default editor for either value is used. The second image shows that the minimum Double value is 4.9E-324, which is very close to 0. So

close, in fact, that the default renderer displays a value of zero instead. The third and fourth images show that the largest Double value has too many significant digits to be displayed in the available space (it has a value of 1.7976931348623157E308). so elipses are used to indicate that the values have been truncated.

The last image shows what happens when a user attempts to enter a value that is outside of the range of valid values. The default editor highlights the cell using a red border, and the user is unable to move the focus away from the cell.
One question that frequently comes to mind relates to the setValueAt(...) method that is called to save userspecified values of various types. Java examples frequently use a switch statement or nested if-then-else statements to deal with this type of thing. Jython, on the other hand, can make this kind of thing trivial, as shown in Listing 12-14.[23]

Listing 12-14. NumbEdit.py getValueAt(...) and setValueAt.(...) Methods

```
11|class tm( DefaultTableModel ) :
| ...
29| def getColumnClass( self, col ) :
30| return [ Byte, Double, Float, Integer, Long, Short ][ col ]
31| def getValueAt( self, row, col ) :
32| return self.data[ row ][ col ]
33| def setValueAt( self, value, row, col ) :
34| print 'tm.setValueAt():', value, type( value )
35| Type = self.getColumnClass( col )
36| self.data[ row ][ col ] = Type( value )
```

All of this capability is provided by the default numeric renderers and editors, which is actually pretty neat. Can you think of a situation where you might need to provide your own numeric editor? What about the situation where you want to allow a specific range of values? You might be able to use a numeric JSpinner as was discussed in Chapter 7, but let’s start with something a little simpler. What does it take to provide a custom numeric editor that verifies the user input before it is accepted?

Custom Numeric Cell Editors

First, you need to decide the range of value with which you want to work. For

example, say your application wants to work with a simple concept like TCP/IP port numbers, which range from 0 to 65535.[24] Since the maximum value is larger than Short.MAX_VALUE (i.e., 32767), you have to use an Integer type field, and the editor has to verify the user input before accepting it.

How, exactly, is the custom editor class supposed to check the user input? Well, first, you have to realize that the editor has to provide some sort of input field. If you base the custom editor on DefaultCellEditor,[22] you can use the fact that one of the constructors requires a JTextField. Listing 12-15 shows one way of doing this.

Listing 12-15. portEditor Class and Table Model from PortEdit.py

```
14|class portEditor( DefaultCellEditor ) :
15| def __init__( self ) :
16| self.textfield = JTextField(
17| horizontalAlignment = JTextField.RIGHT
18| )
19| DefaultCellEditor.__init__( self, self.textfield )
20| def stopCellEditing( self ) :
21| try :
22| val = Integer.valueOf( self.textfield.getText() ) 23| if not ( -1 < val < 65536 )
:
24| raise NumberFormatException()
25| result = DefaultCellEditor.stopCellEditing( self ) 26| except :
27| self.textfield.setBorder( LineBorder( Color.red ) ) 28| result = 0 # false
29| return result
30|class tm( DefaultTableModel ) :
31| def __init__( self ) :
32| head = 'Name,Value'.split( ',' )
33| self.data = [
34| [ 'Min Port', Integer( 0 ) ],
35| [ 'Max Port', Integer( 65535 ) ]
36| ]
37| DefaultTableModel.__init__( self, self.data, head ) 38| def isCellEditable(
self, row, col ) :
39| return col == 1
40| def getColumnClass( self, col ) :
41| return [ String, Integer ][ col ]
```

```
42| def getValueAt( self, row, col ) :
43| return self.data[ row ][ col ]
44| def setValueAt( self, value, row, col ) :
45| print 'tm.setValueAt():', value, type( value ) 46| self.data[ row ][ col ] =
Integer( value ) |...
55| table = JTable( tm() )
56| table.setDefaultEditor( Integer, portEditor() )
```

[23]Again, this script uses a temporary variable, Type, so the line isn't too long.
[24]Ignore the fact that a port number is in fact an *unsigned* short integer, which Java doesn't support.

This example shows how easy it is (line 27) to duplicate the technique used by DefaultCellEditors (that is, highlight the cell with a red border). It's also good to note that since the custom cell editor is based on a JTextField, the value provided to the setValueAt(...) table model method (lines 44-47) is a string, not an integer as you might expect.

How hard do you think it would be to change the code so it would display the port number using a different format (for example, in hexadecimal, octal, or even using a comma for values larger than 999)? Think about this for a moment. One significant decision you need to make is how, exactly, you want to represent port values. If you want to display (render) them using hexadecimal characters, should they be maintained, verified, and edited as character strings, or do you need to work with integer values elsewhere in the application?

These are the kinds of things that you need to consider as an application developer. How should the values be displayed (rendered) for the users? How do you expect the users to provided new values (editor)? What do you need to do to verify user input? Just think of the fun you have ahead of you.

JComboBox Cell Editors

If you look at the constructors for the DefaultCellEditor class,[22] you see one that accepts a JCheckBox argument (which is likely to be the one used by the default Boolean cell editor), one that accepts a JTextField argument, like the one you just saw, and one that accepts a JComboBox argument. What does that look like? Figure 12-11 shows some sample application images for the cbEdit1.py script that use a combo box editor for the values in column 1.

Figure 12-11. *cbEdit1.py output using a JComboBox editor*

It's interesting to note that you can't tell, from the initial display, that a combo box or drop-down list will be displayed, until you invoke the editor on the value displayed in column 1. Is there a way to show this? Certainly! All you have to do is set the cell renderer and the editor. Figure 12-12 shows some images from the cbEdit2.py application, which does this.

Figure 12-12. *cbEdit2.py output using a JComboBox editor and renderer*

A Slight Detour: Table Row Height

Figure 12-12 shows that the default might not always be optimal. So, how do you change the height of the table rows? When one of these kinds of questions arises, the first place that you should look is at the documentation for the class in question. In this case, the JTable[3] class. Search for "rowheight" and you'll find a (protected) rowHeight attribute, as well as some getter and setter methods.

The fact that the class includes multiple getter and setters, as shown in Table 12-4 illustrates that this class provides an opportunity to control the row height used by the rows in the table. In addition, each row can have a different height.

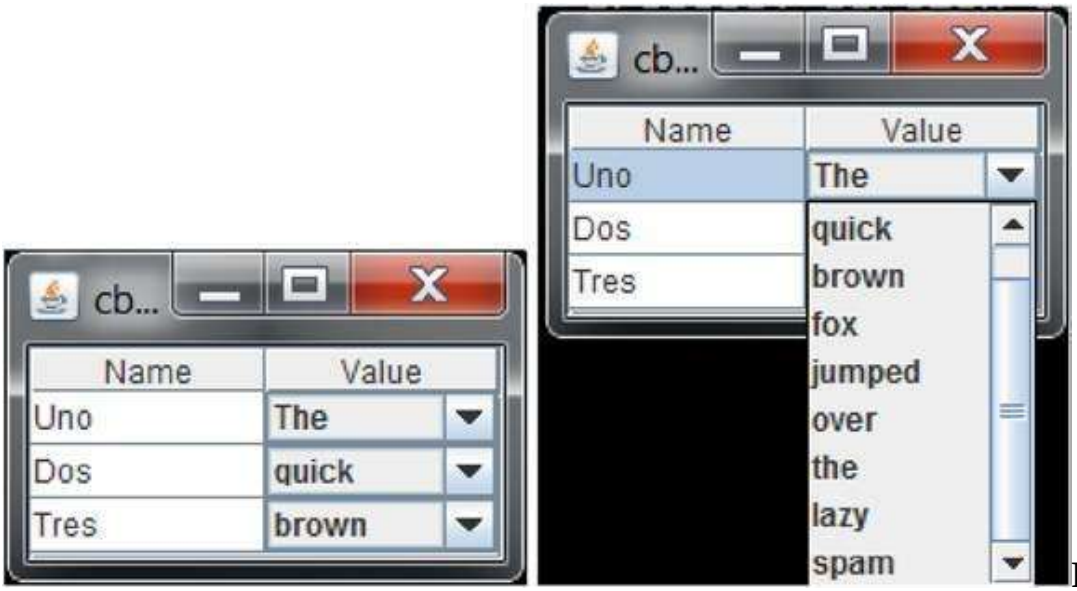
Table 12-4. *JTable rowHeight Getter and Setter Methods* **Getter/Setter Name**
getRowHeight()
getRowHeight(int row)
setRowHeight(int rowHeight)

setRowHeight(int row, int rowHeight)

Description
Returns the JTable rowHeight (in pixels).
Returns the rowHeight of the specified JTable row.

Specifies the rowHeight to be used by all table rows, in pixels, and initiates a table revalidation and repainting.
Specifies the rowHeight to be used by the specified row and initiates a table revalidation and repaint.

The results of using the rowHeight keyword attribute on the JTable constructor call is shown in Figure 12-13. Remember that since the frame containing the JTable instance is contained in a scroll pane, simply changing the height of the rows isn't sufficient. You also need to increase the height of the frame if you don't want the scroll bar to be displayed.



Figure

12-13. *cbEdit3.py output with slightly larger rows*

Interestingly enough, the second image shows that the height of the drop-down list (the ComboBox) isn't affected by the JTable row height. This shouldn't be too much of a surprise when you think about it. The row height that you changed was in the table, not in the ComboBox.

Listing 12-16 shows the slight modifications required to make the application easier to use. The only changes are found in lines 54, where the height of the frame is slightly larger (from 112 to 125 pixels). The other change can be seen in line 58, where you specify the rowHeight keyword argument used to assign the value of 20 (pixels) for each row. **Listing 12-16.** The cbEdit3 Class's run Method

```
51|class cbEdit3( java.lang.Runnable ) :
52| def run( self ) :
53| frame = JFrame(
54| 'cbEdit3',
55| size = ( 200, 125 ),
56| locationRelativeTo = None,
57| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
58| )
59| table = JTable( tm(), rowHeight = 20 )
60| table.setDefaultRenderer( JComboBox, cbRenderer() )
61| table.setDefaultEditor( JComboBox, cbEditor() )
62| frame.add( JScrollPane( table ) )
63| frame.setVisible( 1 )
```

Warning: Ugliness Ahead

Up to this point, I have been completely avoiding the fact that the version of Jython that is provided with the WebSphere Application Server product is, in a word, ancient. I'm sorry, but it's true. Up until now, it hasn't caused any problems, at least nothing significant. Now, however, you're going to see where it makes a difference.

When I tried to implement a simple cell editor based on the JSpinner[25] class, I converted a simple Java application, an abbreviated version of which is shown in Listing 12-17. The point that I want to make with this example is the fact that the

class extends the AbstractCellEditor class and implements the TableCellEditor interface, as you can see in lines 12 and 13.

Listing 12-17. SpinEditor.java JSpinner Cell Editor

```
12|public class SpinEditor extends AbstractCellEditor
13| implements TableCellEditor
14|{
15| JSpinner spinner;
16| public SpinEditor()
17| {
18| String values[] = { "Spam", "Eggs", "Bacon" };
19| spinner = new JSpinner( new SpinnerListModel( values ) );
20| spinner.setEditor( new JSpinner.ListEditor( spinner ) );
21| }
22| public Component getTableCellEditorComponent(
23| JTable table,
24| Object value,
25| boolean isSelected,
26| int row,
27| int column
28| )
29| {
30| spinner.setValue( value ); 31| return spinner;
32| }
33| public Object getCellEditorValue() 34| {
35| return spinner.getValue(); 36| }
37| public static void main(String[] args) 38| {
| ...
63| }
64|}
```

[25]SpinEditor.java is incompletely shown here; only enough is shown for discussion purposes.

Why use this approach? The main reason for doing so relates to the fact that the previous examples used the DefaultCellEditor constructors, as shown in Table 12-5. The Java example, shown in Listing 12-17, illustrates how you might go about creating a different kind of table cell editor, using a totally different

component type.

Table 12-5. *DefaultCellEditor Constructor Signatures*

DefaultCellEditor(JCheckBox checkBox) DefaultCellEditor(JComboBox comboBox) DefaultCellEditor(JTextField textField)

What does this look like when you convert it to Jython? You are likely to get an editor class similar to Listing 12-18. Notice how easy the conversion from Java to Jython is. This might be of assistance to you in the future, should you want to convert a Java Swing application to Jython.

Listing 12-18. SpinEdit1.py Editor Class

```
14|class editor( AbstractCellEditor, TableCellEditor ) :
15| def __init__( self ) :
16| values = 'Bacon,Eggs,Spam'.split( ',' )
17| self.spinner = JSpinner( SpinnerListModel( values ) )
18| self.spinner.setEditor(
19| JSpinner.ListEditor( self.spinner )
20| )
21| def getCellEditorValue( self ) :
22| return self.spinner.getValue()
23| def getTableCellEditorComponent(
24| self, # object reference
25| table, # JTable
26| value, # Object
27| isSelected, # boolean
28| row, # int
29| column # int
30| ) :
31| self.spinner.setValue( value );
32| return self.spinner;
```

Unfortunately, if you were to execute this script using the wsadmin utility from a WebSphere Application Server installation and click one of the values in the right-most column, you would see an exception like the one shown in Figure 12-14.

```
 Press <Enter> to terminate the application:Exception in thread
  "AWT-EventQueue-0" Traceback (innermost last):
  (no code object) at line 0
AttributeError: abstract method "isCellEditable" not implemented
```
Figure 12-14. *The wsadmin exception about the isCellEditable method*

The really strange thing is that when I tried to use this same script with the latest stable build of Jython (Jython 2.5.3), it worked just fine. But how do you get it to work with the version of Jython provided by the WebSphere product?

Instead of inheriting from AbstractCellEditor and TableCellEditor, as shown in Listing 12-17, you can base the editor class on the DefaultCellEditor class, as shown in Listing 12-19.

Listing 12-19. SpinEdit2.py Editor Class Based on DefaultCellEditor

```
13|class editor( DefaultCellEditor ) :
14| def __init__( self ) :
15| DefaultCellEditor.__init__( self, JTextField() )
16| values = 'Bacon,Eggs,Spam'.split( ',' )
17| self.spinner = JSpinner( SpinnerListModel( values ) )
18| self.spinner.setEditor(
19| JSpinner.ListEditor( self.spinner )
20| )
21| def getCellEditorValue( self ) :
22| return self.spinner.getValue()
23| def getTableCellEditorComponent(
24| self, # object reference
25| table, # JTable
26| value, # Object
27| isSelected, # boolean
28| row, # int
29| column # int
30| ) :
31| self.spinner.setValue( value );
32| return self.spinner;
```

This script works using the wsadmin utility and Jython 2.5.3. Since this book is primarily intended for WebSphere script writers, subsequent scripts will be based on the DefaultCellEditor class, and not on the AbstractCellEditor class and the TableCellEditor interface. Nonetheless, it was worth noting, so you don’t waste

the same kind of time that I did when I first encountered this issue.

Figure 12-15 shows some images from this application. The first two use an application frame height of 106 pixels, with each table row using 20 pixels. The next two images show what happens when each row uses 25 pixels and the frame height is set to 116 pixels.

Figure 12-15. *SpinEdit2.py output images*

The interesting note about these images is that when you use a uniform row height for each table row to better display the cell editor representation, the table representation might look quite right. This is due to the fact that you chose to display the cell values using a text, or string renderer, instead of a spinner renderer. If you use a custom renderer instead, you'll see that each cell in this column uses the spinner icons, as shown in Figure 12-16.

Figure 12-16. *SpinEdit3.py output images using a JSpinner renderer*

Did you notice how the cell being edited in the second image shows the text of the spinner selection in bold? It's little things like this that make your application more user friendly. That's one of the best reasons for using a robust and well-designed framework like Swing to build your applications.

Are there other improvements that you should make? Take another look at the application. In fact, let's make this more obvious by comparing the outputs of SpinEdit2 and SpinEdit3, side by side, when the first row in the table is selected. See Figure 12-17.

Figure 12-17. *SpinEdit2 and SpinEdit3 renderer differences*

Seeing them like this makes it more obvious that the custom renderer isn't dealing well with the cell colors. This shouldn't be too difficult to fix, right? Listing 12-12 showed how to solve a similar issue. Listing 12-20 shows the

revision of the sRenderer class from SpinEdit4.py.

Listing 12-20. SpinEdit4.py sRenderer Class

```
34|class sRenderer( DefaultTableCellRenderer ) :
35| def __init__( self ) :
36| self.DTCR = DefaultTableCellRenderer()
37| self.spinner = JSpinner( SpinnerListModel( choices ) )
38| def getTableCellRendererComponent(
39| self,
40| table, # table containing cell being rendered
41| value, # Object - value being rendered
42| isSelected, # boolean - Is value selected?
43| hasFocus, # boolean - Does this cell have focus?
44| row, # int - Row # (0..N)
45| col # int - Col # (0..N)
46| ) :
47| comp = self.DTCR.getTableCellRendererComponent(
48| table, value, isSelected, hasFocus, row, col
49| )
50| result = self.spinner
51| result.setForeground( comp.getForeground() )
52| result.setBackground( comp.getBackground() )
53| result.setValue( value )
54| return result
```

Unfortunately, this doesn’t resolve the problem because the JSpinner class has its own model-specific editor. In order to “fix” this—that is, to have the spinner renderer reflect the appropriate colors—you need to change the text editor field that the spinner is using.

Listing 12-21 shows the revised spinner renderer class from the SpinEdit5.py sample application. Figure 12-18 shows what this does to the application’s output.

Listing 12-21. SpinEdit5.py with Fixed sRenderer Class

```
34|class sRenderer( DefaultTableCellRenderer ) :
35| def __init__( self ) :
36| self.DTCR = DefaultTableCellRenderer()
```

```
37| self.spinner = JSpinner( SpinnerListModel( choices ) )
38| def getTableCellRendererComponent(
39| self,
40| table, # table containing cell being rendered 41| value, # Object - value being
rendered 42| isSelected, # boolean - Is value selected? 43| hasFocus, # boolean -
Does this cell have focus? 44| row, # int - Row # (0..N) 45| col # int - Col #
(0..N) 46| ) :
47| comp = self.DTCR.getTableCellRendererComponent( 48| table, value,
isSelected, hasFocus, row, col 49| )
50| tf = self.spinner.getEditor().getTextField()
51| tf.setForeground( comp.getForeground() )
52| tf.setBackground( comp.getBackground() )
53| self.spinner.setValue( value )
54| return self.spinner
```

Figure 12-18. *SpinEdit5 output*

Column Manipulation

Up to this point, you haven't learned much about column adjustments and manipulations. Back in Figure 12-1, you saw how the default table settings allow users to reorder the columns. Listing 12-3 shows how this feature can be disabled. Now, you're going to take a look at ways that you can manipulate your table columns.

Column Widths

In the section entitled, "Defaults Can Be Harmful to Your . . . Mental Health," you learned that, by default, the available horizontal space will be shared equally among each of the columns. If that is not the best appearance for your application, you need to take control of the way the column space is allocated. The first thing to realize is that your table might not have column headings. If it does, the information in these headings might not affect the way that column space is, or should be, allocated. For example, if the information in your column

headings is always wider than the table data, this can greatly simplify the way that column space should be allocated. On the other hand, if the data is sometimes wider than your column headings, you need to take this into account when your application determines how wide each column should be.

Column Heading Width

In order to determine the amount of space (the width) that should be used to comfortably display your headings, the program needs to make use of the renderer used by the table header. The fact that the table header has its own renderer shouldn't be too much of a surprise to you. Remember that the header will almost always contain strings, whereas the table data will have various types of data in each column. Besides, you previously learned about the JTableHeader class,[10] when you learned how to disable the reordering of columns.

Now, you're going to use this class to obtain the renderer used to display the table header. Let's revisit the application, last seen in Table6c.py, and revise it to adjust the width of the columns using the width of the column headings.

Listing 12-22 shows the code used to do this. You may be surprised by the row number of -1, in line 105. The Javadoc for the TableCellRenderer interface[15] includes a comment in the section on the getTableCellRendererComponent(...) method that states, "When drawing the header, the value of row is -1."

Listing 12-22. Table7.py: Specifying the Column Width Using Header Information Only

```
97| hRenderer = table.getTableHeader().getDefaultRenderer()
98| for col in range( model.getColumnCount() ) :
99| column = table.getColumnModel().getColumn( col ) 100| comp =
hRenderer.getTableCellRendererComponent( 101| None, # Table 102|
column.getHeaderValue(), # value 103| 0, # isSelected = false 104| 0, # hasFocus
= false 105| -1, # row # 106| col # col # 107| )
108| width = comp.getPreferredSize().width
109| column.setPreferredWidth( width )
```

This attempt fails because the preferred size is only a suggestion, not an absolute and firm limitation or restriction.[26] If you want to do this, you have to set the minimum and maximum column widths. Be careful though. In some classes the

method and attribute names have minimum and maximum spelled out entirely. The TableColumn class,[16] on the other hand, has setMinWidth(...), and setMaxWidth(...) methods instead. So, by replacing line 109 in Listing 12-20 with two statements, one to set the minimum column width and one to set the maximum column width, you get the output in Figure 12-19, which was generated using the Table8.py script.

[26]And thirdly, the code is more what you'd call "guidelines" than actual rules.

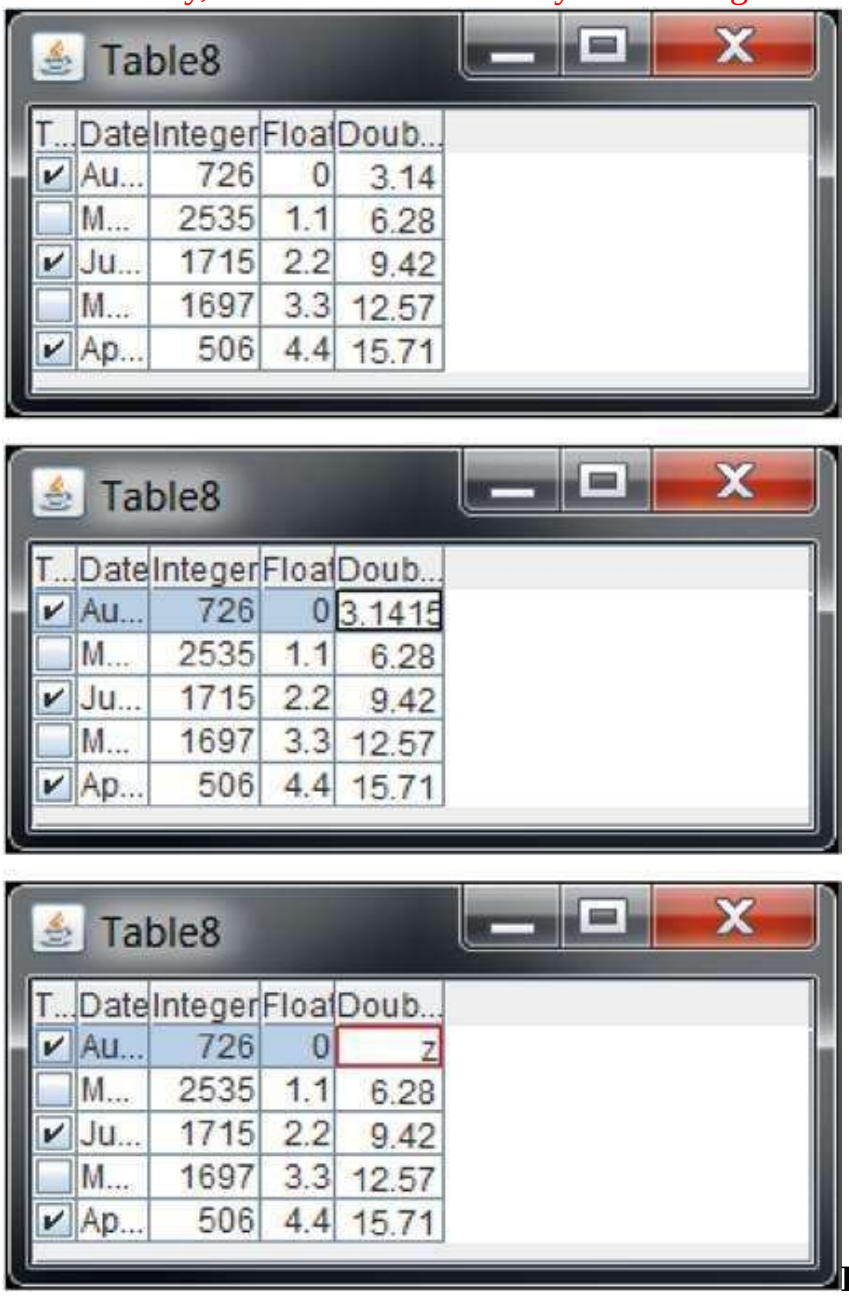


Figure 12-19. *Table8*

with fixed column widths
Determining Column Width

Well, that's a big failure. You certainly don't want to fix the column widths solely on the widths of the column headers, at least not for this kind of column headings that this application is using. Instead, you can process the data in the table, one column at a time, and compute the maximum preferred width for each value in the column. Then, you set the preferred width for the column using the maximum of the preferred header width and the maximum preferred width of all of the values in this column.

That doesn't sound too terribly bad. Are there any problems with this? Yes, unfortunately, there are. While working on a general function for this purpose, I encountered the following issues:
• The data returned by the getValueAt(...) method needs to be processed using the appropriate data type (the result of the getColumnClass(...) for the column). This wasn't too difficult to resolve since you can use the data type identified with this column to provide the renderer with the appropriate kind of value.

• Dealing with Date values can be somewhat of a challenge. In order to resolve this issue, you have to use the appropriate DateFormat instance that matches the way that your Date columns are displayed in the table. In this case, it seemed that the easiest way to deal with this was to use a DefaultRenderer and have it determine the preferred width of the string representation of the formatted date strings.

Listing 12-23 shows excerpts from the setColumnWidths(...) method of the Table9.py sample application. You might want to use something like this to determine how to allocate the column widths.
Listing 12-23. Excerpts from the setColumnWidths(...) Method from Table9.py

```
21|def setColumnWidths( table ) :
22| header = table.getTableHeader()
| ...
27| tcm = table.getColumnModel() # Table Column Model
28| data = table.getModel() # To access table data
29| margin = tcm.getColumnMargin() # gap between columns
30| rows = data.getRowCount() # Number of rows
31| cols = tcm.getColumnCount() # Number of cols
```

```
32| df = DateFormat.getDateInstance( DateFormat.MEDIUM )
33| tWidth = 0 # Table width
34| for i in range( cols ) : # For col 0..N
35| col = tcm.getColumn( i ) # TableColumn: col i
36| idx = col.getModelIndex() # model index: col i
37| render = col.getHeaderRenderer() # header renderer,
38| if not render : #
39| render = hRenderer # or a default
40| if render :
41| comp = render.getTableCellRendererComponent( | ...
48| )
49| cWidth = comp.getPreferredSize().width
50| else :
51| cWidth = -1
52| Type = data.getColumnClass( i ) # dataType: col i
53| for row in range( rows ) :
54| v = data.getValueAt( row, idx ) # value
55| if Type == Date :
56| val = df.format( v ) # formatted date
57| r = table.getDefaultRenderer( String ) 58| else :
59| val = Type( v )
60| r = table.getCellRenderer( row, i )
61| comp = r.getTableCellRendererComponent(
| ...
68| )
69| cWidth = max( cWidth, comp.getPreferredSize().width ) 70| if cWidth > 0 :
71| col.setPreferredWidth( cWidth + margin ) 72| tWidth +=
col.getPreferredWidth()
73| table.setPreferredScrollableViewportSize(
| ...
78| )
```

I think that you'll agree that the column widths shown in Figure 12-20 are a great improvement over the ones in Figure 12-19.

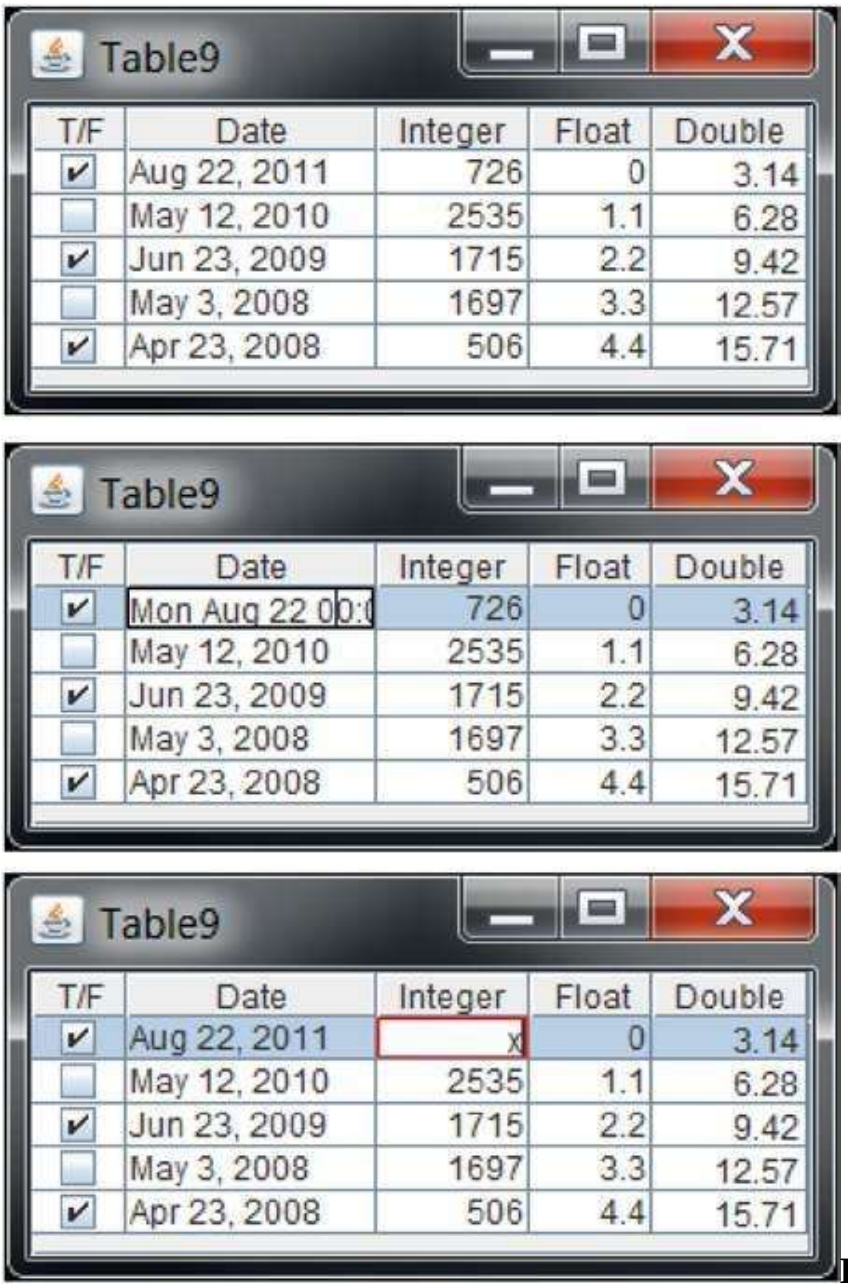


Figure 12-20. *Table9 with compute column widths*

Column Adjustments

If you have played with any of the table samples, you may have wondered why the columns react in the way that they do when you change the column widths. (Drag the vertical bar separating two column headers to the left or to the right to change the widths.)

Did you notice how all of the subsequent columns are adjusted to maintain the

table width? Why is this? Well, the JTable class includes a property called AutoResizeMode that the table uses to determine changes to the other columns' widths. Table 12-6 shows the constants used to modify this JTable property.

Table 12-6. JTable *Auto Resize Constants* **JTable Auto Resize Constant**
AUTO_RESIZE_OFF
AUTO_RESIZE_NEXT_COLUMN

AUTO_RESIZE_SUBSEQUENT_COLUMNS
AUTO_RESIZE_LAST_COLUMN AUTO_RESIZE_ALL_COLUMNS

Description
Columns will not be adjusted automatically.

Only the next column width will be affected by column width changes.

All subsequent columns will have their widths affected, in order to distribute the effects across any subsequent columns.
Note: This is the default setting.

Only the width of the last column will change.
All table columns will be adjusted to account for the userspecified width changes.

Let's see what this means in a real application. To start, disable the auto resize property of the table. Figure 12-21 shows the application when the AUTO_RESIZE_OFF and setColumnWidths(...) functions are used, as shown in Listing 12-21.

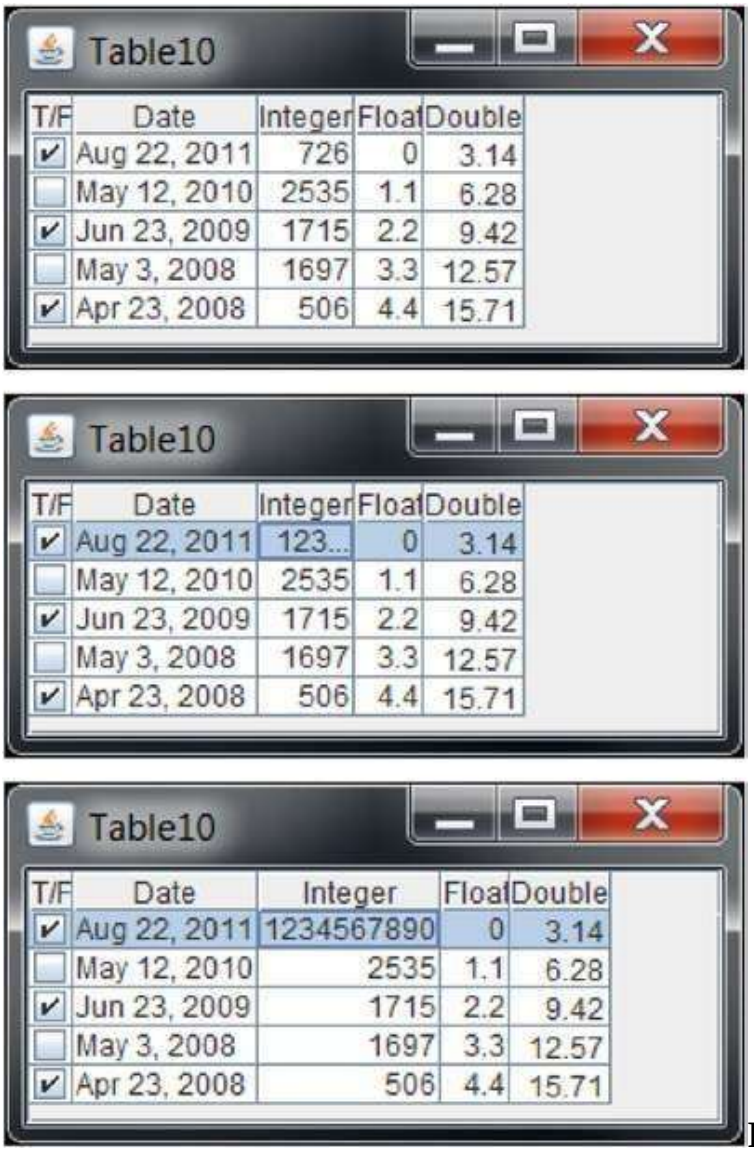


Figure 12-21. *Table10.py: computing column widths with auto resize off*

What would it take to have a simple table application that allowed you to dynamically change the auto resize mode, and then see how each setting affects the application? Figure 12-22 shows the results.


