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, Sections 3.2 and 3.6 of *Advanced Java™ 2 Platform How to Program*. Readers should have a familiarity with Java Swing GUI programming. This article is the first 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.

Model-View-Controller

3.2 Model-View-Controller Architecture

The *model-view-controller* architecture (MVC) separates application data (contained in the *model*) from graphical presentation components (the *view*) and input-processing logic (the *controller*). MVC originally appeared in Smalltalk-80 as a method for separating user interfaces from underlying application data.[1] Figure 3.1 shows the relationships between components in MVC. In our Enterprise Java case study (Chapters 17–20), we will show that MVC is applicable across a wide range of problems and can make applications easier to maintain and extend.

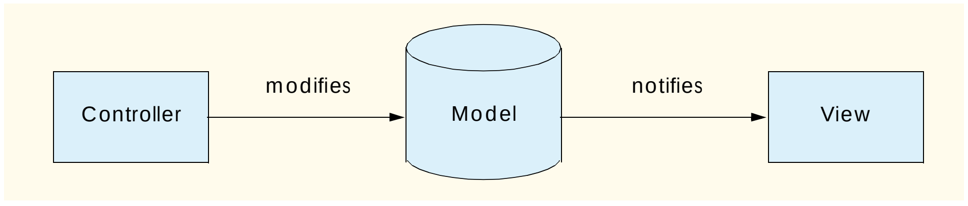


Fig. 3.1 Model-view-controller architecture.

The controller implements logic for processing user input. The model contains application data, and the view generates a presentation of the data stored in the model. When a user provides some input (e.g., by typing text in a word processor,) the controller modifies the model with the given input. It is important to note that the model contains only the raw application data. In a simple text editor, the model might contain only the characters that make up the document. When the model changes, it notifies the view of the change, so that the view can update its presentation with the changed data. The view in a word processor might display the characters on the screen in a particular font, with a particular size, etc.

MVC does not restrict an application to a single view and controller. In a word processor, for example, there might be two views of a single document model. One view might display the document as an outline, and the other might display the document in a print-preview window. The word processor also may implement multiple controllers, such as a controller for handling keyboard input and a controller for handling mouse selections. If either controller makes a change in the model, both the outline view and the print-preview window show the change immediately, because the model notifies all views of any changes. A developer can provide additional views and controllers for the model without changing the existing components.

1. E. Gamma et al., *Design Patterns* (New York: Addison-Wesley Publishing Company, 1995), 4.

© Copyright 2002 by Prentice Hall. All Rights Reserved.

Java’s Swing components implement a variation of MVC that combines the view and controller into a single object, called a *delegate* (Fig. 3.2). The delegate provides both a graphical presentation of the model and an interface for modifying the model. For example, every **JButton** has an associated **ButtonModel** for which the **JButton** is a delegate. The **ButtonModel** maintains state information, such as whether the **JButton** is pressed and whether the **JButton** is enabled, as well as a list of **ActionListener**s. The **JButton** provides a graphical presentation (e.g., a rectangle on the screen with a label and a border) and modifies the **ButtonModel**’s state (e.g., when the user presses the **JButton**). We discuss several Swing components that implement the delegate-model architecture throughout this chapter.

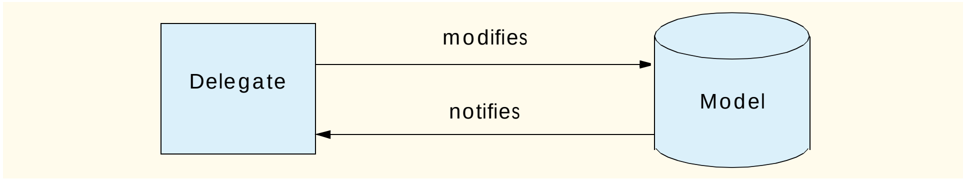


Fig. 3.2 Delegate-model architecture in Java Swing components.

