Chapter 3 – Model-View-Controller

Outline

3.1 Introduction

- MVC
  - Model-view-controller architecture
    - Data components
    - Presentation components
    - Input-processing components
- Delegate-model architecture
- Observer design pattern

3.2 Model-View-Controller Architecture

- Model
  - Application data
- View
  - Graphical presentation components
- Controller
  - Logic for processing user input

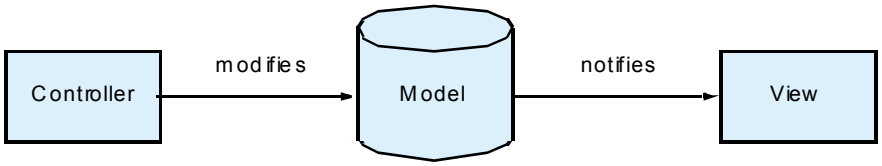


Fig. 3.1 Model-view-controller architecture.

3.2 Model-View-Controller Architecture (Cont.)

- Delegate-model architecture
  - Variant of MVC
  - Combines the view and controller into a single object

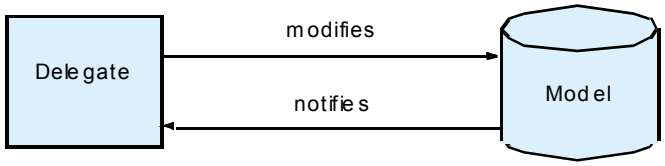


Fig. 3.2 Delegate-model architecture in Java Swing components.

3.3 Observable Class and Observer Interface

- Observer design pattern
  - Loose coupling
- Java implementation of observer design pattern
  - Class java.util.Observable
  - Interface Observer
- Example

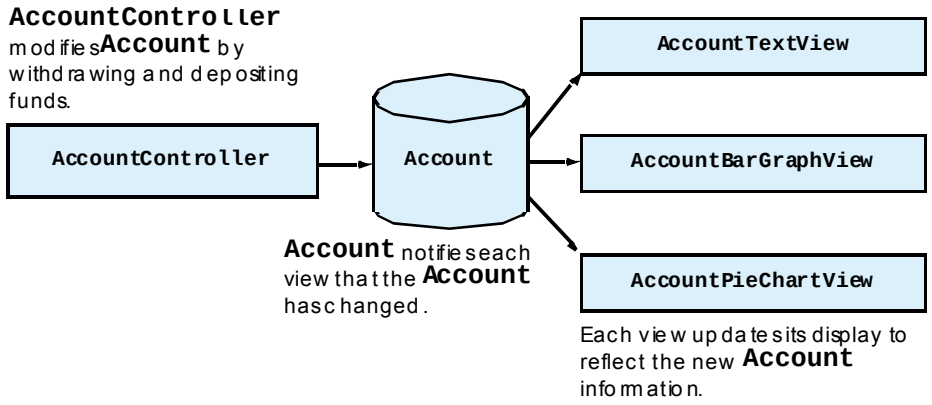
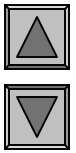


Fig. 3.3 **AccountManager** application MVC architecture.



Outline

Fig. 3.4 Account Observable class that represents a bank account.

Line 9

Lines 18-22

Lines 25-35

Line 31

Line 34

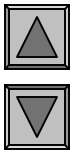
```
1  // Account.java
2  // Account is an Observable class that represents a bank
3  // account in which funds may be deposited or withdrawn.
4  package com.deitel.advjhtp1.mvc.account;
5
6  // Java core packages
7  import java.util.Observable;
8
9  public class Account extends Observable {
10
11    // Account balance
12    private double balance;
13
14    // readonly Account name
15    private String name;
16
17    // Account constructor
18    public Account( String accountName, double openingDeposit )
19    {
20       name = accountName;
21       setBalance( openingDeposit );
22    }
23
24    // set Account balance and notify observers of change
25    private void setBalance( double accountBalance )
26    {
27       balance = accountBalance;
28
29       // must call setChanged before notifyObservers
30       // indicate model has c
31       setChanged();
32
33       // notify Observers that model has changed
34       notifyObservers();
35    }
```

Class **Account** extends class **Observable** and acts as a model in the application.

Initialize the **name** and **balance** properties.

Method **setBalance** changes the model by updating the account balance.

Invokes method **notifyObservers** of class **Observable** to notify all **Account Obserbers** of the change.



Outline

```
36
37       // get Account balance
38       public double getBalance()
39       {
40          return balance;
41       }
42
43       // withdraw funds from Account
44       public void withdraw( double amount )
45          throws IllegalArgumentException
46       {
47          if ( amount < 0 )
48             throw new IllegalArgumentException(
49                "Cannot withdraw negative amount" );
50
51          // update Account balance
52          setBalance( getBalance() - amount );
53       }
54
55       // deposit funds in account
56       public void deposit( double amount )
57          throws IllegalArgumentException
58       {
59          if ( amount < 0 )
60             throw new IllegalArgumentException(
61                "Cannot deposit negative amount" );
62
63          // update Account balance
64          setBalance( getBalance() + amount );
65       }
```

Return the current **Account** balance.

Method **withdraw** subtracts the given **amount** from the **Account balance**.

Method **deposit** adds the **amount** input to the **Account balance**.

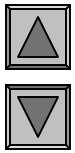
Fig. 3.4 Account Observable class ...epresents account.

Lines 38-41

Lines 44-53

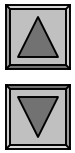
Lines 56-65

```
66
67       // get Account name (readonly)
68       public String getName()
69       {
70          return name;
71       }
72    }
```



Outline

Fig. 3.4 Account Observable class that represents a bank account.



Outline

Fig. 3.5 AbstractAccountView abstract base class for observing Accounts.

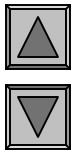
Lines 21-37

Line 32

```
1  // AbstractAccountView.java
2  // AbstractAccountView is an abstract class that represents
3  // a view of an Account.
4  package com.deitel.advjhtp1.mvc.account;
5
6  // Java core packages
7  import java.util.*;
8  import java.awt.*;
9
10 // Java extension packages
11 import javax.swing.JPanel;
12 import javax.swing.border.*;
13
14 public abstract class AbstractAccountView extends JPanel
15    implements Observer {
16
17    // Account to observe
18    private Account account;
19
20    // AbstractAccountView constructor
21    public AbstractAccountView( Account observableAccount )
22       throws NullPointerException
23    {
24       // do not allow null Accounts
25       if ( observableAccount == null )
26          throw new NullPointerException();
27
28       // update account data member to new Accour
29       account = observableAccount;
30
31       // register as an Observer to receive accou
32       account.addObserver( this );
33
```

Constructor sets the **account** member variable to the new **Account**.

Invokes method **addObserver** of class **Observable** to register the newly created **AbstractAccountView** instance as an **Observer** of the new **Account**.



Outline

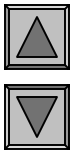
Fig. 3.5 AbstractAccountV

```
34          // set display properties
35          setBackground( Color.white );
36          setBorder( new MatteBorder( 1, 1, 1, 1, Color.black ) );
37       }
38
39       // get Account for which this view is an Observer
40       public Account getAccount()
41       {
42          return account;
43       }
44
45       // update display with Account balance
46       protected abstract void updateDisplay();
47
48       // receive updates from Observable Account
49       public void update( Observable observable, Object object )
50       {
51          updateDisplay();
52       }
53    }
```

Method **updateDisplay** is marked abstract, requiring each **AbstractAccountView** subclass to provide an appropriate implementation for displaying the **Account** information.

Lines 49–52

Method **update** invokes method **updateDisplay** each time an **Account** notifies the **AbstractAccountView** of a change.



Outline

Fig. 3.6 AccountTextView for displaying observed Account information in JTextField.

```
1  // AccountTextView.java
2  // AccountTextView is an AbstractAccountView subclass
3  // that displays an Account balance in a JTextField.
4  package com.deitel.advjhtp1.mvc.account;
5
6  // Java core packages
7  import java.util.*;
8  import java.text.NumberFormat;
9
10 // Java extension packages
11 import javax.swing.*;
12
13 public class AccountTextView extends AbstractAccountView {
14
15    // JTextField for displaying Account balance
16    private JTextField balanceTextField = new
17
18    // NumberFormat for US dollars
19    private NumberFormat moneyFormat =
20       NumberFormat.getCurrencyInstance( Locale.US );
21
22    // AccountTextView constructor
23    public AccountTextView( Account account )
24    {
25       super( account );
26
27       // make balanceTextField readonly
28       balanceTextField.setEditable( false );
29
30       // lay out components
31       add( new JLabel( "Balance: " ) );
32       add( balanceTextField );
33
34       updateDisplay();
35    }
```

Line 13

Extends **AbstractAccountView**

Create a **NumberFormat** field to format the **Account** balance as U.S. dollars.

Line 28

Makes the **balanceTextField** uneditable to prevent users from modifying the balance directly.

```
36
37       // update display with Account balance
38       public void updateDisplay()
39       {
40          // set text in balanceTextField to formatted balance
41          balanceTextField.setText( moneyFormat.format(
42             getAccount().getBalance() ) );
43       }
44    }
```

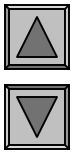
Method **updateDisplay** implements abstract method **updateDisplay** of class **AbstractAccountView**.

Set the **balanceTextField**'s text to the formatted **Account** balance.

Fig. 3.6
AccountTextView
JTextField.

Lines 38-43

Lines 41-42



Outline

Fig. 3.7 AccountBarGraphView for rendering observed Account information as a bar graph.

Lines 21-57

```
1  // AccountBarGraphView.java
2  // AccountBarGraphView is an AbstractAccountView subclass
3  // that displays an Account balance as a bar graph.
4  package com.deitel.advjhtp1.mvc.account;
5
6  // Java core packages
7  import java.awt.*;
8
9  // Java extension packages
10 import javax.swing.*;
11
12 public class AccountBarGraphView extends AbstractAccountView {
13
14    // AccountBarGraphView constructor
15    public AccountBarGraphView( Account account )
16    {
17       super( account );
18    }
19
20    // draw Account balance as a bar graph
21    public void paintComponent( Graphics g )
22    {
23       // ensure proper painting sequence
24       super.paintComponent( g );
25
26       // get Account balance
27       double balance = getAccount().getBalance();
28
29       // calculate integer height for bar graph (graph
30       // is 200 pixels wide and represents Account balances
31       // from -$5,000.00to +$5,000.00)
32       int barLength = ( int ) ( ( balance / 10000.0 ) * 200 );
33
```

Extends **AbstractAccountView** to provide a bar-graph view of **Account** data.