Table11
AUTO_RESIZE
T/F Date Integer Float Double
Aug 22, 2011 726 0 3.14
May 12, 2010 2535 1.1 6.28
Jun 23, 2009 1715 2.2 9.42
May 3, 2008 1697 3.3 12.57
Apr 23, 2008 506 4.4 15.71

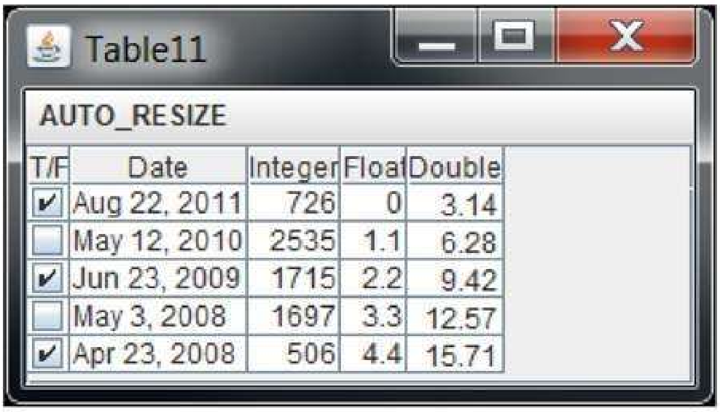


Table11
AUTO_RESIZE
T/F Date Integer Float Double
Aug 22, 2011 726 0 3.14
May 12, 2010 2535 1.1 6.28
Jun 23, 2009 1715 2.2 9.42
May 3, 2008 1697 3.3 12.57
Apr 23, 2008 506 4.4 15.71

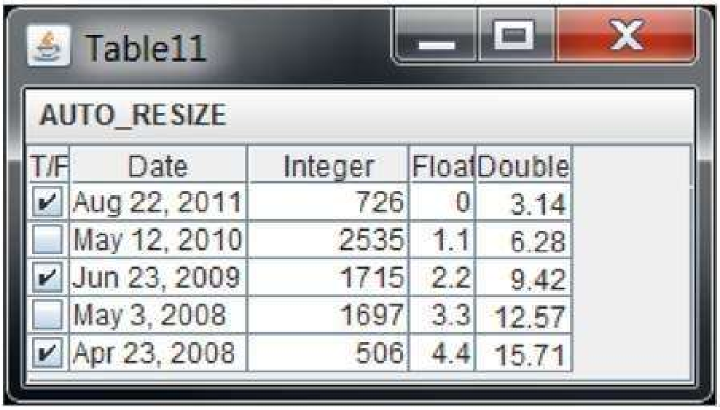


Table11
AUTO_RESIZE
T/F Date Integer Float Double
Aug 22, 2011 726 0 3.14
May 12, 2010 2535 1.1 6.28
Jun 23, 2009 1715 2.2 9.42
May 3, 2008 1697 3.3 12.57
Apr 23, 2008 506 4.4 15.71

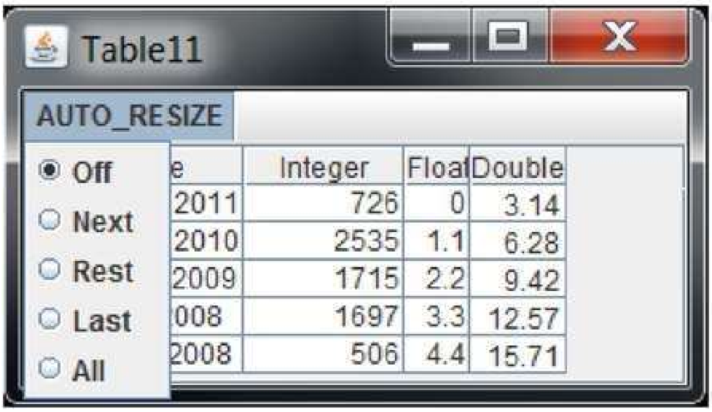


Table11
AUTO_RESIZE
Off
Next
Rest
Last
All
Integer Float Double
2011 726 0 3.14
2010 2535 1.1 6.28
2009 1715 2.2 9.42
008 1697 3.3 12.57
2008 506 4.4 15.71

Figure 12-22. *Table11.py: dynamically selecting auto resize mode*

You can see that the application has a menu that allows you to select a different auto resize setting. Listing 12-24 shows parts of the Table11.py script that were used to produce the output in Figure 12-22. All of the radio button menu items identify the handler method as the action listener event handler. All this routine has to do is match the text of the user selection to the corresponding JTable auto resize constant and use this to change the auto resize mode on the table.

Listing 12-24. Selected Parts of Table11.py

```
130|class Table11( java.lang.Runnable ) :
| ...
144| self.info = [
145| [ 'Off' , JTable.AUTO_RESIZE_OFF ],
146| [ 'Next', JTable.AUTO_RESIZE_NEXT_COLUMN ],
147| [ 'Rest', JTable.AUTO_RESIZE_SUBSEQUENT_COLUMNS ],
148| [ 'Last', JTable.AUTO_RESIZE_LAST_COLUMN ],
149| [ 'All' , JTable.AUTO_RESIZE_ALL_COLUMNS ]
150| ]
| ...
185| def handler( self, event ) :
186| cmd = event.getActionCommand()
187| for name, value in self.info :
188| if cmd == name :
189| self.table.setAutoResizeMode( value )
190| self.table.repaint()
```

This kind of script is easy to produce, and it provides you with a simple test environment to better understand the implications of a table’s property. In this case, you’re looking at the auto resize property.

Summary

This chapter has illustrated some of the power and flexibility of the JTable class. It is important to remember that with great power there must also come great responsibility. This is just a reminder that you can quickly create a table in your application. However, if you do so, it is quite likely that the default settings are going to need some testing, manipulation, and tweaking in order to fit your exact

needs. As this chapter has shown, iterating your script with various enhancements isn’t that difficult. You are encouraged to use this technique to help improve your knowledge of tables and their characteristics.

Chapter 13

Keystrokes, Actions, and Bindings, Oh My!

There are a number of ways that graphical applications obtain user input. This chapter focuses on how applications deal with user input from the keyboard. This is all about what happens when the user presses a key, and how applications perceive and react to this kind of event.

Getting in a Bind: Looking at Bindings

In the section entitled “Menu Mnemonics and Accelerators" in Chapter 10, you learned how convenient it is to associate a particular keystroke[1] with an ActionListener[2] for menu items. This association is called a *binding*. Doing this allowed you to simplify the process of causing some menu-related action to occur. Thus, you were able to initiate the associated menu action using the specified keystroke.

In this chapter, you learn how other parts of your applications can be set up to react to specific keystrokes. You will also learn how to enhance the friendliness of your applications with these kinds of bindings.
What Is Meant by Binding?

Menu entries aren’t the only place in Swing where you can build an association between a keystroke and an action. In fact, creating this kind of association is called *binding*. A keystroke is bound to an action. One point you have to realize is that key bindings relate to a specific context. This has the potential of causing the users some confusion. For example, say you use a keystroke in a particular way for one context and in a different way in a different context. This might confuse or frustrate your users, so consider the consequences of such bindings carefully.

InputMaps and ActionMaps
How are bindings created? To begin, it is important to realize that the abstract JComponent class[3] includes actionMap and inputMap attributes, as you can see

in Listing 13-1.[4]

[1] Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/KeyStroke.html.
[2]See http://docs.oracle.com/javase/8/docs/api/java/awt/event/ActionListener.html.
[3]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JComponent.html.
[4]The classInfo function can be found in code\Chap_04\classInfo.py.

Listing 13-1. JComponent Map Attributes

```
wsadmin>from javax.swing import JComponent wsadmin>
wsadmin>classInfo( JComponent, attr = 'map' ) javax.swing.JComponent

actionMap, inputMap
| java.awt.Container
| | java.awt.Component
| | | java.lang.Object
| | | java.awt.image.ImageObserver
| | | java.awt.MenuContainer
| | | java.io.Serializable
| java.io.Serializable
wsadmin>
```

Let’s take a look at a specific class that descends from JComponent; for example, the JTable class.[5] If you use the classInfo function to display the methods that have the word “map” somewhere in their names, you get the output in Listing 13-2. This makes sense, since it shows that you have getter and setter methods for the inputMap and actionMap attributes. It is important to note, however, that the output of this function can be incomplete.

Listing 13-2. JTable Map Methods

```
wsadmin>from javax.swing import JTable
wsadmin>
wsadmin>classInfo( JTable, meth = 'map' )
javax.swing.JTable
| javax.swing.JComponent
> getActionMap, getInputMap, setActionMap, setInputMap | |
java.awt.Container
```

| | | java.awt.Component
| | | | java.lang.Object
| | | | java.awt.image.ImageObserver
| | | | java.awt.MenuContainer
| | | | java.io.Serializable
| | | java.io.Serializable
| javax.swing.event.TableModelListener
| | java.util.EventListener
| javax.swing.Scrollable
| javax.swing.event.TableColumnModelListener
| | java.util.EventListener
| javax.swing.event.ListSelectionListener
| | java.util.EventListener
| javax.swing.event.CellEditorListener
| | java.util.EventListener
| javax.accessibility.Accessible
| javax.swing.event.RowSorterListener
| | java.util.EventListener
wsadmin>

If you look at the JTable Javadoc,[5] you'll find the inputMap and actionMap getters and setters listed in Table 13-1.

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

The interesting point is that there are two inputMap getters, one without an argument and the other with. What's that all about?

Table 13-1. *JTable inputMap and actionMap Getters and Setters* **Method return type and signature** InputMap getInputMap()

InputMap getInputMap(int condition)
void setInputMap(int condition, InputMap map) ActionMap getActionMap()

void setActionMap(ActionMap am)
Description
Returns the inputMap that is used when the component has focus.

Returns the inputMap that is used during condition. Sets the inputMap to use with the condition to map.

Returns the actionMap used to determine what action to fire for a particular KeyStroke binding.
Sets the actionMap to am.
The condition argument used by the second getter identifies which of three possible inputMaps should be returned to the caller. The constants that should be used are found in the JComponent class and are shown in Table 13-2.

Table 13-2. *JComponent Condition (Binding-Related) Constants* **Constant Name**
WHEN_FOCUSED
WHEN_ANCESTOR_OF_FOCUSED_COMPONENT

WHEN_IN_FOCUSED_WINDOW

Binding Context
Binding applies when the component has the focus.

Binding applies when the receiving component is an ancestor of the focused component or is itself the focused component.
Binding applies when the receiving component is in the window that has the focus or is itself the focused component.

So, it looks like you can use one of these JComponent constants as the condition argument to the getInputMap() method in order to retrieve the KeyStroke InputMap. But which constant is the right one to use?
You have a few options available. You could use all three and see which one results in inputMaps that contain information. Or you could take a look at the JComponent Javadoc[3] and read about the interesting and potentially useful methods that exist. Which ones, you might wonder? Well, the ones listed in Table 13-3 caught my attention (and aren't marked as obsolete).

Table 13-3. *JComponent Keystrokes and Actions Methods* **Method return type and signature Description** KeyStroke[] getRegisteredKeyStrokes() int getConditionForKeyStroke(KeyStroke aKeyStroke) ActionListener getActionForKeyStroke(KeyStroke aKeyStroke)
Returns the keystrokes that will initiate registered actions. Returns the condition that determines whether a registered action occurs in response to the specified keystroke. Returns the object that will perform the action registered for a given keystroke.

Okay, I admit it. I didn’t know to look into the JComponent class for these methods until I used the classInfo function and it showed the KeyStroke methods. Listing 13-3 shows the result of doing this.

Listing 13-3. Keystroke Methods in JTable Hierarchy

```
wsadmin>from javax.swing import JTable
wsadmin>
wsadmin>classInfo( JTable, meth = 'keystroke' )
javax.swing.JTable

getSurrendersFocusOnKeystroke, setSurrendersFocusOnKeystroke |
javax.swing.JComponent
> getActionForKeyStroke, getConditionForKeyStroke >
getRegisteredKeyStrokes
| | java.awt.Container
| | | java.awt.Component
| | | | java.lang.Object
| | | | java.awt.image.ImageObserver
| | | | java.awt.MenuContainer
| | | | java.io.Serializable
| | java.io.Serializable
| javax.swing.event.TableModelListener
| | java.util.EventListener
| javax.swing.Scrollable
| javax.swing.event.TableColumnModelListener
| | java.util.EventListener
| javax.swing.event.ListSelectionListener
| | java.util.EventListener
| javax.swing.event.CellEditorListener
| | java.util.EventListener
| javax.accessibility.Accessible
| javax.swing.event.RowSorterListener
| | java.util.EventListener
wsadmin>
```

JTable Keystrokes

Let’s take advantage of these methods to see how many keystroke bindings exist for the JTable class. Listing 13-4 shows a simple script that can be used to

determine how many of these bindings exist, as well as determine the ones associated specifically with the spacebar.

Listing 13-4. The Keystrokes.py Script

```
1|from javax.swing import JTable
2|table = JTable()
3|keys = [
4| str( key )
5| for key in table.getRegisteredKeyStrokes()
6| ]
7|print 'Number of JTable KeyStrokes:', len( keys )
8|width = max( [ len( key ) for key in keys ] )
9|print 'JTable "Space" Keys:'
10|print '\n'.join(
11| [
12| '%*s' % ( width, key )
13| for key in keys if key.endswith( 'SPACE' ) 14| ]
15| )
```

The output of this script is shown in Figure 13-1. It contains a few interesting things to consider. First, notice that there are almost six dozen keystroke bindings for JTable instances. This should explain why the rest of the output is limited to instances of bindings for the spacebar key. The other point of interest is the way that the keystroke modifiers are presented.

```
Number of JTable KeyStrokes: 71 JTable "Space" Keys:
pressed SPACE ctrl pressed SPACE shift pressed SPACE shift ctrlpressed
SPACE
```

Figure 13-1. *Keystrokes.py output*

Additional tests of the keystroke modifiers show that each of the keystrokes has the four possible modifiers seen in Figure 13-1. This made me wonder how hard it might be to get the names of the actions for these keystrokes.
Listing 13-5 shows the KeyStrokes2.py script, which was written to answer these questions. **Listing 13-5.** KeyStrokes2.py Showing Bound Actions

```
1|from javax.swing import JTable
2|table = JTable()
```

```
3|keys = [
4| key
5| for key in table.getRegisteredKeyStrokes()
6| ]
7|print 'Number of JTable KeyStrokes:', len( keys )
8|width = max( [ len( str( key ) ) for key in keys ] )
9|print 'JTable "Space" Keys:'
10|for key in keys :
11| if str( key ).endswith( 'SPACE' ) :
12| cond = table.getConditionForKeyStroke( key ) 13| act = table.getInputMap( cond ).get( key ) 14| print '%*s : %s' % ( width, str( key ), act )
```

The resulting output is shown in Figure 13-2. I don’t know about you, but I find this really fascinating.

```
Number of JTable KeyStrokes: 71
JTable "Space" Keys:
pressed SPACE : addToSelection ctrl pressed SPACE : toggleAndAnchor shift pressed SPACE : extendTo shift ctrl pressed SPACE : moveSelectionTo
```

Figure 13-2. *Keystrokes2.py output*

Putting It All Together

What would it take to create a simple application that displays a table of information showing the keystroke bindings for the JTable class? You can use a number of things that you’ve learned up to this point to do just that.

```
locationRelativeTo = None
```

While creating this application, you can also get a better understanding of something that you have been taking for granted up to now. Back in Chapter 11, you started seeing some examples that used the locationRelativeTo keyword argument to position the JFrame instance “in the middle of the screen.” So what exactly does that mean, and why am I bringing it up here?

Until this application, you didn't really have much of a reason to dig into it. This application changed that because it became more obvious what setting the locationRelativeTo attribute to None actually does. Essentially, using this setting causes the top-left corner of your application to be placed in the center of the screen. When your application isn’t too big for the screen, this is a very easy and convenient way to position the application window near the center of the screen.

Centering the Application

If you want to center the application's window in the middle of the screen, the application must be able to determine or compute the following:

• The size of the screen
• The size of the application (frame)

Then you have compute the best location for the top-left corner for the application using some simple math; so far, so good. The question is, what's the best way to determine the size of the screen? There's a simple answer to that question because that's something that GUI application developers have needed to know for a long time. Figure 13-3 shows just how easily this can be done.

Figure 13-3. *Determining screen size*

Unfortunately, you might not actually need to use this technique. Consider for a moment what using the locationRelativeTo keyword argument does. It places the application in the center of the screen. To correctly position the application, you only need to reposition it using the size of the application (which you can obtain using the JFrame getSize() method) and the center of the screen (which you can obtain using the JFrame getLocation() method). Listing 13-6 is an excerpt from the KeyBindings.py script; it shows one way that this can be done.

Listing 13-6. Centering an Application

```
134| size = frame.getSize()
135| loc = frame.getLocation()
136| frame.setLocation(
137| Point(
138| loc.x - ( size.width >> 1 ),
139| loc.y - ( size.height >> 1 )
140| )
141| )
```

The code expects that the JFrame instance is in the variable named frame, and that it has been properly populated and sized before the statement in line 134 is executed. It is important to realize what kinds of values are returned by the method calls in lines 134 and 135. The size value is an instance of the java.awt.Dimension class[6] (just like you saw in Figure 13-3). This makes perfect

sense since the application has a width and height. On the other hand, the call to the getLocation() method returns an instance of the java.awt.Point class,[7] because only the X and Y coordinates are needed to identify a point on the screen.

The call to the setLocation(...) method should be fairly clear, once you realize that the positioning adjustment requires that you move the X coordinate by half the width of the application and move the Y coordinate by half the height of the application. This is exactly what the expressions on lines 138 and 139 are doing.

[6]Seehttp://docs.oracle.com/javase/8/docs/api/java/awt/Dimension.html. [7]See http://docs.oracle.com/javase/8/docs/api/java/awt/Point.html. Defining the Table Properties

With a little thought, you'll likely realize that this table will be composed of strings and you need to prevent the users from being able to modify the table's contents. You can define a trivial table model containing only the getColumnClass(...) and isCellEditable(...) methods. Listing 13-7 shows just how simple this class can be.

Listing 13-7. Trivial Custom Table Model Class

```
75|class myTM( DefaultTableModel ) :
76| def getColumnClass( self, col ) :
77| return String
78| def isCellEditable( self, row, col ) :
79| return 0
```

Computing the Table Data

Most of the work needed to create this application has already been completed. The biggest challenge is using the stuff that you learned about earlier—using KeyStrokes.py and KeyStrokes2.py to build the two-dimensional array of strings. The first column should hold the keystroke name and then each of the columns can hold the action name for the corresponding keystroke-modifier column. Listing 13-8 shows the data(...) method, which builds and returns this table for the application.

Listing 13-8. KeyBindings.py Method to Build Table Data

```
80|class KeyBindings( java.lang.Runnable ) :
81| def data( self ) :
82| table = JTable() # use an empty (default) table
83| iMap = table.getInputMap(
84| JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT
85| )
86| keystrokes = [
87| ( key, iMap.get( key ) )
88| for key in table.getRegisteredKeyStrokes()
89| ]
90| keys = {} # Dict, index = key name -> modifiers
91| acts = {} # Dict, index = keyStroke -> actionName
92| for key, act in keystrokes :
93| val = str( key ) # e.g., shift ctrl pressed TAB
94| acts[ val ] = act # e.g., selectNextColumnCell
95| pos = val.rfind( ' ' )
96| prefix, name = val[ :pos ], val[ pos + 1: ]
97| if keys.has_key( name ) :
98| keys[ name ].append( prefix )
99| else :
100| keys[ name ] = [ prefix ]
101| names = keys.keys()
102| names.sort()
103| prefixes = [
104| 'pressed', # unmodified keystroke 105| 'ctrl pressed', # Ctrl-<keystroke>
106| 'shift pressed', # Shift-<keystroke> 107| 'shift ctrl pressed' # Shift-Ctrl-
<keystroke> 108| ]
109| result = [] # The 2D table of strings 110| for name in names : # For each key
name (e.g., TAB) 111| here = [ name ] # Current table row 112| for prefix in
prefixes :
113| kName = ' '.join( [ prefix, name ] ) 114| here.append( acts.get( kName, '' ) )
115| result.append( here )
116| return result
```

Table 13-4 explains how this method produces the desired result. It may help you to realize that the cells for which no bindings exist will contain empty strings.

Table 13-4. *KeyBindings.py data() Method, Explained*

Lines Description
82 Instantiates a JTable to simplify access to its methods.
83-85 Obtains an inputMap of the context used for all key bindings.

86-89 List-comprehension statement that builds a list of tuples for all the keystroke bindings and their associated action names.

90-100 Builds a dictionary, named keys, indexed by the key name (e.g., TAB) and identifying the modifier bindings. Also builds a dictionary, named acts, indexed by the keystroke (e.g., pressed TAB) containing the bound action.

101-102 Builds a sorted list of the bound (unmodified) keystroke names (e.g., SPACE and TAB).
103-108 Builds a list of the keystroke modifiers in column order.
109-116 Builds the table to be returned by the data() method (i.e., the two-dimensional array of strings).

The Fruits of Your Labor
Figure 13-4 shows the results of executing the KeyBindings.py application, before any row selection has been made.

KeyStroke	Unmodified	Ctrl	Shift	Shift-Ctrl
A		selectAll		
BACK_SLASH		clearSelection		
C		copy		
COPY	copy			
CUT	cut			
DELETE			cut	
DOWN	selectNextRow	selectNextRowChangeLead	selectNextRowExtendSelection	selectNextRowExtendSelection
END	selectLastColumn	selectLastRow	selectLastColumnExtendSelection	selectLastRowExtendSelection
ENTER	selectNextRowCell		selectPreviousRowCell	
ESCAPE	cancel			
F2	startEditing			
F8	focusHeader			
HOME	selectFirstColumn	selectFirstRow	selectFirstColumnExtendSelection	selectFirstRowExtendSelection
INSERT		copy	paste	
KP_DOWN	selectNextRow	selectNextRowChangeLead	selectNextRowExtendSelection	selectNextRowExtendSelection
KP_LEFT	selectPreviousColumn	selectPreviousColumnChangeLead	selectPreviousColumnExtendSelection	selectPreviousColumnExtendSelection
KP_RIGHT	selectNextColumn	selectNextColumnChangeLead	selectNextColumnExtendSelection	selectNextColumnExtendSelection
KP_UP	selectPreviousRow	selectPreviousRowChangeLead	selectPreviousRowExtendSelection	selectPreviousRowExtendSelection
LEFT	selectPreviousColumn	selectPreviousColumnChangeLead	selectPreviousColumnExtendSelection	selectPreviousColumnExtendSelection
PAGE_DOWN	scrollDownChangeSelection	scrollRightChangeSelection	scrollDownExtendSelection	scrollRightExtendSelection
PAGE_UP	scrollUpChangeSelection	scrollLeftChangeSelection	scrollUpExtendSelection	scrollLeftExtendSelection
PASTE	paste			
RIGHT	selectNextColumn	selectNextColumnChangeLead	selectNextColumnExtendSelection	selectNextColumnExtendSelection
SLASH		selectAll		
SPACE	addToSelection	toggleAndAnchor	extendTo	moveSelectionTo
TAB	selectNextColumnCell		selectPreviousColumnCell	
UP	selectPreviousRow	selectPreviousRowChangeLead	selectPreviousRowExtendSelection	selectPreviousRowExtendSelection
V		paste		
X		cut		

Figure 13-4. *KeyBindings.py's output, with no row selected*

One of the really neat things about this application is that you can use what you

see to understand the results of using these bindings. For example, take a look at the TAB row and the Unmodified column. The action that is seen here is selectNextColumnCell and the one in you can see that when you press either the Tab or the Shift-Tab key, the table selection moves in the expected fashion.

Take a look at the actions associated with the Ctrl-A and Ctrl-BACK_SLASH keys. Try pressing these keys and see if the results meet your expectations. You should realize that not all of the actions will be allowed. Why not? Well, some of them require that the current cell be editable (e.g., cut or paste). Others depend on the current values of the columnSelectionAllowed and rowSelectionAllowed table attributes. In spite of that, I find the output of this application quite useful. I hope that you do as well.

Binding Reuse

You might be wondering what you can do with this information. I'll get to that, but first I want to explain how I came to investigate keystroke bindings.

I was working with an application that used some JTable instances. One of the challenges that I encountered was that my tables had some read-only and read-write columns. I expected that the users of the application might want to skip from one read-write cell to another using the Tab and Shift-Tab keys. In order to do this, I first had to figure out how the current keystroke bindings worked.

Looking again at Figure 13-4, you can see that the Tab and Shift-Tab keys correspond to selectNextColumnCell and selectPreviousColumnCell, respectively. What can be done with this information? Well, what you want to do is have the Tab and Shift-Tab keys be bound to actions something like selectNextEditableColumnCell and selectPreviousEditableColumnCell.

I don’t know about you, but I would much prefer to use existing code instead of figuring out how to rewrite these two ActionListeners. That should make sense. How should you go about doing that? You have a number of options from which to choose. Let’s start with something simple and decide if and when you can improve things.

Where to Begin: Finding the Appropriate Action Class

With what kind of class do you need to start? Let’s see if you can get by with

something easy, like the AbstractAction class.[8] According to the Javadoc for that class, “The developer need only subclass this abstract class and define the actionPerformed method.” This sounds ideal for this situation.

The next question you need to ask is, “What will our class instance need to access in order to perform its role?” Well, since you are specifically concerned with JTable actions, it would seem likely that the action will need to know with which table instance the action should be working. Additionally, it will need to know the action it will be using to perform its role. In the case of selectNextEditableColumnCell, it needs to know that it will need to use selectNextColumnCell. Or will it? Do you really need to create a completely new action and replace the keystroke binding of the Tab key from selectNextColumnCell to a completely new action? What if your new action simply identifies itself as the original action and uses the original action to perform its role? That seems like a reasonable and fairly simple approach. It also simplifies the code that you need to write to perform the desired action.

What Do You Need to Worry About? Boundary Conditions

Whenever you create something like this, you need to be concerned with *boundary conditions.* What are those in this case? Well, consider a situation in which you have a read-only table. What should happen when the selectNextEditableColumnCell action is invoked? If you’re not careful, you might create an infinite loop situation, which would be a bad thing. You need to be certain that your action deals well with a read-only table.

What’s the next worst-case scenario? In my mind, it would be a table with only one editable cell. In that case, how many cells would your action need to check in order to find the next editable cell? Well, that would depend on the size of the table. Can you figure that out? Given access to the table, you can use the table.getRowCount() and table.getColumnCount() methods to determine how many cells exist.

Listing 13-9 shows a simple implementation of this approach. It includes a call to the default toolkit beep() method if no editable cells are found in the table. You can test this application, which can be found in the code\Chap_13\WoT.py script. You may want to modify the isCellEditable(...) method to return a 0 (false) in order to test the read-only table scenario.

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

Listing 13-9. findEditableCell Class from WoT.py

```
39|class findEditableCell( AbstractAction ) :
40| def __init__( self, table, action ) :
41| self.table = table
42| self.original = table.getActionMap().get( action )
43| self.table.getActionMap().put( action, self )
44| self.beep = Toolkit.getDefaultToolkit().beep
45| def actionPerformed( self, actionEvent ) :
46| table = self.table
47| numCells = table.getRowCount() * table.getColumnCount()
48| for cell in range( numCells ) :
49| self.original.actionPerformed( actionEvent )
50| if table.isCellEditable(
51| table.getSelectedRow(),
52| table.getSelectedColumn()
53| ) :
54| return
55| self.beep()
|...
70| findEditableCell( table, 'selectNextColumnCell' )
71| findEditableCell( table, 'selectPreviousColumnCell' )
```

Summary

This chapter explained how Java Swing applications deal with keystrokes. Rather than have the application monitor the user input, the environment determines the action to be performed for the keystroke event based on the current context. It is important to know how this works so your applications can take advantage of the existing infrastructure.

The next chapter discusses events and event handlers.

Chapter 14

It’s the Event of the Year: Events in Swing Applications

I have been referencing events, of all sorts, in many of the preceding chapters. It seems like a good time to take a better look at how events are handled in Swing

applications.

This chapter is all about events, such as mouse clicks, and preparing an application to handle these kinds of events when they occur. You'll also see how easy it is in Jython to associate a method with an event. It is important to note that methods that are called when specific events occur are called *listeners*. The association of a listener method with a specific kind of event is a kind of registration. Let's begin this chapter by taking a look at the number of classes that have something to do with events and listeners.

If an Event Occurs and No One Hears It . . .

If you look at the "complete list of Java classes,[1] " you'll find over 4,000 items. Of these, almost 12 dozen have "event" somewhere in their name and another seven dozen have "listener" in their name. What are all these things?

You've seen a number of them already. For example, when you were working with the JButton class,[2] you learned that in order to be able to react to user input, you needed an ActionListener[3] or some descendant class instance containing an actionPerformed(...) method. Until you did this association, no method would be called when the event occurred and the events were lost.

What type of argument is supplied to the actionPerformed(...) method? An ActionEvent,[4] that's what. An ActionEvent is an object instance that is used to indicate that some kind of component action has occurred. It can be used by an event handler to do things like identify the component that generated the event. Since ActionEvents are passed to registered listening methods, this chapter focuses more on listeners than on the details about ActionEvents.

Many of the Java Swing examples that you'll investigate are almost certain to include a listener of some type, as well as the method it defines or requires. The interesting thing about many of the Jython script examples that you've seen is that they don't have to explicitly identify, include, and use the particular listener class that is needed for the kind of event to be handled. You simply use the appropriate keyword argument to identify the method to be called when the event occurs.

[1] See http://docs.oracle.com/javase/8/docs/api/allclasses-noframe.html.

[2]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JButton.html.
[3]See http://docs.oracle.com/javase/8/docs/api/java/awt/event/ActionListener.html.
[4]See http://docs.oracle.com/javase/8/docs/api/java/awt/event/ActionEvent.html.

Is this always a good thing? Maybe, maybe not. Consider for a moment the perspective of an experienced Java Swing developer. Their experience may have trained them to look at Swing applications in a specific way to locate event handlers. Their first instinct may be to search for calls to the add*Listener(...) method. Are they going to locate them in Jython scripts that use the appropriate keyword argument? I don't think so, do you?

Personally, I find the use of the keyword argument to be a better approach for a number of reasons, including (but not limited to) these:
• It forces the developer to identify the event handler during component instantiation.

• It allows the event handler routine to be named something other than the generic actionPerformed(...) method name. For example, I prefer to see a method named buttonPressed(...) rather than the ubiquitous actionPerformed(...) variation, don't you?

• It allows multiple event handlers to be defined as part of the same application class. In this way, each button instance can have its own unique event handler, instead of having to share a single actionPerformed(...) method that must determine how it was invoked. This allows each event handler method to be simpler and uncluttered by code that needs to determine or identify the exact source of the associated event that caused the routine to be called.

• It allows the developer to avoid using multiple inheritance in their application class (for example, to include the ActionListener as one of its base classes).

Are there times when it might be a bad idea to use the keyword argument to identify the event handler to be invoked? Think about it for a moment. What does a method name like add*Listener(...) imply? It tells you that there are situations where multiple listeners may be appropriate. If this is the case then you are likely to be better served by using add*Listener(...) to identify all of the event handlers to be registered as event listeners. Otherwise, using the keyword argument approach may very well be the best. I'll talk more about this later in

this chapter.

Using Listener Methods

Most of the examples that you've seen have focused on a small number of listeners.[5] For example, in all of the JButton examples, you have only seen the ActionListener and its actionPerformed(...) method. Is that the only listener available for the JButton class? If you think so, you are sadly mistaken and in for a big surprise. Listing 14-1 shows the various listener methods and class hierarchy where each is defined.

Listing 14-1. JButton Listeners

```
wsadmin>from javax.swing import JButton
wsadmin>
wsadmin>classInfo( JButton, meth = 'listener' )
javax.swing.JButton
| javax.swing.AbstractButton
> addActionListener, addChangeListener, getActionListeners >
getChangeListeners, getItemListeners, removeActionListener >
removeChangeListener
| | javax.swing.JComponent
> > addAncestorListener, addVetoableChangeListener > > getAncestorListeners,
getVetoableChangeListeners > > removeAncestorListener,
removeVetoableChangeListener | | | java.awt.Container
> > > addContainerListener, addPropertyChangeListener > > >
getContainerListeners, removeContainerListener | | | | java.awt.Component
> > > > addComponentListener, addFocusListener
> > > > addHierarchyBoundsListener, addHierarchyListener > > > >
addInputMethodListener, addKeyListener
> > > > addMouseListener, addMouseMotionListener
> > > > addMouseWheelListener, addPropertyChangeListener > > > >
getComponentListeners, getFocusListeners
> > > > getHierarchyBoundsListeners, getHierarchyListeners > > > >
getInputMethodListeners, getKeyListeners, getListeners > > > >
getMouseListeners, getMouseMotionListeners > > > >
getMouseWheelListeners, getPropertyChangeListeners > > > >
removeComponentListener, removeFocusListener > > > >
removeHierarchyBoundsListener, removeHierarchyListener > > > >
```

removeInputMethodListener, removeKeyListener > > > >
removeMouseListener, removeMouseMotionListener > > > >
removeMouseWheelListener, removePropertyChangeListener | | | | |
java.lang.Object
| | | | | java.awt.image.ImageObserver
| | | | | java.awt.MenuContainer
| | | | | java.io.Serializable
| | | java.io.Serializable
| | java.awt.ItemSelectable
> > addItemListener, removeItemListener
| | javax.swing.SwingConstants
| javax.accessibility.Accessible
wsadmin>

[5]The primary exception was shown in Table 6-2, where there were more than a dozen listeners for the JTextArea class.

Is it likely that your applications will be using all of these listeners? No, but there are quite a few that are worth investigating. The information in Listing 14-1 shows a large number of listener methods that are part of the java.awt.Component class. The variety and type of listeners in the hierarchy should encourage you, as a developer, and give you confidence that your application will be able to monitor any type of event you need.

Put Your Listener Where Your Component Is

In this section, you learn what it takes to create a MouseListener[6] for a component.[7] Before you begin, though, it is important to realize that if your application is interested in mouse movement, this requires a different kind of listener (MouseMotionListener).[8] For the moment, however, this section focuses on what it takes to create a MouseListener.