3.6 JTree

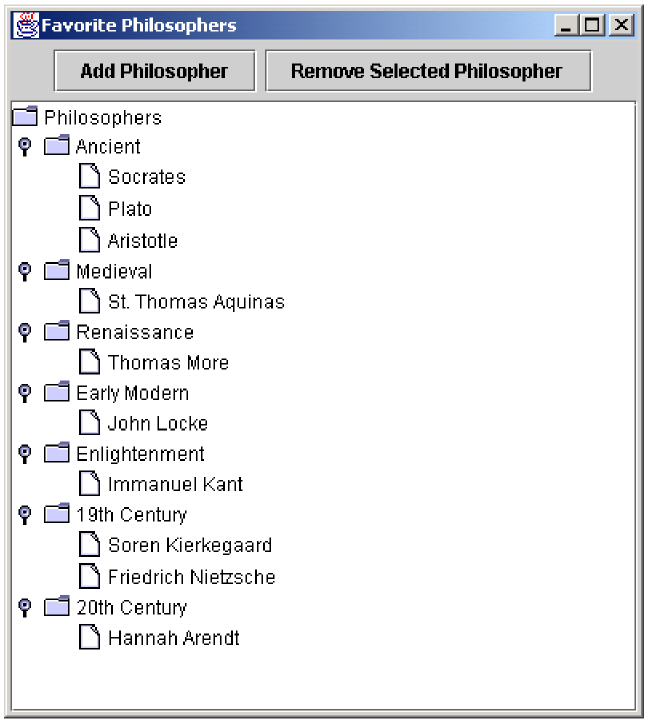
JTree is one of the more complex Swing components that implements the delegate-model architecture. **TreeModel**s represent hierarchical data, such as family trees, certain types of file systems, company management structures and document outlines. **JTree**s act as delegates (i.e., combined view and controller) for **TreeModel**s.

To describe tree data structures, it is common to use terms that more commonly describe family trees.[2] A tree data structure consists of a set of nodes (i.e., members or elements of the tree) that are related as *parents*, *children*, *siblings*, *ancestors* and *descendents*. A parent is a node that has other nodes as its children. A child is a node that has a parent. Sibling nodes are two or more nodes that share the same parent. An ancestor is a node that has children that also have children. A descendent is a node whose parent also has a parent. A tree must have one node—called the *root node*—that is the parent or ancestor of all other nodes in the tree. [*Note*: Unlike in a family tree, in a tree data structure a child node can have only one parent.]

Figure 3.16 shows the relationships among nodes in a tree. The **JTree** contains a hierarchy of philosophers whose root is node **Philosophers**. Node **Philosophers** has seven child nodes, representing the major eras of philosophy—**Ancient**, **Medieval**, **Renaissance**, **Early Modern**, **Enlightenment**, **19th Century** and **20th Century**. Each philosopher (e.g., **Socrates**, **St. Thomas Aquinas** and **Immanuel Kant**) is a child of the philosopher’s era and is a descendent of node **Philosophers**. Nodes **Socrates**, **Plato** and **Aristotle** are sibling nodes, because they share the same parent node (**Ancient**).

2. Note that nodes in the tree data structures we discuss in this section each have only a single parent, unlike a family tree.

© Copyright 2002 by Prentice Hall. All Rights Reserved.



Philosophers
- Ancient
  - Socrates
  - Plato
  - Aristotle
- Medieval
  - St. Thomas Aquinas
- Renaissance
  - Thomas More
- Early Modern
  - John Locke
- Enlightenment
  - Immanuel Kant
- 19th Century
  - Soren Kierkegaard
  - Friedrich Nietzsche
- 20th Century
  - Hannah Arendt

Fig. 3.16 **`JTree`** showing a hierarchy of philosophers.

3.6.1 Using `DefaultTreeModel`

Interface ***`TreeModel`*** declares methods for representing a tree data structure in a **`JTree`**. Objects of any class can represent nodes in a **`TreeModel`**. For example, a **`Person`** class could represent a node in a family tree **`TreeModel`**. Class ***`DefaultTreeModel`*** provides a default **`TreeModel`** implementation. Interface **`TreeNode`** defines common operations for nodes in a **`DefaultTreeModel`**, such as ***`getParent`*** and ***`getAllowsChildren`***. Interface ***`MutableTreeNode`*** extends interface **`TreeNode`** to represent a node that can change, either by adding or removing child nodes or by changing the **`Object`** associated with the node. Class ***`DefaultMutableTreeNode`*** provides a **`MutableTreeNode`** implementation suitable for use in a **`DefaultTreeModel`**.