Method **paintComponent** draws a bar graph for the current **Account** balance.

```
34          // if balance is positive, draw graph in black
35          if ( balance >= 0.0 ) {
36             g.setColor( Color.black );
37             g.fillRect( 105, 15, barLength, 20 );
38          }
39
40          // if balance is negative, draw graph in red
41          else {
42             g.setColor( Color.red );
43             g.fillRect( 105 + barLength, 15, -barLength, 20 );
44          }
45
46          // draw vertical and horizontal axes
47          g.setColor( Color.black );
48          g.drawLine( 5, 25, 205, 25 );
49          g.drawLine( 105, 5, 105, 45 );
50
51          // draw graph labels
52          g.setFont( new Font( "SansSerif", Font.PLAIN, 10 ) );
53          g.drawString( "-$5,000", 5, 10 );
54          g.drawString( "$0", 110, 10 );
55          g.drawString( "+$5,000", 166, 10 );
56
57       } // end method paintComponent
58
59       // repaint graph when display is updated
60       public void updateDisplay()
61       {
62          repaint();
63       }
64
65       // get AccountBarGraphView's preferred size
66       public Dimension getPreferredSize()
67       {
68          return new Dimension( 210, 50 );
```

Draw the bar graph in black for positive **Account** balance and in red for negative **Account** balance.

Method **updateDisplay** invokes method **repaint** to update the ba

Returns a new **Dimension** object that specifies the **AccountBarGraphView**'s preferred size as 210 pixels wide by 50 pixels high.

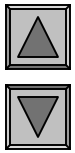
Outline

aphV
iew for rendering observed Account information as a bar graph.

Lines 35-44

Lines 60-63

Line 68



Outline

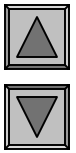
```
69      }
70
71      // get AccountBarGraphView's minimum size
72      public Dimension getMinimumSize()
73      {
74         return getPreferredSize();
75      }
76
77      // get AccountBarGraphView's maximum size
78      public Dimension getMaximumSize()
79      {
80         return getPreferredSize();
81      }
82   }
```

Override methods **`getMinimumSize`** and **`getMaximumSize`** to return the **`AccountBarGraphView`**'s preferred size.

`GraphV`

`ccount`
`information as a`
`bar graph.`

`Lines 72-81`



Outline

Fig. 3.8 AssetPieChartView for rendering multiple observed asset Accounts as a pie chart.

Lines 25-42

Line 35

```
1  // AssetPieChartView.java
2  // AssetPieChartView is an AbstractAccountView subclass that
3  // displays multiple asset Account balances as a pie chart.
4  package com.deitel.advjhtp1.mvc.account;
5
6  // Java core packages
7  import java.awt.*;
8  import java.util.*;
9  import java.util.List;
10
11 // Java extension packages
12 import javax.swing.*;
13 import javax.swing.border.*;
14
15 public class AssetPieChartView extends JPanel
16    implements Observer {
17
18    // Set of observed Accounts
19    private List accounts = new ArrayList();
20
21    // Map of Colors for drawing pie chart wedges
22    private Map colors = new HashMap();
23
24    // add Account to pie chart view
25    public void addAccount( Account account )
26    {
27       // do not add null Accounts
28       if ( account == null )
29          throw new NullPointerException();
30
31       // add Account to accounts Vector
32       accounts.add( account );
33
34       // add Color to Hashtable for drawing Account's wedge
35       colors.put( account, getRandomColor() );
```

Method **addAccount** adds an Account to the List of Accounts shown in the pie chart.

Invokes method **getRandomColor** and adds the random **Color** to the **colors Map**.

```
36
37          // register as Observer to receive A
38          account.addObserver( this );
39
40          // update display with
41          repaint();
42       }
43
44       // remove Account from pie chart view
45       public void removeAccount( Account account )
46       {
47          // stop receiving updates from give
48          account.deleteObserver( this );
49
50          // remove Account from accounts Vector
51          accounts.remove( account );
52
53          // remove Account's Color from Hashtable
54          colors.remove( account );
55
56          // update display to remove Account information
57          repaint();
58       }
59
60       // draw Account balances in a pie chart
61       public void paintComponent( Graphics g )
62       {
63          // ensure proper painting sequence
64          super.paintComponent( g );
65
66          // draw pie chart
67          drawPieChart( g );
68
```

Invokes method **addObserver** of class **Account** to register the **tView** for **Account** updates.

Invokes method **repaint** to display the pie chart with the new **Account**'s information.

Meth… an **A**…

Invokes method **deleteObserver** of class **Account** to **unregister** the **AssetPieChartView** as an **Observer** of the **Account**.

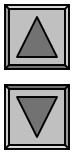
Method **paintComponent** invokes method **drawPieChart** and **drawLegend** to draw the pie chart and chart legend respectively.

AssetPieChartVie w for rendering multiple observed asset as a .

Line 41

Lines 45-58

Line 48



Outline

```
69          // draw legend to describe pie chart wedges
70          drawLegend( g );
71       }
72
73       // draw pie chart on given Graphics context
74       private void drawPieChart( Graphics g )
75       {
76          // get combined Account balance
77          double totalBalance = getTotalBalance();
78
79          // create temporary variables for pie chart calculations
80          double percentage = 0.0;
81          int startAngle = 0;
82          int arcAngle = 0;
83
84          Iterator accountIterator = accounts.iterator();
85          Account account = null;
86
87          // draw pie wedge for each Account
88          while ( accountIterator.hasNext() ) {
89
90             // get next Account from Iterator
91             account = ( Account ) accountIterator.next();
92
93             // draw wedges only for included Accounts
94             if ( !includeAccountInChart( accou
95                continue;
96
97             // get percentage of total balance
98             percentage = account.getBalance() / totalBalance;
99
100            // calculate arc angle for percentage
101            arcAngle = ( int ) Math.round( percentage * 360 );
102
```

Method **drawPieChart** draws a pie-chart wedge for each **Account**.

The **while** loop calculates the percentage of the total balance held in each **Account** and draw the wedges.

Invokes method **includeAccountInChart** to determine if the pie chart should include the current **Account**.

multiple observed asset Accounts as a pie chart.

Lines 74-112

```
103          // set drawing Color for Account pie wedge
104          g.setColor( ( Color ) colors.get( account ) );
105
106          // draw Account pie wedge
107          g.fillArc( 5, 5, 100, 100, startAngle, arcAngle );
108
109          // calculate startAngle for next pie wedge
110          startAngle += arcAngle;
111       }
112    } // end method drawPieChart
113
114    // draw pie chart legend on given Graphics contex
115    private void drawLegend( Graphics g )
116    {
117       Iterator accountIterator = accounts.iterator()
118       Account account = null;
119
120       // create Font for Account name
121       Font font = new Font( "SansSerif", Font.BOLD, 12 );
122       g.setFont( font );
123
124       // get FontMetrics for calculating offsets and
125       // positioning descriptions
126       FontMetrics metrics = getFontMetrics( font );
127       int ascent = metrics.getMaxAscent();
128       int offsetY = ascent + 2;
129
130       // draw description for each Account
131       for ( int i = 1; accountIterator.hasNext(); i++ ) {
132
133          // get next Account from Iterator
134          account = ( Account ) accountIterator.next();
135
```

Invokes method **fillArc** of class **Graphics** to draw the **Account**'s pie wedge.

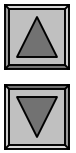
Method **drawLegend** draws a legend to show which color represents each **Account**.

Lines 115-145

Use a **FontMetrics** object to calculate the heights of characters in the current **Font**.

The **for** loop draw the legend item for each **Account**.

Fig. 3.8 AssetPieChartView for rendering multiple observed asset Accounts as a



Outline

```
136          // draw Account color swatch at next offset
137          g.setColor( ( Color ) colors.get( account ) );
138          g.fillRect( 125, offsetY * i, ascent, ascent );
139
140          // draw Account name next to color swatch
141          g.setColor( Color.black );
142          g.drawString( account.getName(), 140,
143             offsetY * i + ascent );
144       }
145    } // end method drawLegend
146
147    // get combined balance of all observed Acco
148    private double getTotalBalance()
149    {
150       double sum = 0.0;
151
152       Iterator accountIterator = accounts.iterator();
153       Account account = null;
154
155       // calculate total balance
156       while ( accountIterator.hasNext() ) {
157          account = ( Account ) accountIterator.next();
158
159          // add only included Accounts to sum
160          if ( includeAccountInChart( account ) )
161             sum += account.getBalance();
162       }
163
164       return sum;
165    }
166
```

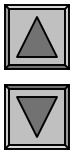
Method **getTotalBalance** calculates the total balance for all included **Account**s.

Adds the **Account**'s balance to variable **sum** if the calculation should include the **Account**.

Fig. 3.8 AssetPieChartView for rendering multiple observed asset[illegible]ts as a [illegible]hart.

Lines 148-164

Line 161



Outline

```
167    // return true if given Account should be included in
168    // pie chart
169    protected boolean includeAccountInChart( Account account )
170    {
171       // include only Asset accounts (Accounts
172       // balances)
173       return account.getBalance() > 0.0;
174    }
175
176    // get a random Color for drawing pie wedg
177    private Color getRandomColor()
178    {
179       // calculate random red, green and blue
180       int red = ( int ) ( Math.random() * 256
181       int green = ( int ) ( Math.random() * 256 );
182       int blue = ( int ) ( Math.random() * 256 );
183
184       // return newly created Color
185       return new Color( red, green, blue );
186    }
187
188    // receive updates from Observable Account
189    public void update( Observable observable, Object object )
190    {
191       repaint();
192    }
193
194    // get AccountBarGraphView's preferred size
195    public Dimension getPreferredSize()
196    {
197       return new Dimension( 210, 110 );
198    }
199
```

Method **includeAccountInChart** returns a **boolean** indicating whether the **Account** should be included in the pie chart.

AssetPieChartView uses method **getRandomColor** to generate a different Color for each **Account** in the pie chart.

Method **update** invokes method **repaint** to update the pie-chart display.

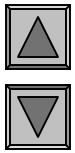
multiple
rved asset
unts as a
chart.