Table 14-1 shows the events defined by the MouseListener class. Each of these methods has a MouseEvent parameter that is passed to the event handler method.

[6] See http://docs.oracle.com/javase/8/docs/api/java/awt/event/MouseListener.html.
[7]It's interesting to note that, at least at the time of this writing, the version 7

java.awt.event.MouseListener Javadoc page has a bad link. The “Tutorial: Writing a Mouse Listener” link should actually point to http://docs.oracle.com/javase/tutorial/ uiswing/events/mouselistener.html. This bad link has been corrected on the version 8 page.

[8]See http://docs.oracle.com/javase/8/docs/api/java/awt/event/MouseMotionListener.ht

Table 14-1. *MouseListener Methods*

Method Name	**Invoked When the Mouse** ...
mouseEntered(...)	Enters a component.
mousePressed(...)	Button has been pressed on a component.
mouseReleased(...)	Button has been released on a component.
mouseClicked(...)	Button has been pressed and released on a component.
mouseExited(...)	Exits a component.

Listing 14-2 shows just how easily you can add a MouseListener to a component like a button. Note how the listener class, in lines 10-24, needs to be provided with access to the specific application component, in this case, a JTextArea. This is because it’s external to the Listen1 application class.

Listing 14-2. Adding a MouseListener to a Button

```
10|class listener( MouseListener ) :
11| def __init__( self, textArea ) :
12| self.textArea = textArea
13| def mouseClicked( self, me ) :
14| self.logEvent( me )
15| def mouseEntered( self, me ) :
16| self.logEvent( me )
17| def mouseExited( self, me ) :
18| self.logEvent( me )
19| def mousePressed( self, me ) :
20| self.logEvent( me )
```

```
21| def mouseReleased( self, me ) :
22| self.logEvent( me )
23| def logEvent( self, me ) :
24| self.textArea.append( me.toString() + '\n' )
25|class Listen1( java.lang.Runnable ) :
26| def run( self ) :
27| frame = JFrame(
28| 'Listen1',
29| layout = FlowLayout(),
30| locationRelativeTo = None,
31| size = ( 512, 256 ),
32| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
33| )
34| self.button = frame.add( JButton( 'Button' ) )
35| self.textarea = JTextArea( rows = 10, columns = 40 )
36| self.button.addMouseListener( listener( self.textarea ) )
37| frame.add( JScrollPane( self.textarea ) )
38| frame.setVisible( 1 )
```

Figure 14-1 shows an image from the Listen1.py application. In this case, I moved the mouse over the button, clicked the button, and then moved the mouse elsewhere. Figure 14-1 shows every event that occurred, in the order that they are shown in Table 14-1.

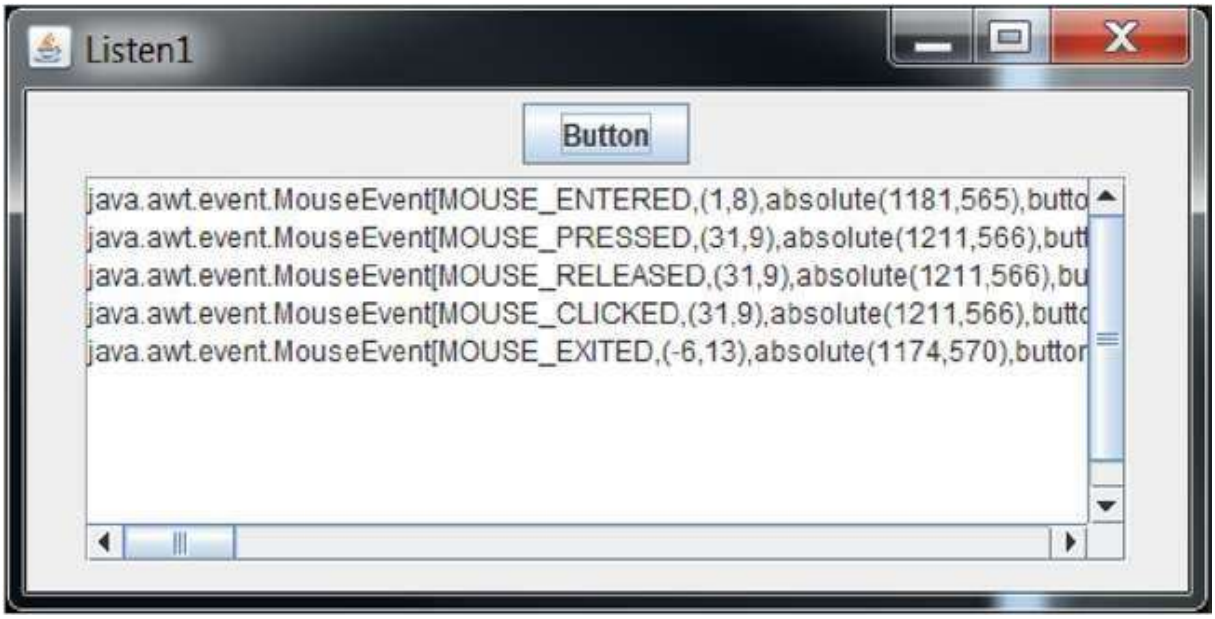


Figure 14-1. *Listen1.py sample output*

What if you aren't interested in monitoring all of the possible events that class provides? Well, one trivial technique is to have the uninteresting methods do nothing (they can contain a simple, pass or return statement). This can be a possible source of confusion and error. I've had at least one instance where I didn't realize that I duplicated a listener method name, where the second instance inadvertently replaced the first. It took me a while to realize and resolve my mistake. What can you do about this kind of situation?

Adapt or Die: Using Adapter Classes

The Swing designers have been very kind to its developers. The hierarchy includes about two dozen "adapter" classes that contain most, if not all, of the methods you'll need, all with empty placeholder statements. Using these as a base class is a great place to start.

One way to find the class you need is by looking at the Javadoc and paying particular interest to the "See Also" sections. For example, the MouseListener Javadoc[6] has, as its first reference in this section, the MouseAdapter class.[9] Using that as a base class for your listener allows you to simplify the listener class from the Listen1 application. Listing 14-2 contains the original listener class in lines 10-24. Listing 14-3 shows a simplified class that assumes you are interested in one only of the available methods.

Listing 14-3. Listen2.py Using MouseAdapter

```
10|class listener( MouseAdapter ) :
11| def __init__( self, textArea ) :
12| self.textArea = textArea
13| def mouseClicked( self, me ) :
14| self.textArea.append( me.toString() + '\n' )
```

Figure 14-2 shows sample output from this iteration of the Listen#.py script. Note that Listen2.py is only interested in mouseClicked events, so you only see references to those events in the text area. You no longer see the other events shown in Figure 14-1.

[9]See http://docs.oracle.com/javase/8/docs/api/java/awt/event/MouseAdapter.html.

Figure 14-2. *Listen2.py sample output*

Listening for Keyboard Events

What if you wanted your application to monitor keystrokes as they are entered, so that the application can be dynamically updated to reflect various information about the user input? If your application has more than one kind of information to be displayed, based on this input, this could make things a little interesting for the developers. One approach would be to have one KeyListener for the user input field that updates multiple fields based on the user's input. One problem with this, though, is that as you continue to add fields to the application, this listener becomes more complex.

Another approach is to have a more generic listener class that is used to monitor one simple property, and then update a single application field based on this property. What would this kind of listener class need in order to perform this role? It would need to have the following parts of the application specified during its instantiation or construction:

• The input field being monitored
• The output field to be updated
• The function used to determine the result displayed in the output field

Say you had a single input field that you wanted to monitor. When the user enters a value, it needs to be checked to see if the value is an even number

(integer). That's not too difficult. What if you also wanted the field to indicate when the value was an odd integer value? It might also be interesting to have a field display whether the value is a prime number.

In order to create this kind of application, you need to understand the kinds of events that are generated for KeyListener objects.[10] How does this class work? Table 14-2 shows the three methods that are part of the KeyListener interface. Each event handler method has a KeyEvent argument that will be passed to the method.

[10]See http://docs.oracle.com/javase/8/docs/api/java/awt/event/KeyListener.html.

Table 14-2. *KeyListener Methods*

Method Name	**Description**
keyPressed(...)	Invoked when a key is pressed.
keyReleased(...)	Invoked when a key is released.
keyTyped(...)	Invoked when a key is typed.

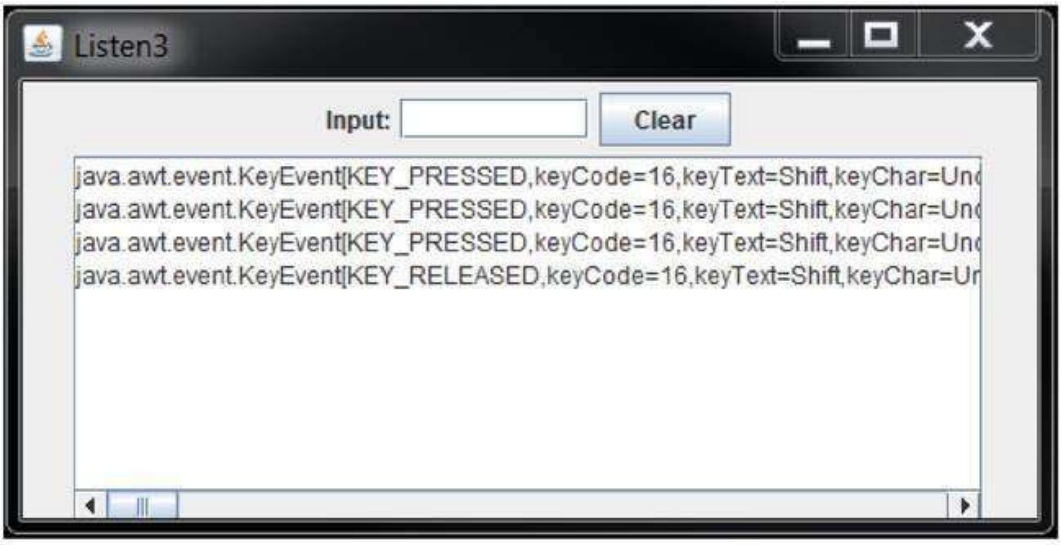
Unfortunately, the description for each method is kind of lacking. What does each mean, and what's the difference between them? When might you be concerned with using one method versus another?
The best way to answer questions like this is to write a program that tries them out. You can then learn more about the methods by seeing the output they produce. Using Listen1.py as a starting point, you can replace MouseListener with KeyListener, replacing each MouseListener method with the appropriate KeyListener method. The result is shown in Listing 14-4.

Listing 14-4. KeyListener Descendant Class from Listen3.py

```
13|class listener( KeyListener ) :
14| def __init__( self, textArea ) :
15| self.textArea = textArea
16| def keyPressed( self, ke ) :
17| self.logEvent( ke )
18| def keyReleased( self, ke ) :
```

```
19| self.logEvent( ke )
20| def keyTyped( self, ke ) :
21| self.logEvent( ke )
22| def logEvent( self, ke ) :
23| self.textArea.append( ke.toString() + '\n' )
```

Sample output from the Listen3.py application is shown in Figure 14-3. The first image shows what happens when you press and hold (and finally release) the Right-Shift key. Notice how multiple KEY_PRESSED events, only one KEY_RELEASED event, and no KEY_TYPED events were generated. The second image shows the key events that were generated when you quickly press and release the 0 key. And the third image shows the keystroke events that can occur when a normal key, in this case the 0 key, is pressed and held just long enough to generate two repeated 0's in the input field.


Listen3
Input:
Clear
java.awt.event.KeyEvent[KEY_PRESSED,keyCode=16,keyText=Shift,keyChar=Unc
java.awt.event.KeyEvent[KEY_PRESSED,keyCode=16,keyText=Shift,keyChar=Unc
java.awt.event.KeyEvent[KEY_PRESSED,keyCode=16,keyText=Shift,keyChar=Unc
java.awt.event.KeyEvent[KEY_RELEASED,keyCode=16,keyText=Shift,keyChar=Ur


Listen3
Input: 0
Clear
java.awt.event.KeyEvent[KEY_PRESSED,keyCode=48,keyText=0,keyChar='0',keyL
java.awt.event.KeyEvent[KEY_TYPED,keyCode=0,keyText=Unknown keyCode: 0x0
java.awt.event.KeyEvent[KEY_RELEASED,keyCode=48,keyText=0,keyChar='0',key

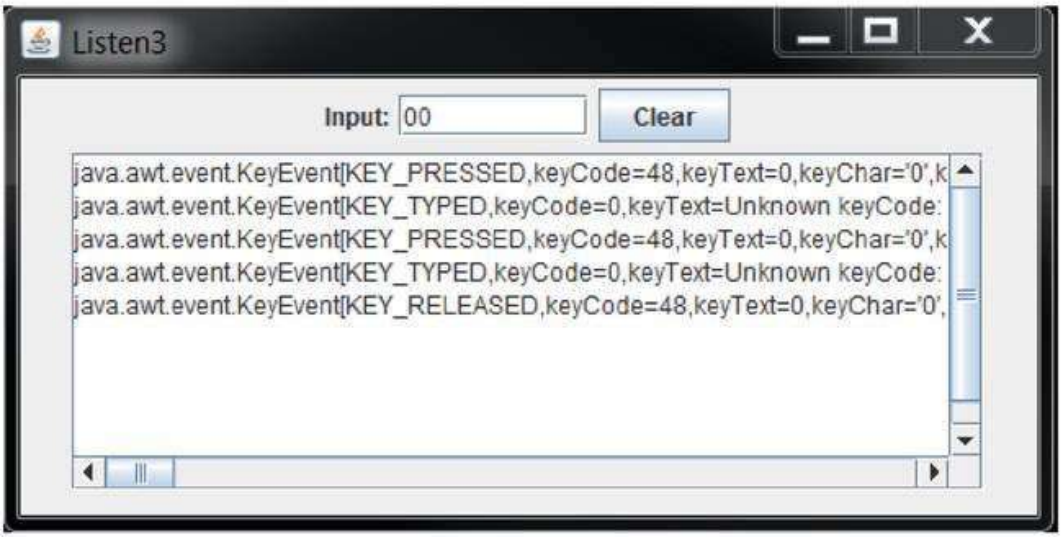

Listen3
Input: 00
Clear
java.awt.event.KeyEvent[KEY_PRESSED,keyCode=48,keyText=0,keyChar='0',k
java.awt.event.KeyEvent[KEY_TYPED,keyCode=0,keyText=Unknown keyCode:
java.awt.event.KeyEvent[KEY_PRESSED,keyCode=48,keyText=0,keyChar='0',k
java.awt.event.KeyEvent[KEY_TYPED,keyCode=0,keyText=Unknown keyCode:
java.awt.event.KeyEvent[KEY_RELEASED,keyCode=48,keyText=0,keyChar='0',

Figure 14-3. *Listen3.py sample output*
From these images, you can see that:
• For repeated keystrokes of characters that don’t produce input (such as the Shift key), only KEY_PRESSED events occur.
• For repeated keystrokes of characters that produce input (such as any of the letters and the numeric keys), the KEY_PRESSED and KEY_TYPED events occur in an alternating pattern.
• The KEY_RELEASED event only occurs once for each key.

You are encouraged to use the Listen3.py script and see the events that are generated when you press and release combinations of keys (such as Ctrl-Shift-Spacebar). Are you surprised, or do you observe what you expected? This exercise should help you better understand the keystroke events that are generated.

Listing 14-5 shows almost all of the Listen3 class that was used to produce the output shown in Figure 14-3. At this point, you shouldn’t be too surprised by how easy it is to create this kind of application.
Listing 14-5. Listen3 Class from Listen3.py

```
24|class Listen3( java.lang.Runnable ) :
25| def run( self ) :
26| frame = JFrame(
27| 'Listen3',
28| layout = FlowLayout(),
29| locationRelativeTo = None,
30| size = ( 512, 256 ),
31| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
32| )
33| frame.add(
34| JLabel(
35| 'Input:',
36| horizontalAlignment = SwingConstants.RIGHT
37| )
38| )
39| self.text = frame.add( JTextField( 8 ) )
40| frame.add(
41| JButton(
42| 'Clear',
```

```
43| actionPerformed = self.clear
44| )
45| )
46| self.textArea = JTextArea( rows = 10, columns = 40 )
47| frame.add( JScrollPane( self.textArea ) )
48| self.text.addKeyListener( listener( self.textArea ) )
49| frame.setVisible( 1 )
```

Listing 14-6, on the other hand, may contain a surprise or two. Initially, I intended that the button should only be used to clear the application TextArea. However, after playing with the application for a very short time, I soon realized that it would also be useful to clear the input text field. The potential surprise, however, is related to the last method call in this routine.

What, exactly, does the requestFocusInWindow() method do, and how did I know to use it? Before including this method call, I was forced to use the mouse to click on the input field to give it focus after the button was pressed. The alternative would be to use the Tab key to move the focus from the button to the input field. The problem with this approach is that it generates input that's written to TextArea.

So, I looked for methods in JTextField that had "focus" in their name. The first one that I found was JComponent.requestFocus(),[11] which looked promising, at least until I read that the use of this routine is discouraged. The good news is that the documentation for this method also provided the name of the recommended routine to be used, instead. So, the purpose of the call to the requestFocusInWindow() method is to restore focus to the input field after the button is pressed.

Listing 14-6. The ActionListener Method Used by the Button

```
50| def clear( self, event ) :
51| self.text.setText( " )
52| self.textArea.setText( " )
53| self.text.requestFocusInWindow()
```

All in all, this is a useful application, especially when you are interested in learning about keystroke events that can be generated and the sequence in which they occur.

Most Objects Never Really Listen

At the beginning of this chapter, you read that there are times when it makes sense for multiple listeners to monitor a particular component for the same kind of event. Fortunately, this is something that can easily be accomplished. Before you looked at the kind of keystroke events that can be generated, you saw an application that had multiple listeners monitoring an input field, each one updating a component based on the value contained in the input field.

Before you look at an example, take a moment to think about situations in which it would make sense to do this. Listing 14-7 provides one example. The class constructor for this listener requires that three things be provided:

- The input field being monitored
- The output field to be updated (in this example, a JLabel)
- The function to be used to determine the message value to be specified

Listing 14-7. Listener Class from Listen4.py

```
10|class listener( KeyAdapter ) :
11| def __init__( self, input, msg, fun ) :
12| self.input = input
13| self.msg = msg
14| self.fun = fun
15| def keyReleased( self, ke ) :
16| text = self.input.text
17| if text :
18| try :
19| value = int( self.input.text )
20| msg = [ 'No', 'Yes' ][ self.fun( value ) ]
21| except :
22| msg = 'invalid integer'
23| else :
24| msg = ''
25| self.msg.setText( msg )
```

This listener class, as you can see, descends from the KeyAdapter class,[12] not from the abstract KeyListener class.[10] This allows you to simplify the class because you can choose to implement only the keyReleased(...) method.[13] Maybe you are confused by the syntax shown on line 20. Here, we take

advantage of some of the power of Jython. The first part of the expression—“['No', 'Yes']”—identifies a list containing two values. The second part of the expression—“[self.fun(value)]”—identifies the index used to select the desired value from the list. In order to determine the value to be used, which should be zero or one, you call the user-specified function (provided when the class was instantiated; see lines 11-14) and pass the value of the user-specified input field (after converting it from a string to an integer).

[11]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JComponent.html#requestF

■ **Note** In case you are wondering, the try/except statement is necessary in case the user specifies an invalid input and the conversion fails.

Listing 14-8. Listen4 Class from Listen4.py

```
26|class Listen4( java.lang.Runnable ) :
27| def run( self ) :
28| def isEven( num ) :
29| return not ( num & 1 )
30| def isOdd( num ) :
31| return num & 1
32| def isPrime( num ):
33| result = 0 # Default = False
34| if num == abs( int( num ) ) : # Only integers allowed
35| if num == 1 : # Special case
36| pass # use default (false)
37| elif num == 2 : # Special case
38| result = 1 #
39| elif num & 1 : # Only odd numbers...
40| for f in xrange( 3, int( num**0.5 ) + 1, 2 ) :
41| if not num % f :
42| break # f is a factor...
43| else :
44| result = 1 # we found a prime
45| return result
46| def label( text ) :
47| return JLabel(
48| text + ' ',
49| horizontalAlignment = SwingConstants.RIGHT
50| )
```

```
51| frame = JFrame(
52| 'Listen4',
53| layout = GridLayout( 0, 2 ),
54| locationRelativeTo = None,
55| size = ( 200, 128 ),
56| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 57| )
58| frame.add( label( 'Integer:' ) )
59| text = frame.add( JTextField( 10 ) )
60| frame.add( label( 'Even?' ) )
61| even = frame.add( JLabel( '' ) )
62| text.addKeyListener( listener( text, even, isEven ) ) 63| frame.add( label(
'Odd?' ) )
64| odd = frame.add( JLabel( '' ) )
65| text.addKeyListener( listener( text, odd, isOdd ) ) 66| frame.add( label(
'Prime?' ) )
67| prime = frame.add( JLabel( '' ) )
68| text.addKeyListener( listener( text, prime, isPrime ) ) 69| frame.setVisible( 1
)
```

[12] See http://docs.oracle.com/javase/8/docs/api/java/awt/event/KeyAdapter.html.
[13]Initially, I mistakenly used the keyTyped(...) method, thinking that it would be the one invoked after the keystroke had been processed. But I was wrong. Fortunately, it was a trivial thing to change the name of the routine fromkeyTyped(...) to keyReleased(...).

Lines 62, 65, and 68 of Listing 14-8 show how the application has three KeyListener methods for the same input field (the JTextField), as well as specifying different output (JLabel) fields and functions to be called to determine the message to be displayed in each.

Figure 14-4 shows some sample output from the Listen4.py application, which has multiple listeners for a single input field.

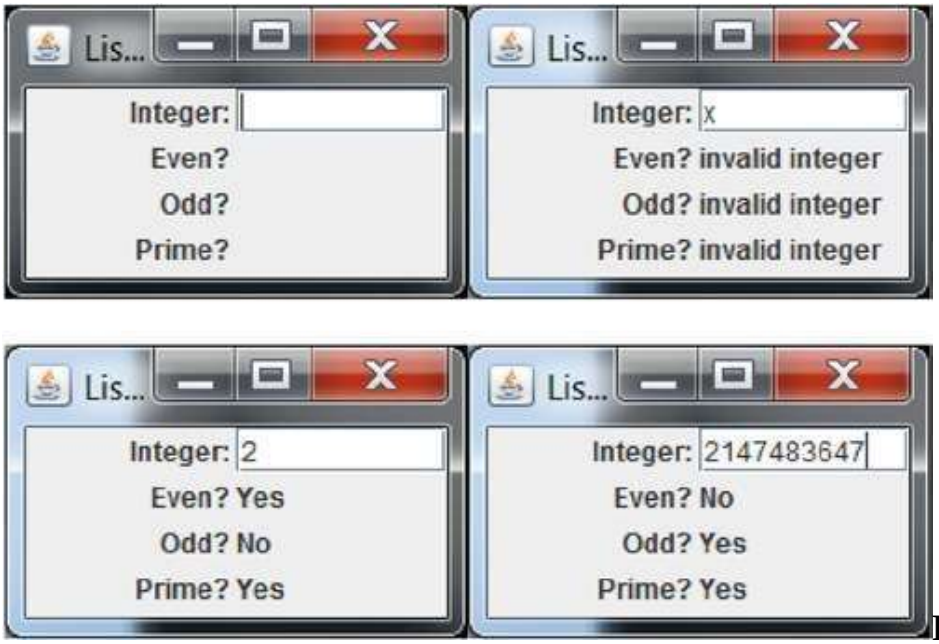


Figure 14-4. *Listen4.py output*

Looking for a Listener in a Haystack

Up to this point, things have sort of been handed to you on a platter, so to speak. Unfortunately, that approach is only useful up to a point. Let's take a look at how a developers can determine the best way to do something.

One of the many challenges with designing and developing an application is trying to figure out what size it should be. Along those lines, I wondered how much time and effort would be required to have the application monitor any resize requests that might occur.

Why would I want to do this? Consider, for a moment, that I have no idea how large or small for that matter, to make my application. I could simply let Swing determine the size for me by using the JFrame pack()(inherited) method.

Unfortunately, I haven't always been happy with the default size. So, I did a quick look at the listeners and the Java Swing Tutorial[14] to see if I could find something useful. It didn't take me long to realize that my photographic memory was out of film, and I would have to use a different technique.

Using one of the various Internet search engines, you can search for "java swing resize listener," which will quickly point you to the "How to Write a Component

Listener" swing tutorial page.[15] Looking at the sample application on that page, called ComponentEventDemo.java,[16] I was able to find some things worth investigating.

• This application, like most from the Java Swing Tutorial pages, has the main(...) method use an anonymous Runnable class within its call to the SwingUtilities invokeLater(...) method.

• Invariably, the run(...) method in this class simply calls the createAndShowGUI(...) method within the application class.

• This application class (ComponentEventDemo) extends JPanel, so the createAndShowGUI(...) method creates JFrame instances and uses the JPanel that is created by the application class as a replacement for the JFrame ContentPane. I haven't used this technique, but it appears to be a common one used by many of the sample applications in the Swing Tutorial pages.

• The sample applications that extend a base class like JPanel almost always use the Java technique of calling the super(...) method to invoke the base class constructor. Unfortunately, this isn't available to Jython developers, so instead, you'll have to invoke the base class constructor—that is, call the JPanel.__init__(self) method.

• The only expression that can't be used "as-is" occurs in the component event handler methods, and looks like this: e.getComponent().getClass().getName(). If you try to use expressions like these in your Jython scripts, you'll get an exception something like this:

TypeError: getName(): expected 1 args; got 0 [14]See https://docs.oracle.com/javase/tutorial/uiswing/. [15]See http://docs.oracle.com/javase/tutorial/uiswing/events/componentlistener.html. [16]See http://docs.oracle.com/javase/tutorial/uiswing/examples/events/ComponentEvent ComponentEventDemo.java.
The reason that this happens is because Jython has two ways for calling a method instance[17]:

• theInstance.theMethod(args)
• TheClass.theMethod(theInstance, args)

Unfortunately, you are trying to use the first technique, but Jython prefers the second. How can you fix this? You have a couple of different options available. You can use the syntax preferred by Jython, which is an expression like this one:

```
java.lang.Class.getName( e.getComponent().getClass() )
```

Unfortunately, this results in a string something like " javax.swing.JButton", instead of the preferred "JButton". A somewhat simpler approach is to use the fact that you can use str(e.getComponent().getClass()) to get this same string value, then split the string with ". " as a delimiter. Finally, you use the negative list indexing technique to access the final part of the list of strings:

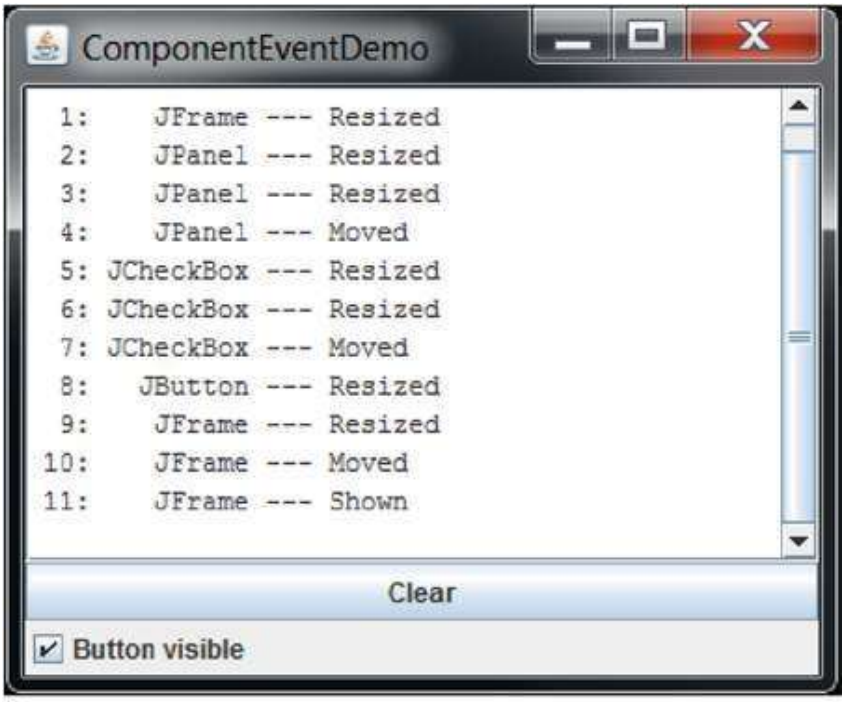
```
name = str( e.getComponent().getClass() ).split( '.' )[ -1 ]
```

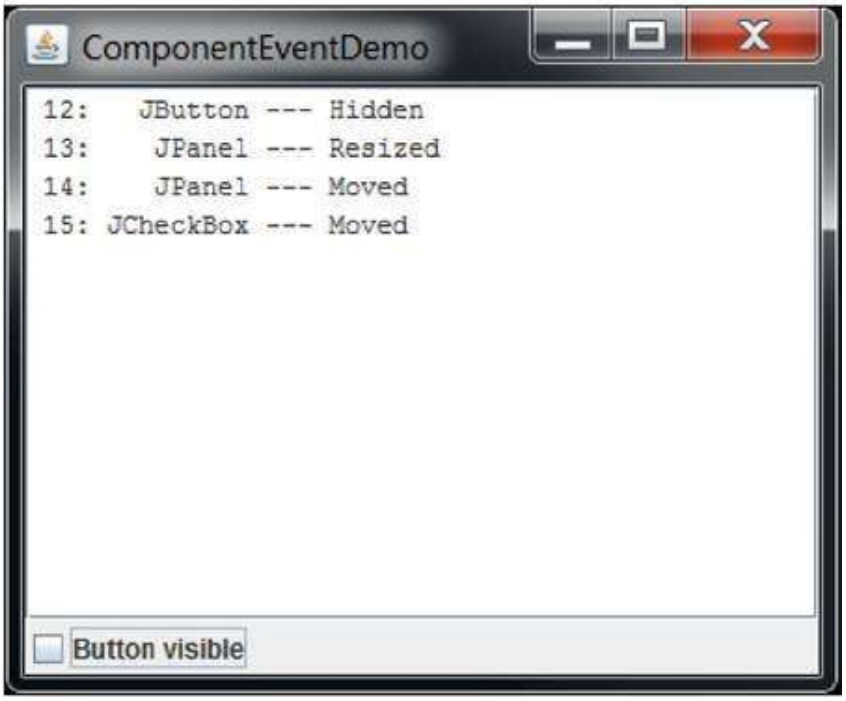
Before you take a look at Jython script that I produced, try to convert the original Java application to Jython yourself. This exercise should provide you with some practice at reading an original Java Swing application and trying to produce a script using the same kind of components and organization.

Using the original Java version—or the one you created, or even the one that can be found in code\Chap_14\ ComponentEventDemo.py—take a look at the events generated when you do something simple like make the application window a tiny bit taller. Figure 14-5 shows multiple images produced by using the Jython script at various times. The most interesting one, at least to me, is the last image, which shows just how many "Moved" events are generated when you make the window a little bit larger.[18] Also interesting is the fact that only one "Resized" event is generated. The really interesting part is that the event identifies the component that was resized, in this case, the JFrame object.

[17]Thanks to Jeff Emanuel for providing this explanation on the jython-users mailing list.

[18]This is why my version of the script includes a counter showing the number of events that have been encountered.


ComponentEventDemo
1: JFrame --- Resized
2: JPanel --- Resized
3: JPanel --- Resized
4: JPanel --- Moved
5: JCheckBox --- Resized
6: JCheckBox --- Resized
7: JCheckBox --- Moved
8: JButton --- Resized
9: JFrame --- Resized
10: JFrame --- Moved
11: JFrame --- Shown
Clear
Button visible


ComponentEventDemo
12: JButton --- Hidden
13: JPanel --- Resized
14: JPanel --- Moved
15: JCheckBox --- Moved
Button visible

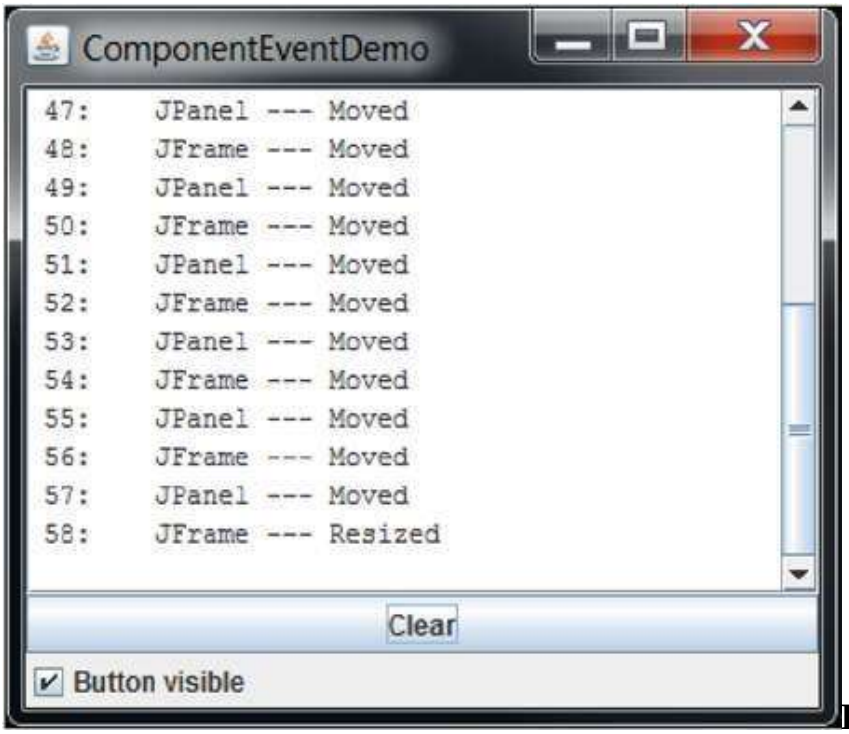


Figure 14-5. *ComponentEventDemo sample output*

Using a ComponentAdapter to Monitor Changes

Using this ComponentEventDemo application made me think about using a ComponentListener[15] to gain a better understanding about the size and position of an application window. Of course, if you aren't interested in monitoring all of the Component events, you might want to consider using a ComponentAdapter[19] instead.

What do you think it would take to have an application that displays the width and height of the frame? The images in Figure 14-6 show one possible representation. Take a few moments and see if you can figure out how to do this before looking at the source, which is available in code\Chap_14\Frame1.py.

