

BERUFSAKADEMIE MANNHEIM
Information Technology International

Lecture Notes

Programming I
Part 5a The Java Foundation Classes

Dipl.-Betriebswirt (BA) VOLKAN YAVUZ
{xmachina GmbH

November 2001–February 2002
Classes: TIM01AGR && TIT01EGR
Term: 1

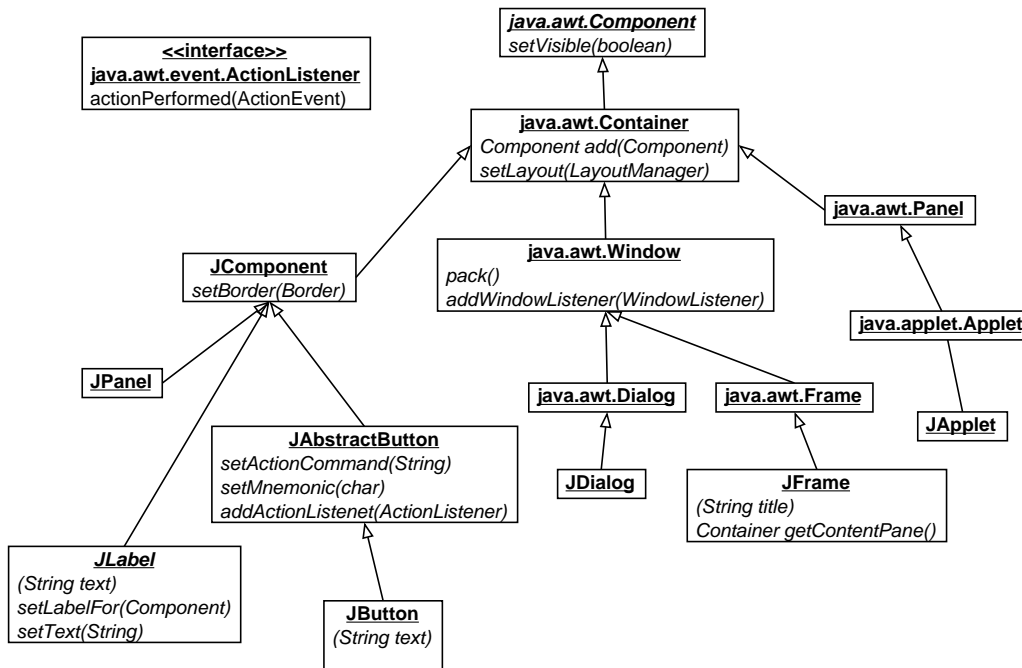
Contents

1	Swing	1
1.1	A Minimal Swing Application	2
1.2	Swing Components and the Containment Hierarchy	4
1.3	Custom Painting and Drawing	5
1.4	Layout Management	6
1.5	Event Handling	6
1.5.1	Implementing ActionListener	8
1.5.2	Implementing MouseListener	8
1.5.3	Extending MouseAdapter	8
1.5.4	Extending MouseAdapter by using an Inner Class	9
1.5.5	Implementing ActionListener by using an Anonymous Inner Class	9
1.5.6	Extending WindowApater by using an Anonymous Inner Class	10
1.6	Models and Views	10
1.6.1	The Model-View-Controller Architecture	10
1.6.2	Separate Data and State Models	11
1.7	Anatomy of a Swing Application	11
	References	12

1 Swing

To implement a GUI (Graphical User Interface), Java supported the AWT (Abstract Window Toolkit). Unfortunately, it wasn't really powerful enough to create complex user interfaces. As it relied heavily onto the underlying operating system's graphical functionality, it more or less only supported a subset of the Macintosh, Motif and Windows user interface functionality. The announcement of the JFC (Java Foundation Classes) in April 1997 brought a much more complete, flexible and portable toolkit to create

Figure 1: The Inheritance Structure of the Example



user interfaces, called “Swing”. All of Swing’s functionality is implemented natively in Java, so it is called *lightweight*, in contrast to the *heavyweight* AWT which relies onto the underlying operating system’s graphical functionality.