Lines 169-174

Lines 177-186

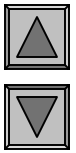
Lines 189-192

```
200     // get AccountBarGraphView's preferred size
201     public Dimension getMinimumSize()
202     {
203        return getPreferredSize();
204     }
205
206     // get AccountBarGraphView's preferred size
207     public Dimension getMaximumSize()
208     {
209        return getPreferredSize();
210     }
211  }
```



Outline

Fig. 3.8 AssetPieChartView for rendering multiple observed asset Accounts as a pie chart.



Outline

Fig. 3.9 Accountcontroller for obtaining user input to modify Account information.

```
1  // AccountController.java
2  // AccountController is a controller for Accounts. It provides
3  // a JTextField for inputting a deposit or withdrawal amount
4  // and JButtons for depositing or withdrawing funds.
5  package com.deitel.advjhtp1.mvc.account;
6
7  // Java core packages
8  import java.awt.*;
9  import java.awt.event.*;
10
11 // Java extension packages
12 import javax.swing.*;
13
14 public class AccountController extends JPanel {
15
16    // Account to control
17    private Account account;
18
19    // JTextField for deposit or withdrawal amount
20    private JTextField amountTextField;
21
22    // AccountController constructor
23    public AccountController( Account controlledAccount )
24    {
25       super();
26
27       // account to control
28       account = controlledAccount;
29
30       // create JTextField for entering amount
31       amountTextField = new JTextField( 10 );
32
```

AccountController implements the controller in the MVC architecture.

Line 31

Creates a **JTextField** into which users can enter an amount to withdraw from, or deposit in, the controlled **Account**.

```
33          // create JButton for deposits
34          JButton depositButton = new JButton( "Deposit" );
35
36          depositButton.addActionListener(
37             new ActionListener() {
38
39                public void actionPerformed( ActionEvent ev
40                {
41                   try {
42
43                      // deposit amount entered in amountTextField
44                      account.deposit( Double.parseDouble(
45                         amountTextField.getText() ) );
46                   }
47
48                   catch ( NumberFormatException exception ) {
49                      JOptionPane.showMessageDialog (
50                         AccountController.this,
51                         "Please enter a valid amount", "Error",
52                         JOptionPane.ERROR_MESSAGE );
53                   }
54                } // end method actionPerformed
55             }
56          );
57
58          // create JButton for withdrawals
59          JButton withdrawButton = new JButton( "Withdraw" );
60
61          withdrawButton.addActionListener(
62             new ActionListener() {
63
64                public void actionPerformed( ActionEvent ev
65                {
```

Create a **JButton** for depositing the given amount into the **Account**.

Create a **JButton** for withdrawing the given amount from the **Account**.

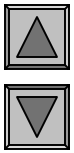
Outline

ntrolle
l for obtaining user input to modify Account information.

Lines 34-56

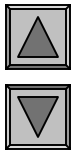
Lines 59-81

```
66                   try {
67
68                      // withdraw amount entered in amountTextField
69                      account.withdraw( Double.parseDouble(
70                         amountTextField.getText() ) );
71                   }
72
73                   catch ( NumberFormatException exception ) {
74                      JOptionPane.showMessageDialog (
75                         AccountController.this,
76                         "Please enter a valid amount", "Error",
77                         JOptionPane.ERROR_MESSAGE );
78                   }
79                } // end method actionPerformed
80             }
81          );
82
83          // lay out controller components
84          setLayout( new FlowLayout() );
85          add( new JLabel( "Amount: " ) );
86          add( amountTextField );
87          add( depositButton );
88          add( withdrawButton );
89       }
90    }
```



Outline

Fig. 3.9 Accountcontroller for obtaining user input to modify Account information.



Outline

Fig. 3.10 AccountManager application for displaying and modifying Account information using the model-view-controller architecture.

```
1  // AccountManager.java
2  // AccountManager is an application that uses the MVC design
3  // pattern to manage bank Account information.
4  package com.deitel.advjhtp1.mvc.account;
5
6  // Java core packages
7  import java.awt.*;
8  import java.awt.event.*;
9
10 // Java extension packages
11 import javax.swing.*;
12 import javax.swing.border.*;
13
14 public class AccountManager extends JFrame {
15
16    // AccountManager no-argument constructor
17    public AccountManager()
18    {
19       super( "Account Manager" );
20
21       // create account1 with initial balance
22       Account account1 = new Account( "Account 1", 1000.00 );
23
24       // create GUI for account1
25       JPanel account1Panel = createAccountPanel( account1 );
26
27       // create account2 with initial balance
28       Account account2 = new Account( "Account 2", 3000.00 );
29
30       // create GUI for account2
31       JPanel account2Panel = createAccountPanel(
32
33       // create AccountPieChartView to show Accou
34       AssetPieChartView pieChartView =
35          new AssetPieChartView();
```

Lines 22 and 28

Creates a new **Account** with the name **Account 1** and a $1,000.00 balance, and **Account 2** with a $3,000.00 balance.

Inv… **ge**… cla… cre… view and controller …**t1**

Create an **AssetPieChartView** for displaying **account1** and **account2** information in a pie chart.

```
36
37          // add both Accounts to AccountPieChartView
38          pieChartView.addAccount( account1 );
39          pieChartView.addAccount( account2 );
40
41          // create JPanel for AccountPieChartView
42          JPanel pieChartPanel = new JPanel();
43
44          pieChartPanel.setBorder(
45             new TitledBorder( "Assets" ) );
46
47          pieChartPanel.add( pieChartView );
48
49          // lay out account1, account2 and pie chart components
50          Container contentPane = getContentPane();
51          contentPane.setLayout( new GridLayout( 3, 1 ) );
52          contentPane.add( account1Panel );
53          contentPane.add( account2Panel );
54          contentPane.add( pieChartPanel );
55
56          setSize( 425, 450 );
57
58       } // end AccountManager constructor
59
60       // create GUI components for given Account
61       private JPanel createAccountPanel( Account account
62       {
63          // create JPanel for Account GUI
64          JPanel accountPanel = new JPanel();
65
66          // set JPanel's border to show Account name
67          accountPanel.setBorder(
68             new TitledBorder( account.getName() ) );
69
```

Invoke method **addAccount** of class **AssetPieChartView** to add **account1** and **account2** to the pie chart.

Create a **JPanel** with a **TitledBorder** for the **AssetPieChartView**.

Lay out the **JPanel**s for each account and **AssetPieChartView**.

Method **createAccountPanel** creates a **JPanel** containing an **A** **A** **A** given **Account**.

Create a **JPanel** with a **TitledBorder** to contain the **Account**'s GUI components.

r
application for
ing and
ng
information
using the model-
view-controller
Lines 38-39

Lines 42-47

Lines 50-54

```
70          // create AccountTextView for Account
71          AccountTextView accountTextView =
72             new AccountTextView( account );
73
74          // create AccountBarGraphView for Account
75          AccountBarGraphView accountBarGraphView =
76             new AccountBarGraphView( account );
77
78          // create AccountController for Account
79          AccountController accountController =
80             new AccountController( account );
81
82          // lay out Account's components
83          accountPanel.add( accountController );
84          accountPanel.add( accountTextView );
85          accountPanel.add( accountBarGraphView );
86
87          return accountPanel;
88
89       } // end method getAccountPanel
90
91       // execute application
92       public static void main( String args[] )
93       {
94          AccountManager manager = new AccountManager();
95          manager.setDefaultCloseOperation( EXIT_ON_CLOSE );
96          manager.setVisible( true );
97       }
98    }
```

Create an **AccountTextView** for the **Account**.

Create an **AccountBarGraphView** for the **Account**.

Create an **AccountController** for the **Account**.

Lay out the **AccountTextview**, **AccountBarGraphView** and **AccountController** components on **accountPanel**.