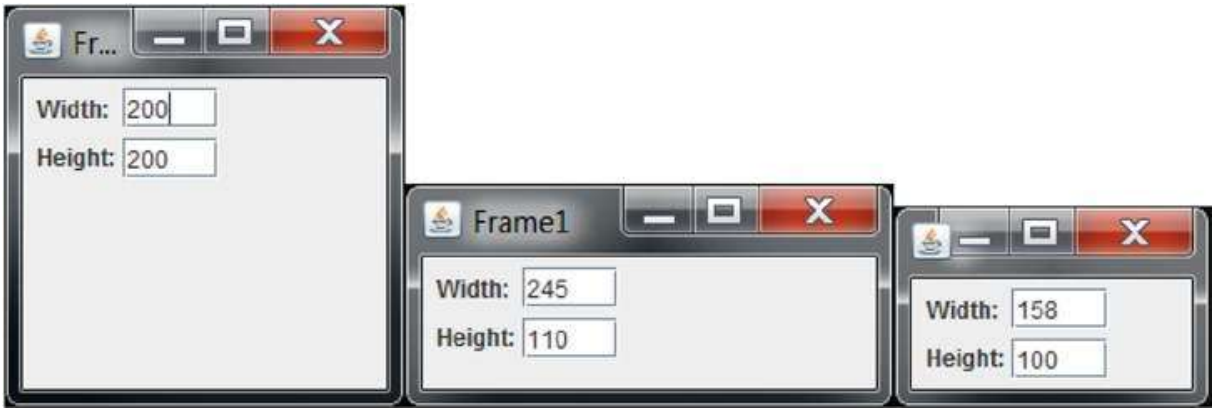


Figure 14-6. *Frame1 sample output*

Issues you might want to consider:
• Which Layout Manager do you want to use?
• For which events does the application need to listen?

Listing 14-9 shows the Frame1 class from the script of the same name. Notice how the Layout Manager attribute of the frame constructor is initialized to None. That means that this application is going to position each component by specifying its size and location.

Listing 14-9. Frame1 Class from Frame1.py

```
16|class Frame1( java.lang.Runnable ) :
17| def run( self ) :
18| self.frame = frame = JFrame(
19| 'Frame1',
20| size = ( 200, 200 ),
21| layout = None,
22| locationRelativeTo = None,
23| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
24| )
25| frame.addComponentListener( listener( self ) )
26| insets = frame.getInsets()
27| self.width = JTextField( 4 )
28| self.height = JTextField( 4 )
29| items = [
30| [ JLabel( 'Width:' ), 7, 7 ], 31| [ self.width , 50, 5 ], 32| [ JLabel( 'Height:' ), 7,
31 ], 33| [ self.height , 50, 30 ] 34| ]
35| for item in items :
```

```
36| thing = frame.add( item[ 0 ] ) 37| size = thing.getPreferredSize() 38|
thing.setBounds(
39| insets.left + item[ 1 ], 40| insets.top + item[ 2 ], 41| size.width,
42| size.height
43| )
44| frame.setVisible( 1 )
```

[19]See http://docs.oracle.com/javase/8/docs/api/java/awt/event/ComponentAdapter.html.

Listing 14-10 shows the simple listener class that is a descendant of the abstract ComponentAdapter class. This allows you to simplify your class and only specify methods of interest, which in this case is the componentResized() method.[20]

Listing 14-10. Listener Class from Frame1.py

```
8|class listener( ComponentAdapter ) : 9| def __init__( self, app ) :
10| self.app = app
11| def componentResized( self, ce ) : 12| app = self.app
13| size = app.frame.getSize() 14| app.width.setText( `size.width` ) 15|
app.height.setText( `size.height` )
```

What would it take to have an application that shows the position or location on the screen, in addition to the size? It would be a good exercise for you to try to expand Frame1.py to add this kind of functionality. Why don't you take some time to do this before you take a look at the one found in code\Chap_14\Frame2.py?

Listing 14-11 shows the revised listener class that was used to generate the output shown in Figure 14-7. Note how additional ComponentListener methods have been included. This allows the application to update the position, that is the X and Y fields, as the application frame is moved. In order to do this, the frame.getBounds() method is called to acquire the current size and position of the frame. In Listing 14-8, you were only using the size of the application, so the frame.getSize() method was used instead.

[20]If you are unfamiliar with the backtic operator, it converts the expression within as a printable string. See the Jython repr() built-in function.

Listing 14-11. Listener Class from Frame2.py

8|class listener(ComponentAdapter) : 9| def __init__(self, app) :
10| self.app = app
11| def updateInfo(self) :
12| app = self.app
13| bounds = app.frame.getBounds() 14| app.width.setText(`bounds.width`) 15| app.height.setText(`bounds.height`) 16| app.x.setText(`bounds.x`)
17| app.y.setText(`bounds.y`)
18| def componentMoved(self, ce) : 19| self.updateInfo()
20| def componentResized(self, ce) : 21| self.updateInfo()
22| def componentShown(self, ce) : 23| self.updateInfo()

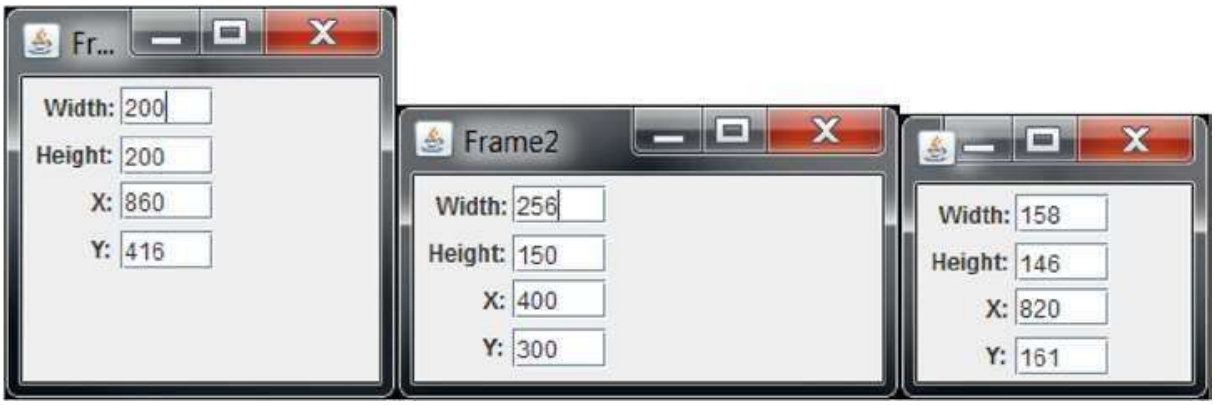


Figure 14-7. *Frame2 sample output*

Listing 14-12 shows just how little needed to be modified in order to have the application monitor the location as well as the size. It also shows how tedious it can be to determine the components' positions so that they can be identified without using a Layout Manager.

Why did I choose to use the absolute layout technique? The primary reason was because I didn't want the components to move around on the application as it moved and was resized.

Listing 14-12. Modified Statements from the Frame2 Class

24|class Frame2(java.lang.Runnable) :
| ...
34| self.width = JTextField(4)
35| self.height = JTextField(4)
36| self.x = JTextField(4)

```
37| self.y = JTextField( 4 )
38| items = [
39| [ JLabel( 'Width:' ) , 11, 7 ],
40| [ self.width , 50, 5 ],
41| [ JLabel( 'Height:' ), 7, 31 ],
42| [ self.height , 50, 30 ], 43| [ JLabel( 'X:' ) , 35, 55 ], 44| [ self.x , 50, 53 ], 45| [
JLabel( 'Y:' ) , 35, 79 ], 46| [ self.y , 50, 78 ] 47| ]
| ...
```

Monitoring the Input Fields

Using the mouse to move and resize the application window can be a frustrating experience, especially if you are trying to precisely adjust a specific value (such as the height). Wouldn’t it be useful to also allow users to enter a value in any of these input fields as well? If you do allow user input, you’ll need to verify that input before it is used. For example, it doesn’t make any sense to allow users to specify a negative width or height. On the other hand, it may make sense for the location values to be negative. On my system, I have a second display, and when I put the application frame in the middle of it, the value of X is -775.

So how do you go about monitoring the input fields? One way is to have an ActionListener for each field. In fact, you could have a different one for each field so that you don’t have to figure out which value was modified. Listing 14-13, which can be found in Frame3.py, shows one way that this can be done.

Listing 14-13. Unique ActionListener for Each Input Field

```
24|class Frame3( java.lang.Runnable ) :
25| def changeWidth( self, event ) :
26| value = event.getActionCommand()
27| try :
28| width = int( value )
29| size = self.frame.getSize()
30| self.frame.setSize( width, size.height )
31| except :
32| print 'Invalid Width: "%s"' % value | ...
57| def run( self ) :
| ...
66| self.width = JTextField(
```

```
67| 4, actionPerformed = self.changeWidth
68| )
69| self.height = JTextField(
70| 4, actionPerformed = self.changeHeight
71| )
72| self.x = JTextField(
73| 4, actionPerformed = self.changeX
74| )
75| self.y = JTextField(
76| 4, actionPerformed = self.changeY
77| )
| ...
```

Unfortunately, this isn't a great approach for a number of reasons, including these:

• Using ActionListeners means that the values aren't verified until the user presses Enter, which isn't very user friendly. It would probably be better if verification occurred when the user does one of the following:

• Presses the Enter key
• Tries to change focus (using the Tab key, Shift-Tab, or even a mouse event)
• Enters the text

• Each ActionListener is very similar. You might be able to refactor, but the technique used to indicate that an invalid value was specified isn't very good or user friendly. Nor does it use the Swing interface, which is a bad thing.

Back in Chapter 7, you read a little about the JFormattedTextField class,[21] including how it is used to restrict user input. Let's revisit that class and see if it can help you improve the user experience of this application. What part of a JFormattedTextField will help you with this application? Well, there are some things in the Javadoc that you should be aware of. Specifically, it contains information about using one of the following:

• A PropertyChangeListener[22] for monitoring editing changes
• An InputVerifier[23] to keep focus from being lost when an invalid value is specified

Using a PropertyChangeListener

One of the interesting things about the JFormattedTextField class is that, in addition to the text attribute that it inherits from JTextField, it also defines a value attribute.[24] Why does it have both a text and a value attribute? More importantly, what does that mean for your applications?

Consider the following scenario: a formatted text field has some kind of value, for example, some kind of currency. When this field has focus, it may be an indication that the user wants to modify the value, which is great. Users start typing and as they do so, the text that they enter might not be valid. As user input is being provided, the text attribute reflects what the user has entered. When entry is complete, the contents of the text attribute can be used to determine if the information is valid or appropriate for the field.

So, the value attribute maintains a valid value, formatted using the appropriate pattern. The text attribute, on the other hand, reflects the current, unverified, and unformatted user input. It's when the user entry process is complete that the verification and formatting can occur, but only if the user-specified value is acceptable. At least that way it is supposed to work, if your applications use the available capabilities.

Let's start by looking at the PropertyChangeListener to see how it fits into this strategy. To begin, consider an application that has a couple of formatted text fields with a property change listener event handler routine to tell you about the PropertyChangeEvents that occur. This routine can update a text area with information about the events, including the PropertyName, the OldValue, and the NewValue attributes identified by PropertyChangeEvent.[25]

Listing 14-14 shows the PropertyChangeListener and ActionListener event handler routines from the PropertyListener.py sample application.

[21] See http://docs.oracle.com/javase/8/docs/api/javax/swing/JFormattedTextField.html.
[22]See http://docs.oracle.com/javase/8/docs/api/java/beans/PropertyChangeListener.html
[23]See http://docs.oracle.com/javase/8/docs/api/javax/swing/InputVerifier.html.
[24]In spite of the fact that the Javadoc for this class doesn't identify this fact, at least not explicitly in the "Field Summary" section.
[25]See

http://docs.oracle.com/javase/8/docs/api/java/beans/PropertyChangeEvent.html.

Listing 14-14. PropertyChangeListener Event Handler Routine

```
17|class PropertyListener( java.lang.Runnable ) :
18| def changed( self, pce ) :
19| format = ' Name: %(name)s\n'
20| format += 'OldValue: %(old)s\n'
21| format += 'NewValue: %(new)s\n\n'
22| name = pce.getPropertyName()
23| old = 'pce.getOldValue()'
24| new = 'pce.getNewValue()'
25| self.textArea.append( format % locals() )
26| def clear( self, e ) :
27| self.textArea.setText( '' )
```

Figure 14-8 shows some images from this application. I'm not sure what I was expecting, but I didn't realize the kind of PropertyChangeEvents that would be generated (which is exactly why this kind of script is so useful).

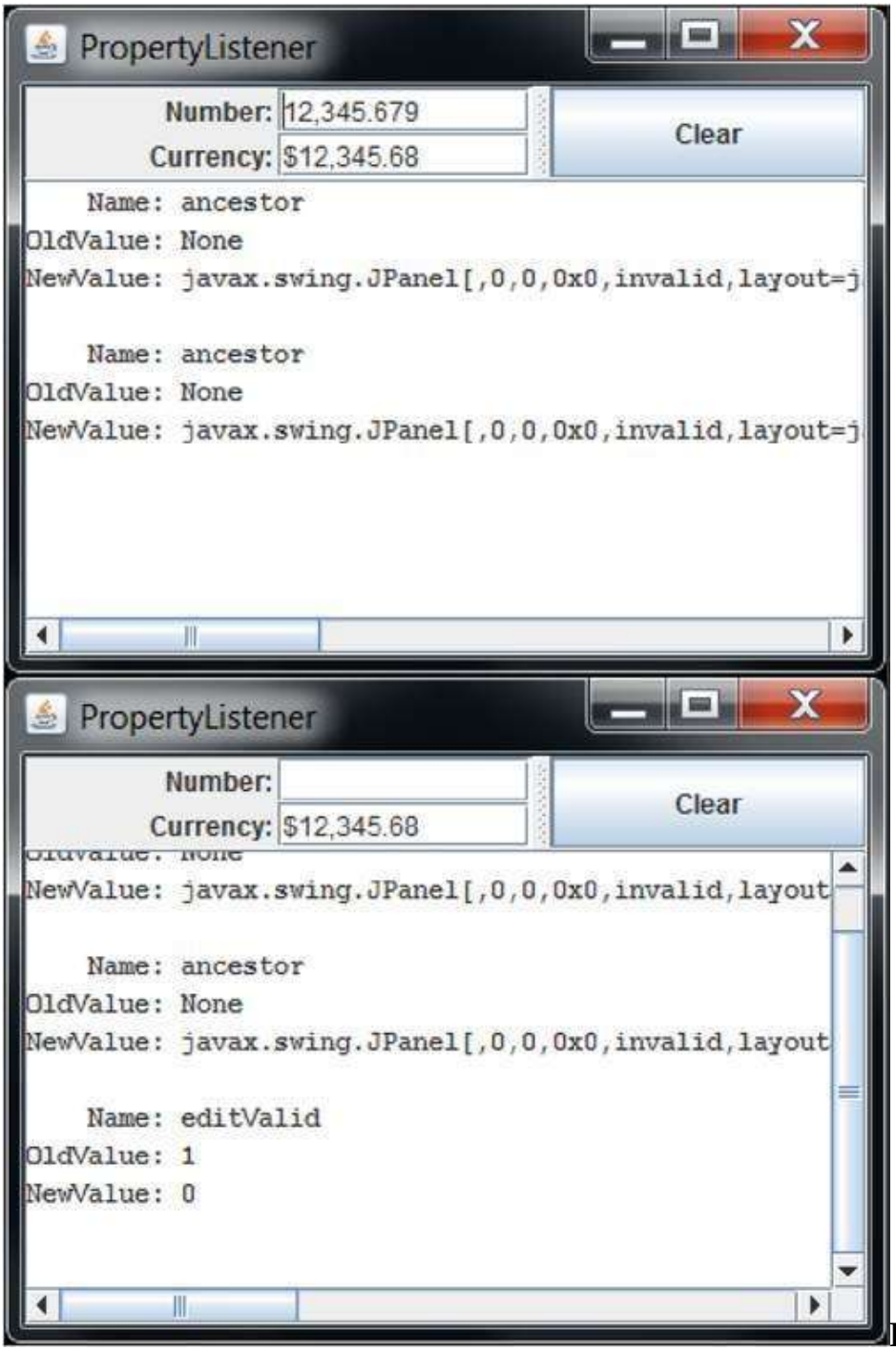


Figure 14-8. *PropertyListener.py sample output*

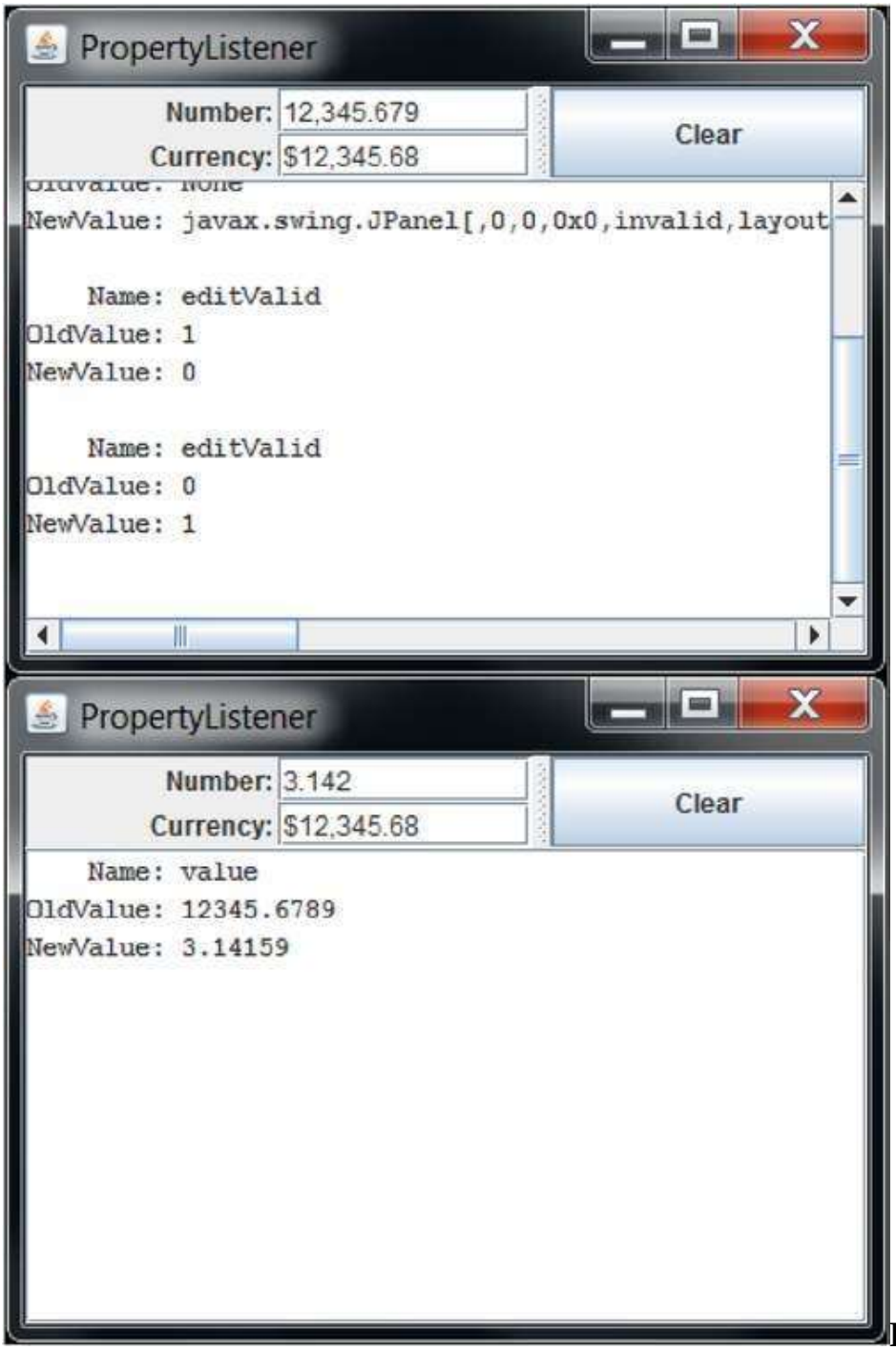


Figure 14-8. (*continued*)

The images in Figure 14-8 demonstrate various events based on the user input. Arguably, the most important is when the user enters a valid value for the formatted text field. This property change event will have a property name of value, as well as an OldValue attribute that can be used to determine the previous

valid value, and a NewValue attribute that can be used to identify the newly entered value. This is shown in the last image in Figure 14-8.

Using an InputVerifier

As mentioned earlier, the InputVerifier class exists to allow your applications to check for valid input before field focus is enabled. Let's take a look at a trivial application that uses an InputVerifier to force the user to enter the value pass before the input focus can change. Figure 14-9 shows the output of this simple application.

Figure 14-9. *VerifyTest1.py sample output*

Listing 14-15 shows how the InputVerifier only needs to implement the verify(...) method. As long as this verify(...) method returns a value of false (0), the input focus isn't allowed to change.

Listing 14-15. Trivial Input Verification from VerifyTest1.py

```
9|class inputChecker( InputVerifier ) :
10| def verify( self, input ) :
11| return input.getText() == "pass"
12|class VerifierTest1( java.lang.Runnable ) :
13| def run( self ) :
14| frame = JFrame(
15| 'VerifierTest1',
16| locationRelativeTo = None,
17| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
18| )
19| frame.add(
20| JTextField(
21| 'Enter "pass"',
22| inputVerifier = inputChecker()
23| ),
24| BorderLayout.NORTH
25| )
```

```
26| frame.add(
27| JTextField( 'TextField 2' ),
28| BorderLayout.SOUTH
29| )
30| frame.pack()
31| frame.setVisible( 1 )
```

The problem with this trivial implementation is that the user doesn't get much feedback that the current value is unacceptable. Wouldn't it be better if you used some of the available techniques to explain to the user that something is wrong?

Figure 14-10 shows one possible approach. Notice what happens when the user simply tries to change focus by pressing the Tab key. The border color around the input field changes to RED and the field text is selected. I don't know about you, but I find this to be a pretty strong indication that something is wrong here.

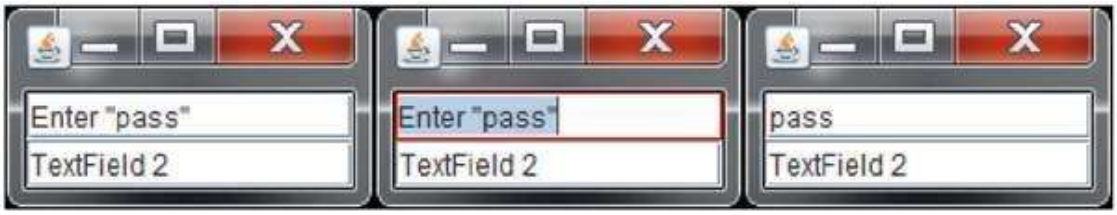


Figure 14-10. *VerifyTest2.py sample output*

Listing 14-16 shows this modified InputVerifier descendant class. In it, the verify(...) method does a little more than the previous version. This additional functionality requires fewer than a dozen lines of code. However, there is a world of difference between the usability of the two applications, don't you think?

Listing 14-16. A Slightly More Involved Input Verifier from VerifyTest2.py

```
11|class inputChecker( InputVerifier ) :
12| def __init__( self ) :
13| self.border = None # holder for "original" border
14| def verify( self, input ) :
15| text = input.getText()
16| if not self.border : # The first time, save the border
17| self.border = input.getBorder()
18| result = input.getText() == "pass"
```

```
19| if result : # valid? Restore original border
20| input.setBorder( self.border )
21| else : # invalid? change border color
22| input.setBorder( LineBorder( Color.red ) )
23| input.selectAll()
24| return result
```

Is there anything missing from this application? Consider what happens when an invalid value is entered. Nothing, that's what. The InputVerifier isn't invoked because it is associated with events that attempt to change the keyboard focus. Pressing the Enter key doesn't do that. This event is associated with ActionEvents, not with InputVerifier events.

Figure 14-11 shows what happens when you associate an ActionListener with the first input field that invokes the InputVerifier to validate the user input. The second image shows that, when the user presses the Enter key without changing the text, the InputVerifier indicates that the text is invalid. The final image shows what happens when the user enters valid text and presses Enter. Note that focus remains with the input field because the change of focus is not a normal result of this action.

Figure 14-11. *VerifyTest3.py sample output*

Listing 14-17 shows how simple it can be to have the ActionListener event handler routine invoke the InputVerifier shouldYieldFocus() method to initiate input verification.

Listing 14-17. Using an ActionListener Method to Verify Input from VerifyTest3.py

```
25|class VerifierTest3( java.lang.Runnable ) :
26| def verify( self, e ) :
27| self.verifier.shouldYieldFocus( e.getSource() )
28| def run( self ) :
| ...
34| self.verifier = inputChecker()
```

```
35| frame.add(
36| JTextField(
37| 'Enter "pass"',
38| actionPerformed = self.verify,
39| inputVerifier = self.verifier
40| ),
41| BorderLayout.NORTH
42| )
| ...
```

Summary

This chapter covered some of events that your Swing applications will encounter. It is important to be familiar with these events so that your applications can react well to user input. Remember, though, that there are many, many events in the Swing class hierarchy. This chapter provides exposure to some of them.

In Chapter 15, you'll investigate how to use open source software libraries to enhance your applications. **Chapter 15**

Nuts to Soup: Using Jsoup to Enhance Applications

Up to this point, the applications in this book have been pretty simple. In this chapter, you learn what it will take to create an application "as you go." By that, I mean you can start simple and iterate over its creation and development, with the intent of making it more useful based on what you learn as you go along. Additionally, you will use an existing open source software (OSS) Java class library to create a useful application built on the work of others.

Using Existing Classes: Creating an HTML Retrieval Application from Scratch

Based on the number of times that I've referred to the Java Swing documentation pages (the Javadoc for the Swing class hierarchy), along with the number of footnote references that point to various class pages, it would be useful if you had an application to help you with this kind of lookup, don't you think? So, what will such an application need to do?

Wouldn’t it be great if this application were able to access and process the Javadoc pages directly? What would that take to achieve? This application needs to be able to access a remote website, request a specific page, and process the results.

Are there any classes in the Swing hierarchy that you might be able to use? Listing 15-1 shows the list of classes on the Java “All Classes” page[1] that include HTML somewhere in their class name.[2]

Listing 15-1. Java “HTML” Classes

```
BasicHTML
HTML
HTML.Attribute
HTML.Tag
HTML.UnknownTag
HTMLDocument
HTMLDocument.Iterator
HTMLEditorKit
HTMLEditorKit.HTMLFactory HTMLEditorKit.HTMLTextAction
HTMLEditorKit.InsertHTMLTextAction HTMLEditorKit.LinkController
HTMLEditorKit.Parser
HTMLEditorKit.ParserCallback HTMLFrameHyperlinkEvent
HTMLWriter
MinimalHTMLWriter
```

[1] See http://docs.oracle.com/javase/8/docs/api/allclasses-noframe.html.

[2]This list of classes can be obtained by executing the following command: jython getLinks.py http://docs.oracle.com/ javase/8/docs/api/allclasses-noframe.html | grep HTML. Then edit the result to remove the URLs that are listed with each class. The getLinks.py script is discussed later in this chapter.

This would appear to be a reasonable place to start an investigation. How could you use some of these classes in order to:

- Connect to a remote website using a uniform resource locator (URL)
- Retrieve the hypertext markup language (HTML) from that page
- Process the contents, with the purpose of locating the hyperlinks[3] and the associated text

After a bit of investigation and work, I was able to get the routine shown in Listing 15-2 working. There may be some way to refactor the code to make it slightly shorter, but I don't see any easy way to make it easier to read and understand.

Listing 15-2. The getLinks Routine

```
8|def getLinks( page ) :
9| url = URI( page ).toURL()
10| conn = url.openConnection()

11| isr = InputStreamReader( conn.getInputStream() )
12| br = BufferedReader( isr )
13| kit = HTMLEditorKit()
14| doc = kit.createDefaultDocument()
15| parser = ParserDelegator()
16| callback = doc.getReader( 0 )
17| parser.parse( br, callback, 1 )
18| iterator = doc.getIterator( HTML.Tag.A )
19| while iterator.isValid() :
20| try :
21| attr = iterator.getAttributes()
22| src = attr.getAttribute( HTML.Attribute.HREF )
23| start = iterator.getStartOffset()
24| fini = iterator.getEndOffset()
25| length = fini - start
26| text = doc.getText( start, length )
27| print '%40s -> %s' % ( text, src )
28| except :
29| pass
30| iterator.next()
```

What does the output of the script containing this routine look like? Well, Listing 15-3 shows the (slightly massaged) output of the script, which allows the output to fit in the available space. This figure shows the hyperlinks found on the IBM website (http://www.ibm.com).[4]

[3]See http://en.wikipedia.org/wiki/Hyperlink.

Listing 15-3. Absolute Links from http://www.ibm.com

```
United States -> http://www.ibm.com/planetwide/select/selector.html IBM?? ->
```

http://www.ibm.com/us/en/
Site map -> http://www.ibm.com/sitemap/us/en/
Figure 15-1 shows output from the Merriam Webster page. It contains a number of relative hyperlinks instead of the absolute links shown in Listing 15-3.

```
                -> /
    "Cartouche" -> /dictionary/cartouche
   "Praemunire" -> /dictionary/praemunire
   "Positivity" -> /dictionary/positivity
Quizzes & Games -> /game/index.htm
Word of the Day -> /word-of-the-day/
          Video -> /video/index.php
      New Words -> http://nws.merriam-webster.com/opendictionary/
   My Favorites -> /my-saved-words/manage-list.htm
```

Figure 15-1. *Relative links from http://www.merriam-webster.com/*

Dealing with relative versus absolute hyperlinks is just the kind of thing that your application should take into consideration. Additionally, dealing with HTML includes the risk that it may not be well formed (i.e., syntactically correct).

Based on the hope that others are likely to have encountered and already solved these kinds of problems, I decided to search the Internet for possible solutions. An initial search for “Jython HTML parsing” results in some interesting possibilities. Unfortunately, subsequent investigation identifies some prohibiting factors, including the fact that some Python modules (such as urllib2, HTMLParser, BeautifulSoup, and lxml) don’t work well or at all with Jython environments or the possible solutions have significant performance issues.

Wouldn’t It Be Nice: Using Java Libraries

If the Jython or Python modules can’t help , what’s left? Don’t forget that one of the most significant advantages to Jython is that it is running on a Java Virtual Machine. So, you can investigate the possibility of using one or more Java libraries to help.

[4]This output is the result of executing wsadmin -f %SwJCode%\Chap_15\getLinks.py -conntype none with the SwJCode environment variable containing the path to the code folder that contains the book’s scripts.

When I asked on the Jython user's mailing list[5] for suggestions, the *Jsoup*[6] *library* was quickly recommended. Jsoup is a Java library for working with "real-world" HTML (such as pages with syntax issues). The Jsoup page includes the following quote:

■**Note** Jsoup is a Java library for working with real-world htMl. it provides a very convenient api for extracting and manipulating data, using the best of DoM, Css, and jQuery-like methods.
Jsoup implements the WhatWg htMl5[7] specification and parses htMl to the same DoM as modern browsers do. This sounds exactly like the kind of thing you need in this case, don't you think?

Working with the Jsoup Library

Follow these steps to use a new library with wsadmin and Jython:
1. First, you need to download the Jsoup archive library:

• Point your favorite browser to http://jsoup.org/.
• Select the Download link.
• Right-click on the core library file[8] and save it in a convenient directory.

2. For wsadmin, you need to add this Java archive to the classpath. Fortunately, a wsadmin command-line option exists to do exactly this. If the Jsoup JAR file is in the C:\temp directory, this can be as simple as:[9]

wsadmin -wsadmin_classpath c:\temp\jsoup-1.8.1.jar -f scriptName.py
For Jython, you only need to have this library (JAR file) as part of the JYTHONPATH environment variable. For example:
Set JYTHONPATH=C:\Programs\jsoup\jsoup-1.8.1.jar;%JYTHONPATH%
jython scriptName.py

[5] See jython-users@lists.sourceforge.net.
[6]See http://jsoup.org/.
[7]See http://whatwg.org/html.
[8]At the time of this writing, it was jsoup-1.8.1.jar.
[9]The first time that Jython is told about this Java archive, you will see a message something like this to indicate that Jython is aware of the archive and has processed it to store information about the JAR file for later use by Jython

scripts.

sys-package-mgr: processing new jar, 'C:\temp\jsoup-1.8.1.jar'.

Listing 15-4 shows a simple, interactive wsadmin session that demonstrates how easy it is for scripts to use the Jsoup library.
Listing 15-4. Simple Jsoup Demonstration

```
1|wsadmin>from org.jsoup import Jsoup
2|wsadmin>
3|wsadmin>doc = Jsoup.connect( 'http://www.ibm.com' ).get()
4|wsadmin>doc.title()
5|'IBM - United States'
6|wsadmin>for link in doc.getElementsByTag( 'a' ) :
7|wsadmin> print link.attr( 'abs:href' )
8|wsadmin>
9|http://www.ibm.com/planetwide/select/selector.html
10|http://www.ibm.com/us/en/
11|http://www.ibm.com/sitemap/us/en/
12|http://www.ibm.com/software/marketing-solutions/benchmark-hub/...
13|http://www.ibm.com/systems/infrastructure/us/en/it-infrastruct...
14|http://www-935.ibm.com/services/us/en/it-services/business-con...
15|http://www.ibm.com/common/twitter/ibm.xml
16|http://www.ibm.com/news/us/en
17|...
```

Table 15-1 contains a description of each line of the simple interactive session.

Table 15-1. *Simple Jsoup Demonstration, Explained* **Lines Description**
1 Import statement used to add the Jsoup library to the current namespace. 2 An empty line to make the session input easier to read.

3 A statement used to retrieve the contents of the www.ibm.com web page.
Note: It is possible for an exception to be raised.
4 Demonstration of using the title() method[10] to display the title of the retrieved web page.
6-8 For loop used to retrieve the HTML links and display the absolute URLs[11] to which they refer.
9. . . The remainder of the session shows the first few lines that are output by the

previous loop.

Isn’t this easier to read, understand, and use than the ones from the Swing hierarchy? I think so, and I’m pretty sure that you’ll agree.
[10]See http://jsoup.org/apidocs/org/jsoup/nodes/Document.html#title(). [11]See http://jsoup.org/cookbook/extracting-data/working-with-urls.

Jsoup Call May Appear to Hang

So now you’re all set, right? No, not quite. Remember that the Swing applications need to be able to deal with events. More importantly, you don’t want the application to wait or appear to be “hung,” while waiting for things (such as like the retrieval of web pages) to complete. As mentioned before, you’re going to start simple and make decisions based on how things look.

