Ordering Information: **Advanced Java™ 2 Platform How to Program**

- View the complete **Table of Contents**
- Read the **Preface**
- Download the **Code Examples**

To view all the Deitel products and services available, visit the Deitel Kiosk on InformIT at **www.informIT.com/deitel**

To follow the Deitel publishing program, sign-up now for the *DEITEL™ BUZZ ONLINE* e-mail newsletter at **www.deitel.com/newsletter/subscribeinformIT.html** To learn more about Deitel instructor-led corporate training delivered at your location, visit **www.deitel.com/training** or contact Christi Kelsey at (978) 461-5880 or e-mail: **christi.kelsey@deitel.net**.

Note from the Authors: This article is an excerpt from Chapter 3, Section 3.6 of *Advanced Java™ 2 Platform How to Program*. Readers should have a familiarity with Java Swing GUI programming. This article is the second in a two-part series that overviews the Model-View-Controller architecture and Java's variation of it called the delegate model. The code examples included in this article show readers examples of applying the delegate model with the JTree GUI component. All of our articles are written using the Deitel signature *LIVE-CODE™ Approach*, which presents all concepts in the context of complete working programs followed by the screen shots of the actual inputs and outputs.

3.6.2 Custom `TreeModel` Implementation

If the **`DefaultTreeModel`** implementation is not sufficient for an application, developers also can provide custom implementations of interface **`TreeModel`**. **`FileSystemModel`** (Fig. 3.18) implements interface **`TreeModel`** to provide a model of a computer's file system. A file system consists of directories and files arranged in a hierarchy. Line 17 declares a **`File`** reference **`root`** that serves as the root node in the hierarchy. This **`File`** is a directory that contains files and other directories. The **`FileSystemModel`** constructor (lines 23–26) takes a **`File`** argument for the **`FileSystemModel`** root. Method **`getRoot`** (lines 29–32) returns the **`FileSystemModel`**'s root node.

When building its view of a **`TreeModel`**, a **`JTree`** repeatedly invokes method **`getChild`** (lines 35–46) to traverse the **`TreeModel`**'s nodes. Method **`getChild`** returns argument **`parent`**'s child node at the given **`index`**. The nodes in a **`TreeModel`** need not implement interface **`TreeNode`** or interface **`MutableTreeNode`**; any **`Object`** can be a node in a **`TreeModel`**. In class **`FileSystemModel`**, each node is a **`File`**. Line 38 casts **`Object`** reference **`parent`** to a **`File`** reference. Line 41 invokes method **`list`** of class **`File`** to get a list of file names in **`directory`**. Line 45 returns a new **`TreeFile`** object for the **`File`** at the given **`index`**. **`JTree`** invokes method **`toString`** of class **`TreeFile`** to get a label for the node in the **`JTree`**.

```
1  // FileSystemModel.java
2  // TreeModel implementation using File objects as tree nodes.
3  package com.deitel.advjhtp1.mvc.tree.filesystem;
4
5  // Java core packages
6  import java.io.*;
7  import java.util.*;
8
9  // Java extension packages
10 import javax.swing.*;
11 import javax.swing.tree.*;
12 import javax.swing.event.*;
13
14 public class FileSystemModel implements TreeModel {
15
16    // hierarchy root
17    private File root;
18
19    // TreeModelListeners
20    private Vector listeners = new Vector();
21
22    // FileSystemModel constructor
23    public FileSystemModel( File rootDirectory )
24    {
25      root = rootDirectory;
26    }
27
```

Fig. 3.18 **`FileSystemModel`** implementation of interface **`TreeModel`** to represent a file system (part 1 of 5).

```
28     // get hierarchy root (root directory)
29     public Object getRoot()
30     {
31        return root;
32     }
33
34     // get parent's child at given index
35     public Object getChild( Object parent, int index )
36     {
37        // get parent File object
38        File directory = ( File ) parent;
39
40        // get list of files in parent directory
41        String[] children = directory.list();
42
43        // return File at given index and override toString
44        // method to return only the File's name
45        return new TreeFile( directory, children[ index ] );
46     }
47
48     // get parent's number of children
49     public int getChildCount( Object parent )
50     {
51        // get parent File object
52        File file = ( File ) parent;
53
54        // get number of files in directory
55        if ( file.isDirectory() ) {
56
57           String[] fileList = file.list();
58
59           if ( fileList != null )
60              return file.list().length;
61        }
62
63        return 0; // childCount is 0 for files
64     }
65
66     // return true if node is a file, false if it is a directory
67     public boolean isLeaf( Object node )
68     {
69        File file = ( File ) node;
70        return file.isFile();
71     }
72
73     // get numeric index of given child node
74     public int getIndexOfChild( Object parent, Object child )
75     {
76        // get parent File object
77        File directory = ( File ) parent;
78
```