Fig. 3.10

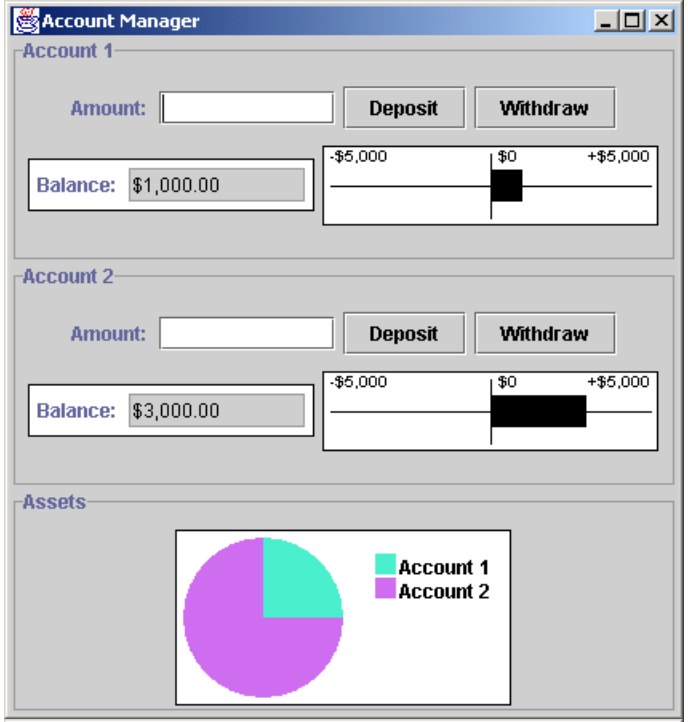
displaying and

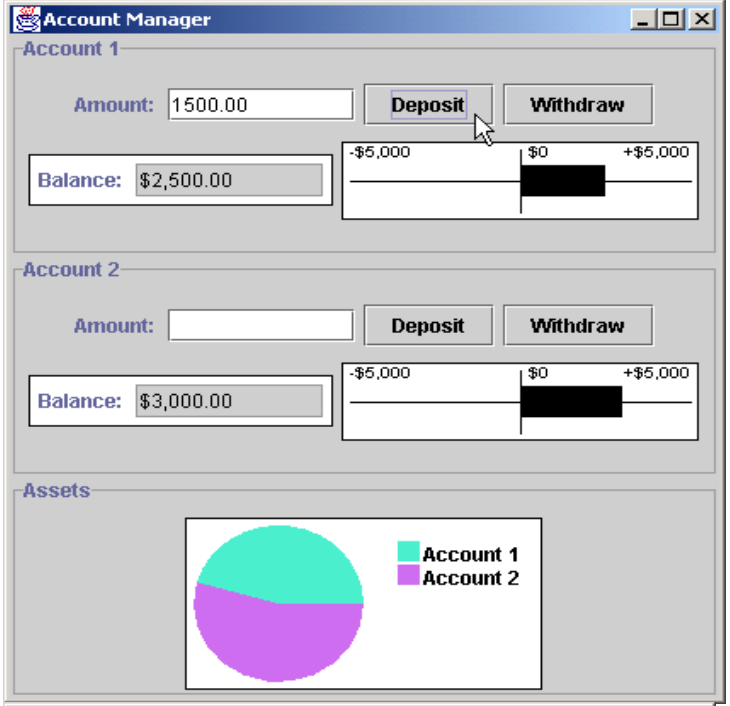
Lines 71-72

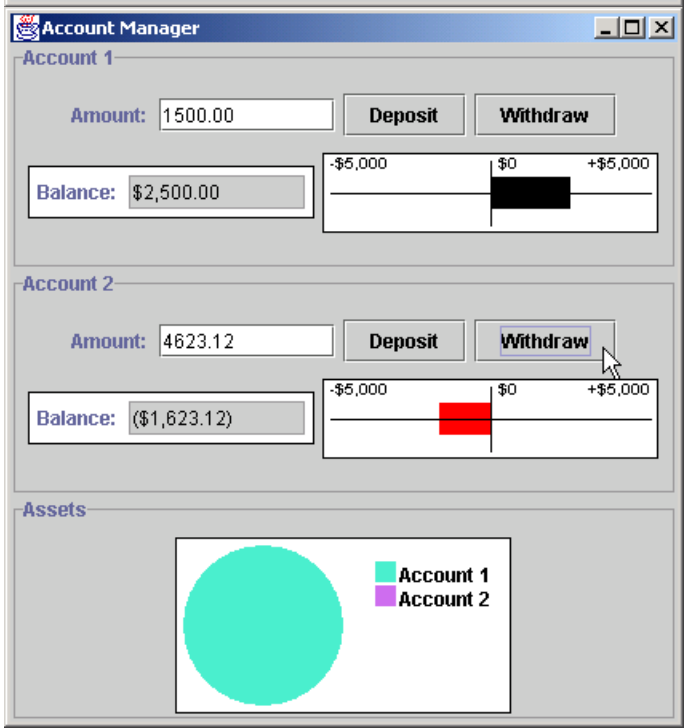
Lines 75-76

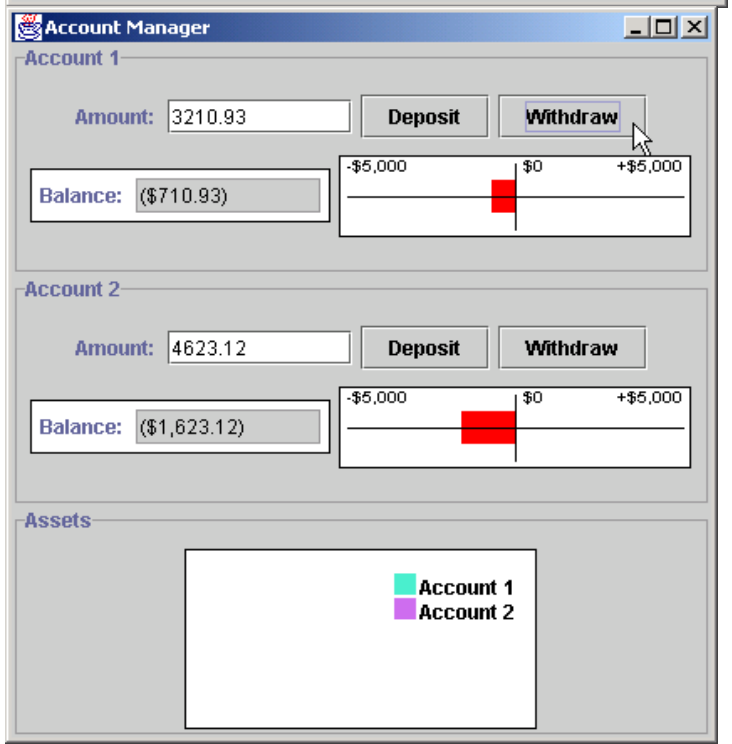
Lines 79-80

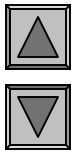
Lines 83-85











Outline

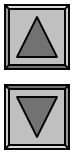
Fig. 3.10 AccountManager application for displaying and modifying Account information using the model-view-controller architecture.

Program output

3.4 JList

- **`JList`**
  - Implements the delegate-model architecture
  - Delegates for **`ListModel`**s
- **`ListModel`**
  - Define methods
  - Register/unregister **`ListDataListener`**

Fig. 3.11 **`JList`** and **`ListModel`** delegate-model architecture.



Outline

Fig. 3.12 PhilosophersJList application demonstrating Jlist and DefaultListModel.

Line 23

Lines 24-31

Line 34

```
1  // PhilosophersJList.java
2  // MVC architecture using JList with a DefaultListModel
3  package com.deitel.advjhtp1.mvc.list;
4
5  // Java core packages
6  import java.awt.*;
7  import java.awt.event.*;
8
9  // Java extension packages
10 import javax.swing.*;
11
12 public class PhilosophersJList extends JFrame {
13
14    private DefaultListModel philosophers;
15    private JList list;
16
17    // PhilosophersJList constructor
18    public PhilosophersJList()
19    {
20       super( "Favorite Philosophers" );
21
22       // create a DefaultListModel to store philosoph
23       philosophers = new DefaultListModel();
24       philosophers.addElement( "Socrates" );
25       philosophers.addElement( "Plato" );
26       philosophers.addElement( "Aristotle" );
27       philosophers.addElement( "St. Thomas Aquinas" );
28       philosophers.addElement( "Soren Kierkegaard" );
29       philosophers.addElement( "Immanuel Kant" );
30       philosophers.addElement( "Friedrich Nietzsche" );
31       philosophers.addElement( "Hannah Arendt" );
32
33       // create a JList for philosophers Default
34       list = new JList( philosophers );
35
```

Creates a new **DefaultListModel** which provides a basic **ListModel** implementation.

Add several philosophers to the **DefaultListModel**.

Creates a new **JList** and passes the philosophers **DefaultListModel** to the **JList** constructor.

```
36          // allow user to select only one philosoph
37          list.setSelectionMode(
38             ListSelectionModel.SINGLE_SELECTION );
39
40          // create JButton for adding philosophers
41          JButton addButton = new JButton( "Add Philosopher" );
42          addButton.addActionListener(
43             new ActionListener() {
44
45                public void actionPerformed( ActionEvent event )
46                {
47                   // prompt user for new philosopher's name
48                   String name = JOptionPane.showInputDialog(
49                      PhilosophersJList.this, "Enter Name" );
50
51                   // add new philosopher to model
52                   philosophers.addElement( name );
53                }
54             }
55          );
56
57          // create JButton for removing selected philosopher
58          JButton removeButton =
59             new JButton( "Remove Selected Philosopher" );
60
61          removeButton.addActionListener(
62             new ActionListener() {
63
64                public void actionPerformed( ActionEvent event )
65                {
66                   // remove selected philosopher from model
67                   philosophers.removeElement(
68                      list.getSelectedValue() );
69                }
70             }
```

Set the **JList**'s selection mode to allow the user to select only one philosopher at a time.

Create a **JButton** for adding new philosophers to the **DefaultListModel**.

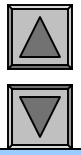
Invokes method **addElement** of class **DefaultListModel** to add the new philosopher to the list.

Create a **JButton** for deleting a philosophers from the **DefaultListModel**.

Fig. 3.12 PhilosophersJLis
Jlist and DefaultListModel
.

Line 52

Lines 58-71



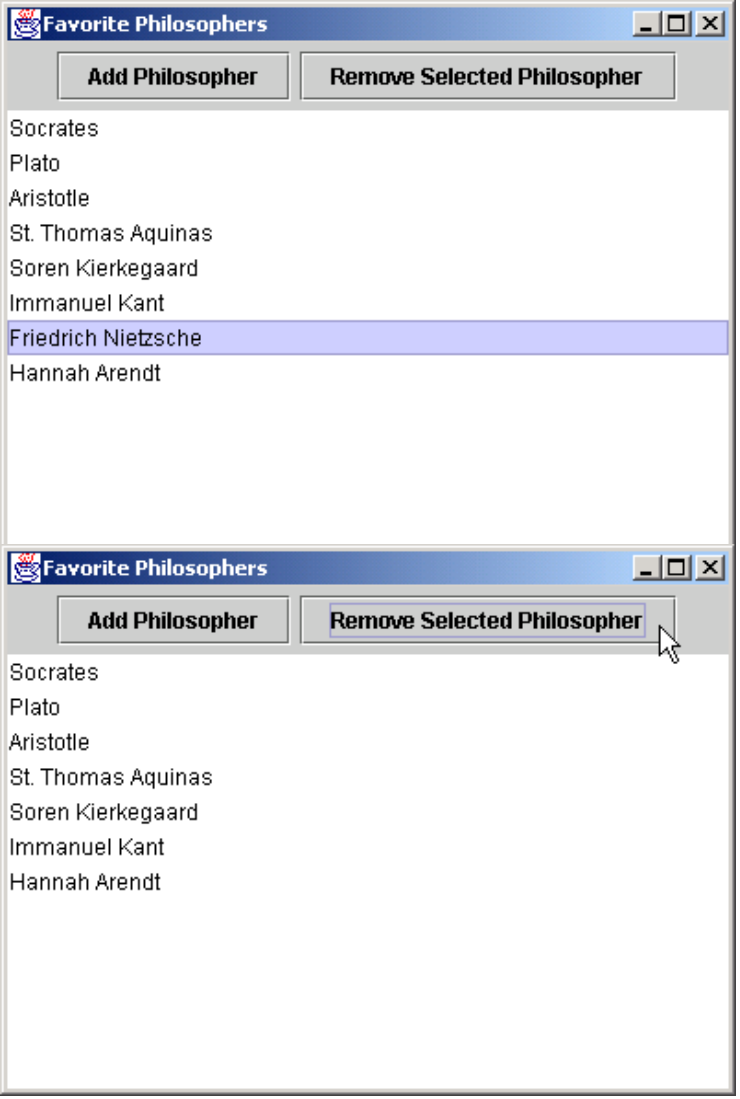
Outline

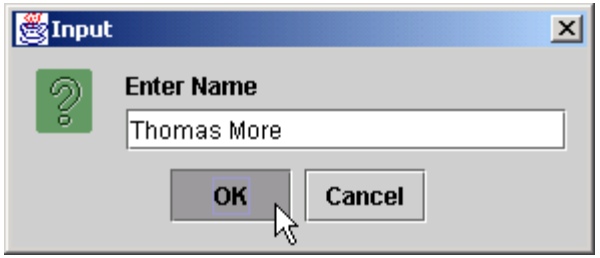
```
71          );
72
73          // lay out GUI components
74          JPanel inputPanel = new JPanel();
75          inputPanel.add( addButton );
76          inputPanel.add( removeButton );
77
78          Container container = getContentPane();
79          container.add( list, BorderLayout.CENTER );
80          container.add( inputPanel, BorderLayout.NORTH );
81
82          setDefaultCloseOperation( EXIT_ON_CLOSE );
83          setSize( 400, 300 );
84          setVisible( true );
85
86       } // end PhilosophersJList constructor
87
88       // execute application
89       public static void main( String args[] )
90       {
91          new PhilosophersJList();
92       }
93    }
```

