Dancing Lesson One

Babysteps towards building Graphical User Interfaces using Java and Swing

by Bastiaan Schönhage

1 Swingtroduction

All beginning is difficult. As a little child you have to learn how to eat, walk and sit. Then you have to learn how to speak. While growing up, you must learn how to read, write, behave, and so forth. And each time you want to learn something new, you will have to start from the beginning, just like a baby.

When learning computer programming, the analogy holds very well. You think you have managed how to learn programming the computer in Java. You know how to write interactive command line applications and how to program difficult data structures such as double-linked ordered lists. However, the one thing you are still missing is a graphical user interface for your application.

Learning to program graphical applications, might be a little difficult in the beginning, but as soon as you get the grip, it will make you feel very good. Because suddenly you are able to demonstrate your applications to your friends. And they will be impressed by what you have made.

However, getting back to the first sentence of this introduction: *all beginning is difficult*. And exactly to help you with those difficult first steps, is the purpose of this document.

Java, AWT and Swing

When Java first appeared it consisted of a number of packages. Of course, there was a core language package (`java.lang`), but also some packages for networking (`java.net`), input and output (`java.io`) and more were available. The package used for creating graphical applications was called the *Abstract Window Toolkit (AWT)* (`java.awt`) and consisted of a number of simple classes to create windows, frames, buttons etcetera.[1]

However, this graphical toolkit appeared to be too simplistic and, in addition, it simply was not flashy enough. Java applications were looking dull and often contained a poor interface. This was the main motivation for creating a new graphical toolkit (based on AWT) which is far more appealing and is part of an application development framework called the *Java Foundation Classes*. The new graphical toolkit is called Swing (`com.sun.java.swing`).

2 First swinging steps

Now you know a little bit about the background of Swing, it is time to start writing our first Swing-enabled Java application. And since these are your first dancing lessons, let us start extremely simple by just creating a Swing Frame (Figure 1). A frame is the Java synonym for what most people would call a window: the box on your screen that you can move around, resize, close and minimize.

```
// SimpleFrame.java

import com.sun.java.swing.*;
```

[1]Caution: although the Java Development Kit (JDK) 1.0 and JDK 1.1 both call the graphical toolkit AWT, there are some major differences between them.

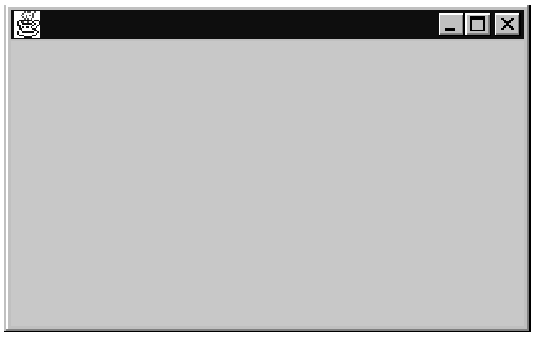


Figure 1: A simple JFrame

```
 4
 5  class SimpleFrame
 6  {
 7
 8      public static void main(String args[])
 9      {
10          JFrame frame = new JFrame();
11          frame.setSize(320, 200);
12          frame.setVisible(true);
13      }
14  }
```

Before discussing the example above, you must realize that since Java is a completely Object Oriented language everything is an Object. As a consequence this also holds for the graphical user interface. So to create a Swing Frame, we will have to create an instance of the Swing Frame class. The example above shows our first Swing application (note that the line numbers are not part of the program but are included here so I can refer to the code easily). Line 3 contains the import of all necessary Swing classes (in this case only the `JFrame` is used). Lines 10-13 contain the interesting code: first, we create an object `frame` of class `JFrame`, which is not yet visible on the screen. To display the object on the screen, we will have to set some properties of the frame to make it the right size and to make it visible. This is achieved in line 11 and 12.

After you have compiled the application (make sure that the Swing package is available in your CLASSPATH), you can run the application. Now, an empty frame should be visible which you can play around with for a while. To stop the application, you will have to use the Ctrl-C keys in the window where you started it. Later on, we will see how an application can be ended more elegantly.