Fig. 3.18 **`FileSystemModel`** implementation of interface **`TreeModel`** to represent a file system (part 2 of 5).

```
79          // get child File object
80          File file = ( File ) child;
81
82          // get File list in directory
83          String[] children = directory.list();
84
85          // search File list for given child
86          for ( int i = 0; i < children.length; i++ ) {
87
88             if ( file.getName().equals( children[ i ] ) ) {
89
90                // return matching File's index
91                return i;
92             }
93          }
94
95          return -1; // indicate child index not found
96
97       } // end method getIndexOfChild
98
99       // invoked by delegate if value of Object at given
100      // TreePath changes
101      public void valueForPathChanged( TreePath path,
102         Object value )
103      {
104         // get File object that was changed
105         File oldFile = ( File ) path.getLastPathComponent();
106
107         // get parent directory of changed File
108         String fileParentPath = oldFile.getParent();
109
110         // get value of newFileName entered by user
111         String newFileName = ( String ) value;
112
113         // create File object with newFileName for
114         // renaming oldFile
115         File targetFile = new File(
116            fileParentPath, newFileName );
117
118         // rename oldFile to targetFile
119         oldFile.renameTo( targetFile );
120
121         // get File object for parent directory
122         File parent = new File( fileParentPath );
123
124         // create int array for renamed File's index
125         int[] changedChildrenIndices =
126            { getIndexOfChild( parent, targetFile) };
127
128         // create Object array containing only renamed File
129         Object[] changedChildren = { targetFile };
130
```

Fig. 3.18 **FileSystemModel** implementation of interface **TreeModel** to represent a file system (part 3 of 5).

```
131         // notify TreeModelListeners of node change
132         fireTreeNodesChanged( path.getParentPath(),
133            changedChildrenIndices, changedChildren );
134
135      } // end method valueForPathChanged
136
137      // notify TreeModelListeners that children of parent at
138      // given TreePath with given indices were changed
139      private void fireTreeNodesChanged( TreePath parentPath,
140         int[] indices, Object[] children )
141      {
142         // create TreeModelEvent to indicate node change
143         TreeModelEvent event = new TreeModelEvent( this,
144            parentPath, indices, children );
145
146         Iterator iterator = listeners.iterator();
147         TreeModelListener listener = null;
148
149         // send TreeModelEvent to each listener
150         while ( iterator.hasNext() ) {
151            listener = ( TreeModelListener ) iterator.next();
152            listener.treeNodesChanged( event );
153         }
154      } // end method fireTreeNodesChanged
155
156      // add given TreeModelListener
157      public void addTreeModelListener(
158         TreeModelListener listener )
159      {
160         listeners.add( listener );
161      }
162
163      // remove given TreeModelListener
164      public void removeTreeModelListener(
165         TreeModelListener listener )
166      {
167         listeners.remove( listener );
168      }
169
170      // TreeFile is a File subclass that overrides method
171      // toString to return only the File name.
172      private class TreeFile extends File {
173
174         // TreeFile constructor
175         public TreeFile( File parent, String child )
176         {
177            super( parent, child );
178         }
179
```

Fig. 3.18 **`FileSystemModel`** implementation of interface **`TreeModel`** to represent a file system (part 4 of 5).

```
180         // override method toString to return only the File name
181         // and not the full path
182         public String toString()
183         {
184            return getName();
185         }
186      } // end inner class TreeFile
187   }
```

Fig. 3.18 **`FileSystemModel`** implementation of interface **`TreeModel`** to represent a file system (part 5 of 5).

Method **`getChildCount`** (lines 49–64) returns the number of children contained in argument **`parent`**. Line 52 casts **`Object`** reference **`parent`** to a **`File`** reference named **`file`**. If **`file`** is a directory (line 55), lines 57–60 get a list of file names in the directory and return the **`length`** of the list. If **`file`** is not a directory, line 63 returns **`0`**, to indicate that **`file`** has no children.

A **`JTree`** invokes method **`isLeaf`** of class **`FileSystemModel`** (lines 67–71) to determine if **`Object`** argument **`node`** is a *leaf node*—a node that does not contain children.[3] In a file system, only directories can contain children, so line 70 returns **`true`** only if argument **`node`** is a file (not a directory).