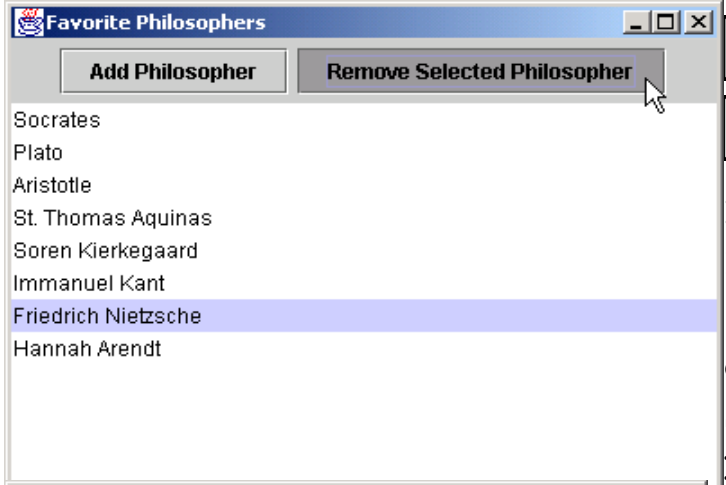
Lay out the GUI components and set **`JFrame`** properties for the application window.

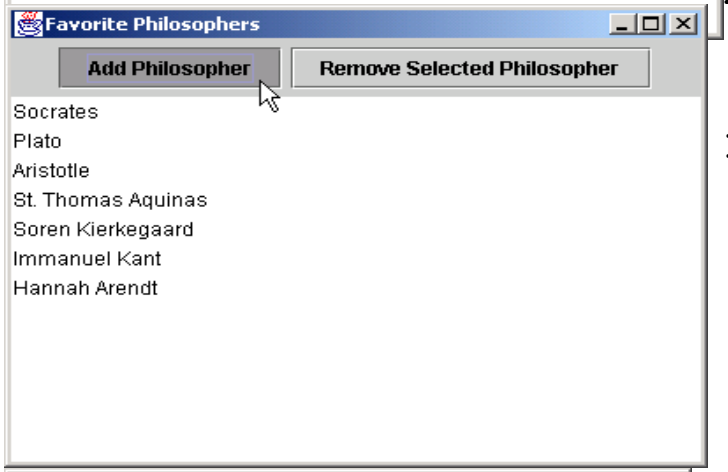
`Lis`
`t application demonstrating Jlist and DefaultListModel.`

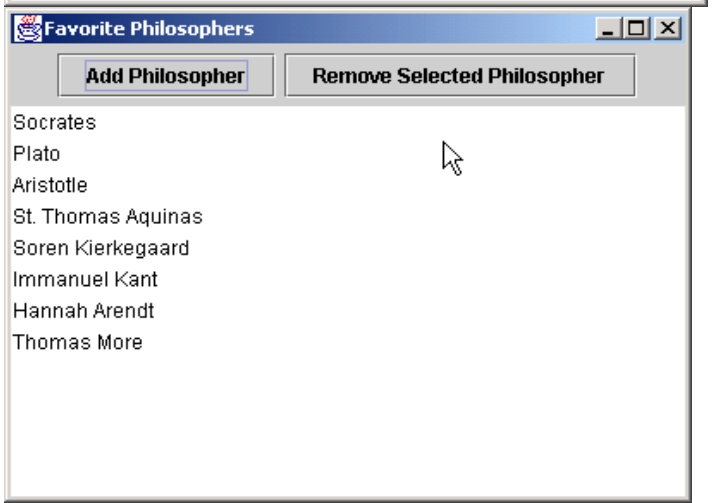
`Lines 74-84`

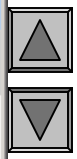












Outline

Fig. 3.12 PhilosophersJList application demonstrating Jlist and DefaultListModel.

Program output

3.5 JTable

- **`JTable`**
  - Implements the delegate-model architecture
  - Delegates for **`TableModel`**s
- **`TableModel`**
  - Declare methods
    - Retrieving and modifying data

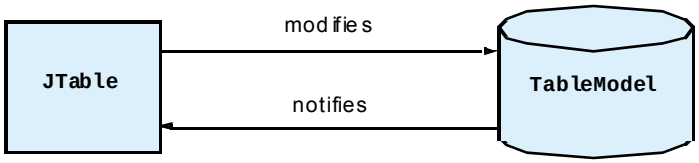


Fig. 3.14 **`JTable`** and **`TableModel`** delegate-model architecture.

3.5 Jtable (Cont.)

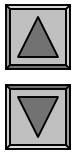
Method	Description
`void addTableModelListener( TableModelListener listener )`	
	Add a `TableModelListener` to the `TableModel`. The `TableModel` will notify the `TableModelListener` of changes in the `TableModel`.
`void removeTableModelListener( TableModelListener listener )`	
	Remove a previously added `TableModelListener` from the `TableModel`.
`Class getColumnClass( int columnIndex )`	
	Get the `Class` object for values in the column with specified `columnIndex`.
`int getColumnCount()`	
	Get the number of columns in the `TableModel`.
`String getColumnName( int columnIndex )`	
	Get the name of the column with the given `columnIndex`.
`int getRowCount()`	
	Get the number of rows in the `TableModel`.

Fig. 3.13 (Part 1 of 2) `TableModel` interface methods and descriptions.

3.5 JTable (Cont.)

`Object getValueAt( int rowIndex, int columnIndex )`	
	Get an **Object** reference to the value stored in the **TableModel** at the given row and column indices.
`void setValueAt( Object value, int rowIndex, int columnIndex )`	
	Set the value stored in the **TableModel** at the given row and column indices.
`boolean isCellEditable( int rowIndex, int columnIndex )`	
	Return **true** if the cell at the given row and column indices is editable.

Fig. 3.13 (Part 2 of 2) TableModel interface methods and descriptions.



Outline

Fig. 3.15 PhilosophersJTable application demonstrating JTable and DefaultTableModel.

Line 24

Lines 27-29

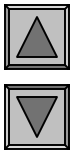
Lines 32-53

```
1  // PhilosophersJTable.java
2  // MVC architecture using JTable with a DefaultTableModel
3  package com.deitel.advjhtp1.mvc.table;
4
5  // Java core packages
6  import java.awt.*;
7  import java.awt.event.*;
8
9  // Java extension packages
10 import javax.swing.*;
11 import javax.swing.table.*;
12
13 public class PhilosophersJTable extends JFrame {
14
15    private DefaultTableModel philosophers;
16    private JTable table;
17
18    // PhilosophersJTable constructor
19    public PhilosophersJTable()
20    {
21       super( "Favorite Philosophers" );
22
23       // create a DefaultTableModel to store philo
24       philosophers = new DefaultTableModel();
25
26       // add Columns to DefaultTableModel
27       philosophers.addColumn( "First Name" );
28       philosophers.addColumn( "Last Name" );
29       philosophers.addColumn( "Years" );
30
31       // add philosopher names and dates to DefaultTableModel
32       String[] socrates = { "Socrates", "", "469-399 B.C." };
33       philosophers.addRow( socrates );
34
```

Creates the philosophers **`DefaultTableModel`**.

Add columns to the **`DefaultTableModel`** for the philosophers' first names, last names and years in which they lived.

Create rows for seven philosophers.



Outline

Fig. 3.15 PhilosophersJTable application demonstrating JTable and Mode

Lines 32-53

Line 56

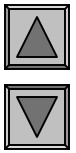
-72

```
35          String[] plato = { "Plato", "", "428-347 B.C." };
36          philosophers.addRow( plato );
37
38          String[] aquinas = { "Thomas", "Aquinas", "1225-1274" };
39          philosophers.addRow( aquinas );
40
41          String[] kierkegaard = { "Soren", "Kierkegaard",
42             "1813-1855" };
43          philosophers.addRow( kierkegaard );
44
45          String[] kant = { "Immanuel", "Kant", "1724-1804" };
46          philosophers.addRow( kant );
47
48          String[] nietzsche = { "Friedrich", "Nietzsche",
49             "1844-1900" };
50          philosophers.addRow( nietzsche );
51
52          String[] arendt = { "Hannah", "Arendt", "1906-1975" };
53          philosophers.addRow( arendt );
54
55          // create a JTable for philosophers Defaul
56          table = new JTable( philosophers );
57
58          // create JButton for adding philosophers
59          JButton addButton = new JButton( "Add Philosopher" );
60          addButton.addActionListener(
61             new ActionListener() {
62
63                public void actionPerformed( Ac
64                {
65                   // create empty array for new philosopher row
66                   String[] philosopher = { "", "", "" };
67
68                   // add empty philosopher row to model
69                   philosophers.addRow( philosopher );
```

Create rows for seven philosophers.

Creates the **JTable** that will act as a delegate for the philosophers **DefaultTableModel**.

Create a **JButton** and **ActionListener** for adding a new philosopher to the **DefaultTableModel**.



Outline

Fig. 3.15 PhilosophersJTableModel.