Fundamental concepts that Swing is based on include:

- Containment, i. e. how the components a GUI is constructed from are contained within each other
- Layout Management. Swing uses, unlike most other graphical user interfaces, a rather sophisticated way in layout out it’s components.
- The event handling model describes how user actions are translated into *events*, which are then processed further.
- The MVC (Model View Controller) paradigm
- Painting

1.1 A Minimal Swing Application

Following is an almost minimal, but functional example of a Swing application. Most of the used classed and their inheritance hierarchy are shown in figure 1

```

1 import java.awt.*;
2 import java.awt.event.*;
3 import javax.swing.*;

```

The main Swing functionality is provided in the package ‘*javax.swing*’. As Swing uses the AWT infrastructure, including event handling, the relevant packages are ‘*java.awt*’ and ‘*java.awt.event*’.

```

4 public class SimpleExample extends JPanel {
5     static JFrame frame;

```

The main class can extend a 'JPanel' to contain all its atomic components. The top-level container 'JFrame' is declared as static and will be created and used by the also static 'main'-method.

```
6     JRadioButton metalButton, motifButton, windowsButton;
7
8     public SimpleExample() {
9         // Create the buttons.
10        JButton button = new JButton("Hello, world");
11
12        metalButton = new JRadioButton(metal);
13        metalButton.setMnemonic('o');
14        metalButton.setActionCommand(metalClassName);
15        ...
16        // Group the radio buttons.
17        ButtonGroup group = new ButtonGroup();
18        group.add(metalButton);
19        ...
20
21        // Register a listener for the radio buttons.
22        RadioListener myListener = new RadioListener();
23        metalButton.addActionListener(myListener);
24
25        add(button);
26        add(metalButton);
27    }
```

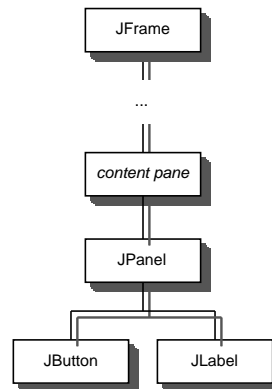
The constructor is used to create all atomic components. A new 'ActionListener' subclass 'RadioListener' is created to listen for events originating from the radio buttons. They are finally added to this 'JPanel'.

```
28     /** An ActionListener that listens to the radio buttons. */
29     class RadioListener implements ActionListener {
30         public void actionPerformed(ActionEvent e) {
31             String lnfName = e.getActionCommand();
32
33             try {
34                 UIManager.setLookAndFeel(lnfName);
35                 SwingUtilities.updateComponentTreeUI(frame);
36                 frame.pack();
37             }
38             catch (Exception exc) {
39                 JRadioButton button = (JRadioButton)e.getSource();
40                 button.setEnabled(false);
41                 updateState();
42                 System.err.println("Could not load LookAndFeel: " + lnfName);
43             }
44         }
45     }
46 }
```

This 'ActionListener' is implemented as an *inner class*.

```
47     public void updateState() {
48         String lnfName = UIManager.getLookAndFeel().getClass().getName();
49         if (lnfName.indexOf(metal) >= 0) {
50             metalButton.setSelected(true);
51         } else if (lnfName.indexOf(windows) >= 0) {
52             windowsButton.setSelected(true);
53         } else if (lnfName.indexOf(motif) >= 0) {
54             motifButton.setSelected(true);
55         } else {
```

Figure 2: The Containment Hierarchy of a Typical Example



```

56         System.err.println("SimpleExample is using an unknown L&F: " + lnfName);
57     }
58 }
    
```

The method ‘updateState’ is just called the first time to get the used look & feel and the button selections in sync. It will also be called whenever installation of a new look & feel was not successful.

```

59     public static void main(String s[]) {
60         SimpleExample panel = new SimpleExample();
61
62         frame = new JFrame("SimpleExample");
63         frame.addWindowListener(new WindowAdapter() {
64             public void windowClosing(WindowEvent e) {System.exit(0);}
65         });
    
```