Try a little harder

Although the previous example paints a frame on your screen, the frame is still very empty. Our next step towards a full user interface will be to draw some lines and text in the frame (Figure 2). The straightforward way to achieve this, is by using inheritance. We do not create a standard `JFrame` any longer, instead we will create our own PaintFrame which is based on the standard Swing `JFrame`. The following application creates an instance of our new PaintFrame class, containing some simple line drawings, and shows it on the screen.

```
 1  // PaintFrame.java
 2
 3  import com.sun.java.swing.*;
```

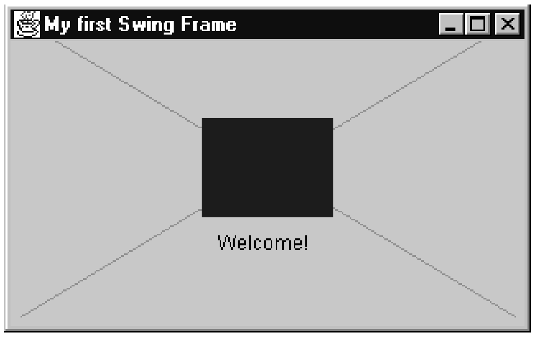


Figure 2: A frame with a redefined paint method

```
 4  import java.awt.Graphics;
 5  import java.awt.Color;
 6
 7  class PaintFrame extends JFrame
 8  {
 9
10      PaintFrame()
11      {
12          setTitle("My first Swing Frame");
13          setSize(320, 200);
14      }
15
16      public void paint(Graphics g)
17      {
18          g.setColor(Color.green);
19          g.drawLine(10, 10,  310, 190);
20          g.drawLine(10, 190, 310, 10);
21          g.setColor(Color.blue);
22          g.fillRect(120, 70 , 80, 60);
23          g.setColor(Color.black);
24          g.drawString("Welcome!", 130, 150);
25      }
26
27      public static void main(String args[])
28      {
29          JFrame frame = new PaintFrame();
30          frame.setVisible(true);
31      }
32  }
```

The imports in lines 3-5 include the needed Swing and AWT classes[2]. Our new `PaintFrame` is defined in lines 7-32, derived from the standard Swing `JFrame`. In the constructor of our class (lines 10-14) we set the title and size of our frame. The most important part of this example comes next in lines 16-25. This is the redefined `paint` method, which is called each time the frame is (re)drawn. Using the AWT `Graphics` object we draw two green lines, a blue rectangle and some black text. Finally, lines 27-31 contain the `main` method of our application. It creates the `PaintFrame` object and displays it by setting the visible property to `true`.

[2]Currently, we need the AWT classes to draw the lines on the screen. However, future releases of the Java Foundation Classes will include a 2D package for more extensive graphics.

3 Graphical User Interfaces

To create Graphical User Interfaces (GUIs) we need more than a simple frame and some line drawings. To create GUIs we must have buttons, drop-down boxes, menus and so on. The next example (PointClick.java, Figure 3) is an example of a GUI consisting of a text-label (`JLabel`), a button (`JButton`) and a field to enter some text (`JTextField`).

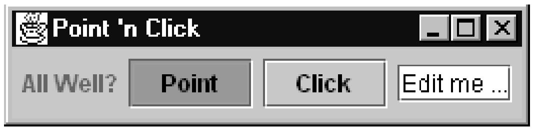


Figure 3: A GUI containing a label, button and text field

```
 1  // PointClick.java
 2
 3  import com.sun.java.swing.*;
 4  import java.awt.*;
 5
 6  class PointClick extends JFrame
 7  {
 8
 9      PointClick()
10      {
11          setTitle("Point ’n Click");
12          JLabel label = new JLabel("All Well?");
13          JButton point = new JButton("Point");
14          JButton click = new JButton("Click");
15          JTextField textField = new JTextField("Edit me ...");
16
17          Container c = getContentPane();
18          c.setLayout(new FlowLayout());
19          c.add(label);
20          c.add(point);
21          c.add(click);
22          c.add(textField);
23      }
24
25      public static void main(String args[])
26      {
27          JFrame frame = new PointClick();
28          frame.pack();
29          frame.setVisible(true);
30      }
31  }
```

After the imports and the class definition (again inheriting from JFrame), we create the GUI in the constructor of our class, lines 9-23. Line 12 creates a JLabel object containing the text “All Well?”. In lines 13 and 14, we create two button objects containing respectively the text “Point” and “Click”. Finally, in line 15, we create the textfield containing the text “Edit me...”.

The GUI objects that have just been created are added to our frame in lines 17-22. First, in line 17, we ask our frame for the `Container` object to which we can add the GUI objects. Then we set the layout manager in line 18. How layout managers work, will be explained in Section 4. Finally, lines 19-22 add the GUI objects to the container of our frame.

Note that line 28 contains a new method call to our JFrame: `frame.pack()`. This method causes subcomponents of this frame to be laid out at their preferred size. So we do not have to use `setSize(x, y)` anymore, instead the frame will have exactly the right size.