Lines 75-88

Lines 96-97

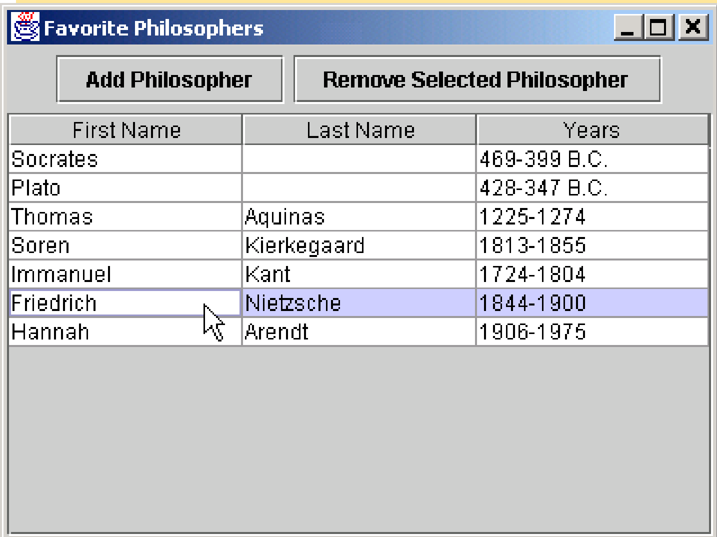
```
71          }
72       );
73
74       // create JButton for removing selected philosopher
75       JButton removeButton =
76          new JButton( "Remove Selected Philosopher" );
77
78       removeButton.addActionListener(
79          new ActionListener() {
80
81             public void actionPerformed( ActionEvent event )
82             {
83                // remove selected philosopher from model
84                philosophers.removeRow(
85                   table.getSelectedRow() );
86             }
87          }
88       );
89
90       // lay out GUI components
91       JPanel inputPanel = new JPanel();
92       inputPanel.add( addButton );
93       inputPanel.add( removeButton );
94
95       Container container = getContentPane();
96       container.add( new JScrollPane( table ),
97          BorderLayout.CENTER );
98       container.add( inputPanel, BorderLayout.NORTH );
99
100      setDefaultCloseOperation( EXIT_ON_CLOSE );
101      setSize( 400, 300 );
102      setVisible( true );
103
104   } // end PhilosophersJTable constructor
```

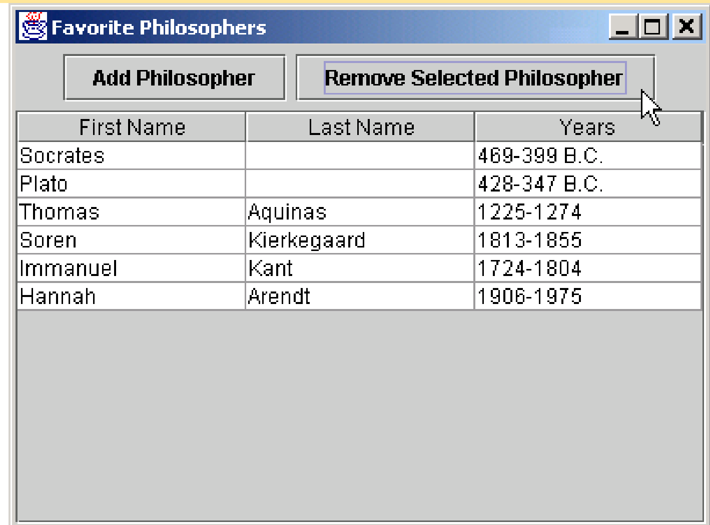
Create a **JButton** and **ActionListener** for removing a philosopher from the **DefaultTableModel**.

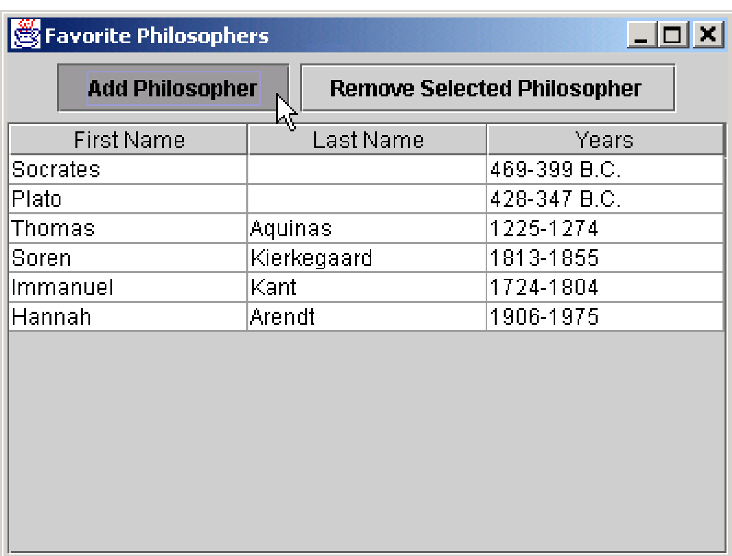
Add the **JTable** to a **JScrollPane**.

```java

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







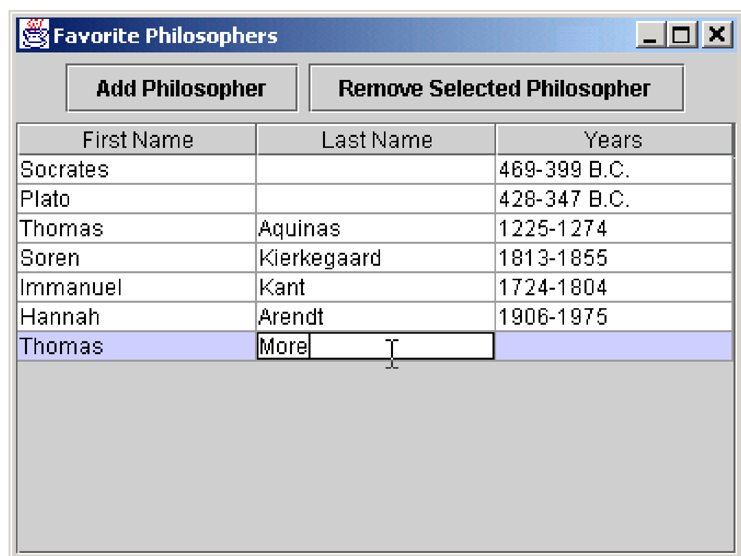


Fig. 3.15 PhilosophersJTable application demonstrating JTable and DefaultTableModel.

Program output

3.6 JTree

- **`JTree`**
  - Implements the delegate-model architecture
  - Delegates for **`TreeModel`**s
- **`TreeModel`**
  - Hierarchical data
    - Parents
    - Children
    - Siblings
    - Ancestors
    - Descendents

3.6 Jtree (Cont.)

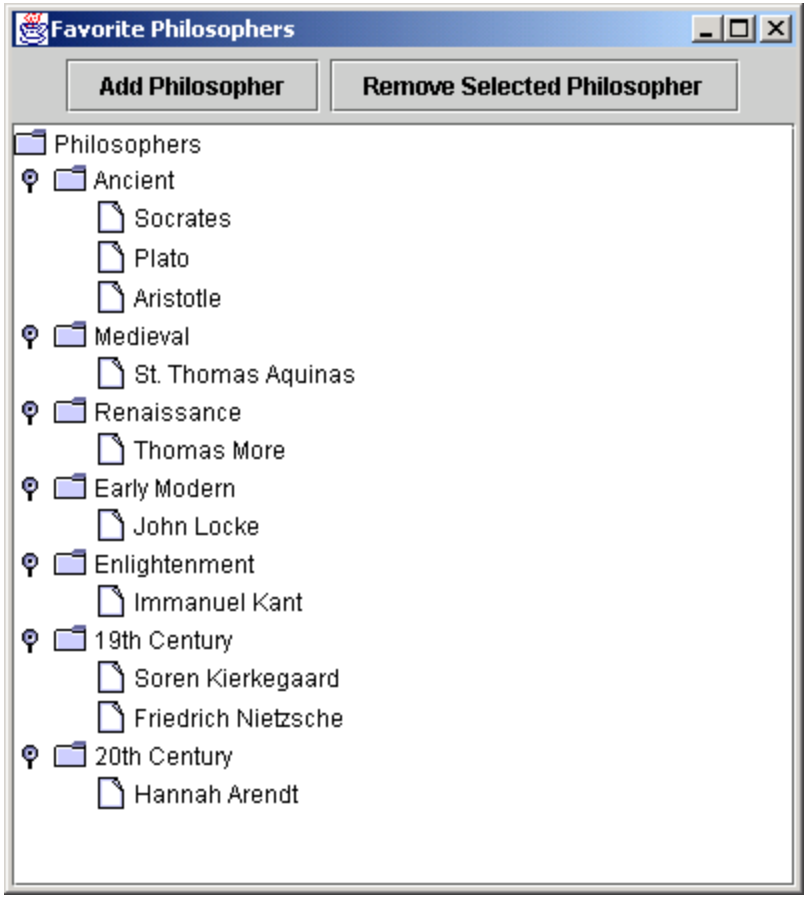
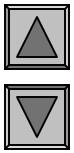


Fig. 3.16 **JTree** showing a hierarchy of philosophers.

3.6.1 Using `DefaultTreeModel`

- Interface **`TreeModel`**
  - Declares methods for representing tree structure
- Class **`DefaultTreeModel`**
  - Default **`TreeModel`** implementation
    - **`TreeNode`**
    - **`MutableTreeNode`**
    - **`DefaultMutableTreeNode`**



Outline

Fig. 3.17 PhilosophersJTree application demonstrating Jtree and DefaultTreeModel.

Lines 26-27

Line 30

Line 33

```
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 DefaultMutableTreeNo
26       DefaultMutableTreeNode philosophersNode =
27          createPhilosopherTree();
28
29       // create philosophers DefaultTreeModel
30       philosophers = new DefaultTreeModel( philoso
31
32       // create JTree for philosophers DefaultTreeModel
33       tree = new JTree( philosophers );
34
```

Invoke method

Creates a **DefaultTreeModel** and passes the **philosophersNode**

Creates a **JTree** and passes **DefaultTreeModel** philosophers to the **JTree** constructor.

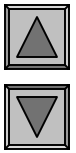
```
35          // create JButton for adding philosophers
36          JButton addButton = new JButton( "Add" );
37          addButton.addActionListener(
38             new ActionListener() {
39
40                public void actionPerformed( ActionEvent event )
41                {
42                   addElement();
43                }
44             }
45          );
46
47          // create JButton for removing selected philosopher
48          JButton removeButton =
49             new JButton( "Remove" );
50
51          removeButton.addActionListener(
52             new ActionListener() {
53
54                public void actionPerformed( ActionEvent event )
55                {
56                   removeElement();
57                }
58             }
59          );
60
61          // lay out GUI components
62          JPanel inputPanel = new JPanel();
63          inputPanel.add( addButton );
64          inputPanel.add( removeButton );
65
66          Container container = getContentPane();
67
68          container.add( new JScrollPane( tree ),
69             BorderLayout.CENTER );
```

Create a **JButton** and an **ActionListener** for adding a philosopher to the **philosophers DefaultTreeModel**.

Create a **JButton** and an **ActionListener** for removing a philosopher from the philosophers **DefaultTreeModel**.

ine

PhilosophersJTree application demonstrating Jtree and DefaultTreeModel .



Outline

Fig. 3.17 PhilosophersJTree application rating and TreeModel

Lines 80-103

Lines 99-101

Line 101