First you need a descendant of the SwingWorker class to perform the Jsoup retrieval and to process requests on a separate thread. Listing 15-5 shows the soupTask class from this simple application.
Listing 15-5. soupTask Class from javadocInfo_01.py

```
23|class soupTask( SwingWorker ) :
24| def __init__( self, comboBox, label, url ) :
25| self.cb = comboBox # Save provided references
26| self.msg = label
27| self.url = url # URL to be used
28| self.doc = None
29| SwingWorker.__init__( self )
30| def doInBackground( self ) :
31| try :
32| self.msg.setText( 'working...' )
33| self.doc = Jsoup.connect( self.url ).get()
34| self.msg.setText( 'ready' )
35| except :
36| Type, value = sys.exc_info()[ :2 ]
37| print 'Error:', str( Type )
38| print 'value:', str( value )
39| self.msg.setText( str( value ) )
40| if self.doc :
41| self.cb.removeAllItems()
```

```
42| for link in self.doc.getElementsByTag( 'a' ) :
43| self.cb.addItem( str( link.text() ) )
44| def done( self ) :
45| pass
```

Figure 15-2 shows some sample images from the first Swing application using the Jsoup library.

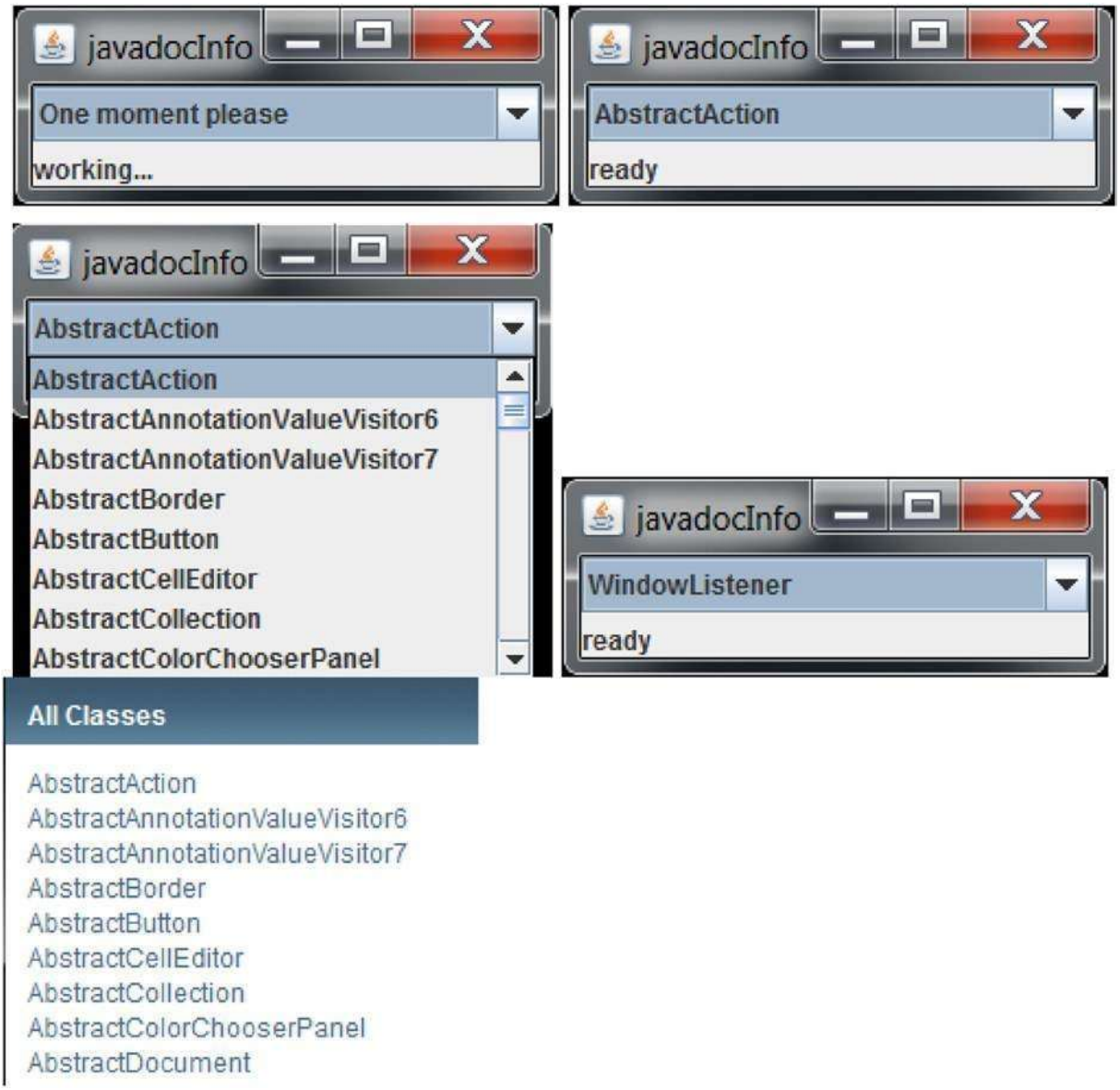


Figure 15-2. *javadocInfo_01.py sample output*

The last image in Figure 15-2 shows that a browser appears when the same URL is used. It certainly looks like the Jsoup application is working as expected.

From a Combo Box to a List Box

That application is quite simple and demonstrates the feasibility of using the Jsoup library to process the current Java documentation URL.[12] Unfortunately, a combo box doesn't appear to be a good choice for showing the list of available entries, does it? What else is missing? Well, I would like to be able to see the URL of the selected entry, wouldn't you? Let's take a look at how to do that.

Figure 15-3 shows some images from this iteration of the application. The first image shows the application as a connection to the remote host is being established. The second image, which is very short lived, shows that the processing of the retrieved information is in progress. Then you see an image of the list of items from the page, and finally, you see that when you make a selection, the text field is updated to reflect the associated URL for the selection.

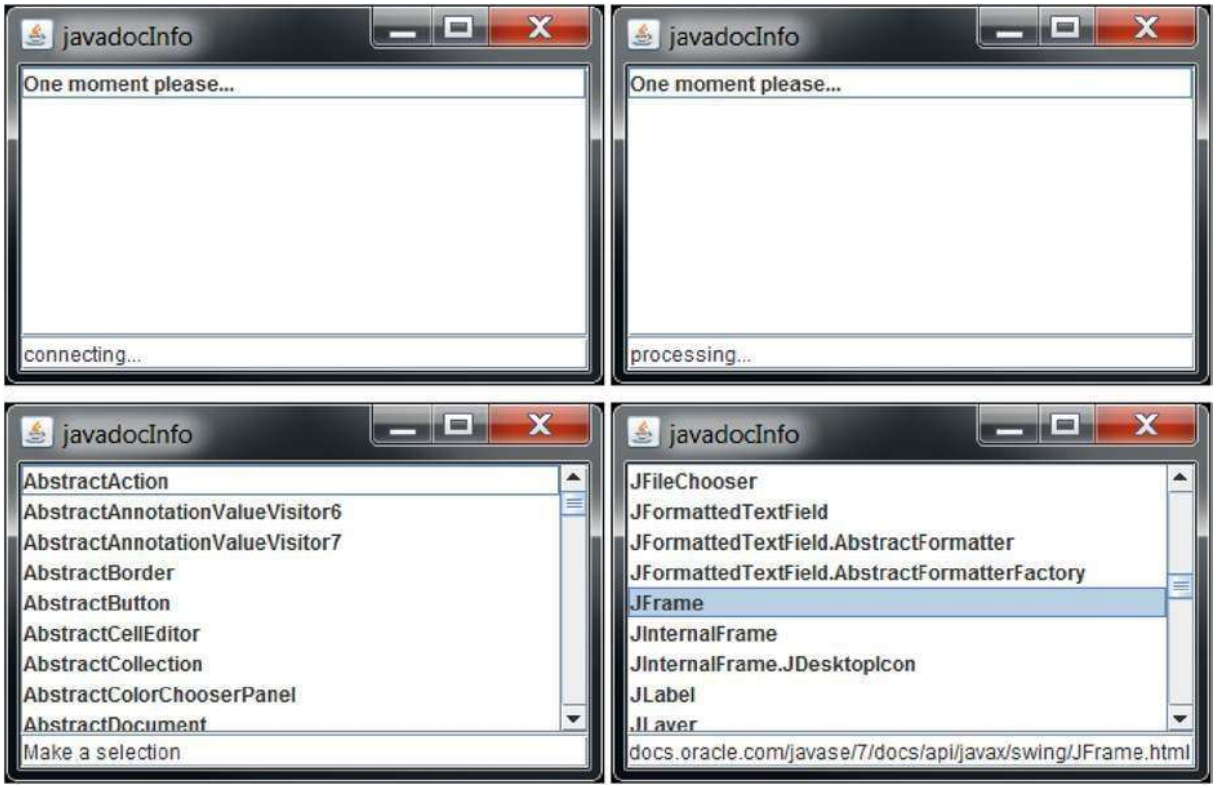


Figure 15-3. *javadocInfo_02.py sample output*

To make this happen, you have to change the JComboBox into a JList, which allows you to display more entries. This means that you need to change the soupTask class to deal with these changes. Additionally, a new argument is added to this constructor, so that the task can return a dictionary of the links found on the page, indexed by the text associated with each link. Listing 15-6 shows the most significant changes that were made to the application.

[12]See http://docs.oracle.com/javase/8/docs/api/allclasses-noframe.html.

Listing 15-6. soupTask, Iteration 2, from javadocInfo_02.py

```
26|class soupTask( SwingWorker ) :
```

```
27| def __init__( self, List, label, url, result ) :
28| self.List = List # Save provided references
29| self.msg = label
30| self.url = url # URL to be used
31| self.doc = None
32| self.docLinks = result # lookup result
33| SwingWorker.__init__( self )
34| def doInBackground( self ) :
35| try :
36| self.msg.setText( 'connecting...' )
37| self.doc = Jsoup.connect( self.url ).get()
38| self.msg.setText( 'processing...' )
39| model = DefaultListModel()
40| for link in self.doc.getElementsByTag( 'a' ) :
41| name = link.text()
42| href = link.attr( 'abs:href' )
43| self.docLinks[ name ] = href
44| model.addElement( name )
45| self.List.setModel( model )
46| self.msg.setText( 'Make a selection' )
47| except :
48| Type, value = sys.exc_info()[ :2 ]
49| Type, value = str( Type ), str( value )
50| print 'Error:', Type
51| print 'value:', value
52| self.msg.setText( value )
53| def done( self ) :
54| pass
```

That’s pretty good, but it’s still not a great application. What else can you do to make it more useful and usable? Listing 15-7 shows the main application class of the second iteration of the Javadoc lookup application. Overall, it’s amazing how much this little application can do with fewer than 100 lines of Jython code.

Listing 15-7. javadocInfo_02 Application Class

```
55|class javadocInfo_02( java.lang.Runnable ) :
56| def run( self ) :
57| frame = JFrame(
58| 'javadocInfo_02',
```

```
59| locationRelativeTo = None,
60| size = ( 350, 225 ),
61| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
62| )
63| model = DefaultListModel()
64| model.addElement( 'One moment please...' )
65| self.List = JList(
66| model,
67| valueChanged = self.pick,
68| selectionMode = ListSelectionModel.SINGLE_SELECTION
69| )
70| frame.add(
71| JScrollPane( self.List ),
72| BorderLayout.CENTER
73| )
74| self.msg = JTextField()
75| frame.add( self.msg, BorderLayout.SOUTH )
76| self.Links = {}
77| soupTask(
78| self.List, # The visible JList instance 79| self.msg, # The message area
(JTextField) 80| JAVADOC_URL, # Remote web page URL to be processed 81|
self.Links # Dictionary of links found 82| ).execute()
83| frame.setVisible( 1 )
```

Listing 15-8 shows the ListSelectionListener[13] event handler that's invoked when the user makes a selection. It takes a bit of work (code) to figure out when this handler actually has some work to do. Why is that? One reason is that it is invoked when a ListSelectionEvent[14] occurs. So, the first thing it does is determine if the event that occurred is related to a selection adjustment, which is ignored. Then it checks the number of entries on the list. If only one element is present, which only occurs when the list is created with the "One moment please. . ." message, this event is ignored.

Finally, it checks to see if an element is selected, in which case the URL associated with this item is displayed in the message text field. Otherwise, the "Make a selection" message is displayed.

Listing 15-8. javadocInfo_02.py ListSelectionListener Event Handler

```
84| def pick( self, e ) :
```

```
85| if not e.getValueIsAdjusting() :
86| List = self.List
87| model = List.getModel()
88| if model.getSize() > 1 :
89| index = List.getSelectedIndex()
90| if index > -1 :
91| choice = model.elementAt( index )
92| self.msg.setText( self.Links[ choice ] )
93| else :
94| self.msg.setText( 'Make a selection' )
```

Adding a TextArea to Show the HTML

From here, it's easy to imagine how useful it would be to retrieve the contents of the page that is selected by the user. You really don't need to create another web browser.[15] It is potentially useful if the application can retrieve, process, and analyze a user-selectable page. Let's begin by displaying the page's contents (the HTML), and then decide how best to proceed. Listing 15-9 contains the modified textTask class, which is where the most interesting changes occur.

[13] See http://docs.oracle.com/javase/8/docs/api/javax/swing/event/ListSelectionListener.
[14]See http://docs.oracle.com/javase/8/docs/api/javax/swing/event/ListSelectionEvent.ht
[15]In fact, if you try to display a complete HTML page using one of the HTML-aware components, you are likely to encounter a large number of Java exceptions.

Listing 15-9. textTask Class from javadocInfo_03.py

```
60|class textTask( SwingWorker ) :
61| def __init__( self, area, url ) :
62| self.area = area # Save area to be updated
63| self.url = url # URL to be retrieved
64| SwingWorker.__init__( self )
65| def doInBackground( self ) :
66| try :
67| self.area.setText( 'connecting...' )
```

```
68| doc = Jsoup.connect( self.url ).get()
69| self.area.setText( str( doc.normalise() ) )
70| except :
71| Type, value = sys.exc_info()[ :2 ]
72| Type, value = str( Type ), str( value )
73| self.area.setText(
74| '\nError: %s\nValue: %s' % ( Type, value )
75| )
76| def done( self ) :
77| pass
```

Figure 15-4 shows some sample output from the javadocInfo_03.py script. The first image shows the HTML text from the JFrame Javadoc web page. It should be no surprise that the TextArea containing the HTML needs to be in a scroll pane. The second image shows the portion of the HTML that describes the Field Summary portion of the page. The last image shows how this HTML is displayed in a browser.

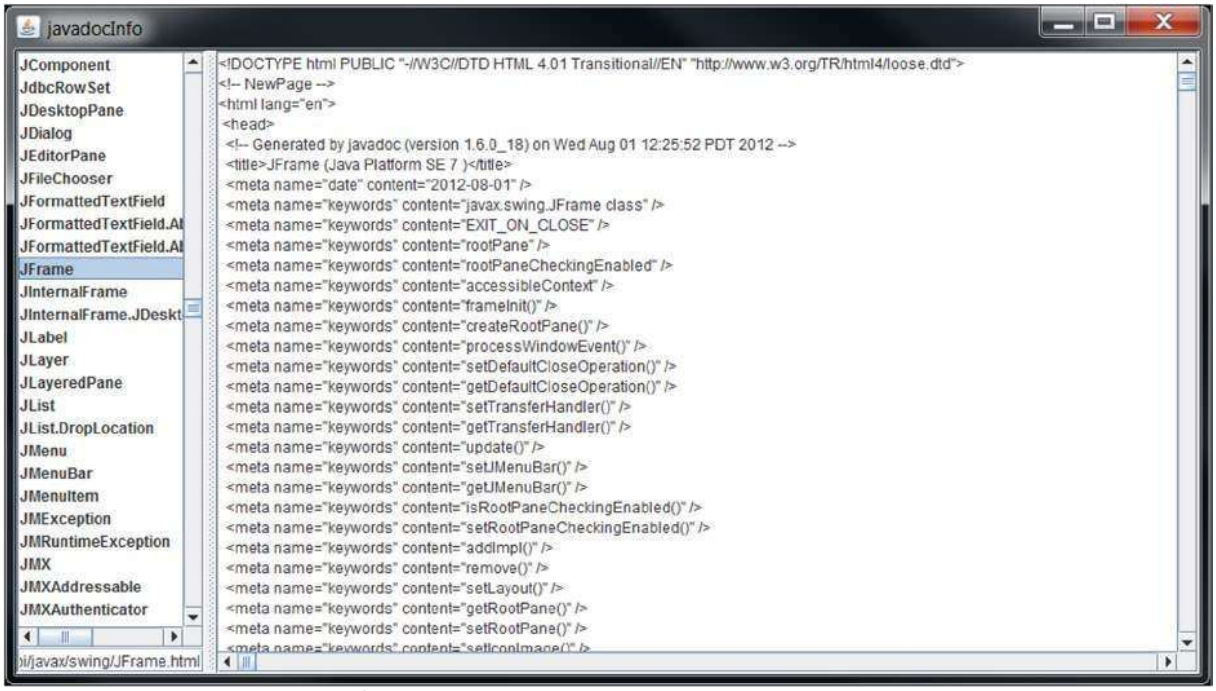
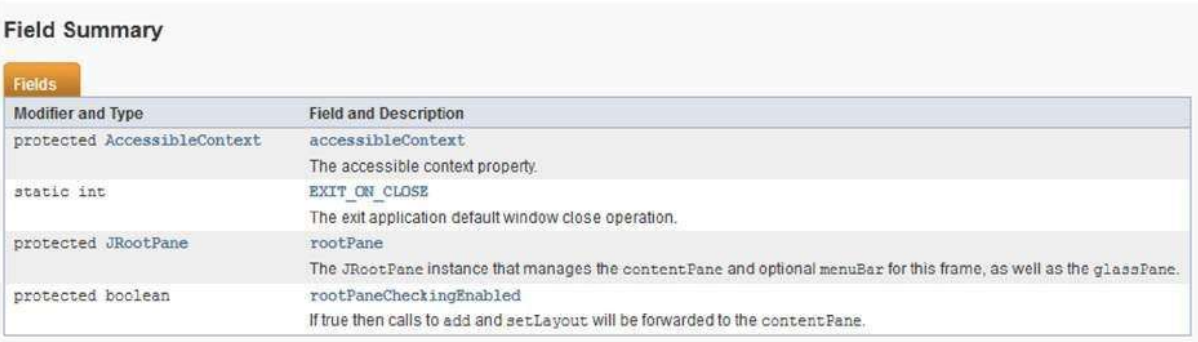


Figure 15-4. *javadocInfo_03 sample output*



Field Summary

Fields

Modifier and Type	Field and Description
protected AccessibleContext	accessibleContext The accessible context property.
static int	EXIT_ON_CLOSE The exit application default window close operation.
protected JRootPane	rootPane The JRootPane instance that manages the contentPane and optional menuBar for this frame, as well as the glassPane.
protected boolean	rootPaneCheckingEnabled If true then calls to add and setLayout will be forwarded to the contentPane.

Figure 15-4. (*continued*)

Rendering HTML

It certainly would be nice if you could actually use HTML tags to tell how the Swing component should display information. Well, you can, at least to a limited extent. Some, but not all, Swing components are capable of using HTML tags to determine how information should be displayed. You saw from the most recent application output in Figure 15-4 that the JTextArea component doesn’t do this.

Don’t get your hopes up too quickly. You won’t be able to create a fully capable web browser in your applications simply by selecting the “right” Swing components.
Another caveat you need to be aware of is the fact that the HTML specification

implemented in the Swing component hierarchy is 3.2, so it’s not as feature-rich as the latest HTML specification (version 5), which is starting to show up on the Internet.
What does it take to have one of the HTML capable components display text as the HTML tags indicate? Well, it’s as simple as having the text attribute string begin with <html>.
Listing 15-10 contains the trivial script that demonstrates just how easy it is to use HTML to do this. Please be careful though—just because you can do something, doesn’t mean that you should.

Listing 15-10. HTMLtext_01.py Script Showing an HTML Label

```
1|from javax.swing import JFrame
2|from javax.swing import JLabel
3|frame = JFrame(
4| 'HTMLtext_01',
5| size = ( 200, 200 ),
6| locationRelativeTo = None,
7| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
8| )
9|text = '<html>'
10|text += '<sup>My</sup> '
11|text += '<sub><i>Label</i></sub> '
12|text += '<font color="#FF0000">is</font> '
13|text += '<font color="#00FF00"><b>far</b></font> ' 14|text += '<font
color="#0000FF">too busy,</font> ' 15|text += "<u>isn't it?</u>"
16|label = frame.add( JLabel( text ) )
17|frame.setVisible( 1 )
18|raw_input()
```

Figure 15-5 demonstrates this axiom quite vividly. Keep in mind that you should be interested in making your application more appealing, not less.

Figure 15-5. *HTMLtext_01.py sample output*

If you are familiar with HTML, you know that you can do all sorts of interesting things with it. For example, as you can see in Figure 15-6, you can create a simple unordered list tag (starts on line 10 and ends on line 14 of Listing 15-?) to display a bulleted list with each item in a different color.

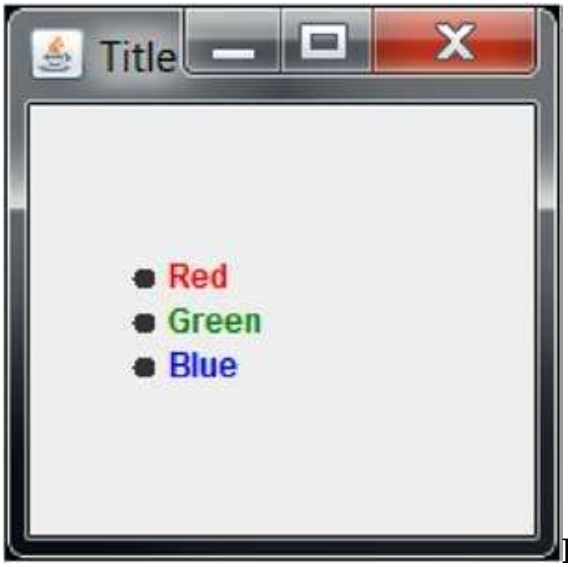


Figure 15-6. *HTMLtext_02.py sample output*

Listing 15-11 shows the trivial script used to produce the output in Figure 15-6. As you can see, it's up to the developer how to build the string. In this case, using multiple concatenation steps to build the HTML string makes for a better

and more complete understanding of the various parts of the HTML.

Listing 15-11. HTMLtext_02.py Source

```
1|from javax.swing import JFrame
2|from javax.swing import JLabel
3|frame = JFrame(
4| 'HTMLtext_02',
5| size = ( 200, 200 ),
6| locationRelativeTo = None,
7| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
8| )
9|text = '<html>'
10|text += '<ul compact>'
11|text += '<li><font color="red">Red</font></li>'
12|text += '<li><font color="green">Green</font></li>'
13|text += '<li><font color="blue">Blue</font></li>'
14|text += '</ul>'
15|label = frame.add( JLabel( text ) )
16|frame.setVisible( 1 )
17|raw_input()
```

Modifying the HTML Text

In addition to static components, your applications might have some dynamic components that might be improved by the users of the HTML. Do you remember the JToggleButton component discussed in Chapter 8? Figure 15-7 shows one way to use HTML to make the button state more obvious to the users.

Figure 15-7. *HTMLtext_03.py sample output*

Listing 15-12 shows part of the HTMLtext_03.py script that was used to produce

this output. If you're uncomfortable with the statement in lines 24-29, don't worry. It uses Jython's flexibility to determine which color and text values should be used by indexing an array using the result of calling the button.isSelected() method.

Listing 15-12. Using HTML in a JToggleButton from HTMLtext_03.py

```
6|class HTMLtext_03( java.lang.Runnable ) :
7| def run( self ) :
8| frame = JFrame(
9| 'HTMLtext_03',
10| size = ( 100, 100 ),
11| locationRelativeTo = None,
12| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
13| )
14| text = '<html><font color="red">Off</font>'
15| label = frame.add(
16| JToggleButton(
17| text,
18| itemStateChanged = self.toggle
19| )
20| )
21| frame.setVisible( 1 )
22| def toggle( self, event ) :
23| button = event.getItem()
24| button.setText(
25| '<html><font color="%s">%s</font>' % [
26| ( 'red' , 'Off' ),
27| ( 'green', 'On' )
28| ][ button.isSelected() ]
29| )
```

Listing 15-13 shows some additional ways that this might be done.[16] I'm quite certain that you can come up with some other alternatives yourself.

Listing 15-13. Alternative Ways of Setting the JToggleButton Text if button.isSelected() :
text = '<html>Off' else :
text = '<html>On' button.setText(text)
... or even ...

```
if button.isSelected() :
color, value = 'red', 'Off'
else :
color, value = 'green', 'On'
text = '<html><font color="%s">%s</font>' % ( color, value )
button.setText( text )
```

Identifying the Sections

Now that you've learned about the components that you can use to display text that's been marked up in HTML, it's time to return to Jsoup to see what you can do to improve your Javadoc application.

Note in Figure 15-4 that the HTML for a Swing class can be quite extensive. The application that I'm considering, at least for this chapter, can display a subset of the information that's available on the user selected entry, not the complete details.

Let's work on this in steps, so that you can learn as you go along. To begin, let's see what it takes to build a tiny application that will display the header text for a specific Javadoc page. The getHeaders class that is used in all three of the getHeader scripts is shown in Listing 15-14. As you can see, it is a simple application that uses a SwingWorker descendent class to populate a scrollable text area with the HTML heading tags from the JFrame Javadoc URL.

Listing 15-14. The getHeaders Class[17]

```
62|class getHeaders1( java.lang.Runnable ) :
63| def run( self ) :
64| frame = JFrame(
65| 'JFrame headers',
66| size = ( 500, 250 ),
67| locationRelativeTo = None,
68| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
69| )
70| textArea = JTextArea()
71| frame.add( JScrollPane( textArea ) ) 72| url =
'.../docs/api/javax/swing/JFrame.html' 73| headerTask( url, textArea ).execute()
74| frame.setVisible( 1 )
```

[16] This is a kind of a tip of the hat to Perl's "Tim Toady" (TMTOWTDI) motto at http://en.wikipedia.org/wiki/There%27s_ more_than_one_way_to_do_it and the "Zen of Python," which can be seen at http://www.python.org/dev/peps/pep-0020/.
[17]The complete URL of the JFrame Javadoc is http://docs.oracle.com/javase/8/docs/api/javax/swing/JFrame.html.

Listing 15-15 shows the headerTask that uses the Jsoup library to retrieve the specified HTML, process it, and display the result in the user-specified area. If you watch closely while executing any of these getHeader scripts, you might be able to see the connecting... and processing... messages that this class produces.

It is important to note that this class uses the Jsoup items to process the retrieved HTML. The ones new to this application are found in the getPlainText(...) method, as shown on lines 38-42.
Listing 15-15. The headerTask Class from the getHeader1.py Script Files

```
38|class headerTask( SwingWorker ) :
39| def __init__( self, url, result ) :
40| self.url = url # URL to be retrieved
41| self.result = result
42| SwingWorker.__init__( self )
43| def getPlainText( self, element ) :
44| visitor = FormattingVisitor( self.url )
45| walker = NodeTraversor( visitor )
46| walker.traverse( element )
47| return visitor.toString()
48| def doInBackground( self ) :
49| try :
50| self.result.setText( 'connecting...' )
51| doc = Jsoup.connect( self.url ).get()
52| self.result.setText( 'processing...' )
53| self.text = self.getPlainText( doc )
54| except :
55| Type, value = sys.exc_info()[ :2 ]
56| Type, value = str( Type ), str( value )
57| print '\nError:', Type
58| print 'value:', value
59| self.result.setText( 'Exception: %s' % value )
```

```
60| def done( self ) :
61| self.result.setText( self.text )
```

The FormattingVisitor(...) method is a descendant of the Jsoup NodeVisitor class, and its implementation is shown in Listing 15-16. This is the only class in the three getHeader#.py scripts that changes. In this class, the application appends the text of any HTML header tags that it finds to be displayed in the application text area. **Listing 15-16.** The FormattingVisitor() Method from getHeader1.py

```
23|class FormattingVisitor( NodeVisitor ) :
24| def __init__( self, url ) :
25| self.result = StringBuilder()
26| def append( self, text ) :
27| newline = [ '', '\n' ][ self.result.length() > 0 ]
28| self.result.append( newline + text )
29| def head( self, node, depth ) :
30| name = node.nodeName()
31| if name in [ 'h1', 'h2', 'h3', 'h4', 'h5', 'h6' ] :
32| if node.hasText() :
33| self.append( '%s: %s' % ( name, node.text() ) )
34| def tail( self, node, depth ) :
35| name = node.nodeName()
36| def toString( self ) :
37| return str( self.result )
```

The only method that changes in the three getHeader scripts is head(...). The one from getHeader2.py is shown in Listing 15-17. Note how this iteration is only interested in the H3 (Header 3) tags.
Listing 15-17. The head(...) Method from getHeader2.py

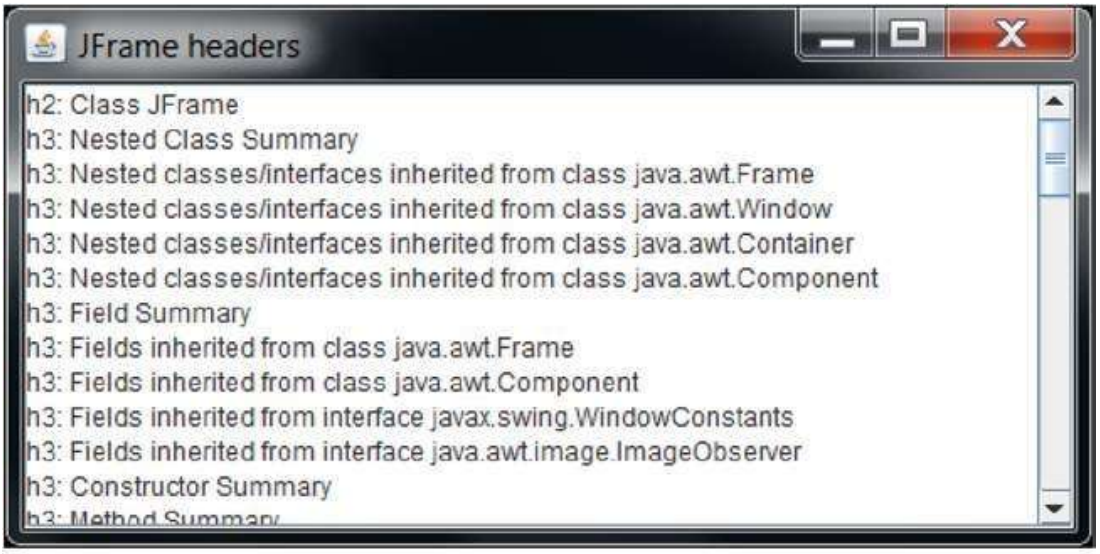
```
29| def head( self, node, depth ) :
30| name = node.nodeName()
31| if name == 'h3' :
32| if node.hasText() :
33| self.append( '%s: %s' % ( name, node.text() ) )
```

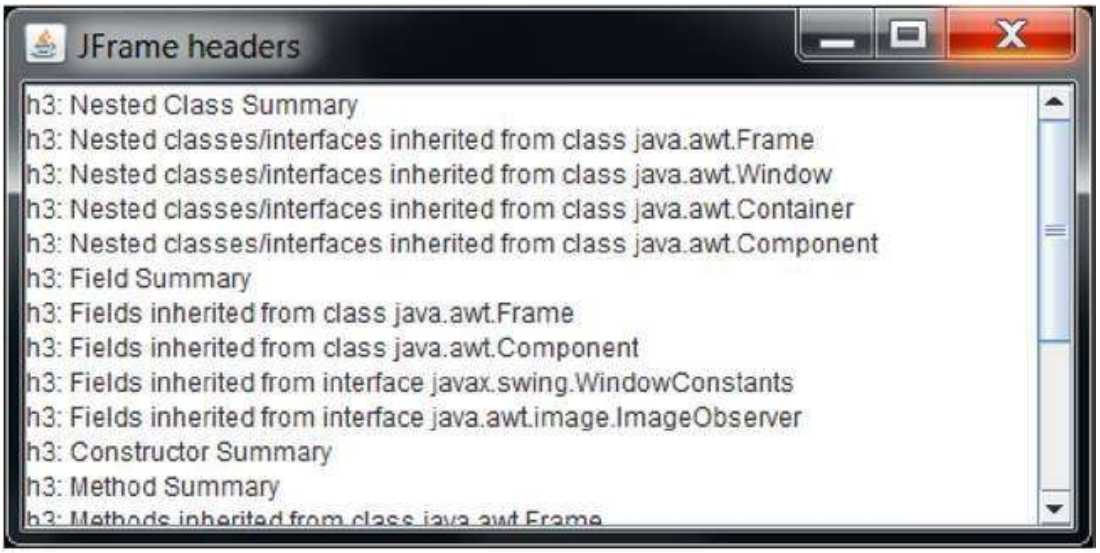
Listing 15-18 shows the last version of the head(...) method. This version is only interested in H3 tags whose text doesn’t contain the word “inherited.”

Listing 15-18. The head(...) Method from getHeader3.py

```
29| def head( self, node, depth ) :
30| name = node.nodeName()
31| if name == 'h3' and node.hasText() :
32| text = node.text()
33| if text.find( 'inherited' ) < 0 :
34| self.append( '%s: %s' % ( name, text ) )
```

Figure 15-8 shows the output of these three getHeader scripts. From this output, it shouldn’t be too much of a leap to figure out how to change the Javadoc script to display the most useful H3 text. Before you do that, though, take a moment to figure out where you want to go next.


JFrame headers
h2: Class JFrame
h3: Nested Class Summary
h3: Nested classes/interfaces inherited from class java.awt.Frame
h3: Nested classes/interfaces inherited from class java.awt.Window
h3: Nested classes/interfaces inherited from class java.awt.Container
h3: Nested classes/interfaces inherited from class java.awt.Component
h3: Field Summary
h3: Fields inherited from class java.awt.Frame
h3: Fields inherited from class java.awt.Component
h3: Fields inherited from interface javax.swing.WindowConstants
h3: Fields inherited from interface java.awt.image.ImageObserver
h3: Constructor Summary


JFrame headers
h3: Nested Class Summary
h3: Nested classes/interfaces inherited from class java.awt.Frame
h3: Nested classes/interfaces inherited from class java.awt.Window
h3: Nested classes/interfaces inherited from class java.awt.Container
h3: Nested classes/interfaces inherited from class java.awt.Component
h3: Field Summary
h3: Fields inherited from class java.awt.Frame
h3: Fields inherited from class java.awt.Component
h3: Fields inherited from interface javax.swing.WindowConstants
h3: Fields inherited from interface java.awt.image.ImageObserver
h3: Constructor Summary
h3: Method Summary


JFrame headers
h3: Nested Class Summary
h3: Field Summary
h3: Constructor Summary
h3: Method Summary
h3: Field Detail
h3: Constructor Detail
h3: Method Detail

Figure 15-8. *Sample output from the getHeader scripts*

Fixing the Great Divide

One of the problems that I have with the way the javadocInfo_03.py application looks is the fact that the divider doesn't provide enough space to see the longer class names. How can you fix this?