Software Engineering Observation 3.1

*Although a **`TreeModel`** implementation can use objects of any class to represent the **`TreeModel`**'s nodes, the **`TreeModel`** implementation must be able to determine the hierarchical relationships among those objects. For example, a **`Person`** class would have to provide methods such as **`getParent`** and **`getChildren`** for use in a family tree **`TreeModel`**.*

© Copyright 2002 by Prentice Hall. All Rights Reserved.

JTree employs two interfaces to implement the **JTree**'s delegate functionality. Interface **TreeCellRenderer** represents an object that creates a view for each node in the **JTree**. Class **DefaultTreeCellRenderer** implements interface **TreeCellRenderer** and extends class **JLabel** to provide a **TreeCellRenderer** default implementation. Interface **TreeCellEditor** represents an object for controlling (i.e., editing) each node in the **JTree**. Class **DefaultTreeCellEditor** implements interface **TreeCellEditor** and uses a **JTextField** for the **TreeCellEditor** default implementation.

PhilosophersJTree (Fig. 3.17) uses a **DefaultTreeModel** to represent a set of philosophers. The **DefaultTreeModel** organizes the philosophers hierarchically according to their associated eras in the history of philosophy. Lines 26–27 invoke method **createPhilosopherTree** to get the root, **DefaultMutableTreeNode**, which contains all the philosopher nodes. Line 30 creates a **DefaultTreeModel** and passes the **philosophersNode DefaultMutableTreeNode** to the **DefaultTreeModel** constructor. Line 33 creates a **JTree** and passes **DefaultTreeModel philosophers** to the **JTree** constructor.

```
1   // PhilosophersJTree.java
2   // MVC architecture using JTree with a DefaultTreeModel
3   package com.deitel.advjhtp1.mvc.tree;
4
5   // Java core packages
6   import java.awt.*;
7   import java.awt.event.*;
8   import java.util.*;
9
10  // Java extension packages
11  import javax.swing.*;
12  import javax.swing.tree.*;
13
14  public class PhilosophersJTree extends JFrame {
15
16     private JTree tree;
17     private DefaultTreeModel philosophers;
18     private DefaultMutableTreeNode rootNode;
19
20     // PhilosophersJTree constructor
21     public PhilosophersJTree()
22     {
23        super( "Favorite Philosophers" );
24
25        // get tree of philosopher DefaultMutableTreeNodes
26        DefaultMutableTreeNode philosophersNode =
27           createPhilosopherTree();
28
29        // create philosophers DefaultTreeModel
30        philosophers = new DefaultTreeModel( philosophersNode );
31
```

Fig. 3.17 **PhilosophersJTree** application demonstrating **JTree** and **DefaultTreeModel** (part 1 of 6).

© Copyright 2002 by Prentice Hall. All Rights Reserved.

```
      // create JTree for philosophers DefaultTreeModel
      tree = new JTree( philosophers );

      // create JButton for adding philosophers
      JButton addButton = new JButton( "Add" );
      addButton.addActionListener(
         new ActionListener() {

            public void actionPerformed( ActionEvent event )
            {
               addElement();
            }
         }
      );

      // create JButton for removing selected philosopher
      JButton removeButton =
         new JButton( "Remove" );

      removeButton.addActionListener(
         new ActionListener() {

            public void actionPerformed( ActionEvent event )
            {
               removeElement();
            }
         }
      );

      // lay out GUI components
      JPanel inputPanel = new JPanel();
      inputPanel.add( addButton );
      inputPanel.add( removeButton );

      Container container = getContentPane();

      container.add( new JScrollPane( tree ),
         BorderLayout.CENTER );

      container.add( inputPanel, BorderLayout.NORTH );

      setDefaultCloseOperation( EXIT_ON_CLOSE );
      setSize( 400, 300 );
      setVisible( true );

   } // end PhilosophersJTree constructor

   // add new philosopher to selected era
   private void addElement()
   {
      // get selected era
      DefaultMutableTreeNode parent = getSelectedNode();
```

Fig. 3.17 **PhilosophersJTree** application demonstrating **JTree** and **DefaultTreeModel** (part 2 of 6).