Method **`getIndexOfChild`** (lines 74–98) returns argument **`child`**'s index in the given **`parent`** node. For example, if **`child`** were the third node in **`parent`**, method **`getIndexOfChild`** would return zero-based index **`2`**. Lines 77 and 80 get **`File`** references for the **`parent`** and **`child`** nodes, respectively. Line 83 gets a list of files, and lines 86–93 search through the list for the given **`child`**. If the filname in the list matches the given child (line 88), line 91 returns the index **`i`**. Otherwise, line 95 returns **`-1`**, to indicate that **`parent`** did not contain **`child`**.

The **`JTree`** delegate invokes method **`valueForPathChanged`** (lines 101–135) when the user edits a node in the tree. A user can click on a node in the **`JTree`** and edit the node's name, which corresponds to the associated **`File`** object's file name. When a user edits a node, **`JTree`** invokes method **`valueForPathChanged`** and passes a **`TreePath`** argument that represents the changed node's location in the tree, and an **`Object`** that contains the node's new value. In this example, the new value is a new file name **`String`** for the associated **`File`** object. Line 105 invokes method **`getLastPathComponent`** of class **`TreePath`** to obtain the **`File`** object to rename. Line 108 gets **`oldFile`**'s parent directory. Line 111 casts argument **`value`**, which contains the new file name, to a **`String`**. Lines 115–116 create **`File`** object **`targetFile`** using the new file name. Line 119 invokes method **`renameTo`** of class **`File`** to rename **`oldFile`** to **`targetFile`**.

After renaming the file, the **`FileSystemModel`** must notify its **`TreeModelListener`**s of the change by issuing a **`TreeModelEvent`**. A **`TreeModelEvent`** that indicates a node change includes a reference to the **`TreeModel`** that generated the event, the **`TreePath`** of the changed nodes' parent node, an integer array containing the changed nodes' indices and an **`Object`** array containing references to the changed nodes themselves. Line 122 creates a **`File`** object for the renamed file's parent directory. Lines 125–126 create an integer array for the indices of changed nodes. Line 128 creates an **`Object`**

3. Leaf node controls the initial screen display of the expand handle.

array of changed nodes. The integer and **Object** arrays have only one element each because only one node changed. If multiple nodes were changed, these arrays would need to include elements for each changed node. Lines 132–133 invoke method **fireTreeNodesChanged** to issue the **TreeModelEvent**.

Performance Tip 3.1

***JTree** uses the index and **Object** arrays in a **TreeModelEvent** to determine which nodes in the **JTree** need to be updated. This method improves performance by updating only the nodes that have changed, and not the entire **JTree**.*

Method **fireTreeNodesChanged** (lines 139–154) issues a **TreeModelEvent** to all registered **TreeModelListener**s, indicating that nodes in the **TreeModel** have changed. **TreePath** argument **parentPath** is the path to the parent whose child nodes changed. The integer and **Object** array arguments contain the indices of the changed nodes and references to the changed nodes, respectively. Lines 143–144 create the **TreeModel** event with the given event data. Lines 150–153 iterate through the list of **TreeModelListener**s, sending the **TreeModelEvent** to each. Methods **addTreeModelListener** (lines 157–161) and **removeTreeModelListener** (lines 164–168) allow **TreeModelListener**s to register and unregister for **TreeModelEvent**s.

Inner-class **TreeFile** (lines 172–186) overrides method **toString** of superclass **File**. Method **toString** of class **File** returns a **String** containing the **File**'s full path name (e.g., **D:\Temp\README.TXT**). Method **toString** of class **TreeFile** (lines 182–185) overrides this method to return only the **File**'s name (e.g., **README.TXT**). Class **JTree** uses a **DefaultTreeCellRenderer** to display each node in its **TreeModel**. The **DefaultTreeCellRenderer** invokes the node's **toString** method to get the text for the **DefaultTreeCellRenderer**'s label. Class **TreeFile** overrides method **toString** of class **File** so the **DefaultTreeCellRenderer** will show only the **File**'s name in the **JTree**, instead of the full path.

FileTreeFrame (Fig. 3.19) uses a **JTree** and a **FileSystemModel** to allow the user to view and modify a file system. The user interface consists of a **JTree** that shows the file system and a **JTextArea** that shows information about the currently selected file. Lines 33–34 create the uneditable **JTextArea** for displaying file information. Lines 37–38 create a **FileSystemModel** whose root is **directory**. Line 41 creates a **JTree** for the **FileSystemModel**. Line 44 sets the **JTree**'s editable property to **true**, to allow users to rename files displayed in the **JTree**.

```
1  // FileTreeFrame.java
2  // JFrame for displaying file system contents in a JTree
3  // using a custom TreeModel.
4  package com.deitel.advjhtp1.mvc.tree.filesystem;
5
6  // Java core packages
7  import java.io.*;
8  import java.awt.*;
9  import java.awt.event.*;
10
```