4 Layout managers

Although we now know how to create our own frames and how to add GUI components, two important things are still missing. First, the GUI is still dead. You can press the buttons or edit the text in the textfield but the application does not respond to it. Second, we do not know how to lay out our GUI. The first issue is covered in the next section. The subject of layout and layout managers will be the subject of this section.

The layout of your GUI in Swing (and AWT) is being done by a layout manager. Currently, around five types of layout managers exist. In this dancing lesson, we will cover the three most often deployed layout managers (`FlowLayout, BorderLayout, GridLayout`) using the following example (Figure 4).

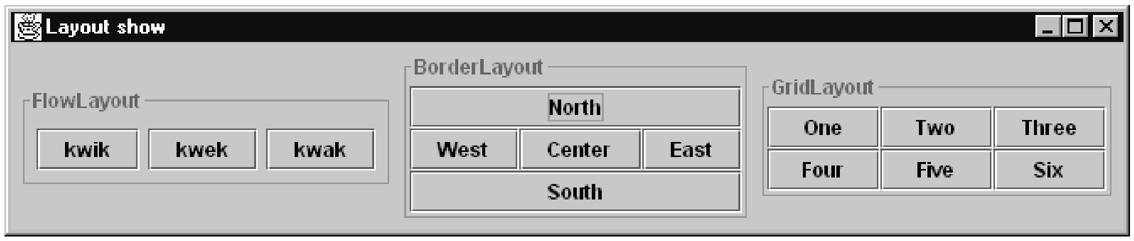


Figure 4: Some layout managers at work

```
// ShowLayout.java

import com.sun.java.swing.*;
import java.awt.*;

class ShowLayout extends JFrame
{

    ShowLayout()
    {
        setTitle("Layout show");

        // FlowLayout
        JPanel panelFlow = new JPanel();
        panelFlow.setBorder(
            BorderFactory.createTitledBorder("FlowLayout"));
        panelFlow.setLayout( new FlowLayout() );
        JButton kwik = new JButton("kwik");
        JButton kwek = new JButton("kwek");
        JButton kwak = new JButton("kwak");
        panelFlow.add(kwik);
        panelFlow.add(kwek);
        panelFlow.add(kwak);

        // BorderLayout
        JPanel panelBorder = new JPanel();
        panelBorder.setBorder(
            BorderFactory.createTitledBorder("BorderLayout"));
        panelBorder.setLayout( new BorderLayout() );
        JButton north = new JButton("North");
        JButton east = new JButton("East");
        JButton south = new JButton("South");
        JButton west = new JButton("West");
        JButton center = new JButton("Center");
        panelBorder.add(north, "North");
        panelBorder.add(east, "East");
        panelBorder.add(south, "South");
        panelBorder.add(west, "West");
        panelBorder.add(center, "Center");

        // GridLayout
        JPanel panelGrid = new JPanel();
        panelGrid.setBorder(
            BorderFactory.createTitledBorder("GridLayout"));
        panelGrid.setLayout(new GridLayout(2,3));
        JButton one = new JButton("One");
        JButton two = new JButton("Two");
        JButton three = new JButton("Three");
        JButton four = new JButton("Four");
        JButton five = new JButton("Five");
        JButton six = new JButton("Six");
        panelGrid.add(one);
        panelGrid.add(two);
        panelGrid.add(three);
        panelGrid.add(four);
        panelGrid.add(five);
        panelGrid.add(six);

        // the Frame contains all panels
        Container c = getContentPane();
        c.setLayout(new FlowLayout());
        c.add(panelFlow);
        c.add(panelBorder);
        c.add(panelGrid);
    }

    public static void main(String args[])
    {
        JFrame frame = new ShowLayout();
        frame.pack();
        frame.setVisible(true);
    }
}
```

Before describing the used layout managers, I will briefly say something about the used `JPanel` object. A panel is used to contain a number of other GUI objects. It has its own layout manager and a `JPanel` can be used inside another `JPanel` or `JFrame`. As an example look at lines 14-16. First, we create a panel that is added to the frame in line 59. Second, we create a border around the panel using the `BorderFactory`. And, finally, we specify the layout manager of this panel to be the flow layout manager.