© Copyright 2002 by Prentice Hall. All Rights Reserved.

```java

      // ensure user selected era first
      if ( parent == null ) {
         JOptionPane.showMessageDialog(
            PhilosophersJTree.this, "Select an era.",
            "Error", JOptionPane.ERROR_MESSAGE );

         return;
      }

      // prompt user for philosopher's name
      String name = JOptionPane.showInputDialog(
         PhilosophersJTree.this, "Enter Name:" );

      // add new philosopher to selected era
      philosophers.insertNodeInto(
         new DefaultMutableTreeNode( name ),
         parent, parent.getChildCount() );

   } // end method addElement

   // remove currently selected philosopher
   private void removeElement()
   {
      // get selected node
      DefaultMutableTreeNode selectedNode = getSelectedNode();

      // remove selectedNode from model
      if ( selectedNode != null )
         philosophers.removeNodeFromParent( selectedNode );
   }

   // get currently selected node
   private DefaultMutableTreeNode getSelectedNode()
   {
      // get selected DefaultMutableTreeNode
      return ( DefaultMutableTreeNode )
         tree.getLastSelectedPathComponent();
   }

   // get tree of philosopher DefaultMutableTreeNodes
   private DefaultMutableTreeNode createPhilosopherTree()
   {
      // create rootNode
      DefaultMutableTreeNode rootNode =
         new DefaultMutableTreeNode( "Philosophers" );

      // Ancient philosophers
      DefaultMutableTreeNode ancient =
         new DefaultMutableTreeNode( "Ancient" );
      rootNode.add( ancient );

```

Fig. 3.17 **`PhilosophersJTree`** application demonstrating **`JTree`** and **`DefaultTreeModel`** (part 3 of 6).

© Copyright 2002 by Prentice Hall. All Rights Reserved.

```
      ancient.add( new DefaultMutableTreeNode( "Socrates" ) );
      ancient.add( new DefaultMutableTreeNode( "Plato" ) );
      ancient.add( new DefaultMutableTreeNode( "Aristotle" ) );

      // Medieval philosophers
      DefaultMutableTreeNode medieval =
         new DefaultMutableTreeNode( "Medieval" );
      rootNode.add( medieval );

      medieval.add( new DefaultMutableTreeNode(
         "St. Thomas Aquinas" ) );

      // Renaissance philosophers
      DefaultMutableTreeNode renaissance =
         new DefaultMutableTreeNode( "Renaissance" );
      rootNode.add( renaissance );

      renaissance.add( new DefaultMutableTreeNode(
         "Thomas More" ) );

      // Early Modern philosophers
      DefaultMutableTreeNode earlyModern =
         new DefaultMutableTreeNode( "Early Modern" );
      rootNode.add( earlyModern );

      earlyModern.add( new DefaultMutableTreeNode(
         "John Locke" ) );

      // Enlightenment Philosophers
      DefaultMutableTreeNode enlightenment =
         new DefaultMutableTreeNode( "Enlightenment" );
      rootNode.add( enlightenment );

      enlightenment.add( new DefaultMutableTreeNode(
         "Immanuel Kant" ) );

      // 19th Century Philosophers
      DefaultMutableTreeNode nineteenth =
         new DefaultMutableTreeNode( "19th Century" );
      rootNode.add( nineteenth );

      nineteenth.add( new DefaultMutableTreeNode(
         "Soren Kierkegaard" ) );

      nineteenth.add( new DefaultMutableTreeNode(
         "Friedrich Nietzsche" ) );

      // 20th Century Philosophers
      DefaultMutableTreeNode twentieth =
         new DefaultMutableTreeNode( "20th Century" );
      rootNode.add( twentieth );

```

Fig. 3.17 **PhilosophersJTree** application demonstrating **JTree** and **DefaultTreeModel** (part 4 of 6).