```
70
71          container.add( inputPanel, BorderLayout.NORTH );
72
73          setDefaultCloseOperation( EXIT_ON_CLOSE );
74          setSize( 400, 300 );
75          setVisible( true );
76
77       } // end PhilosophersJTree constructor
78
79       // add new philosopher to selected era
80       private void addElement()
81       {
82          // get selected era
83          DefaultMutableTreeNode parent = getSelectedNode();
84
85          // ensure user selected era first
86          if ( parent == null ) {
87             JOptionPane.showMessageDialog(
88                PhilosophersJTree.this, "Select an era.",
89                "Error", JOptionPane.ERROR_MESSAGE );
90
91             return;
92          }
93
94          // prompt user for philosopher's name
95          String name = JOptionPane.showInputDialog(
96             PhilosophersJTree.this, "Enter Name:" );
97
98          // add new philosopher to selected era
99          philosophers.insertNodeInto(
100            new DefaultMutableTreeNode( name ),
101            parent, parent.getChildCount() );
102
103       } // end method addElement
104
```

Method **addElement** gets the currently selected node in the **JTree** and inserts the new philosopher node as a child of the currently selected node.

Invoke method **insertNodeInto** of class … the n…

Invokes method **getChildCount** of class **DefaultMutableTreeNode** to get the total number of children in node **parent**.

```
105    // remove currently selected philosopher
106    private void removeElement()
107    {
108       // get selected node
109       DefaultMutableTreeNode selectedNode = get
110
111       // remove selectedNode from model
112       if ( selectedNode != null )
113          philosophers.removeNodeFromParent( sel
114    }
115
116    // get currently selected node
117    private DefaultMutableTreeNode getSelectedNode
118    {
119       // get selected DefaultMutableTreeNode
120       return ( DefaultMutableTreeNode )
121          tree.getLastSelectedPathComponent();
122    }
123
124    // get tree of philosopher DefaultMutableTreeNodes
125    private DefaultMutableTreeNode createPhilosopherTree()
126    {
127       // create rootNode
128       DefaultMutableTreeNode rootNode =
129          new DefaultMutableTreeNode( "Philosophers
130
131       // Ancient philosophers
132       DefaultMutableTreeNode ancient =
133          new DefaultMutableTreeNode( "Ancient" );
134       rootNode.add( ancient );
135
136       ancient.add( new DefaultMutableTreeNode( "So
137       ancient.add( new DefaultMutableTreeNode( "Plato" ) );
138       ancient.add( new DefaultMutableTreeNode( "Aristotle" ) );
139
```

Invokes method **getSelectedNode** to get the currently selected node in the **JTree**.

Invokes method **removeNodeFromParent** of class **DefaultTreeModel** to remove **selectedNode** from the model.

Invokes method **getLastSelectedPathComponent** of class **JTree** to get a reference to the currently selected node.

Create a **DefaultMutableTreeNode** for the tree's root

Create **DefaultMutableTreeNodes** for three ancient philosophers and add each as a child of **DefaultMutableTreeNode**.

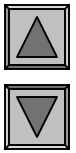
Fig. 3.17

rsJTre
ion
ing

DefaultTreeModel

Line 113

Lines 117-122

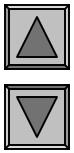


Outline

Fig. 3.17 PhilosophersJTree application demonstrating Jtree and DefaultTreeModel

```
140         // Medieval philosophers
141         DefaultMutableTreeNode medieval =
142            new DefaultMutableTreeNode( "Medieval" );
143         rootNode.add( medieval );
144
145         medieval.add( new DefaultMutableTreeNode(
146            "St. Thomas Aquinas" ) );
147
148         // Renaissance philosophers
149         DefaultMutableTreeNode renaissance =
150            new DefaultMutableTreeNode( "Renaissance" );
151         rootNode.add( renaissance );
152
153         renaissance.add( new DefaultMutableTreeNode(
154            "Thomas More" ) );
155
156         // Early Modern philosophers
157         DefaultMutableTreeNode earlyModern =
158            new DefaultMutableTreeNode( "Early Modern" );
159         rootNode.add( earlyModern );
160
161         earlyModern.add( new DefaultMutableTreeNode(
162            "John Locke" ) );
163
164         // Enlightenment Philosophers
165         DefaultMutableTreeNode enlightenment =
166            new DefaultMutableTreeNode( "Enlightenment" );
167         rootNode.add( enlightenment );
168
169         enlightenment.add( new DefaultMutableTreeNode(
170            "Immanuel Kant" ) );
171
```

Create several additional **DefaultMutableTreeNodes** for other eras in the history of philosophy and for philosophers in those eras.



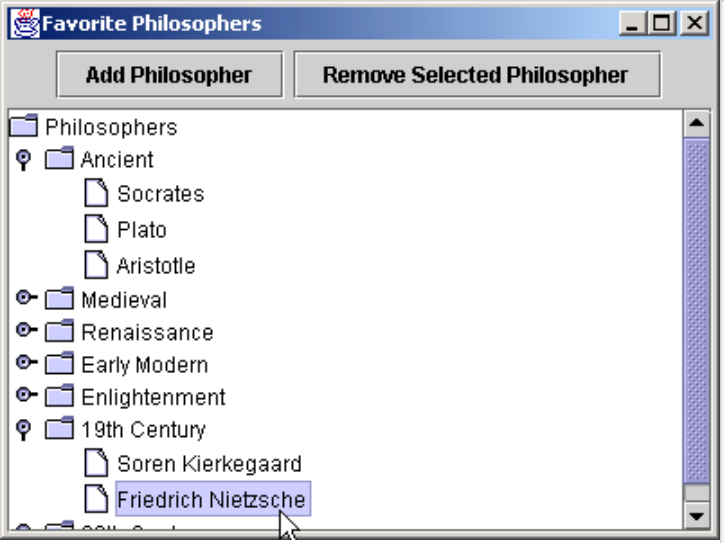
Outline

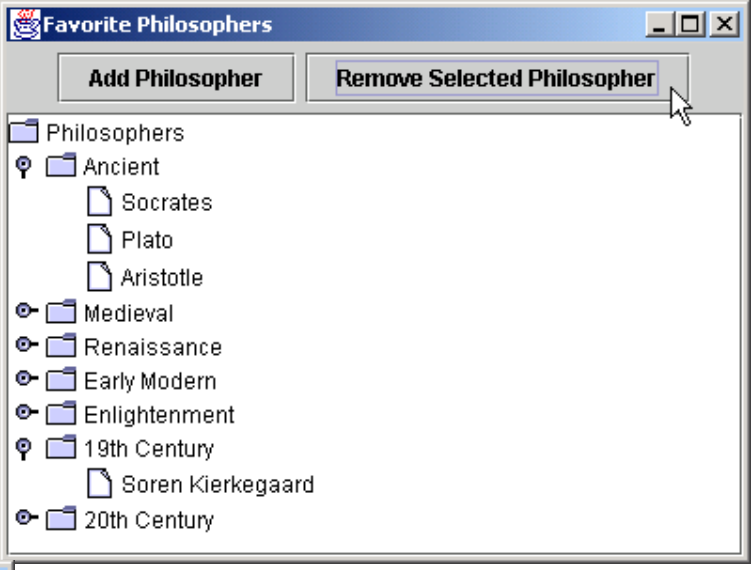
```
172         // 19th Century Philosophers
173         DefaultMutableTreeNode nineteenth =
174            new DefaultMutableTreeNode( "19th Century" );
175         rootNode.add( nineteenth );
176
177         nineteenth.add( new DefaultMutableTreeNode(
178            "Soren Kierkegaard" ) );
179
180         nineteenth.add( new DefaultMutableTreeNode(
181            "Friedrich Nietzsche" ) );
182
183         // 20th Century Philosophers
184         DefaultMutableTreeNode twentieth =
185            new DefaultMutableTreeNode( "20th Century" );
186         rootNode.add( twentieth );
187
188         twentieth.add( new DefaultMutableTreeNode(
189            "Hannah Arendt" ) );
190
191         return rootNode;
192
193      } // end method createPhilosopherTree
194
195      // execute application
196      public static void main( String args[] )
197      {
198         new PhilosophersJTree();
199      }
200   }
```

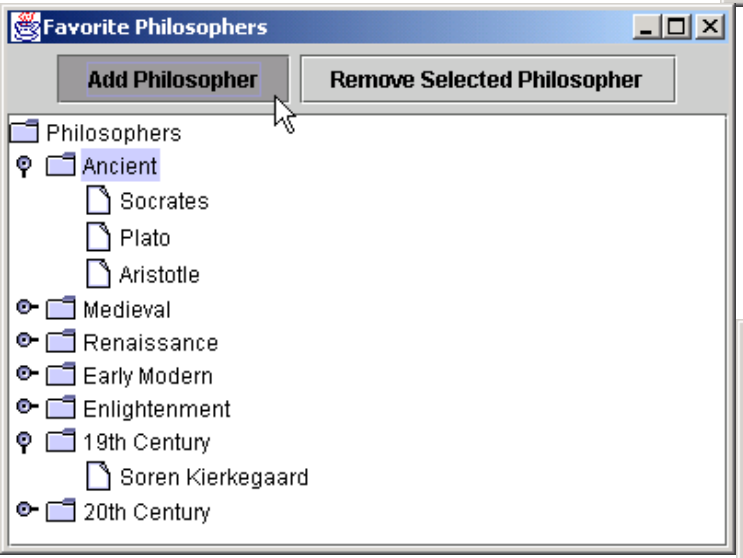
Create several additional **DefaultMutableTreeNodes** for other eras in the history of philosophy and for philosophers in those eras.

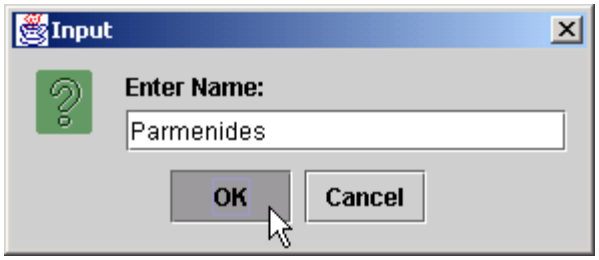
Fig. 3.17 PhilosophersJTree application

Lines 141-189









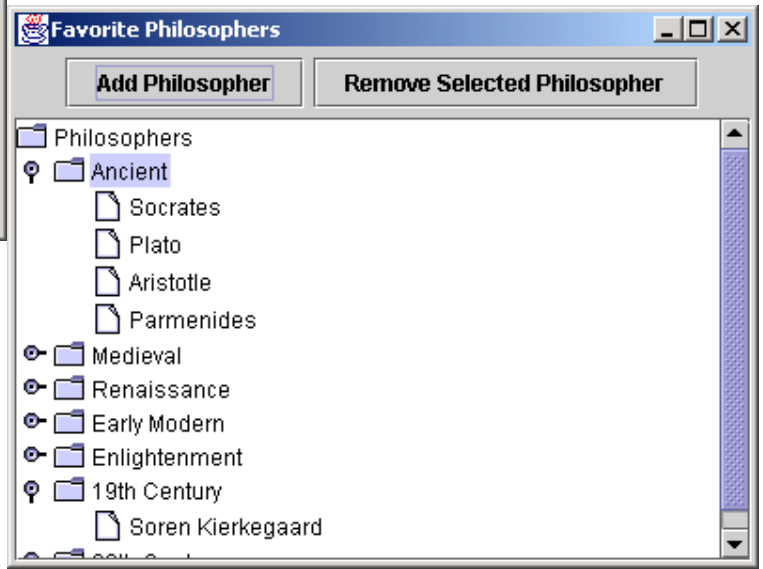
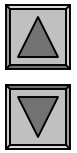


Fig. 3.17 PhilosophersJTree application demonstrating Jtree and DefaultTreeModel.

Program output

3.6.2 Custom `TreeModel` Implementation

- Implement interface **`TreeModel`**
  - Example: **`FileSystemModel`**



Outline

Fig. 3.18 FileSystemModel implementation of interface TreeModel to represent a file