FlowLayout

The `FlowLayout` is used in the example on two places: in the `panelFlow` panel (line 16) and in the `ShowLayout` frame (line 58). The flow layout manager works by adding objects to the first row until that row is full. Then, it adds the next GUI objects to the next row until that one is full again etcetera. Try to understand the flow layout manager by making the displayed frame somewhat smaller and longer, then you will see that some panels are moved from the first row to next rows. Note that because the `panelFlow` panel does not change size, the buttons in this frame are not moved to a new row.

BorderLayout

The next layout manager, the `BorderLayout`, divides the available space into five parts: north, east, south, west and center. Each of these spaces can contain a single GUI object (this might also be a panel, which can contain more than one GUI object). Lines 33-37 add five buttons to the five parts of the `panelBorder` panel. Note that the `add` method differs from previous `add` methods because it contains a second parameter indicating to which part the GUI object is added.

GridLayout

Finally, the `GridLayout` arranges the components in a number of rows and columns (lines 39-54). When creating the `GridLayout` object we can specify the number of rows and columns in the constructor. In this case we have two rows and three columns (line 42). When adding GUI objects, the grid is filled from left-to-right and top-to-bottom. A disadvantage of the GridLayout is that all components occupy exactly one grid element, which implies that objects in the same row are equally high and objects in the same column equally wide. A much more powerful, but inherently more difficult, variant of this layout manager is the `GridBagLayout` where objects can occupy the space of a number of rows and columns (for references to more information see Section 8).

5 Events and Listeners

Finally, in this section, our GUIs will come to life. They will respond to user actions such as mouse moves and clicks. Additionally, we will learn how close a program gracefully (without Ctrl-C). But before going to the examples we must first learn two new concepts:

- **Event**
  Every time something happens because of user interaction, an *event* is created. For instance, when you click on a button (user interaction) an event is generated by Java.

- **Listener**
  Events are handled and/or processed by *listener* objects. For example, the event generated by your button click is handled by your listener object(s).

Both examples given below show how events can be handled. In the first example, the handling is rather simple; we only have to deal with a single button. The second example is more complicated, because we are managing events caused by buttons, sliders and frames.

To understand the examples, one thing is very important. To be a listener, you will have to implement a listener interface, e.g. `ActionListener` for button events and `WindowListener` for window (frame) events. This leaves us with three options to create listener objects. First, we can use a separate listener object that implements the needed interface. Second, we can use an existing object, for example the parent frame object, to handle the generated events. This option has the advantage of being able to access private member data that is hidden from a separate listener object. The last option is to create an internal class object that handles the events. This internal class can be inherited from an existing class which saves some work. The last two possibilities are illustrated in the examples given below.

Click That Button

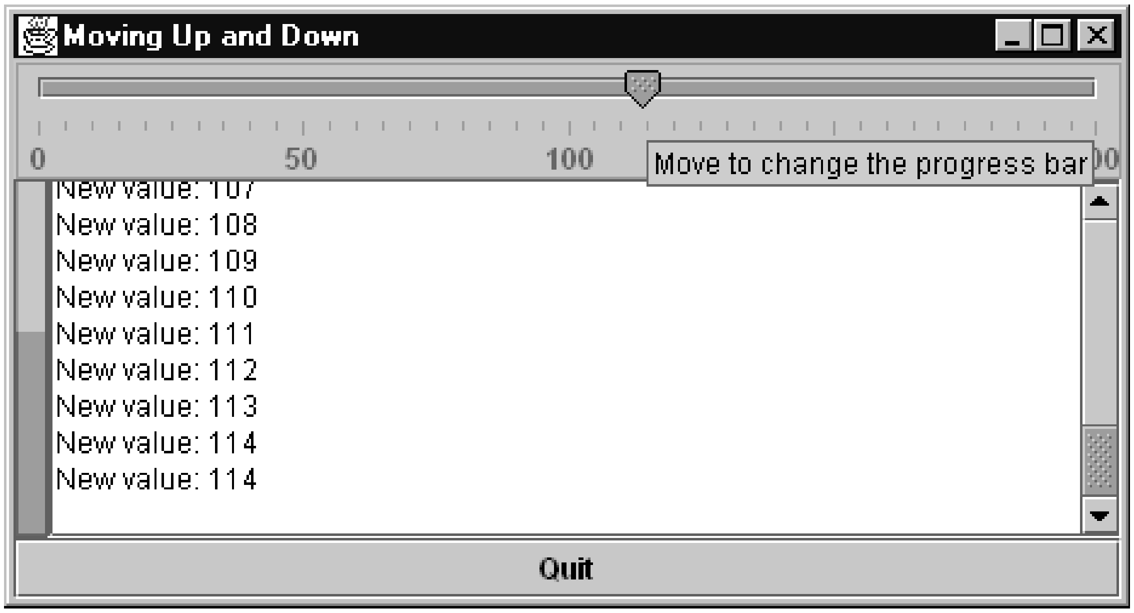
This example creates a button that is sensitive to your mouse clicks. It prints “Beep” to standard output each time you press the button. To achieve this, our newly defined `ClickThatButton` frame implements the `ActionListener` interface (the second option discussed above).

```
 1  // ClickThatButton.java
 2
 3  import com.sun.java.swing.*;
 4  import java.awt.event.*;
 5
 6  class ClickThatButton extends JFrame
 7  implements ActionListener
 8  {
 9      public ClickThatButton()
10      {
11          setTitle("Click That Button");
12          JButton button = new JButton("Beep");
13          button.addActionListener(this);
14          getContentPane().add(button);
15      }
16
17      public void actionPerformed(ActionEvent e)
18      {
19          System.out.println("Beep!");
20      }
21
22      public static void main(String[] args)
23      {
24          JFrame frame = new ClickThatButton();
25          frame.pack();
26          frame.setVisible(true);
27      }
28  }
```