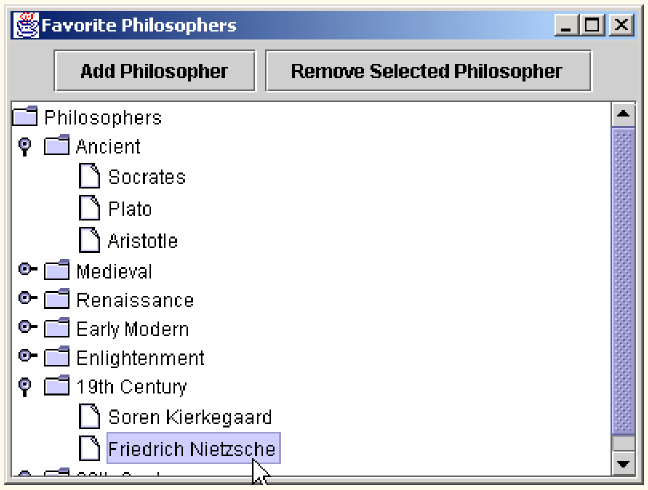
© Copyright 2002 by Prentice Hall. All Rights Reserved.

```
         twentieth.add( new DefaultMutableTreeNode(
            "Hannah Arendt" ) );

         return rootNode;

      } // end method createPhilosopherTree

      // execute application
      public static void main( String args[] )
      {
         new PhilosophersJTree();
      }
   }
```



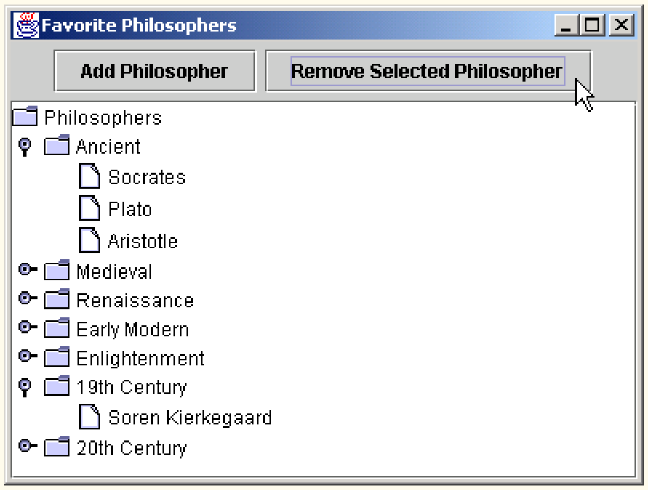


Fig. 3.17 **`PhilosophersJTree`** application demonstrating **`JTree`** and **`DefaultTreeModel`** (part 5 of 6).