This ‘WindowAdapter’ is implemented as an *anonymous inner class*.

```

66         frame.getContentPane().add("Center", panel);
67         frame.pack();
68         frame.setVisible(true);
69
70         panel.updateState();
71     }
72 }
    
```

1.2 Swing Components and the Containment Hierarchy

Every Swing program must contain at least one top-level Swing container, which is needed for all Swing components to perform their paintings and event-handling. A typical containment hierarchy is show in figure 2.

There exist the following three top-level Swing containers:

- ‘JFrame’ A single main window
- ‘JDialog’ A secondary window, dependent on another window
- ‘JApplet’ An applet’s display area within a browser window

Intermediate containers are used to simply the positioning of other *atomic* components. Typical examles are

- ‘JPanel’

'JScrollPane'

'JTabbedPane'

Finally, *atomic* components exist, whose job is not to contain other components, but that are capable to present themselves and react to user actions, like

'JButton'

'JLabel'

'JComboBox'

'JTable'

1.3 Custom Painting and Drawing

Painting is performed by extending e.g. 'JPanel' and overriding the method 'paintComponent'. That method is called a 'java.awt.Graphics' context object, which is used to perform specific drawing operations, which include:

- drawing lines, rectangles, text, different shapes like ovals, arcs, polygons,
- setting the color and font to use
- loading and drawing images.

```
73 class DrawPanel extends JPanel {  
74     private static final int WIDTH = 400;  
75     private static final int HEIGHT = 150;  
76     Dimension preferredSize = new Dimension(WIDTH, HEIGHT);  
77     ...
```

First, 'JPanel' is extended. A 'Dimension' object is used to set the preferred size.

```
78     public DrawPanel(Draw controller) {  
79         this.controller = controller;  
80  
81         ...  
82  
83         addMouseListener(new MouseAdapter() {  
84             public void mousePressed(MouseEvent e) {  
85                 int x = e.getX();  
86                 int y = e.getY();  
87                 if (point == null) {  
88                     point = new Point(x, y);  
89                 } else {  
90                     point.x = x;  
91                     point.y = y;  
92                 }  
93                 repaint();
```

The action listener reacts to mouse events, and calls the 'repaint()' method. This schedules this component for a repaint, and the Swing machinery makes sure, that 'paintComponent' will be called in accordance to other repainting actions.

```
94             }  
95         });  
96     }  
97
```

```
98     ...
99
100     public void paintComponent(Graphics g) {
101         super.paintComponent(g); //paint background
```

First, make sure that ‘JPanel’ can do it’s own painting operations.

```
102
103         //If user has chosen a point, paint a tiny rectangle on top.
104         if (point != null) {
105             controller.updateLabel(point);
106             g.fillRect(point.x - 1, point.y - 1, 2, 2);
```

Finally, using the given graphics context, a rectangle is painted.

1.4 Layout Management

The Java platform supplies five commonly used layout managers:

1. ‘BorderLayout’,
2. ‘BoxLayout’,
3. ‘FlowLayout’,
4. ‘GridBagLayout’,
5. ‘GridLayout’, and finally the
6. ‘CardLayout’ which is a special purpose layout manager to be used in conjunction with other layout managers.

By default,

- ‘JPanel’ objects use a ‘FlowLayout’, and
- content panes (like ‘JApplet’, ‘JDialog’, and ‘JFrame’) use ‘BorderLayout’.

It is possible to explicitly disable layout management by calling a panel’s ‘.setLayout(**null**)’. As a consequence, absolute positioning and sizing of every component is necessary. The drawback to this is that this strategy doesn’t adjust well if the top-level container is resized and to differences in systems like font sizes, etc.

1.5 Event Handling

Event handling using Swing can be done in two ways, either by implementing ‘ActionListener’ (or it’s many siblings), or by using an *anonymous inner class* and either extending the abstract class(es) ‘WindowAdapter’ (or it’s many siblings) or implementing ‘ActionListener’ (or it’s many siblings).