To begin with, it's a good idea to adjust the position to something more convenient. If you do that, though, you need to have the application tell you the new position, so that you can use it as the starting position for the revised application.

You need to add a PropertyChangeListener event handler to the JSplitPane and watch for the right kind of changes. This is all very easy to say, but how easy is it to do? Listing 15-19 shows just how easy this can be. In line 109, you see how to specify the PropertyChangeListener event handler method, and in lines 137-146, you see how the event handler determines and displays the divider-related changes.

Listing 15-19. PropertyChangeListener Changes in javadocInfo_04.py

```
| ...
104| frame.add(
105| JSplitPane(
106| JSplitPane.HORIZONTAL_SPLIT,
107| pane,
108| JScrollPane( self.area ),
109| propertyChange = self.propChange
110| )
111| )
| ...
137| def propChange( self, pce ) :
138| src = pce.getSource() # Should be our JSplitPane
139| prop = pce.getPropertyName()
140| if prop == JSplitPane.LAST_DIVIDER_LOCATION_PROPERTY :
141| curr = src.getDividerLocation()
142| last = pce.getNewValue()
143| prev = pce.getOldValue()
```

```
144| print '\nlast: %4d prev: %4d curr: %4d' % (
145| last, prev, curr
146| ),
```

Figure 15-9 shows the sample output from an execution of the script that determined a more appropriate divider value.

```
last:   -1  prev:    0  curr:  129
last:  129  prev:   -1  curr:  129
last:   -1  prev:  129  curr:  129
last:  129  prev:   -1  curr:  271
last:  271  prev:  129  curr:  271
last:  129  prev:  271  curr:  271
```

Figure 15-9. *SplitPane PropertyChangeListener output*

So what can you do with this newfound knowledge? Well, my initial attempt consisted of adding a dividerLocation keyword argument to the JSplitPane[18] constructor call. Unfortunately, this caused an exception indicating that this is a read-only value.

Looking at the Javadoc, you can see that there are multiple setDividerLocation(...) methods, which is likely to be the source of the problem. So you need to call one of these setter methods after JSplitPane has been instantiated. Listing 15-20 shows how this is done in the javadocInfo_05.py script.

Listing 15-20. Using a DividerLocation Setter Method Call

```
106| sPane = JSplitPane(
107| JSplitPane.HORIZONTAL_SPLIT,
108| pane,
109| JScrollPane( self.area ),
110| propertyChange = self.propChange
111| )
112| sPane.setDividerLocation( 234 )
```

Filtering the List

One of the problems with the complete list of classes that's initially displayed by the javadocInfo script is that it's too long. If you drag the scroll bar down, you'll quickly see that there are something like 4,000 classes listed. This makes it harder for the users to deal with.

Wouldn't it be nice to use the JTextField as an input field filter that allowed users to enter text to be matched? That's what you'll do next. The changes that need to be made include the following:

• Locate the places in the script where the setText(...) method is used to show the URL associated with the user selection.

• Add a listener to JTextField that allows you to monitor user input and locate class names containing the user-specified text. For this, I chose to use a CaretListener and check for changes in the user-specified text.

The javadocInfo_06.py script contains these changes and Figure 15-10 shows some sample screens. [18]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JSplitPane.html.

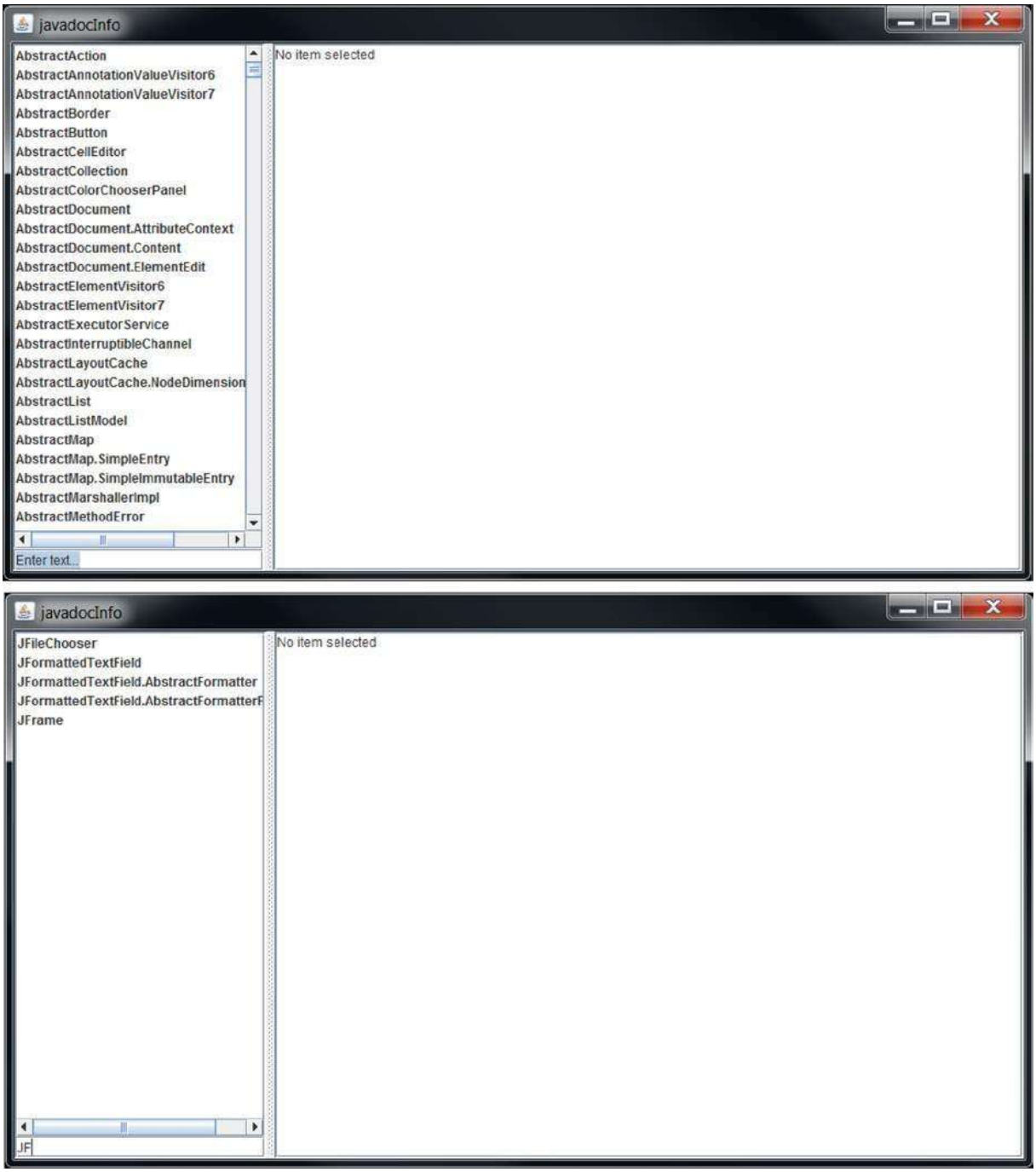


Figure 15-10. *javadocInfo_06.py sample output with list filtering*

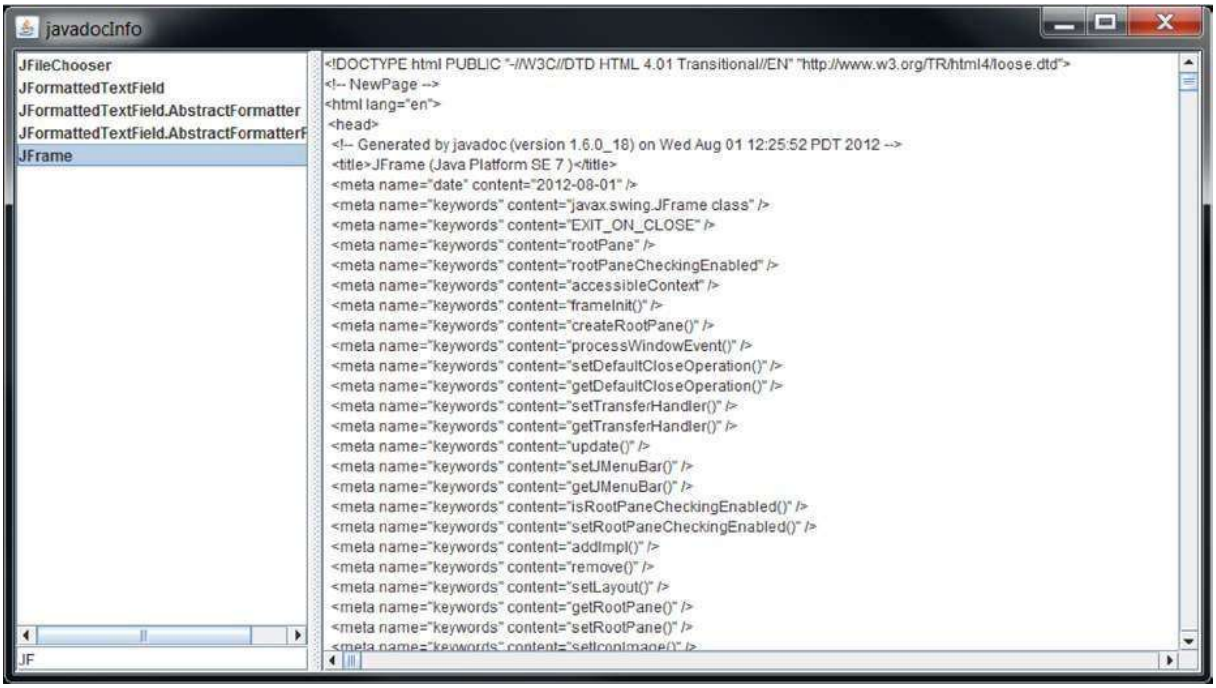
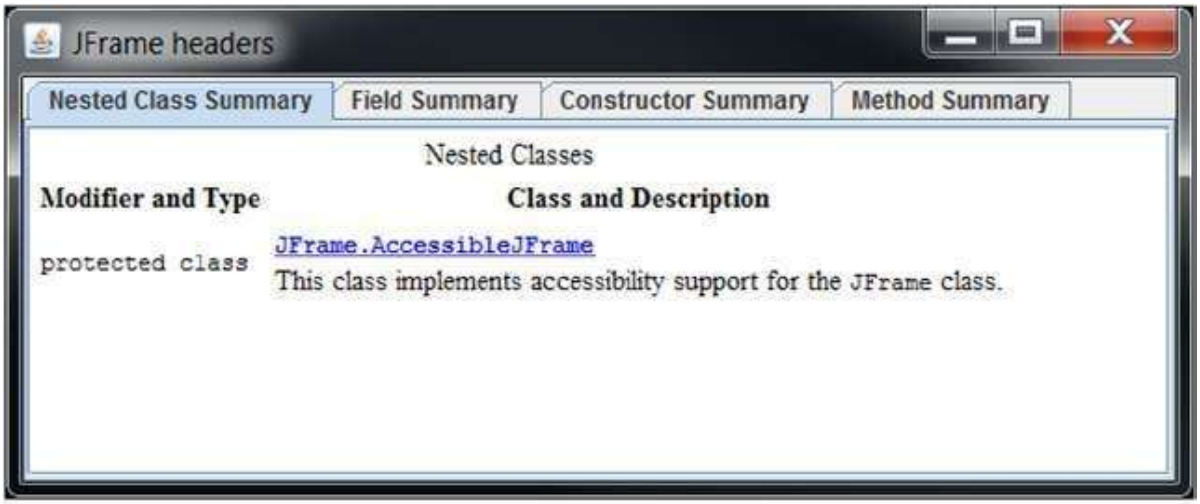


Figure 15-10. (*continued*)

Using Jsoup to Pick Up the Tab: Adding a JTabbedPane

I wonder how difficult it would be for the Jsoup library to process the Javadoc HTML and find the tables that are frequently found in the heading 3 (h3) sections of the documentation? While I was thinking about this, it came to me that I might be able to use a JTabbedPane, with the tab name being the heading 3 text and the contents being the HTML table.

You could even use a suitable pane to render the table properly. This is sounding interesting. To simplify things during my testing (and learning) phases, I chose to start with the getHeaders script.
Figure 15-11 shows the sample output of the getHeaders5.py script, which retrieves and processes the Javadoc for the JFrame class.


JFrame headers
Nested Class Summary
Field Summary
Constructor Summary
Method Summary
Nested Classes
Modifier and Type
Class and Description
protected class
JFrame.AccessibleJFrame
This class implements accessibility support for the JFrame class.


JFrame headers
Nested Class Summary
Field Summary
Constructor Summary
Method Summary
Fields
Modifier and Type
Field and Description
protected
AccessibleContext
accessibleContext
The accessible context property.
static int
EXIT_ON_CLOSE
The exit application default window close operation.
protected JRootPane
rootPane
The JRootPane instance that manages the contentPane and

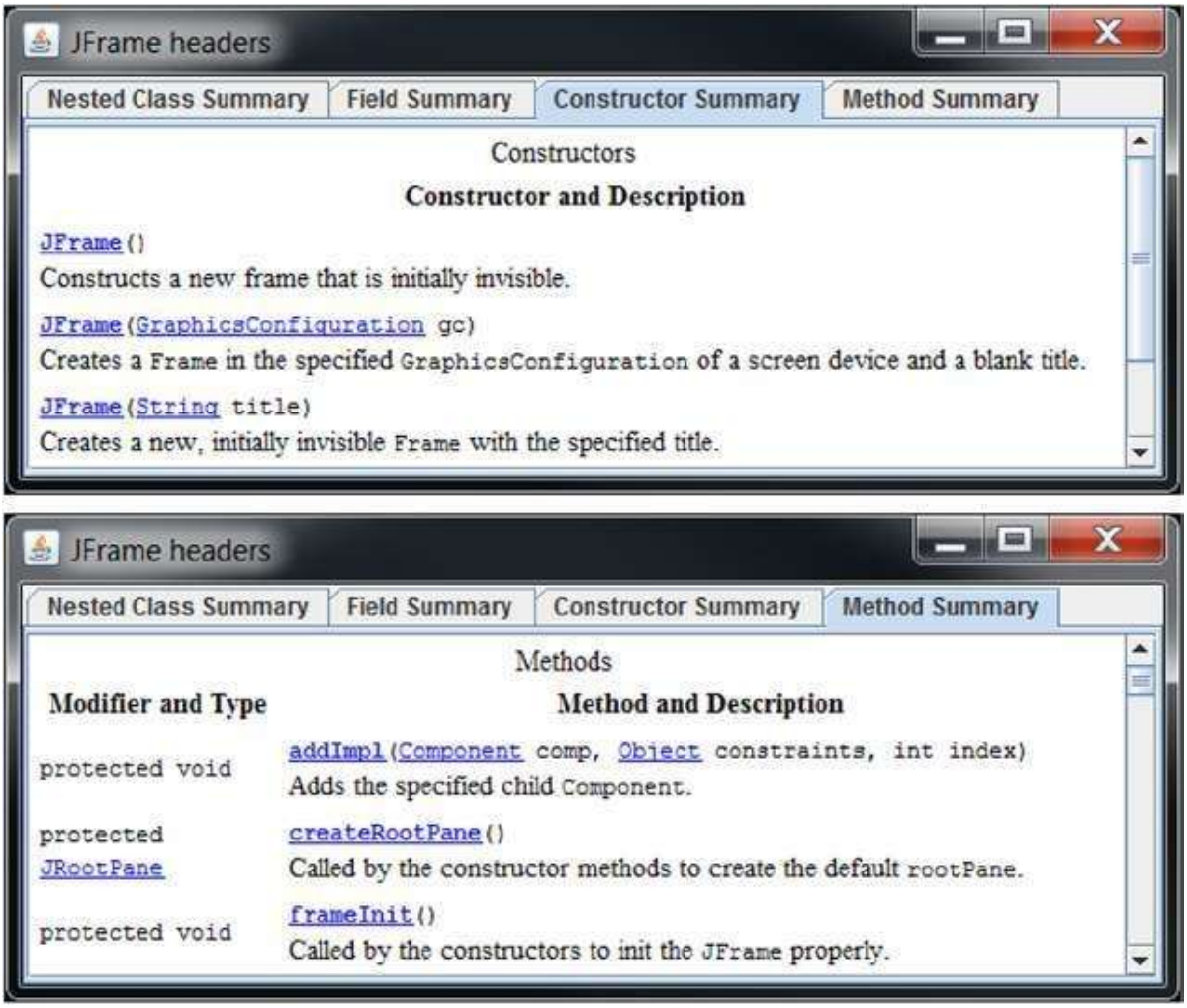


Figure 15-11. *getHeaders5.py output with JFrame h3 tables on JTabbedPane*

The most significant changes are shown in Listing 15-21, which contains the FormattingVisitor class from the getHeaders5.py script.

Listing 15-21. FormattingVisitor Class Used to Build the JTabbedPane

```
25|class FormattingVisitor( NodeVisitor ) :
26| def __init__( self, url ) :
27| self.Tabs = JTabbedPane()
28| self.header = re.compile( '[hH][1-6]$' )
29| self.h3Text = ''
30| def head( self, node, depth ) :
31| name = node.nodeName()
32| if re.match( self.header, name ) :
33| self.h3Text = [ '', node.text() ] [ name[ 1 ] == '3' ]
34| def tail( self, node, depth ) :
```

```
35| name = node.nodeName()
36| if self.h3Text and name == 'table' :
37| ePane = JEditorPane(
38| 'text/html', # mime type
39| '<html>' + str( node ), # content
40| editable = 0
41| )
42| self.Tabs.addTab( self.h3Text, JScrollPane( ePane ) )
43| def toString( self ) :
44| tp = self.Tabs
45| return '\n'.join(
46| [
47| tp.getTitleAt( i )
48| for i in range( tp.getTabCount() )
49| ]
50| )
```

Table 15-2 explains how this class works, in detail.

Table 15-2. *FormattingVisitor Class, Explained* **Lines Description**

26-29 Class constructor used to instantiate the JTabbedPane, as well as define a regular expression (RegExp) to identify heading tags, specifically h3 tags.

30-33 The head(...) method is called by the NodeTraversor() method as the HTML is processed. It is invoked when any open tag is encountered (such as <h2> or <table...>). The value of the h3Text variable is set to an empty string (when any other head level is encountered) or to the text associated with the heading tag when an h3 tag is found.

34-42 The tail(...) method is called as the HTML is processed, specifically when an end tag (such as </h3> or </table>) is found. When a table “end” tag is found after an <h3> tag, a read-only JEditorPane is created containing the whole <table> HTML. This JEditorPane instance is placed in a JScrollPane and added as a tab to the JTabbedPane.

43-50 The toString(...) method is provided and returns a string with all of the JTabbedPane tab names, separated by newline \n characters.

The only other significant difference in this script relates to how the

SwingWorker descendent class (headerTask) updates the application in a way that allows the updates to be shown to the users. Listing 15-22 shows the headerTask class from getHeader5.py.

Listing 15-22. The headerTask Class from the getHeaders5.py Script

```
51|class headerTask( SwingWorker ) :
52| def __init__( self, url, frame ) :
53| self.url = url # URL to be retrieved
54| self.frame = frame
55| SwingWorker.__init__( self )
56| def doInBackground( self ) :
57| try :
58| print 'connecting...'
59| doc = Jsoup.connect( self.url ).get()
60| print 'processing...'
61| visitor = FormattingVisitor( self.url )
62| walker = NodeTraversor( visitor )
63| walker.traverse( doc )
64| self.frame.add( visitor.Tabs )
65| self.frame.validate()
66| except :
67| Type, value = sys.exc_info()[ :2 ]
68| Type, value = str( Type ), str( value )
69| print '\nError:', Type
70| print 'value:', value
71| def done( self ) :
72| print 'done'
```

Table 15-3 describes the headerTask class in detail. The most important part of this is how this class informs the Swing framework that it needs to validate the application (frame) updates. If this isn’t done, the user won’t see any of the changes.

Table 15-3. *The headerTask Class, Explained*

Lines	Description
52-55	Class constructor used to save the specified URL, as well as a reference to the application frame to be modified. This method also has to invoke the

SwingWorker constructor in order to properly initialize the class as a SwingWorker instance.

56-70 The doInBackground() method is where the actual (generally long-running) processing occurs.
59 This is where the HTML from the specified URL is retrieved.
61-63 A NodeVisitor call uses the FormattingVisitor instance to process the Javadoc from the specified URL.

64 The result of the processing is a JTabbedPane instance, which is added to the application's JFrame instance. Unfortunately, this call to the frame.add(...) method is not likely to happen in a way that lets the Swing framework know that a change has occurred.

65 The call to frame.validate() forces the framework to recognize that it needs to process the changes and update the user display.

Note: Lines 58, 60, and 72 write messages to the console to let you know that things are progressing. They certainly aren't required and may be commented out.
If you comment out line 72, you also need to add a pass statement to satisfy language syntax requirements.

The Jsoup library is quite powerful. Even a tiny bit of testing shows some neat capabilities. For example, I find the output shown in Figure 15-11 a bit hard to read. When I looked at the HTML found in the Javadoc pages, I saw that all of the tables have a border="0" attribute that keeps the grid lines between columns and rows from being shown. Do you know how difficult it is to use the Jsoup library to change the table border attributes to "1"?

Take a look at Listing 15-22, specifically at line 54 where the HTML document is retrieved from the remote website. All you need to do is add the statement in Listing 15-23 after the HTML has been retrieved, and before it is processed by the rest of the code.

Listing 15-23. Changing the border Attribute of All the Tables doc.select('table').attr('border', '1')
It's as simple as that. The result of adding this statement can be seen in Figure 15-12.

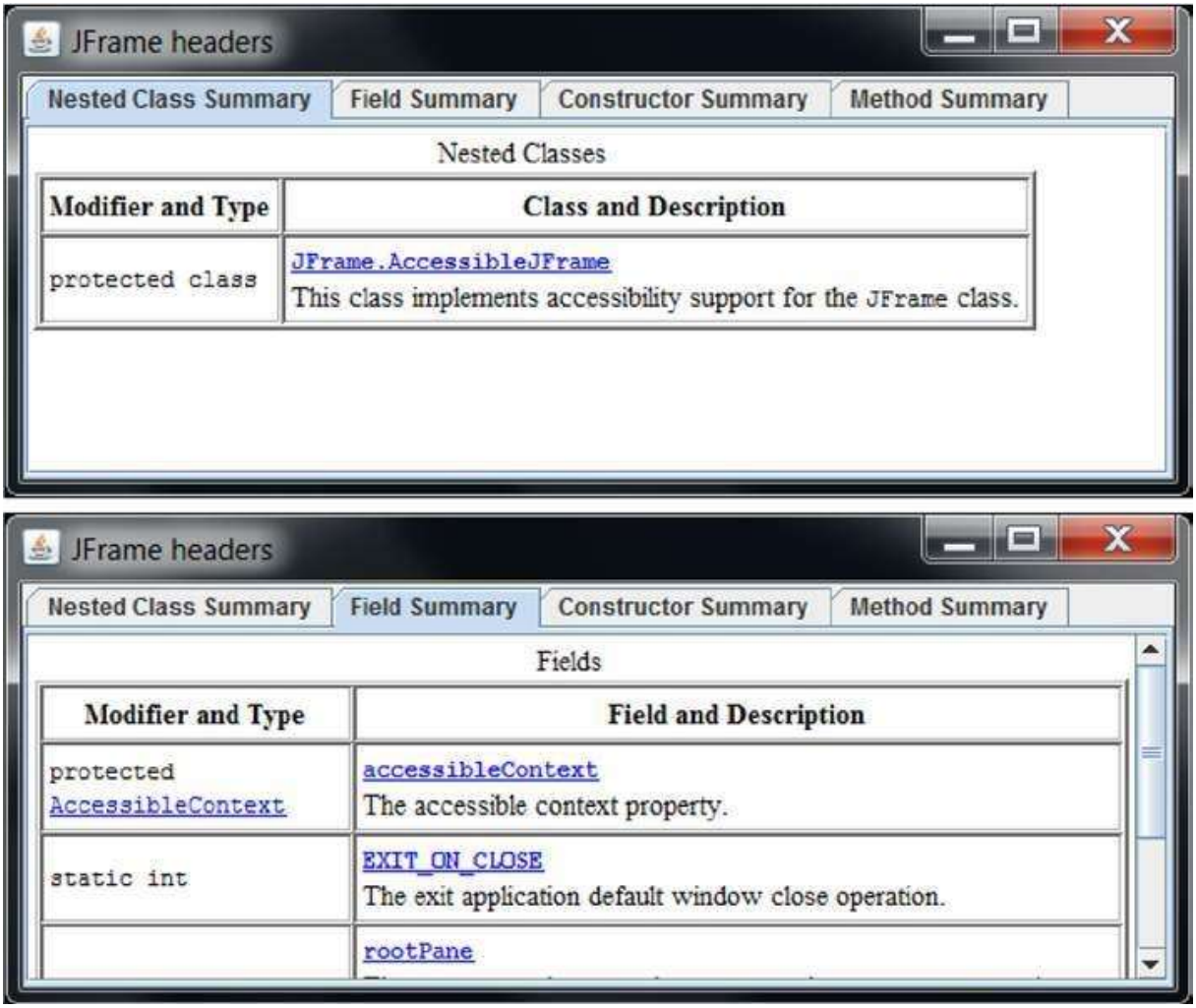


Figure 15-12. *getHeaders6.py output with JFrame h3 tables on JTabbedPane*

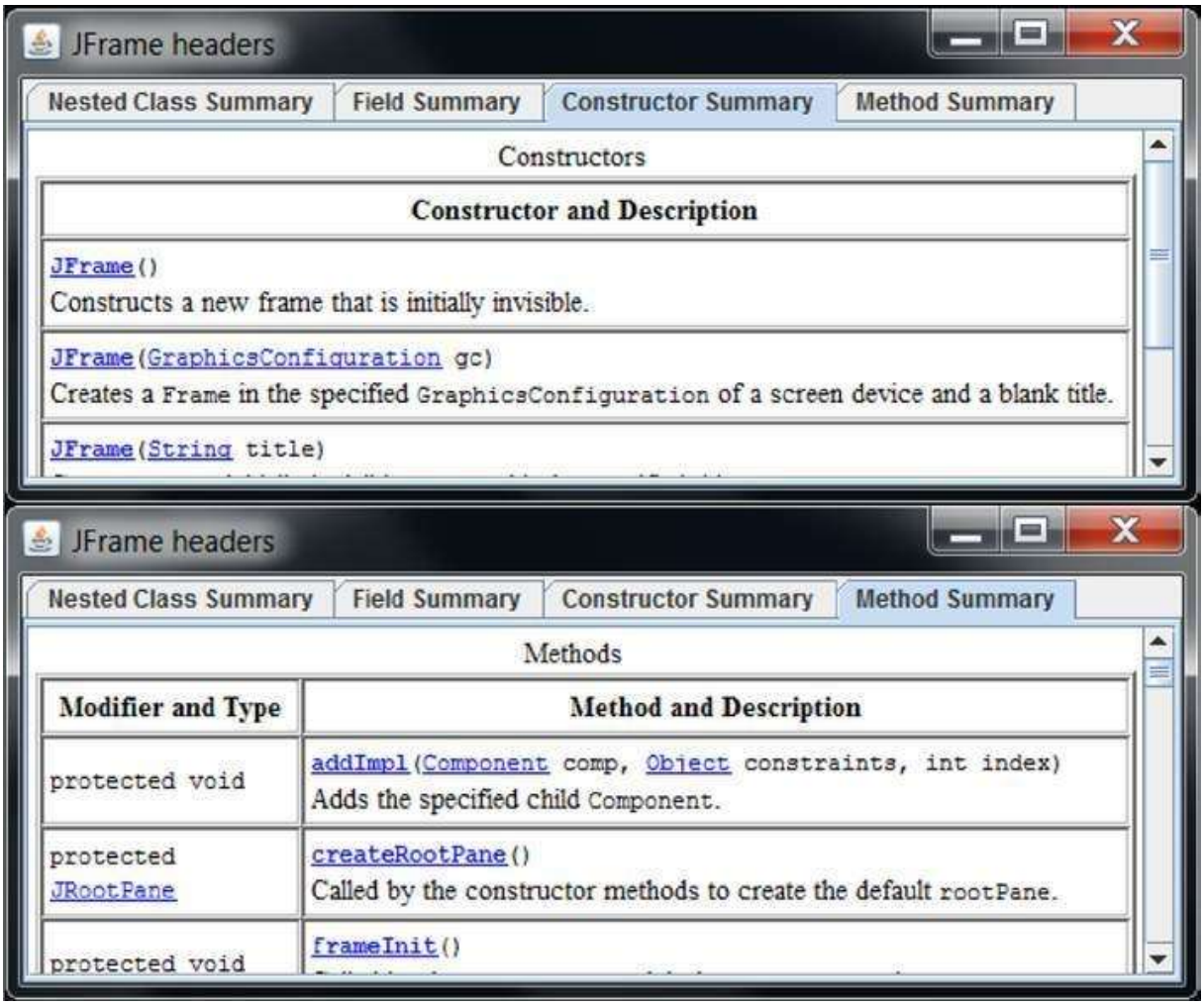


Figure 15-12. (*continued*)

If you think about it, using the statement in Listing 15-23 to find and change all of the table border attributes might be extra work. This would be true if there were other tables in the document that aren't associated with the h3 sections that you want to display. The alternative is to change the attribute only on those tables that you will be adding to the JEditorPane in the tail(...) method of the FormattingVisitor class (shown in Listing 15-21). If you do this, you need to add the statement in Listing 15-24 between lines 31 and 32 of Listing 15-21.

Listing 15-24. Changing the border Attribute of the Current Table
node.attr('border', '1')

Adding Tabbed Editor Panes to the Javadoc Application

What do you need to do in order to add the functionality of the getHeader6.py script to the latest javadocInfo_06.py script?

• The right pane needs to be changed. It isn't a simple JTextArea instance any longer. You have to decide what you want to display when nothing is selected, and you have to figure out how to display the JTabbedPane after a class' HTML has been processed.

• What do you want the application to do as the data is being retrieved and processed? Currently, you can display a status message in the JTextArea. It would seem that you need to dynamically change the right panel of SplitPane, based on what needs to be displayed.

In order for the application to display different kinds of panes in the frame, you need to use a JSplitPane method that wasn't covered in Chapter 5. Since the SplitPane has left and right parts, you need to use the setRightComponent(...) method to dynamically replace that part of the application.

In order for the application to display tabbed JEditorPanes, you'll also need to replace the textTask class (as seen in Listing 15-9) with a modified version of the headerTask class shown in Listing 15-22. This class needs to have the right information so that it can display a status message or the final JTabbedPane result in the application's SplitPane.

What's the result of making these changes? Figure 15-13 shows some sample images from the modified script (javadocInfo_07.py).