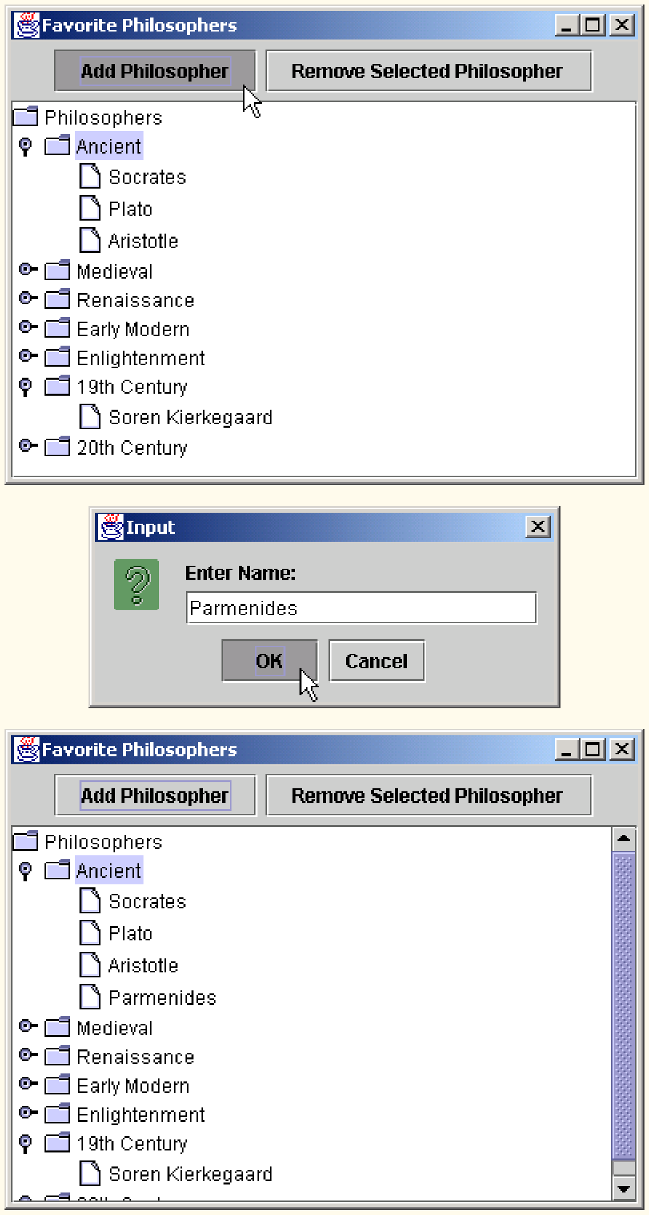


Fig. 3.17 **`PhilosophersJTree`** application demonstrating **`JTree`** and **`DefaultTreeModel`** (part 6 of 6).

Lines 36–45 create a **`JButton`** and an **`ActionListener`** for adding a philosopher to the **`philosophers DefaultTreeModel`**. Line 42 in method **`actionPer-formed`** invokes method **`addElement`** to add a new philosopher. Lines 48–59 create a

© Copyright 2002 by Prentice Hall. All Rights Reserved.

JButton and an **ActionListener** for removing a philosopher from the **philosophers** **DefaultTreeModel**. Line 56 invokes method **removeElement** to remove the currently selected philosopher from the model.

Method **addElement** (lines 80–103) gets the currently selected node in the **JTree** by invoking method **getSelectedNode** (line 83). Method **addElement** inserts the new philosopher node as a child of the currently selected node. If there is no node currently selected, line 91 returns from method **addElement** without adding a new node. Lines 95–96 invoke **static** method **showInputDialog** of class **JOptionPane** to prompt the user for the new philosopher's name. Lines 99–101 invoke method **insertNodeInto** of class **DefaultTreeModel** to insert the new philosopher in the model. Line 100 creates a new **DefaultMutableTreeNode** for the given philosopher. Line 101 specifies the parent node to which the new philosopher should be added. The final argument to method **insertNodeInto** specifies the index at which the new node should be inserted. Line 101 invokes method **getChildCount** of class **DefaultMutableTreeNode** to get the total number of children in node **parent**, which will cause the new node to be added as the last child of **parent**.

Method **removeElement** (lines 106–114) invokes method **getSelectedNode** (line 109) to get the currently selected node in the **JTree**. If **selectedNode** is not **null**, line 113 invokes method **removeNodeFromParent** of class **DefaultTreeModel** to remove **selectedNode** from the model. Method **getSelectedNode** (lines 117–122) invokes method **getLastSelectedPathComponent** of class **JTree** to get a reference to the currently selected node (line 121). Line 120 casts the selected node to **DefaultMutableTreeNode** and returns the reference to the caller.

Method **createPhilosopherTree** (lines 125–192) creates **DefaultMutableTreeNode**s for several philosophers and for the eras in which the philosophers lived. Lines 128–129 create a **DefaultMutableTreeNode** for the tree's root. Class **DefaultMutableTreeNode** has property **userObject** that stores an **Object** that contains the node's data. The **String** passed to the **DefaultMutableTreeNode** constructor (line 129) is the **userObject** for **rootNode**. The **JTree**'s **TreeCellRenderer** will invoke method **toString** of class **DefaultMutableTreeNode** to get a **String** to display for this node in the **JTree**.

Software Engineering Observation 3.2

*Method **toString** of class **DefaultMutableTreeNode** returns the value returned by its **userObject**'s **toString** method.*

Lines 132–134 create a **DefaultMutableTreeNode** for the **ancient** era of philosophy and add node **ancient** as a child of **rootNode** (line 134). Lines 136–138 create **DefaultMutableTreeNodes** for three ancient philosophers and add each **DefaultMutableTreeNode** as a child of **DefaultMutableTreeNode** **ancient**. Lines 141–189 create several additional **DefaultMutableTreeNodes** for other eras in the history of philosophy and for philosophers in those eras. Line 191 returns **rootNode**, which now contains the era and philosopher **DefaultMutableTreeNodes** as its children and descendents, respectively.