```
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
28    // get hierarchy root (root directory)
29    public Object getRoot()
30    {
31       return root;
32    }
33
```

FileSystemModel implements **TreeModel** interface.

Lines 23-26

Constructor takes a File argument for the **FileSystemModel** root.

Lines 29-32

Returns the **FileSystemModel**'s root node.

```
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
```

Method **getChild** returns argument **parent**'s child node at the given **index**.

FileSystemModel implementation of interface TreeModel to represent a file system.

Method **getChildCount** returns the number of children contained in argument **parent**.

```
66       // return true if node is a file, false if it is
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
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
```

Method **isLeaf** determines if **Object** argument **node** is a leaf node.

Method **getIndexOfChild** returns argument **child**'s index in the given **parent** node.

This **for** loop search through the list for the given child

Fig. 3.18 FileSystemModel implementation of interface TreeModel to present a file system.

Lines 67-71

Lines 74-98

Outline

```
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
```

Method **valueForPathChanged** is invoked by **JTree** delegate when the user edits a node in the tree.

implementation

Invokes method **getLastPathComponent** of class **TreePath** to obtain the File object to rename.

e

Lines 101-135

Create **File** object **targetFile** using the new file name.

05

Invokes method **renameTo** of class **File** to rename **oldFile** to **targetFile**.

Create a **File** object for the renamed file's parent directory.

Line 122

```
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
```

Invoke method **fireTreeNodesChanged** to issue the **TreeModelEvent**.

Method **fireTreeNodesChanged** issues a **TreeModelEvent** to all **...ners**,

Create the **TreeModel** event with the given event data.

This **while** loop iterates through the list of **TreeModelListeners**, sending the **TreeModelEvent** to each.

Method **addTreeModelListener** allow **TreeModelListeners** to register for **TreeModelEvents**.

FileSystemModel implementation of interface

Lines 143-144

Lines 150-153

-161

```
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
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  }
```

Method **removeTreeModelListener** allow **TreeModelListeners** to unregister for **TreeModelEvents**.

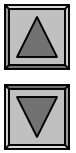
Inner-class **TreeFile** overrides method **toString** of superclass **File**.

FileSystemModel implementation

system.

Lines 164-168

Lines 172-186



Outline

Fig. 3.19 FileTreeFrame application for browsing and editing a file system using JTree and FileSystemModel.

Lines 33-34

```
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
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
```

Create the uneditable **`JTextArea`** for displaying file information.

```
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 Fi
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
63
64          // put fileTree and fileDetailsTextArea in a JSp
65          JSplitPane splitPane = new JSplitPane(
66             JSplitPane.HORIZONTAL_SPLIT, true,
67             new JScrollPane( fileTree ),
68             new JScrollPane( fileDetailsTextArea ) );
69
70          getContentPane().add( splitPane );
```

Create a **FileSystemModel** whose root is **directory**.

Creates a **JTree** for the **FileSystemModel**.

Sets the **JTree**'s editable property to **true**, to allow users to rename files displayed in the **JTree**.

Create a **TreeSelectionListener** to listen for **TreeSelectionEvents** in the **JTree**.

Get the selected **File** object from the **JTree**.

Create a **JSplitPane** to separate the **JTree** and **JTextArea**.

ine

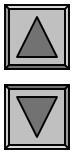
3.19
reeFrame
cation for
and
a file
sing

Lines 37-38

Line 44

Lines 47-62

Lines 55-56



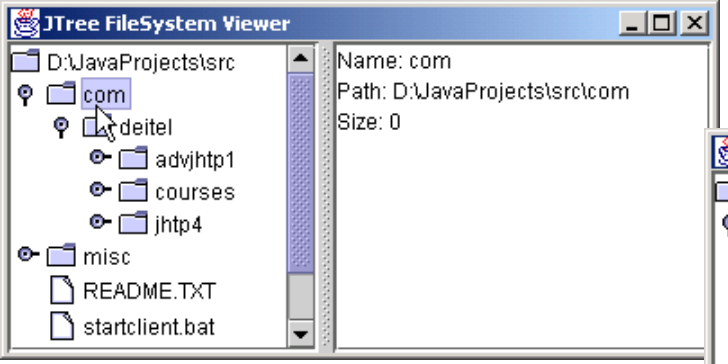
Outline

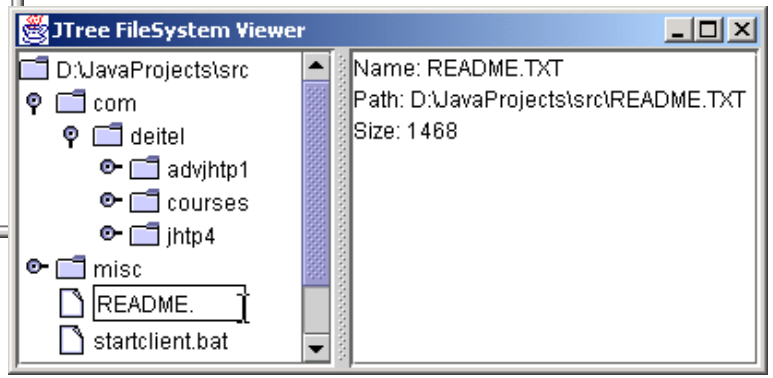
Fig. 3.19 FileTreeFrame for JTree and FileSystemModel.

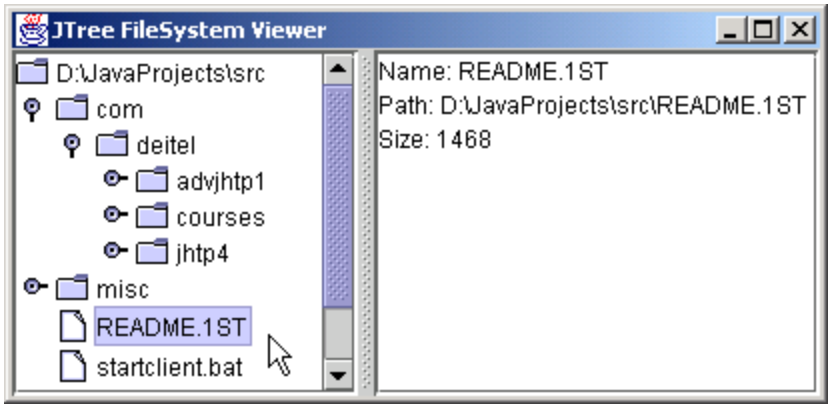
Lines 78-91

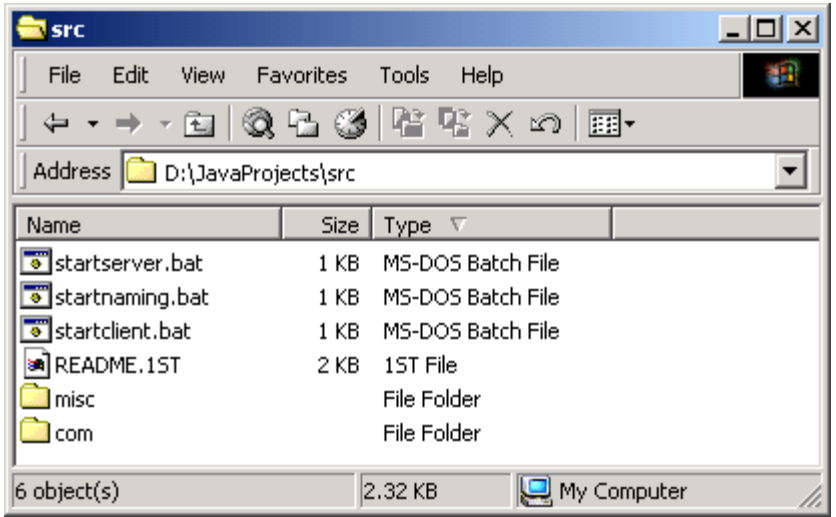
Method **getFileDetails** takes a **File** argument and returns a String containing the **File**'s name, path and length.

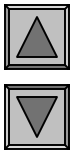
```
71
72          setDefaultCloseOperation( EXIT_ON_CLOSE );
73          setSize( 640, 480 );
74          setVisible( true );
75       }
76
77       // build a String to display file details
78       private String getFileDetails( File file )
79       {
80          // do not return details for null 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   }
```











Outline

Fig. 3.19 FileTreeFrame application for browsing and editing a file system using JTree and FileSystemModel.

Program output