A typical example is show in figure 3 on the following page, where a containment hierarchy is depicted, and two action listeners associated with the respective components.

Table 1 on the next page are *some* Swing listener interfaces and their corresponding abstract classes (as adapters), all of which are defined in package ‘java.awt.Event’. Note that for all more complex listener interfaces, *so-called* adapters are defined as *abstract* classes with empty methods, to make implementing them easier. As implementing an interface involves implementing *all* methods, extending abstract classes (like those provided by the several adapters), only involves implementing the *relevant* methods.

Figure 3: Action Listener Installation

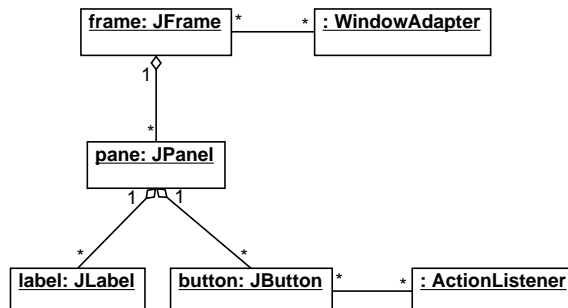


Table 1: Swing Event Listeners and their corresponding Adapters

Listener	Adapter	Example Act that results in the event
'AWTEventListener'		
'ActionListener'		User clicks a button, presses Return while typing in a text field, or chooses a menu item.
'AdjustmentListener'		
'ComponentListener'		Component becomes visible
'ContainerListener'		
'FocusListener'		Component gets the keyboard focus
'InputMethodListener'		
'ItemListener'		
'KeyListener'		
'ListSelectionListener'		Table or list selection changes
'MouseListener'		User presses a mouse button while the cursor is over a component.
'MouseMotionListener'		User moves the mouse over a component
'TextListener'		
'WindowListener'		User closes a frame (main window).

1.5.1 Implementing ActionListener

Event handling by implementing 'ActionListener' works as follows:

```
107 public class SwingApplication implements ActionListener
108 {
109     private void someMethodToConstructTheGui()
110     {
111         ...
112         button.addActionListener(this);
113         ...
114     }
115
116     public void actionPerformed(ActionEvent e) {
117         // do something with the action
118     }
119 }
```

1.5.2 Implementing MouseListener

The preceding approach has works well if the 'ActionListener' interface just contains the needed methods. But consider this example, where a *MouseListener* is implemented. As can be seen, the code is cluttered because of the many empty methods that have to be declared to satisfy that a class that implements an interface has to implement *all* of the methods declared in that interface. The advantage, however is as Java does not support multiple inheritance, the class can *extend* e.g. 'JApplet' and still *implement* 'MouseListener'.

```
120 public class SwingApplication extends JApplet implements MouseListener
121 {
122     // ...
123
124     /* Empty method definition. */
125     public void mousePressed(MouseEvent e) {
126     }
127
128     /* Empty method definition. */
129     public void mouseReleased(MouseEvent e) {
130     }
131
132     /* Empty method definition. */
133     public void mouseEntered(MouseEvent e) {
134     }
135
136     /* Empty method definition. */
137     public void mouseExited(MouseEvent e) {
138     }
139
140     public void mouseClicked(MouseEvent e) {
141         //Event handler implementation goes here...
142     }
143 }
```

1.5.3 Extending MouseAdapter

The Swing package provides so-called adapters for as implementations of 'ActionListener' interfaces which have more than one method, e.g. the 'MouseAdapter'. The above functionality can then be written as follows, but with the drawback that 'SwingApplication' then cannot extend 'JApplet'.

```
144 public class SwingApplication extends MouseAdapter
```

```
145 {
146     // ...
147     // this would NOT work as this does not implement MouseListener
148     someObject.addMouseListener(this);
149     // ...
150
151     public void mouseClicked(MouseEvent e) {
152         //Event handler implementation goes here...
153     }
154 }
```