Fig. 3.19 **FileTreeFrame** application for browsing and editing a file system using **JTree** and **FileSystemModel** (part 1 of 4).

```
11  // Java extension packages
12  import javax.swing.*;
13  import javax.swing.tree.*;
14  import javax.swing.event.*;
15
16  public class FileTreeFrame extends JFrame {
17
18     // JTree for displaying file system
19     private JTree fileTree;
20
21     // FileSystemModel TreeModel implementation
22     private FileSystemModel fileSystemModel;
23
24     // JTextArea for displaying selected file's details
25     private JTextArea fileDetailsTextArea;
26
27     // FileTreeFrame constructor
28     public FileTreeFrame( String directory )
29     {
30        super( "JTree FileSystem Viewer" );
31
32        // create JTextArea for displaying File information
33        fileDetailsTextArea = new JTextArea();
34        fileDetailsTextArea.setEditable( false );
35
36        // create FileSystemModel for given directory
37        fileSystemModel = new FileSystemModel(
38           new File( directory ) );
39
40        // create JTree for FileSystemModel
41        fileTree = new JTree( fileSystemModel );
42
43        // make JTree editable for renaming Files
44        fileTree.setEditable( true );
45
46        // add a TreeSelectionListener
47        fileTree.addTreeSelectionListener(
48           new TreeSelectionListener() {
49
50              // display details of newly selected File when
51              // selection changes
52              public void valueChanged(
53                 TreeSelectionEvent event )
54              {
55                 File file = ( File )
56                    fileTree.getLastSelectedPathComponent();
57
58                 fileDetailsTextArea.setText(
59                    getFileDetails( file ) );
60              }
61           }
62        ); // end addTreeSelectionListener
```

Fig. 3.19 **FileTreeFrame** application for browsing and editing a file system using **JTree** and **FileSystemModel** (part 2 of 4).

```
63
64          // put fileTree and fileDetailsTextArea in a JSplitPane
65          JSplitPane splitPane = new JSplitPane(
66             JSplitPane.HORIZONTAL_SPLIT, true,
67             new JScrollPane( fileTree ),
68             new JScrollPane( fileDetailsTextArea ) );
69
70          getContentPane().add( splitPane );
71
72          setDefaultCloseOperation( EXIT_ON_CLOSE );
73          setSize( 640, 480 );
74          setVisible( true );
75       }
76
77       // build a String to display file details
78       private String getFileDetails( File file )
79       {
80          // do not return details for null Files
81          if ( file == null )
82             return "";
83
84          // put File information in a StringBuffer
85          StringBuffer buffer = new StringBuffer();
86          buffer.append( "Name: " + file.getName() + "\n" );
87          buffer.append( "Path: " + file.getPath() + "\n" );
88          buffer.append( "Size: " + file.length() + "\n" );
89
90          return buffer.toString();
91       }
92
93       // execute application
94       public static void main( String args[] )
95       {
96          // ensure that user provided directory name
97          if ( args.length != 1 )
98             System.err.println(
99                "Usage: java FileTreeFrame <path>" );
100
101         // start application using provided directory name
102         else
103            new FileTreeFrame( args[ 0 ] );
104      }
105   }
```

Fig. 3.19 **FileTreeFrame** application for browsing and editing a file system using **JTree** and **FileSystemModel** (part 3 of 4).