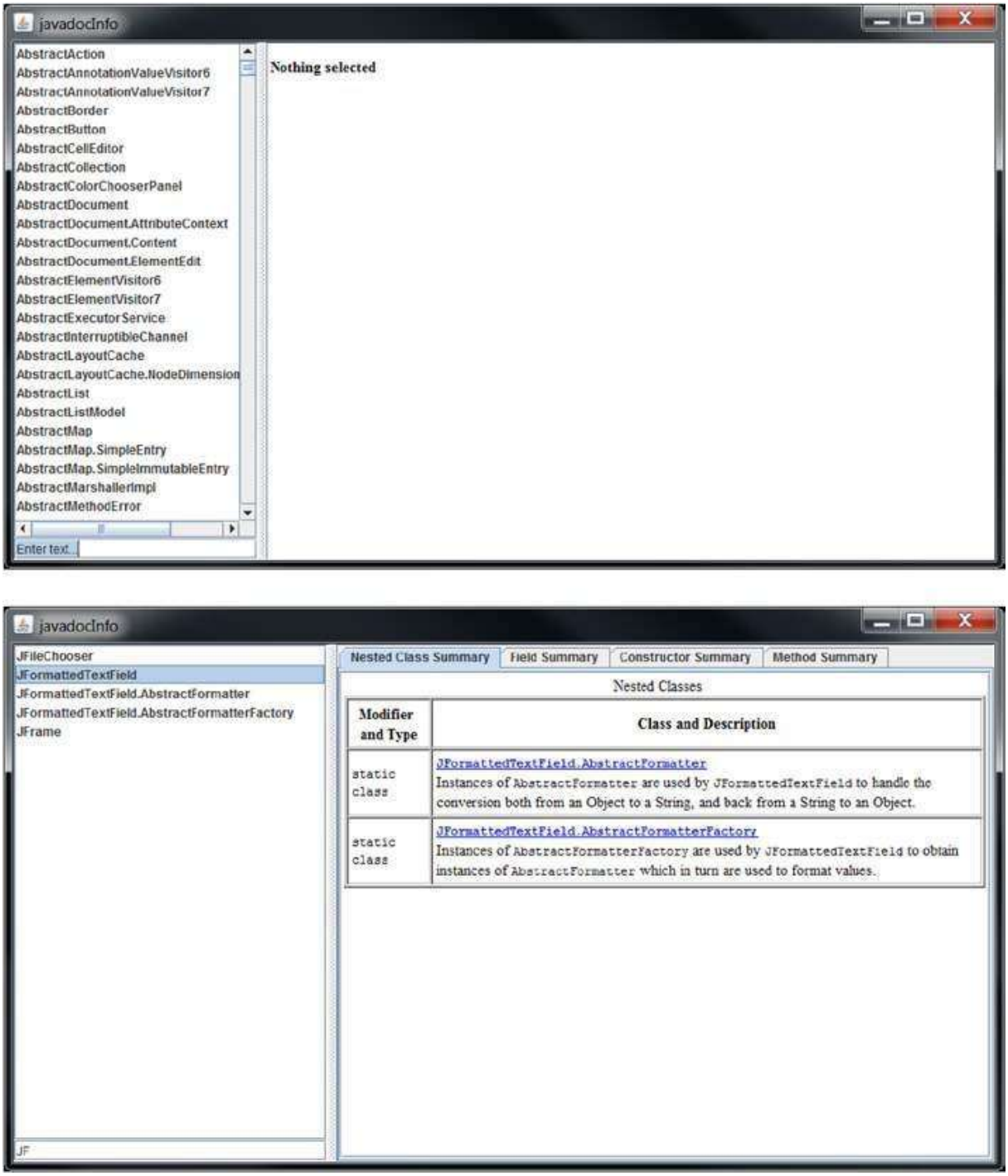


Figure 15-13. *javadocInfo_07.py sample output*

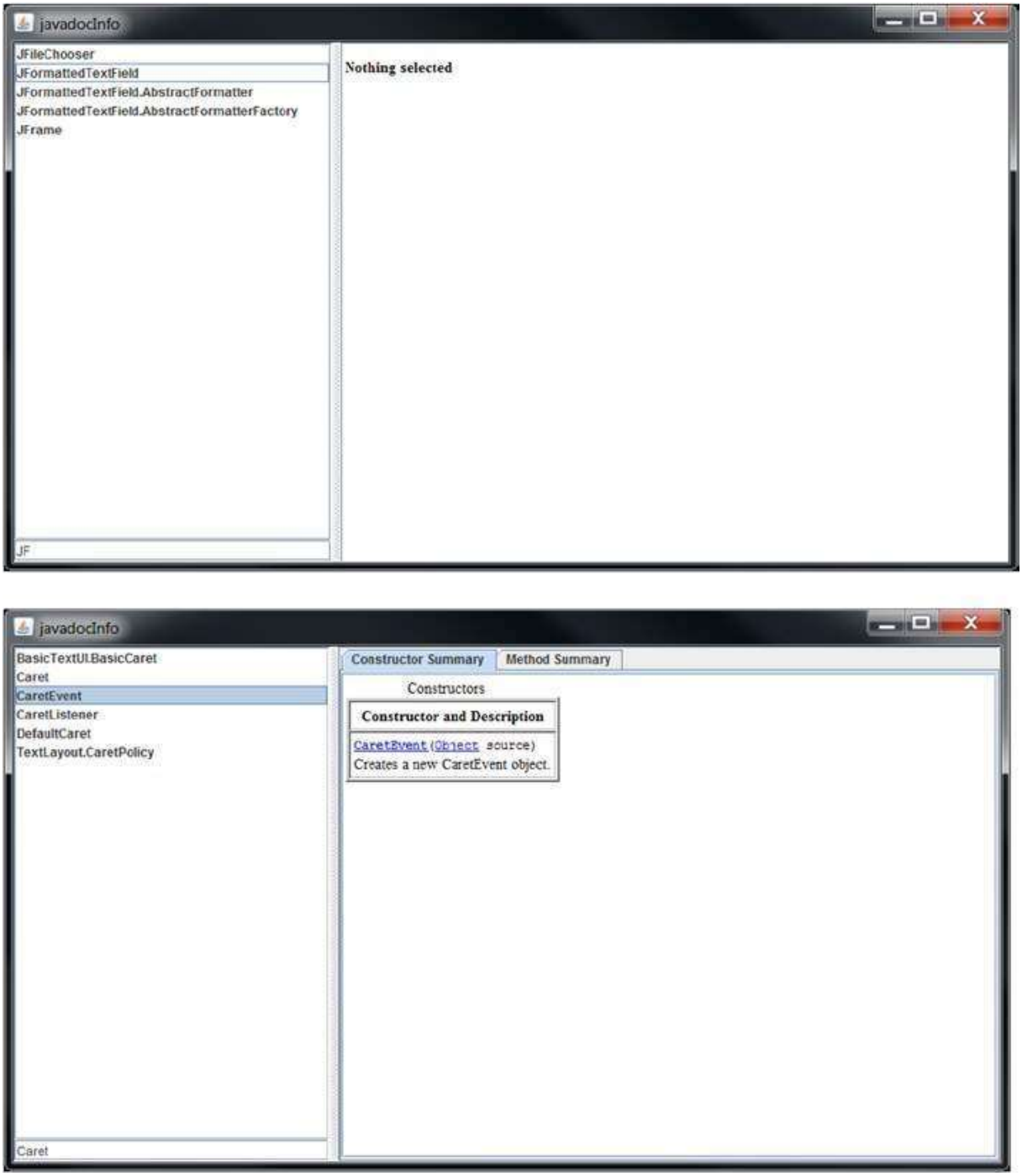


Figure 15-13. (*continued*)

Were there any surprises that happened during this combination process? A few, but not nearly as many as I feared. The biggest surprise was that the first attempt showed that when the JTabbedPane replaced the right component of the SplitPane, the width of the left component used to display the list of classes

narrowed drastically. So, I had to set the minimum size of the JScrollPane containing this list.

What Improvements/Enhancements Remain?

As is frequently the case when using any program, possible enhancements come to mind. You might want to consider adding some menu items. This would make sense if you wanted to allow the users to specify the URL of the page from which it should load the list of Java classes. Another possible menu entry might be used to allow the user to filter class entries (such as AWT and Swing).

You might also want to add a tab to the tabbed pane that shows the class hierarchy produced by the classInfo function mentioned in Chapter 4.
If you are really ambitious, you could add functionality to the Editor Panes so that the links in the displayed HTML open the specified website in a browser.
I haven't delved too deeply into the subject of Editor Panes. There is a decent page in the Java Swing Tutorials entitled "How to Use Editor Panes and Text Panes"[19] that you might want to take a look at.

Summary

One of the useful things about this chapter is that it demonstrates how easy it is to use existing Java classes and libraries. There is an enormous number of class libraries on the Internet. Hopefully, this chapter gives you the courage to use them in your Jython scripts.

The next chapter discusses dialog boxes and how they can be used to interact with users to obtain input for scripts.

[19] See http://docs.oracle.com/javase/tutorial/uiswing/components/editorpane.html.

Chapter 16

Conversing with a User with Dialog Boxes

According to several dictionaries, a "dialogue" is an exchange of information between two entities, at least one of whom is a person. Maybe that's why a window for providing input to an application is called a "dialog box."

Way back in Chapter 1, you read all about the top-level containers. The examples throughout the book have used the JFrame class as the top-level container of choice. Now you're going take some time to learn about the JDialog[1] container class and dialog boxes in general. This chapter also touches on a few related issues. In Chapter 17, you'll continue by learning about some specialized dialog boxes that you can use to make your life as an application developer much easier.

This chapter is all about using dialog boxes to get information from the user into an application. It starts by discussing where dialog boxes fit in the Swing class hierarchy. Then is discusses how you position them, especially when multiple displays exist. Additionally, you'll see how various dialog boxes can be created using JOptionPane methods.

What Are Dialog Boxes?

Dialog boxes are separate windows that are used to convey information to, and from, the application user. They can also be used to interact with the user to obtain input that is returned to the application. You are likely very familiar with the kind of message windows that display informational, warning, and error messages. It is also likely that you've used a dialog box to provide information to applications. Now you'll see what you need to do to use these dialog boxes in your applications.

What's a JDialog?

Listing 16-1 shows the class hierarchy for the JDialog class. It's interesting to note how similar this hierarchy is to the JFrame class,[2] shown in Listing 16-2.
[1]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JDialog.html. [2]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JFrame.html. **Listing 16-1.** JDialog Class Hierarchy

```
wsadmin>from javax.swing import JDialog wsadmin>
wsadmin>classInfo( JDialog )
javax.swing.JDialog
| java.awt.Dialog
| | 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 | javax.swing.WindowConstants
| javax.accessibility.Accessible | javax.swing.RootPaneContainer
wsadmin>

This similarity should help you understand that JDialog instances should be able to contain the same kind of Swing components that JFrame instances do.

Listing 16-2. JFrame Class Hierarchy

```
wsadmin>from javax.swing import JFrame 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>
```

These similarities made me wonder just how similar the two top-level containers might be as well. Using the final version of the javadocInfo.py script file discussed in Chapter 15,[3] take a quick look at the JDialog methods shown in Figure 16-1.

[3]Which is ...\Code\Chap_15\javadocInfo_07.py.

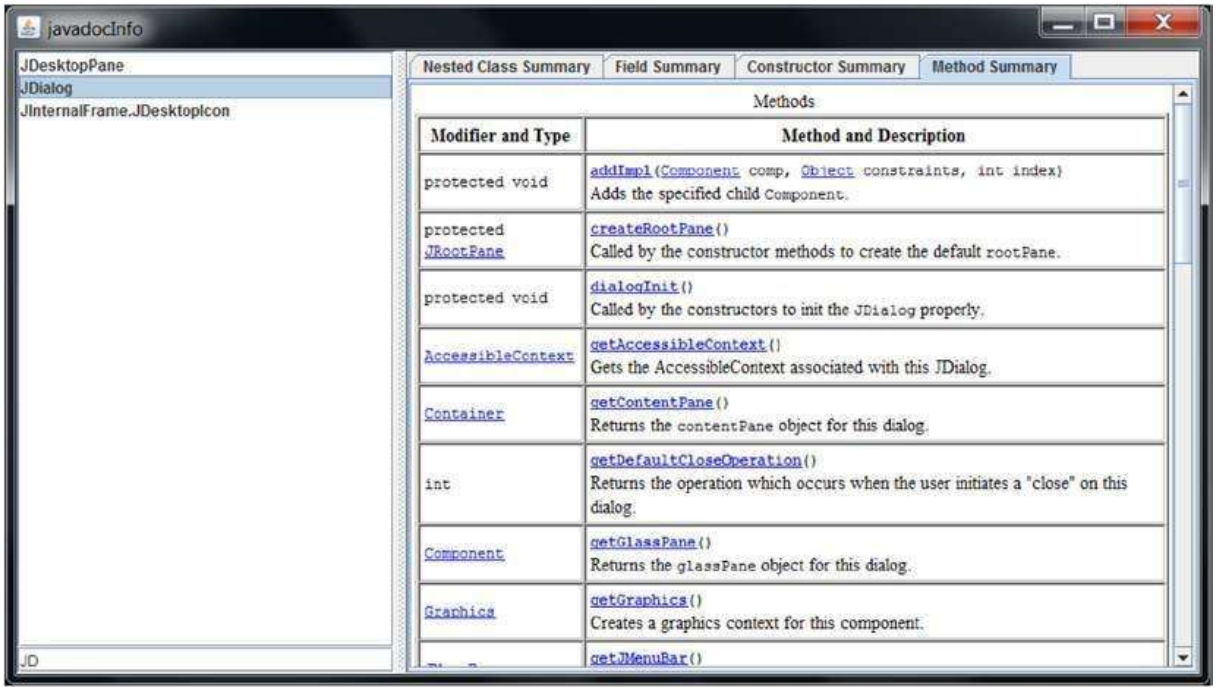


Figure 16-1. *JDialog methods*

It also makes me feel good about the effort that was invested in creating that script/application. After starting it, you only need to type two letters and use two mouse clicks to display the information about the class methods:

1. Type JD to filter the list of classes.
2. Select the JDialog entry on the short list of classes.
3. Select the Method Summary tab. You're done!

A quick glance at these methods shows that the JFrame and JDialog classes both have RootPanes, ContentPanes, GlassPanes, MenuBars, and many other things in common. There are a number of differences as well. If you start by looking at the class constructors for each class, it should quickly become obvious that there are a lot more JDialog constructors than there are JFrame constructors. It seems that this is the result of the presence in the JDialog constructors of two additional parameters:

• The owner argument (dialog, frame, or window)
• The modality argument (a Boolean flag or a Dialog.ModalityType)

What's the GraphicsConfiguration Component Do?

One issue that pops up time and time again in the Javadocs for various Swing-

related classes is that they contain so much information that it is often difficult to comprehend. This is a long way of saying that the first few (perhaps dozen) times that I looked at the JFrame and JDialog Javadoc pages, I didn't take the time or effort to see the GraphicsConfiguration[4] parameter in some of the constructors.

[4]See http://docs.oracle.com/javase/8/docs/api/java/awt/GraphicsConfiguration.html.

When I finally did see it, it took me a while to realize how useful this particular parameter can be. However, this is only the case when the users can have multiple displays. You can use this parameter to identify on which of the available displays the application will open a JFrame or JDialog instance.

Listing 16-3 shows one technique for determining the position, width, and height of the available displays using the GraphicsEnvironment class. Did you notice that this class comes from java.awt and not the javax.swing library?

Listing 16-3. Using the GraphicsEnvironment Class

```
wsadmin>from java.awt import GraphicsEnvironment
wsadmin>
wsadmin>LGE = GraphicsEnvironment.getLocalGraphicsEnvironment()
wsadmin>for SD in LGE.getScreenDevices() :
wsadmin> for GC in SD.getConfigurations() :
wsadmin> print GC.getBounds()
wsadmin>
java.awt.Rectangle[x=1920,y=0,width=1920,height=1080]
java.awt.Rectangle[x=0,y=0,width=1920,height=1080]
wsadmin>
```

Table 16-1 describes each of the statements in the interactive wsadmin session shown in Figure 16-1.

Table 16-1. *Using the GraphicsEnvironment Session*

Lines	Description
1	Add the GraphicsEnvironment class from the java.awt library to the local namespace.
3	Use the getLocalGraphicsEnvironment() method to obtain information about

the local graphics environment.

4 Loop over the ScreenDevice objects in the environment.
5 Loop over the GraphicConfiguration objects in each ScreenDevice object.
6 Display the bound instance values for the current GraphicConfiguration object.
8 - 9 Results generated by the print statement in line 6.

The output shown in Listing 16-3 is specific to my environment and shows that it has two displays, logically configured to be side by side, with the primary display on the right. How do you tell this? Well, the primary display (“Screen 0”) is considered to be at location [x=0, y=0]. The next display (“Screen 1”) has a positive x value, which indicates that it is logically to the right of the primary display. If a third display were present that was logically considered to be on the left of the primary display, it would have a negative value of x. Adding the values of x and width from the second display (1920 + 1920 = 3840) results in the logical width of the combined display.

Using a GraphicsConfiguration Object

Listing 16-4 shows how to obtain a GraphicsConfiguration object from a ScreenDevice object. Once you have it, how can it be used? The JDialog and JFrame classes both have constructors that include GraphicsConfiguration objects. So, once you have a configuration object that identifies a display, this can be provided on the JDialog or JFrame constructor call to identify the display with which the application should be associated. Listing 16-4 shows an example class that uses a GraphicsConfiguration object to identify the user display on which the JFrame instance should be displayed.

Listing 16-4. Using a GraphicsConfiguration Object

8|class ScreenLoc(java.lang.Runnable) :
9| def run(self) :
10| d = 0
11| LGE = GraphicsEnvironment.getLocalGraphicsEnvironment() 12| for GD in LGE.getScreenDevices() :
13| CO = GD.getDefaultConfiguration()
14| for GC in GD.getConfigurations() :
15| b = GC.getBounds()
16| frame = JFrame(

```
17| 'Screen: %d' % d,
18| CO,
19| size = (
20| int( b.getWidth() ) >> 1,
21| int( b.getHeight() ) >> 3
22| ),
23| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 24| )
25| d += 1
26| frame.add( JLabel('CO' ) )
27| frame.setVisible( 1 )
```

Table 16-2 explains the ScreenLoc class shown in Listing 16-4 in detail. One way to figure out what the code is doing is to keep an eye out while viewing the Javadoc pages of the code snippets. In this case, some of this is explained on the GraphicsConfiguration and GraphicsDevice pages, which help explain how they can be used. The ScreenLoc class in Figure 16-1 is based on these code snippets.

Table 16-2. *ScreenLoc Class Explained*

Lines	Description
8-9	Define the class as a descendent of the java.lang.Runnable class; its run() method will be called by the Swing Dispatch thread.
10	d is a simple integer identifying the device number 0..N-1.
11	Call the static getLocalGraphicsEnvironment() method from the GraphicsEnvironment[5] class to obtain information about the local graphics environment (LGE).
12-26	Process each GraphicsDevice[6] (GD) in the local graphics environment.
13	Get the default GraphicsConfiguration object (CO) associated with the current device.
14-24	Process all of the GraphicsConfiguration (GC) objects for the current GraphicsDevice.
15	Get the bounds (java.awt.Rectangle[7]) describing the device coordinates.
16-23	Create a JFrame instance on the current GraphicsDevice using the configuration object.

Note: Since only the size is provided, the JFrame instance will be positioned in the top-left corner of the display.

19-22 The size of the frame is defined to be half the display width and an eighth of the display height.
24 Increment the display counter (d).
25 Add a JLabel instance to the current frame containing the information about the configuration object.
26 Make the frame instance visible.

Is a GraphicsConfiguration Object Really Necessary?

If you want to create a frame or a dialog window using a configuration object, you can. But it would be just as easy to use the device bounds rectangle to position the window, especially in Jython. Listing 16-5 shows the ScreenPos class, from the script of the same name, which shows how easily this can be done.

Listing 16-5. Using Device Bounds to Position Items

```
8|class ScreenPos( java.lang.Runnable ) :
9| def run( self ) :
10| d = 0
11| LGE = GraphicsEnvironment.getLocalGraphicsEnvironment() 12| for GD in LGE.getScreenDevices() :
13| for GC in GD.getConfigurations() :
14| b = GC.getBounds()
15| w = int( b.getWidth() ) >> 1
16| h = int( b.getHeight() ) >> 3
17| x = int(
18| ( int( b.getWidth() - w ) >> 1 ) + b.getX() 19| )
20| y = int(
21| ( int( b.getHeight() - h ) >> 1 ) + b.getY() 22| )
23| frame = JFrame(
24| 'Screen: %d' % d,
25| bounds = ( x, y, w, h ),
26| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 27| )
28| d += 1
29| frame.add( JLabel('GC' ) )
30| frame.setVisible( 1 )
```

[5] See

http://docs.oracle.com/javase/8/docs/api/java/awt/GraphicsEnvironment.html.
[6]See http://docs.oracle.com/javase/8/docs/api/java/awt/GraphicsDevice.html.
[7]See http://docs.oracle.com/javase/8/docs/api/java/awt/Rectangle.html.

Table 16-3 explains the statements shown in Listing 16-5 in detail. Remember that these listings sometimes have more lines than might normally be used only because of the narrow width required for these pages.

Table 16-3. *ScreenPos Example, Explained*
Lines Description

8-9 Define the class as a descendent of the java.lang.Runnable class; its run() method will be called by the Swing Dispatch thread.
10 d is a simple integer identifying the device number 0..N-1.
11 Call the static getLocalGraphicsEnvironment() method from the GraphicsEnvironment class to obtain information about the local graphics environment (LGE).

12-29 Process each GraphicsDevice (GD) in the local graphic environment.
13-29 Process all of the GraphicsConfiguration (GC) objects for the current GraphicsDevice.
14 Get the bounds (b) Rectangle defining the GraphicsDevice.

15 The width (w) of the window to be instantiated should be half the width of the graphic device on which it will be seen.
16 The height (h) of the window should be an eighth the height of the display screen.
17-19 The window's upper-left corner (the X coordinate) is half the screen width minus the width of the window. Adding the screen's X coordinate positions it properly.
20-22 The window's upper-left corner (the Y coordinate) is half the screen height minus the height of the window. Adding the screen's Y coordinate positions it properly.

23-26 Instantiate the JFrame using the given title and the previously computed coordinates.
27 Increment the display counter (d).

28 Add a JLabel instance to the current frame containing the information about

the GraphicsConfiguration object.
29 Make the frame instance visible.

The result of all of this simply shows that using a GraphicsConfiguration object isn't required. It is generally easier to use the Rectangle object available with the GraphicsDevice to determine where to position the window, based on the window and screen sizes.

What About an Owner?

In the section entitled, "What's a JDialog?," I mentioned that there are two parameters that exist in most of the JDialog constructors that don't exist in the JFrame constructors. These are the owner and modality parameters, and they are closely associated. When a JDialog instance is configured to be modal and the window is visible, all user input is blocked to all of the other application windows. This forces the user to interact with the modal dialog window first.

Note if the owner parameter is specified as None, then a shared, hidden frame will be identified as the owner.
Figure 16-2 shows some sample output generated by the SimpleDialog.py script. It shows how you can create multiple non-modal dialog windows, but only one modal one.

Figure 16-2. *SimpleDialog sample output*

Listing 16-6 shows the SimpleDialog class from this application. It only needs an ActionListener event handler to respond to when one of the application buttons is clicked. Note how the JDialog constructor specifies the owner using a value of None (line 36). This forces the Swing library to use a (shared) hidden frame, which forces the modal dialog box to be handled first.

Listing 16-6. The SimpleDialog Class from SimpleDialog.py

```
9|class SimpleDialog( java.lang.Runnable ) :
10| def run( self ) :
11| self.frame = frame = JFrame(
```

```
12| 'SimpleDialog',
13| size = ( 250, 100 ),
14| layout = GridLayout( 0, 2 ),
15| locationRelativeTo = None,
16| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
17| )
18| frame.add(
19| JButton(
20| 'Modal',
21| actionPerformed = self.makeDialog
22| )
23| )
24| frame.add(
25| JButton(
26| 'non-Modal',
27| actionPerformed = self.makeDialog
28| )
29| )
30| self.boxNum = 0
31| frame.setVisible( 1 )
32| def makeDialog( self, event ) :
33| cmd = event.getActionCommand()
34| isModal = ( cmd == 'Modal' )
35| dialog = JDialog(
36| self.frame, # Try None as owner... 37| title = '%d: %s' % ( self.boxNum, cmd
), 38| modal = isModal,
39| size = ( 200, 100 ),
40| locationRelativeTo = self.frame
41| )
42| self.boxNum += 1
43| dialog.add( JLabel( cmd ) )
44| dialog.setVisible( 1 )
```

What happens if you iconify the application when non-modal dialog windows exist? Well, if they were instantiated with the owner parameters specified as None, nothing happens to them. They remain as they are. However, if the application is identified as the owner (in this case specifying an argument of self.frame), then all of the associated non-modal windows will be iconified as well. And if you deiconify the application, all of the non-modal windows will

follow suit. Knowing this might help you decide if you want to specify the owner argument when you instantiate your JDialog instance.

Where's the Dialog?

At the start of this chapter, you saw that a dialog is about communication. Up to now, you haven't done anything with the JDialog instances that you've created other than make them appear and disappear. Where's the communication in that? This section explains how you enable a JDialog instance to communicate with the application or the component that created it.

Let's start with the simple case. If your application creates a modal JDialog instance, the application will be in suspended animation until the dialog is hidden or closed. Since the application created the object instance, it can now call any of its getter methods to retrieve information from the object. Figure 16-3 shows some sample output from a trivial application that uses a custom JDialog box to obtain user input.

Figure 16-3. *CustomDialog1.py sample output*

Listing 16-7 shows the CustomDialog class from this sample application. Since there are some points of interest, this section will describe in more detail how this application works.

Listing 16-7. CustomDialog Sample Class

```
12|class CustomDialog( java.lang.Runnable ) :
13| def run( self ) :
14| self.frame = frame = JFrame(
15| 'CustomDialog',
```

```
16| size = ( 200, 100 ),
17| locationRelativeTo = None,
18| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
19| )
20| self.label = frame.add( JLabel( " ) )
21| frame.add(
22| JButton(
23| 'Prompt user',
24| actionPerformed = self.popup
25| ),
26| BorderLayout.SOUTH
27| )
28| frame.setVisible( 1 )
29| def popup( self, event ) :
30| self.dialog = dialog = JDialog(
31| self.frame,
32| 'Name prompt',
33| 1,
34| layout = GridLayout( 0, 2 ),
35| locationRelativeTo = self.frame
36| )
37| dialog.add( JLabel( "What's your name?" ) )
38| self.text = dialog.add(
39| JTextField(
40| 10,
41| actionPerformed = self.enter
42| )
43| )
44| self.result = None
45| dialog.pack()
46| dialog.setVisible( 1 )
47| self.label.setText( 'Name = "%s"' % self.result )
48| def enter( self, event ) :
49| self.result = self.text.getText()
50| self.dialog.setVisible( 0 )
```

Table 16-4 describes the CustomDialog class shown in Listing 16-7 in more detail. For simplicity's sake, the code that's similar to other examples is described in less detail.

Table 16-4. *CustomDialog Class, Explained*

Lines	Description
13-28	CustomDialog run() method that populates the JFrame instance and displays it for the user.
20	The popup() ActionListener method needs to access the label field, so addressability to it must be saved.
29-47	popup() ActionListener event handler, which is invoked when the button is activated. 30-36 Create a modal JDialog instance that identifies the application frame as the owner and tell the Swing library to display the JDialog instance relative to it.
37-46	Populate the JDialog and make it visible.
47	This is the point where control returns when the JDialog is no longer visible (it's either closed or hidden).
48-50	ActionListener event handler for the JDialog user input field (JTextField). **Note:** This event handler expects to be associated with a modal dialog instance. It does not include any protection from concurrency issues.

The last row of Table 16-4 includes one of the significant reasons for using modal dialog boxes. Dealing with concurrency issues can add complexity to your applications. So think carefully before using non-modal dialog boxes that can update values in the component that creates it.

Multiple Modal Dialog Boxes Are Annoying

Please be considerate of your users. Even though it's possible for your application to create multiple modal dialog boxes, don't do it. Unfortunately, I see this quite often, especially when a web application uses Adobe Flash. I tend to see something like the image shown in Figure 16-4. Unfortunately, these boxes can stack up, so I find myself clicking on the Deny ("No") button a lot. This can be frustrating. As a user, I try to avoid using this kind of application whenever possible. Keep this in mind when you create your applications. Don't create a "user-hostile" application that does something annoying like this.

Figure 16-4. *Please take "no" for an answer*

Using JOptionPane Methods

It's commonplace to come across applications that use modal dialog boxes to obtain user input, so it should be no surprise that there's an easy way to create a variety of commonly used dialog boxes.

There are four major and two minor variants of methods in the JOption class for creating modal dialog boxes to obtain user input. All of these methods have names that adhere to the following naming convention: show%1%2Dialog(), where %2 represents the major portion, which will be one of the following four types:

• Input
• Confirm
• Message
• Option

And %1 represents the minor portion, which will be empty or internal. This means that you should be able to produce the complete list of method names by using a simple nested loop, right? Listing 16-8 shows this simple list.
Listing 16-8. JOption show*Dialog Method Name Variants

```
wsadmin>num = 1
wsadmin>for minor in [ '', 'Internal' ] :
wsadmin> for major in [ 'Input', 'Confirm', 'Message', 'Option' ] : wsadmin> print
'%2d: show%s%sDialog()' % ( num, minor, major ) wsadmin> num += 1
wsadmin>

1: showInputDialog()
2: showConfirmDialog()
3: showMessageDialog()
```

4: showOptionDialog()
5: showInternalInputDialog()
6: showInternalConfirmDialog()
7: showInternalMessageDialog()
8: showInternalOptionDialog()
wsadmin>

Unfortunately, it doesn't take into consideration the fact that Java allows method overloading, so each method name can have multiple signatures. What does this mean for Jython programmers? It means that you have to take this into consideration.

■ **Note** For now, ignore the showInternal variants. You'll see them again in Chapter 19, where you learn about JInternalFrames.

The JOptionPane.showMessageDialog() Method

The simplest dialog box that can be displayed by the JOptionPane class is called a MessageDialog. It is the simplest primarily because it returns no value. The purpose of a MessageDialog is to inform users about some event. Since it doesn't provide the user with any opportunity to provide input, the showMessageDialog() method doesn't have to return any value.

Figure 16-5 shows the Java method signature and a simple image from the MessageDialogDemo.py application.

Notice that the dialog box has a simple title of "Message," an informational icon, and an OK button.

Figure 16-5. *Simple MessageDialog signature and sample image*

Figure 16-6 shows another showMessageDialog() method signature, as well as an example image from the MessageDialogDemo.py application. This version of the method allows you to specify the title to be displayed in the title bar. It also allows you to specify the messageType and an optional icon to be displayed. If an icon isn't provided, then the icon that is displayed on the MessageDialog is based on the messageType that was specified.

```
showMessageDialog(
    Component parentComponent,
    Object    message,
    String    title,
    int       messageType,
    Icon      icon
)
```

Figure 16-6. *Complete MessageDialog signature and sample image*

Figure 16-7 shows the list of MessageType constants that are defined in the JOptionPane class. These should be used for messageType arguments in the show*Dialog() method calls. If you don't provide an icon parameter, the icon displayed on the dialog box will be based on the MessageType.

```
PLAIN_MESSAGE

ERROR_MESSAGE

INFORMATION_MESSAGE

WARNING_MESSAGE

QUESTION_MESSAGE
```

Figure 16-7. *MessageType constants defined in the JOptionPane class*

The JOptionPane.showOptionDialog() Method

The next most useful show*Dialog method is showOptionDialog(). It is useful, and frequently used, because of its flexibility. It can display a custom dialog. Figure 16-8 shows the list of parameters that should be specified when this method is called.

```
showOptionDialog(
    Component parentComponent,
    Object    message,
    String    title,
    int       optionType,
    int       messageType,
    Icon      icon,
    Object[]  options,
    Object    initialUalue
)
```

Figure 16-8. *Complete showOptionDialog signature*

This method creates a dialog with a specified icon, where the initial choice is

determined by the initialValue parameter and the number of choices is determined by the optionType parameter.
You saw how the messageType value can be used to determine the icon to be shown if it's not specified. How is the optionType parameter used and what values can be specified? Figure 16-9 shows the JOptionPane constants that can be used for the optionType argument.

Figure 16-9. *OptionType constants defined in the JOptionPane class*

It would seem that you can use the optionType and messageType arguments to tell the showOptionDialog() method how many buttons to display, as well as their labels. Listing 16-9 shows the simplest form of a call to the showOptionDialog() method. Note how you can use an optionType of DEFAULT_OPTION and None for the options and initialValue arguments.

Listing 16-9. Sample showOD() Method Used to Call showOptionDialog

```
26| def showOD( self, event ) :
27| options = 'Bacon,Eggs,Spam'.split( ',' )
28| result = JOptionPane.showOptionDialog(
29| self.frame, # parentComponent
30| 'What goes good with spam?', # message text
31| 'This is a test!', # title
32| JOptionPane.DEFAULT_OPTION, # optionType 33|
JOptionPane.QUESTION_MESSAGE, # messageType 34| None, # icon 35|
None, # options 36| None # initialValue 37| )
38| self.label.setText( 'result = %d' % result )
```

Figure 16-10 shows the different buttons that are displayed on the OptionDialog when you specify the various constants listed in Figure 16-9.

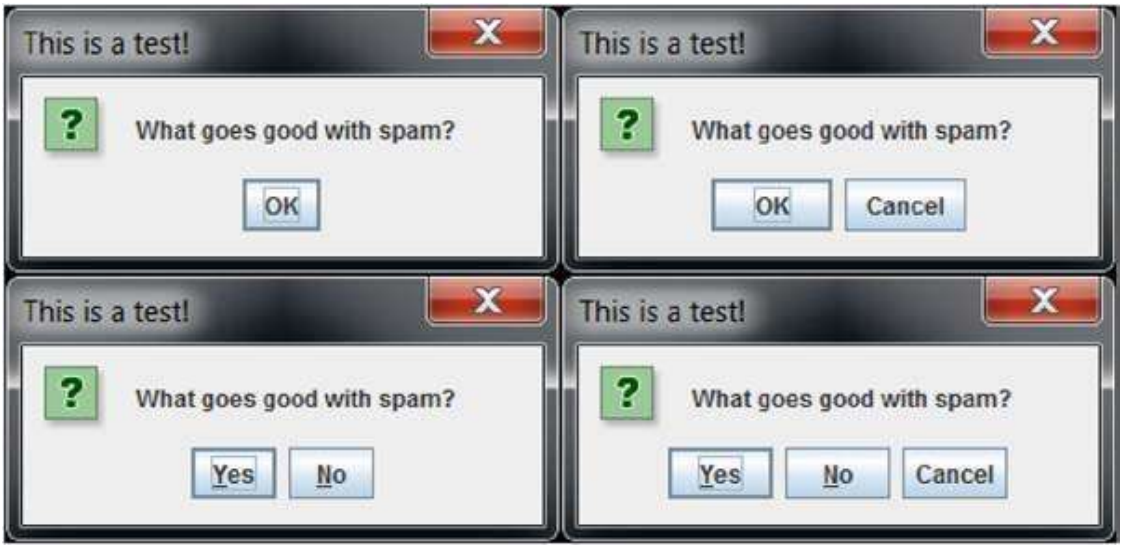


Figure 16-10. *Simple OptionDialog examples*

What if you don't like these kinds of buttons? Can you specify button labels other than Yes, No, OK, and Cancel? Sure, that's what the options and initialValue arguments are for. Listing 16-10 shows how to provide your own button labels for the OptionDialog box.

Listing 16-10. Modified showOD() Method Specifying Options

```
26| def showOD( self, event ) :
27| options = 'Bacon,Eggs,Spam'.split( ',' )
28| result = JOptionPane.showOptionDialog(
29| self.frame, # parentComponent
30| 'What goes good with spam?', # message text
31| 'This is a test!', # title
32| JOptionPane.DEFAULT_OPTION, # optionType
33| JOptionPane.QUESTION_MESSAGE, # messageType
34| None, # icon
35| options, # options
36| options[ -1 ] # initialValue
37| )
38| self.label.setText( 'result = %d' % result )
```

Figure 16-11 shows the OptionDialog that results from this showOD() method.

Figure 16-11. *OptionDialog example using non-standard button labels*

Are you limited to one, two, or three buttons? As a matter of fact, you're not. If you provide an array (list) of options with more than three values, the showOptionDialog() is able to show them to you.
The example OptionDialog shown in Figure 16-12 can easily be produced by changing line 27 in Listing 16-10 to the statement in Listing 16-11.

Figure 16-12. *OptionDialog example with eight buttons*

Listing 16-11. Specifying Many OptionDialog Button Labels

```
27| options = 'Now is the time for all good spam'.split( ' ' )
```

Certainly this example is a bit excessive, but it does demonstrate an interesting point. It should also raise a question or two. When you display a variety of buttons on a dialog box, how can you tell which one the user pressed? If you use one of the standard OptionDialog objects, you can use the return value constants defined in the JOptionPane class and listed in Figure 16-13 to make this determination.

Figure 16-13. *JOptionPane constants for the showOptionDialog returned value*

If you specify your own list of button labels, you can use the returned value as an index into the options array to determine the label of the selected button. Be careful, though. Before using it as an index, you should check to see if a value of -1 (the JOptionPane.CLOSED_OPTION) was returned in case the user closed the dialog box.

The JOptionPane.showConfirmDialog() Method

By now, you should be very comfortable with the parameters that exist for the JOptionPane.show*Dialog() methods. This will make the explanation of the showConfirmDialog() method that much easier. The simplest form of this method is shown in Figure 16-14.