1.5.4 Extending MouseAdapter by using an Inner Class

The Swing package provides so-called adapters for as implementations of ‘`ActionListener`’ interfaces which have more than one method, e.g. the ‘`MouseAdapter`’. The above functionality can then be written as follows, but with the drawback that ‘`SwingApplication`’ then cannot extend ‘`JApplet`’.

```
155 public class MyClass extends JApplet
156 {
157     // ...
158     someObject.addMouseListener(new MyAdapter());
159     // ...
160     class MyAdapter extends MouseAdapter
161     {
162         public void mouseClicked(MouseEvent e)
163         {
164             //Event handler implementation goes here...</em>
165         }
166     }
167 }
```

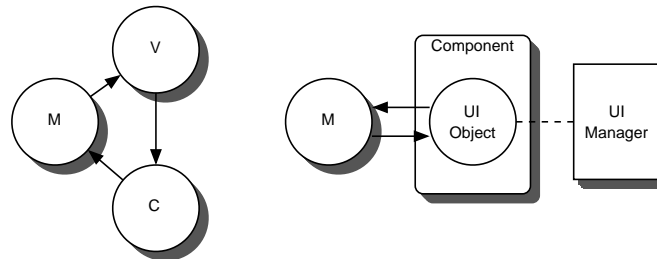
1.5.5 Implementing ActionListener by using an Anonymous Inner Class

As an alternative, event handling can be performed by using an *anonymous inner class*, which has the advantage that the code implementing the action is close to the point where that action is bound to a component. The statement ‘`new ActionListener()...`’ actually means

- create an instance of a class that
- ‘implements’ ‘`ActionListener`’, and
- ‘implements’ the method ‘`actionPerformed(ActionEvent e)`’

```
168 public class SwingApplication
169 {
170     ...
171     private void someMethodToConstructTheGui()
172     {
173         // construct the GUI
174         JButton button = new JButton("I'm a Button!");
175         button.addActionListener(new ActionListener() {
176             public void actionPerformed(ActionEvent e) {
177                 // do something with the action
178             }
179         });
180     }
181     ...
182 }
```

Figure 4: The Model-View-Controller and Swing Architecture



1.5.6 Extending WindowAdapter by using an Anonymous Inner Class

The advantage is, that using anonymous inner classes also works well for extending *WindowAdapter*, as can be seen in the following code. The statement ‘`new WindowAdapter()...`’ actually means

- create an instance of a class that
- ‘extends’ ‘*WindowAdapter*’, and
- ‘implements’ the method ‘`windowClosing(WindowEvent e)`’

```

183 public class Hello
184 {
185     // ...
186     frame.addWindowListener(new WindowAdapter()
187     {
188         public void windowClosing(WindowEvent e)
189         {
190             System.exit(0);
191         }
192     });
193     // ...
194 }
    
```

1.6 Models and Views

1.6.1 The Model-View-Controller Architecture

It is considered good practice to separate an application's data from its representation, and also, to model an application around its data instead of around its presentation. The user interaction with and presentation of data has to be separated by a clear design.

The Swing architecture is rooted in the MVC architecture, which dates back to SmallTalk. A visual application is broken up into three separate parts:

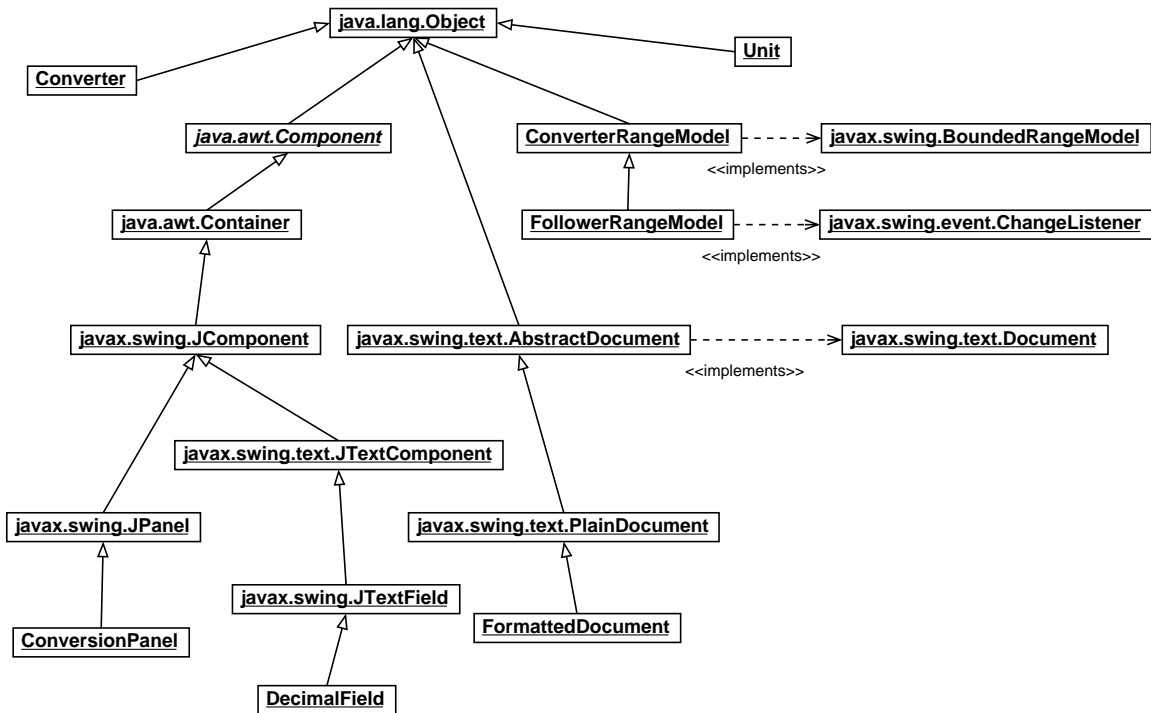
Model The model represents the data the application works with

View The view is the visual representation of the data contained in the *model*.

Controller The controller's responsibility is to accept user input (from the view) and translate that into changes that directly affect the model. The model then affects the view.

Swing itself is only loosely based on a MVC-architecture. The difference is that in Swing, the *view* and *controller* functionality are combined in a *UI Object*, which is shown in figure 4. The *UI Object* provides all functionality combined in both the *view* and *controller* parts of the MVC architecture. Additionally, all *look&feel* specific aspects are delegated to the *UI Manager* part.

Figure 5: The Inheritance Structure of the Example



1.6.2 Separate Data and State Models

Swing provides some additional separation of the *model* part of its architecture.

GUI-state models GUI-state models have only meaning in combination with a graphical representation. A typical example of a GUI-state model would be whether a button is pressed or not, which items of a list are selected, etc. For simple applications this kind of models is sufficient, and all interaction with the user and manipulation of the data can be done directly on the GUI-state models.

Application-data models Whenever the data to be manipulated gets more complex, like in ‘JTree’ and ‘JTable’ components, it is advisable to interact with the Application-data models. While the Application-data model contains the complete data, the GUI-state model just contains the information specific to the visualization.

1.7 Anatomy of a Swing Application

A complete example of a Swing application is given in project ‘swing09_Anatomy’. The inheritance structure is given in figure 5.

List of Tables

1	Swing Event Listeners and their corresponding Adapters	7
---	------------------------------------------------------------------	---

List of Figures

1	The Inheritance Structure of the Example	2
2	The Containment Hierarchy of a Typical Example	4
3	Action Listener Installation	7
4	The Model-View-Controller and Swing Architecture	10
5	The Inheritance Structure of the Example	11

List of Acronyms

AWT.....	Abstract Window Toolkit
GUI.....	Graphical User Interface
JFC.....	Java Foundation Classes
MVC.....	Model View Controller

References