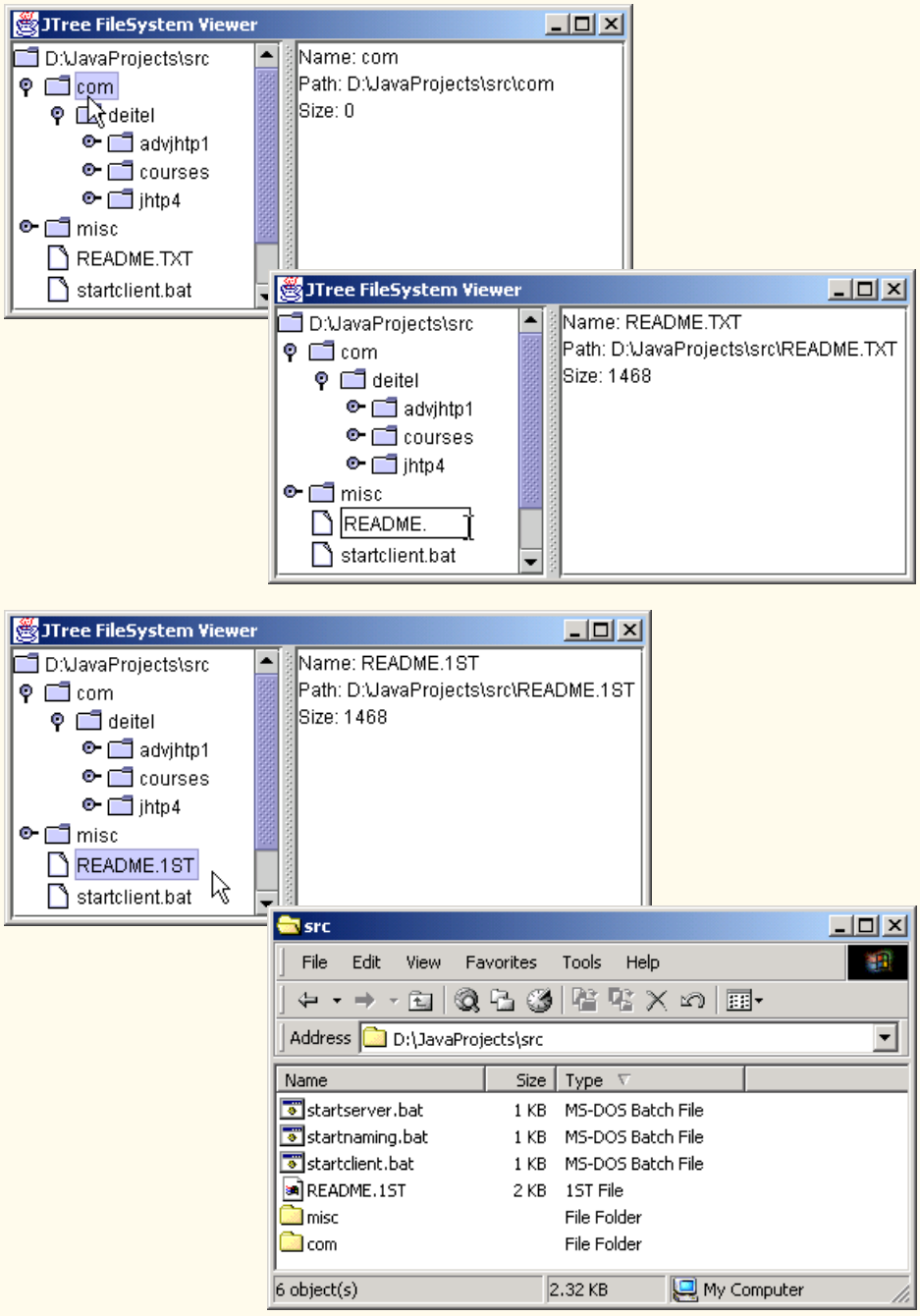

Fig. 3.19 **`FileTreeFrame`** application for browsing and editing a file system using **`JTree`** and **`FileSystemModel`** (part 4 of 4).

Lines 47–62 create a **`TreeSelectionListener`** to listen for **`TreeSelectionEvent`**s in the **`JTree`**. Lines 55–56 of method **`valueChanged`** get the selected **`File`** object from the **`JTree`**. Lines 58–59 invoke method **`getFileDetails`** to retrieve information about the selected **`File`** and to display the details in **`fileDetailsTextArea`**. Lines 65–69 create a **`JSplitPane`** to separate the **`JTree`** and **`JTextArea`**. Lines 67 and 68 place the **`JTree`** and **`JTextArea`** in **`JScrollPane`**s. Line 70 adds the **`JSplitPane`** to the **`JFrame`**.

Method **`getFileDetails`** (lines 78–91) takes a **`File`** argument and returns a **`String`** containing the **`File`**'s name, path and length. If the **`File`** argument is **`null`**, line 81 returns an empty **`String`**. Line 85 creates a **`StringBuffer`**, and lines 86–88 append the **`File`**'s name, path and length. Line 90 invokes method **`toString`** of class **`StringBuffer`** and returns the result to the caller.

Method **`main`** (lines 94–104) executes the **`FileTreeFrame`** application. Lines 97–99 check the command-line arguments to ensure that the user provided a path for the **`FileTreeModel`**'s root. If the user did not provide a command-line argument, lines 98–99 display the program's usage instructions. Otherwise, line 103 creates a new **`FileTreeFrame`** and passes the command-line argument to the constructor.

In this chapter, we introduced the model-view-controller architecture, the Observer design pattern and the delegate-model architecture used by several Swing components. In later chapters, we use MVC to build a Java2D paint program (Chapter 6), database-aware programs (Chapter 8, JDBC) and an Enterprise Java case study (Chapters 16–19).