```
| showConfirmDialog(                                                    |
|     Component parentComponent,                                        |
|     Object    message                                                 |
| )                                                                     |
```

Figure 16-14. *Simple showConfirmDialog() arguments*

This method can be used to display a simple dialog box with three choices (buttons): Yes, No, and Cancel. This is identical to calling the showOptionDialog() method and specifying an optionType value of JOptionPane.YES_NO_ CANCEL_OPTION. The default title is “Select an Option,” and you can use the JOptionPane constants listed in Figure 16-13 to determine which selection the user chose.

The other showConfirmDialog() signatures follow this same pattern, with one allowing an optionType, the next having an optionType as well as a messageType, and the last allowing these two arguments in addition to an icon. Again, when the optionType is provided, it determines the number of buttons and which values should be shown. The messageType identifies the icon to be

shown, if the icon parameter isn’t specified.

The JOptionPane.showInputDialog() Method

The last of the show*Dialog() methods discussed in this chapter is showInputDialog(). It has a number of variations. You’ll take a look at these variations so you can decide which make most sense for your needs. Let’s start by looking at the simplest one, which has only one parameter and work your way toward the most complex version, which has seven parameters.

Figure 16-15 shows the simplest variant, which requires only a string parameter containing the message to be displayed. In fact, there are two forms of this variant—with and without the parentComponent parameter.

```
| showInputDialog(                                                    |
|     Object    message                                               |
| )                                                                   |
```

```
| showInputDialog(                                                    |
|     Component parentComponent,                                      |
|     Object    message                                               |
| )                                                                   |
```

Figure 16-15. *Simplest showInputDialog() method signatures*

Listing 16-12 shows the showID() method from the InputDialogDemo.py application, with the optional parentComponent specified.

Listing 16-12. Simplest showInputDialog() Example

```
26| def showID( self, event ) :
27| result = JOptionPane.showInputDialog(
28| self.frame, # parentComponent
29| 'What is your favorite color?' # message text
30| )
31| self.label.setText( 'result = "%s"' % result )
```

The result of executing the showInputDialog() method call shown in Listing 16-12 is illustrated in Figure 16-16. Personally, I haven’t been able to figure out a difference between these two variants. Regardless of whether the optional parentComponent is specified, you see a dialog box with a title of “Input” and the specified message, as well as OK and Cancel buttons.

Figure 16-16. *Sample showInputDialog() window*

One interesting thing to note is that the value that is returned is not an integer, as you have seen with other show*Dialog() methods. It is either the user-specified string (the contents of the input field) or None if the dialog box is closed (which is done by using the Close icon or pressing the Cancel button).

Figure 16-17 shows the next pair of signatures for this method. In it you see that an initial value for the input field can be specified. Again, there are two variants of this showInputDialog() method, with the parentComponent being optional.

```
| showInputDialog(                                                    |
|     Object    message,                                              |
|     Object    initialSelectionValue                                 |
| )                                                                   |
```

```
| showInputDialog(                                                    |
|     Component parentComponent,                                      |
|     Object    message,                                              |
|     Object    initialSelectionValue                                 |
| )                                                                   |
```

Figure 16-17. *showInputDialog() with initialSelectionValue specified*

Listing 16-13 shows an example of this method call from the InputDialogDemo.py sample application. In this particular example, the optional parentComponent is specified in line 28. Feel free to try this version and then comment out that line and try it again.

Listing 16-13. showInputDialog() with an Initial Value

```
26| def showID( self, event ) :
27| result = JOptionPane.showInputDialog(
28| self.frame, # parentComponent
```

```
29| 'What is your favorite color?', # message text
30| 'Spam' # initialValue
31| )
32| self.label.setText( 'result = "%s"' % result )
```

When I tried this, I saw output similar to what is shown in Figure 16-18. Note how the initial value is provided and selected.

Figure 16-18. *Sample showInputDialog() window showing an initial value*

The next variation allows you to specify the title of the dialog box, as well as the messageType parameter that determines the icon to display. The signature for this version is shown in Figure 16-19.

```
| showInputDialog(                                                    |
|     Component parentComponent,                                      |
|     Object    message,                                              |
|     String    title,                                                |
|     int       messageType                                           |
| )                                                                   |
```

Figure 16-19. *showInputDialog() method with messageType parameter*

Listing 16-14 shows how the title and messageType parameters are specified. The constants to be used for the messageType parameter can be found in Figure 16-7.

Listing 16-14. showInputDialog() with Title and messageType Parameters

```
26| def showID( self, event ) :
27| result = JOptionPane.showInputDialog(
28| self.frame, # parentComponent
29| 'What is your favorite color?', # message text
30| 'Asked by the bridge guardian', # title
31| JOptionPane.QUESTION_MESSAGE # messageType
32| )
33| self.label.setText( 'result = "%s"' % result )
```

The output of the version of the showID() method in Listing 16-14 is shown in

Figure 16-20.

Figure 16-20. *Sample output showing the title and messageType*

Finally, you get to the last of the showInputDialog() method variants. As mentioned earlier, its signature is shown in Figure 16-21. The main difference between this method and the other signatures just discussed is the presence of the icon, selectionValues, and initialSelectionValue arguments.

```
| showInputDialog(                                                    |
|     Component parentComponent,                                      |
|     Object    message,                                              |
|     String    title,                                                |
|     int       messageType,                                          |
|     Icon      icon,                                                 |
|     Object[]  selectionValues,                                      |
|     Object    initialSelectionValue                                 |
| )                                                                   |
```

Figure 16-21. *Seven argument showInputDialog() method signature*

Listing 16-15 shows an example of this seven argument variant of the showInputDialog() method.[8] As mentioned, if an icon isn't provided, the icon that's displayed on the MessageDialog is based on the specified messageType value.

[8]As you've seen previously, the lines have been shortened to fit in the available space. This is most obvious with lines 27 and 28, which really don't require two statements.

Listing 16-15. Sample Use of Seven Argument showInputDialog() Method

```
26| def showID( self, event ) :
27| COLORS = 'Red,Orange,Yellow,Green,Blue,Indigo,Violet'
28| colors = COLORS.split( ',' )
29| result = JOptionPane.showInputDialog(
```

```
30| self.frame, # parentComponent
31| 'What is your favorite color?', # message text
32| 'Asked by the bridge guardian', # title
33| JOptionPane.QUESTION_MESSAGE, # messageType
34| None, # icon
35| colors, # selectionValues
36| colors[ -1 ] # initialSelectionValue
37| )
38| self.label.setText( 'result = "%s"' % result )
```

Figure 16-22 shows images from the application containing the method shown in Listing 16-15. This variant of the showInputDialog() method provides a combo box containing the values provided by the selectionValues argument. The last parameter is used to identify the initial combo box entry to be displayed.

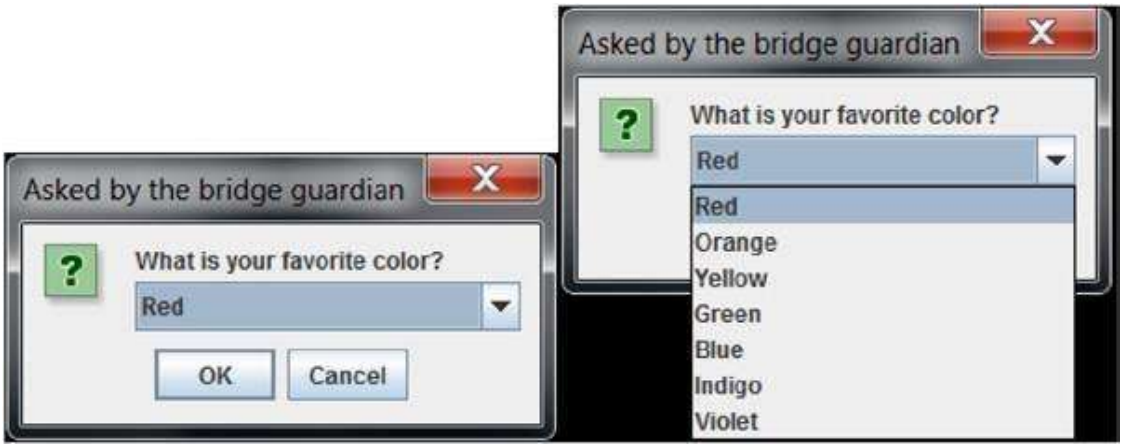


Figure 16-22. *Sample output from seven argument showInputDialog() call*

Summary

This chapter has been all about dialog boxes and some of the ways they can be used in your applications. You've seen how simple these techniques can be. The large variety of ways that dialog boxes can be created and used should be a clue about how useful they can be. I encourage you to take a look at the sample scripts in the code\Chap_16 directory and modify them to test the various techniques described in this chapter.

Additionally, you are encouraged to take a look at the "How to Make Dialogs"[9] portion of the Java Swing Tutorial, which contains much of this same information but uses Java instead of Jython as the programming language. One

thing you'll take away from reading that document is how much easier it can be to create and use dialog boxes in Jython.

In the next chapter, you investigate specialized dialog boxes, including how to create and use them in your applications.

[9]See http://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html.

Chapter 17

Specialized Dialog Boxes

In Chapter 16, you saw how to create a completely customized dialog box using the JDialog[1] class. You also saw how to take advantage of the JOptionPane[2] methods to quickly and easily display some simple, generalized dialog boxes. Now you are going to look at some dialog boxes that can be used to make your life as a developer significantly easier. You'll be able to use dialog boxes that have been designed, implemented, and tested for you. The chapter begins by looking at the JFileChooser[3] class, which the Swing developers were kind enough to provide for creating dialog boxes. These are great examples of the kind of user input that can be performed by dialog boxes. Keep these in mind when you encounter a situation where you want to provide your own dialog boxes.

The JFileChooser Class

One of the most common types of dialog boxes is the ones that let the users traverse the filesystem and specify a file or directory to be used by the application. It is important to remember, though, that the JFileChooser instance helps your application to interact with the users to choose a particular file or directory. It is the responsibility of the application to do something with the specified file or directory.

Let's take a look at just how difficult it is to use the JFileChooser class. Listing 17-1 shows a simple event handler that displays a File Chooser dialog box that allows the users to traverse the local filesystem and select a file to be opened.

Listing 17-1. Trivial Sample JFileChooser() Routine

```
26| def showFC( self, event ) :
27| fc = JFileChooser()
```

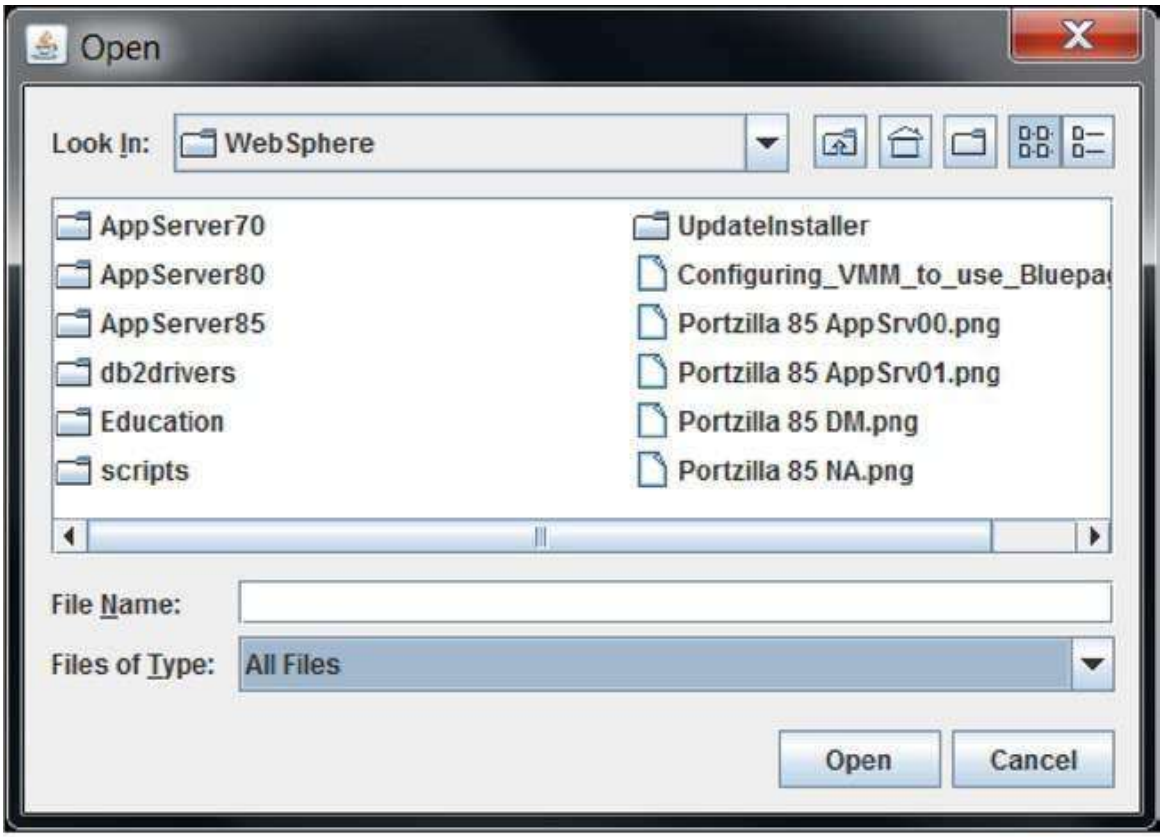
```
28| result = fc.showOpenDialog( None )
29| if result == JFileChooser.APPROVE_OPTION :
30| message = 'result = "%s"' % fc.getSelectedFile()
31| else :
32| message = 'Request canceled by user'
33| self.label.setText( message )
```

Figure 17-1 shows an example dialog box after the user has moved about the filesystem; it’s displaying the contents of the C:\IBM\WebSphere directory.

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

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

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


Open
Look In:
WebSphere
AppServer70
AppServer80
AppServer85
db2drivers
Education
scripts
UpdateInstaller
Configuring_VMM_to_use_Bluepa
Portzilla 85 AppSrv00.png
Portzilla 85 AppSrv01.png
Portzilla 85 DM.png
Portzilla 85 NA.png
File Name:
Files of Type:
All Files
Open
Cancel

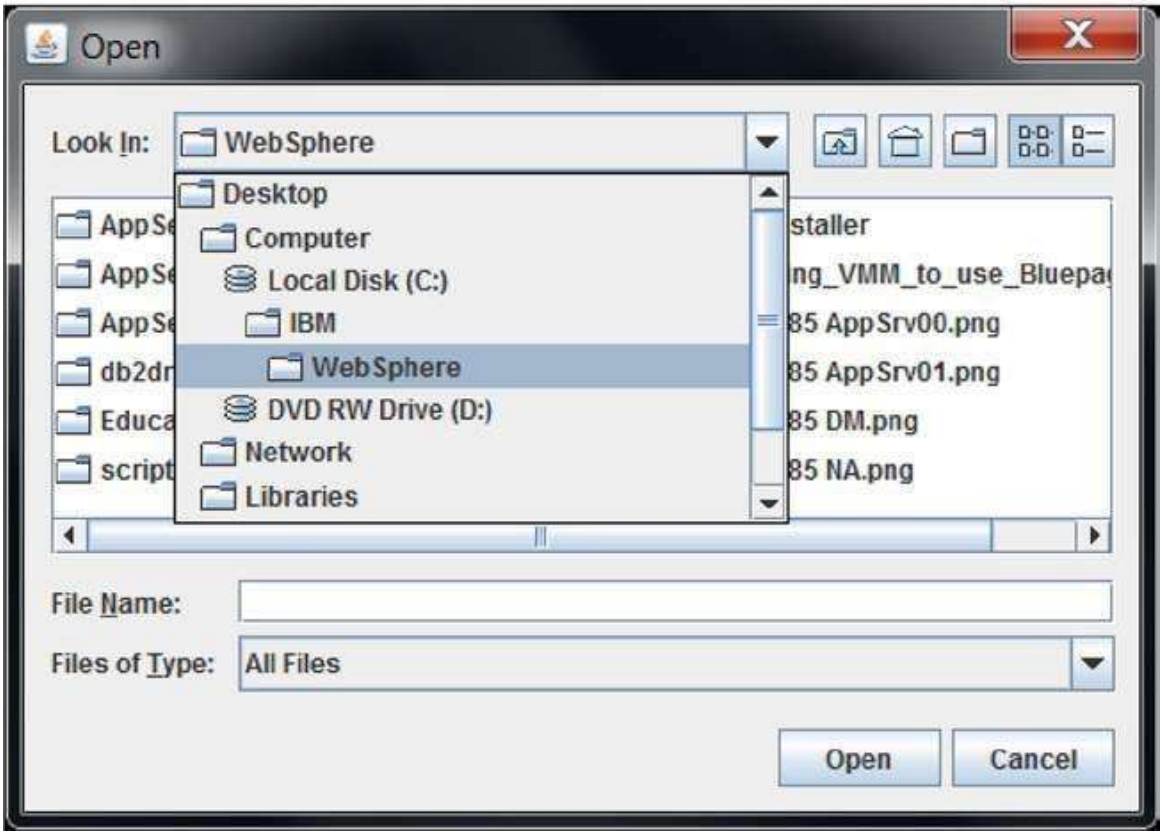

Open
Look In:
WebSphere
Desktop
Computer
Local Disk (C:)
IBM
WebSphere
DVD RW Drive (D:)
Network
Libraries
File Name:
Files of Type:
All Files
Open
Cancel

Figure 17-1. *Sample JFileChooser() Open dialog box from FileChooserDemo1.py*

It doesn't take much to realize how powerful this dialog box can be. For example, clicking on the Look In dropdown shows a combo box with indented entries for each directory level, as well as some specialized icons to indicate directories and disk drives.

Now let's take a look at which constructors are available with the JFileChooser class, so that you can have your applications create the appropriate kind of JFileChooser instance best suited for your needs.
JFileChooser Constructors

Which constructors exist for the JFileChooser class? Table 17-1 shows the Java constructor signatures and includes a short description about each constructor. The first constructor is the one that was used in Listing 17-1, in line 27. The next two signatures allow you to identify the starting view of the JFileChooser dialog box using either java.io.File or java.lang.String to identify the initial view to be shown. That makes it easy to indicate the starting directory to be displayed by the JFileChooser instance.

Table 17-1. *JFileChooser Constructors* **Signature Description**

JFileChooser()
JFileChooser(File currentDirectory) JFileChooser(String currentDirectoryPath) JFileChooser(FileSystemView fsv)

JFileChooser(File currentDirectory, FileSystemView fsv)
JFileChooser(String currentDirectoryPath, FileSystemView fsv)

Constructs a JFileChooser pointing to the user's default directory. Constructs a JFileChooser using the given File as the path. Constructs a JFileChooser using the given Path.
Constructs a JFileChooser using the given FileSystemView.

Constructs a JFileChooser using the given currentDirectory and FileSystemView.
Constructs a JFileChooser using the given currentDirectoryPath and FileSystemView.

The last three JFileChooser constructors include a FileSystemView parameter

that can be used to limit the portion of the filesystem that the JFileChooser can display to the user. This way, you don’t have to allow the user to see the whole filesystem, just the portion they need.

Using a FileSystemView

The default FileSystemView instance that is used by the JFileChooser allows you to access the complete filesystem. Listing 17-2 shows a verbose way to provide or identify the default FileSystemView instance. It is, in fact, identical to the default JFileChooser( ) instantiation when no parameter is specified.

Listing 17-2. JFileChooser( ) Instance with Default FileSystemView from FileChooserDemo2.py

```
27| def showFC( self, event ) :
28| fc = JFileChooser( FileSystemView.getFileSystemView() )
29| result = fc.showOpenDialog( None )
30| if result == JFileChooser.APPROVE_OPTION :
31| message = 'result = "%s"' % fc.getSelectedFile()
32| else :
33| message = 'Request canceled by user'
34| self.label.setText( message )
```

If, on the other hand, you are interested in limiting the user to a specific subset of the filesystem, you can implement a class that identifies the virtual “root” of the filesystem that you want the user to be able to traverse. Listing 17-3 shows one implementation of this kind of FileSystemView descendent class.[4]

Listing 17-3. FileSystemView Class that Limits Filesystem Access from FileChooserDemo3.py

```
11|class RestrictedFileSystemView( FileSystemView ) :
12| def __init__( self, root ) :
13| FileSystemView.__init__( self )
14| self.root = root
15| self.Roots = [ root ]
16| def createNewFolder( self, containingDir ) :
17| folder = File( containingDir, 'New Folder' )
18| folder.mkdir()
19| return folder
```

```
20| def getDefaultDirectory( self ) :
21| return self.root
22| def getHomeDirectory( self ) :
23| return self.root
24| def getRoots( self ) :
25| return self.Roots
```

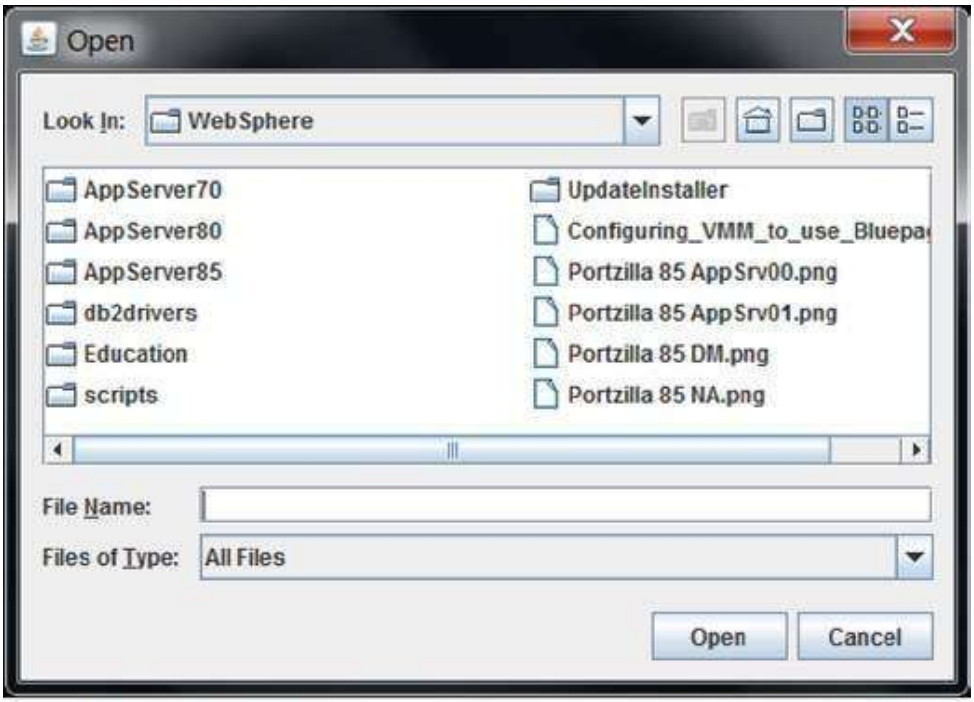
Listing 17-4 shows how this RestrictedFileSystemView class might be used to limit the JFileChooser to the specified directory and its subdirectories. It is unlikely, however, that you are going to want the “root” directory specified using a string like this one. It would be more reasonable to include statements to determine the root system based on the specific environment in which the script is executing.

Listing 17-4. JFileChooser() Using RestrictedFileSystemView from FileChooserDemo3.py

```
43| def showFC( self, event ) :
44| fc = JFileChooser(
45| RestrictedFileSystemView(
46| File( r'C:\IBM\WebSphere' )
47| )
48| )
49| result = fc.showOpenDialog( None )
50| if result == JFileChooser.APPROVE_OPTION :
51| message = 'result = "%s"' % fc.getSelectedFile()
52| else :
53| message = 'Request canceled by user'
54| self.label.setText( message )
```

Figure 17-2 shows some sample images from the application from Listings 17-3 and 17-4. It is interesting to see how the initial directory is WebSphere, and that the “up one level” button is disabled because WebSphere is considered the “root” directory of this restricted filesystem.

[4]Note: The FileSystemView roots attribute is read-only, so this class works around that limitation by using a variable named Roots instead.


Open
Look In:
WebSphere
AppServer70
AppServer80
AppServer85
db2drivers
Education
scripts
UpdateInstaller
Configuring_VMM_to_use_Bluepa
Portzilla 85 AppSrv00.png
Portzilla 85 AppSrv01.png
Portzilla 85 DM.png
Portzilla 85 NA.png
File Name:
Files of Type:
All Files
Open
Cancel

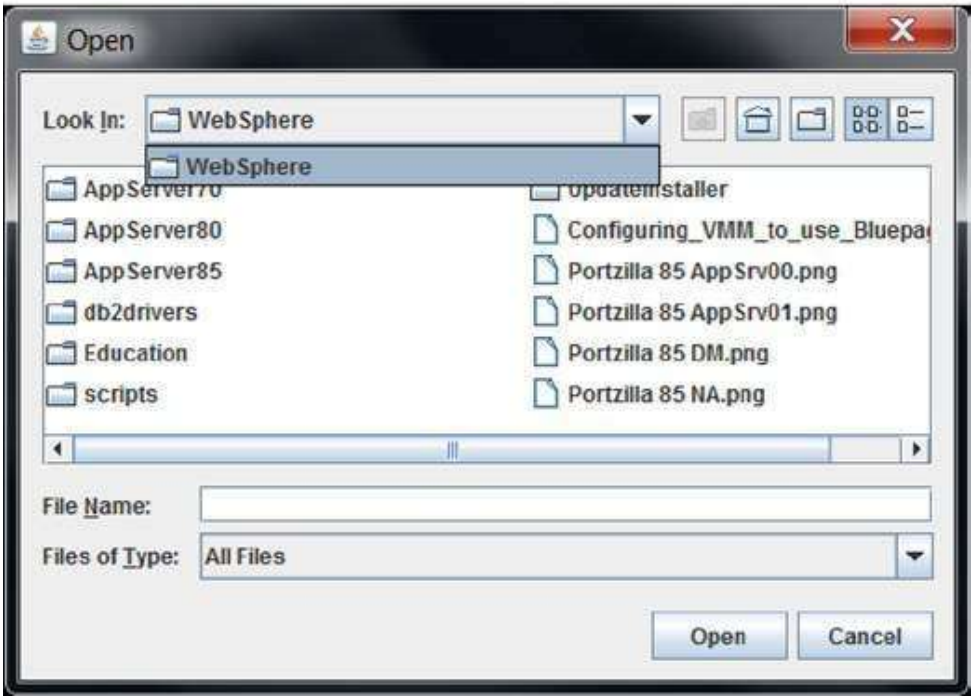

Open
Look In:
WebSphere
WebSphere
AppServer80
AppServer85
db2drivers
Education
scripts
Configuring_VMM_to_use_Bluepa
Portzilla 85 AppSrv00.png
Portzilla 85 AppSrv01.png
Portzilla 85 DM.png
Portzilla 85 NA.png
File Name:
Files of Type:
All Files
Open
Cancel

Figure 17-2. *Sample RestrictedFileSystem JFileChooser() images*

Does this FileSystemViewer class completely limit the users from accessing the rest of the system? No, not really. For example, if the user enters something like C:\ in the File Name input field, the view will display the contents of that directory, unfortunately. There may be a way to add an input verifier to the JFileChooser input field to intercept this kind of thing, but I haven't investigated that option. If you have some success with that approach, please let me know.

File Filtering

By default, a JFileChooser instance will show all of the (non-hidden) files and directories to the user. There are times when you might prefer to limit the kinds of files to be displayed. For example, you might want to allow the user to see only the XML, text, or image files. The way to do this is by adding one or more FileFilter[5] instances to the JFileChooser. The JFileChooser then determines which files and directories should be visible.

One of the most common FileFilter mechanisms is to identify the viewable files based on file extension. To simplify this process, the Swing developers have provided a FileNameExtensionFilter[6] class.
Listing 17-5 shows how easily you can add multiple FileNameExtensionFilter instances to a file chooser. This allows users to select the kinds of files to be

shown based on the filename extensions.

Listing 17-5. The showFC Method from the FileChooserDemo4.py Script

```
44| def showFC( self, event ) :
45| fc = JFileChooser(
46| RestrictedFileSystemView(
47| File( r'C:\IBM\WebSphere' )
48| )
49| )
50| fc.addChoosableFileFilter(
51| FileNameExtensionFilter(
52| 'XML files',
53| [ 'xml' ]
54| )
55| )
56| fc.addChoosableFileFilter(
57| FileNameExtensionFilter(
58| 'Image files',
59| 'bmp,jpg,jpeg,gif,png'.split( ',' )
60| )
61| )
62| fc.addChoosableFileFilter(
63| FileNameExtensionFilter(
64| 'Text files',
65| [ 'txt' ]
66| )
67| )
68| result = fc.showOpenDialog( None )
69| if result == JFileChooser.APPROVE_OPTION :
70| message = 'result = "%s"' % fc.getSelectedFile()
71| else :
72| message = 'Request canceled by user'
73| self.label.setText( message )
```

Figure 17-3 shows an example image from the FileChooserDemo4.py script. It shows the filters available when the Files of Type combo box is selected. It is interesting to note that the initial filter to be displayed is the last one that was added. It is also interesting to note that the All Files filter is available by default. So, if you don’t want this filter to be available, you need to use the

removeChoosableFileFilter() method and pass it the result of calling the getAcceptAllFileFilter(). The FileChooserDemo5.py script includes these calls, in case you are interested.

[5]See http://docs.oracle.com/javase/8/docs/api/javax/swing/filechooser/FileFilter.html.
[6]See http://docs.oracle.com/javase/8/docs/api/javax/swing/filechooser/FileNameExtens

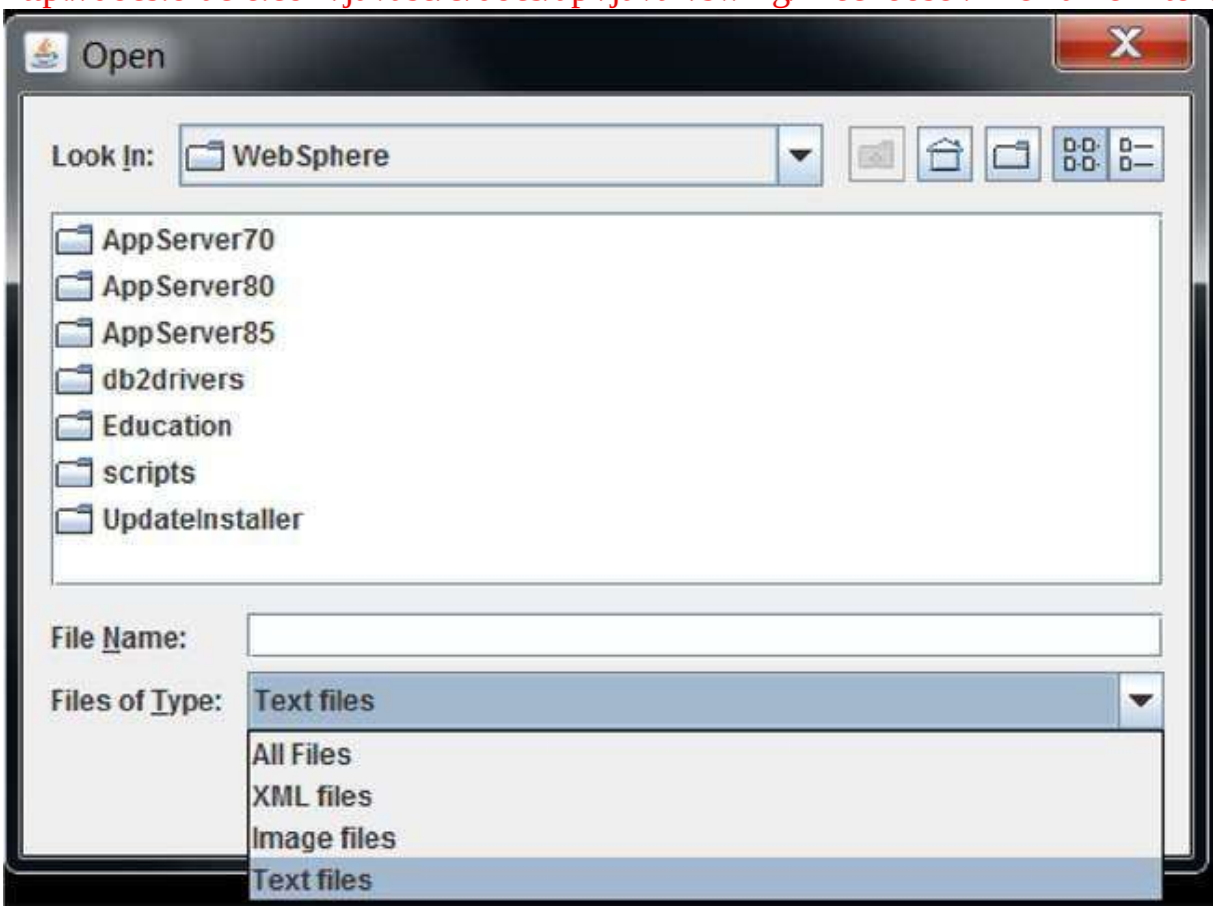


Figure 17-3. *Sample output from the FileChooserDemo4.py script*

Chooser Dialog Types

Up to now, all of the examples that you've seen have used the showOpenDialog() method to display the modal dialog box. In each of these, the primary button displayed the Open text. There are two other ways to display the dialog box. The first uses the showSaveDialog() method, which includes a Save button. This shouldn't be too much of a surprise. The second allows you to customize the text to be displayed on the primary button and uses the showDialog() method to do so. In each case, the application uses the return value

to determine which choice the user made so that the appropriate action can be taken. The possible return values are:

• JFileChooser.APPROVE_OPTION
• JFileChooser.CANCEL_OPTION
• JFileChooser.ERROR_OPTION

The last of these occurs only if some kind of error is encountered or if the dialog is somehow dismissed.
Selection Types

There are times when you'll want your users to be able to choose something other than a file. For these times, you need to tell the JFileChooser instance the kinds of selections that are acceptable. To do this, you can use the setFileSelectionMode() method or the fileSelectionMode attribute keyword on the constructor call. In either case, the values that can be specified are as follows:

• JFileChooser.FILES_ONLY
• JFileChooser.DIRECTORIES_ONLY
• JFileChooser.FILES_AND_DIRECTORIES

The default value is JFileChooser.FILES_ONLY, which shouldn't be much of a surprise. You might wonder why the primary button isn't disabled when this in effect. I was a little surprised about this myself, until I selected a directory and clicked on the Open button. The JFileChooser understood this to be the same as double-clicking on the directory, so I guess it does make some sense.

You might want to use the JFileChooser.DIRECTORIES_ONLY value for a chooser that allows the user to identify a source or destination directory for some operation (such as for copying files). There are a number of other JFileChooser methods as well. You might want to take a look at the Swing Tutorial page entitled "How to Use File Choosers"[7] if you are interested.

The JColorChooser Class

Another specialized dialog box that is included in the Swing hierarchy is the JColorChooser[8] class. It allows users to display a variety of techniques that can be used to select a color. A trivial script can be used to display a modal color