Line 4 imports the event classes needed for this example (most Swing events are the same as AWT event classes). Line 7 specifies that our `ActionListener` interface implements the `ActionListener` interface which is needed to be able to listen to button events. After the button object is created, the `ClickThatButton` frame (`this`) is added as a listener of the button in line 13. The handling of the event is specified in lines 17-20. The implemented method, `actionPerformed(ActionEvent e)`, is the only method that has to be defined to implement the `ActionListener` interface. Because we only have one object generating events and one object functioning as listener, we do not need to check what event has been generated in the `actionPerformed` method; instead we immediately print “Beep”.

Moving Up And Down

The next example is getting more complicated for a number of reasons. First of all, we will use some extra GUI objects and features such as sliders and tool tips. Second, we will listen to events coming from the button, the slider and from the frame itself. And, finally, we will use an internal class to handle some of the generated events.



```
New value: 107
New value: 108
New value: 109
New value: 110
New value: 111
New value: 112
New value: 113
New value: 114
New value: 114
```

Figure 5: A more complicated GUI

The moving up and down example consists of a frame containing four GUI objects, see Figure 5. A horizontal slider (`JSlider`) can be slid from left to right and vice versa by dragging the selector with your mouse. The vertical progress bar on the left reflects the current position of the selector in the slider. The scrollable text area in the middle (scrollbars appear when necessary) prints the current value of the slider. The forth GUI object is a quit button to gracefully exit the application.

```java
// MovingUpAndDown.java

import com.sun.java.swing.*;
import com.sun.java.swing.event.*;
import java.awt.*;
import java.awt.event.*;

class MovingUpAndDown extends JFrame
implements ActionListener, ChangeListener
{
    private JTextArea m_text;
    private JProgressBar m_bar;

    MovingUpAndDown()
    {
        JSlider slider = new JSlider(SwingConstants.HORIZONTAL,
            0, 200, 50);
        slider.setMajorTickSpacing(50);
        slider.setMinorTickSpacing(5);
        slider.setPaintTicks(true);
        slider.setPaintLabels(true);
        slider.setToolTipText("Move to change the progress bar");
        slider.addChangeListener(this);

        m_bar = new JProgressBar();
        m_bar.setOrientation(SwingConstants.VERTICAL);
        m_bar.setMinimum(0);
        m_bar.setMaximum(200);
        m_bar.setValue(50);

        m_text = new JTextArea(5, 40);
        JScrollPane scroll = new JScrollPane(m_text);

        JButton quit = new JButton("Quit");
        quit.setToolTipText("Use this button or windowclose to exit");
        quit.setActionCommand("quit");
        quit.addActionListener(this);

        addWindowListener(new MyWindowAdapter());
        setTitle("Moving Up and Down");
        Container c = getContentPane();
        c.setLayout(new BorderLayout());
        c.add(slider, "North");
        c.add(m_bar, "West");
        c.add(quit, "South");
        c.add(scroll, "Center");
    }

    // ActionListener
    public void actionPerformed(ActionEvent event)
    {
        String command = event.getActionCommand();
        if (command.equals("quit")) quit();
    }

    // ChangeListener
    public void stateChanged(ChangeEvent event)
    {
        JSlider slider = (JSlider) event.getSource();
        int value = slider.getValue();
        m_text.append("New value: "+value+"\n");
        m_bar.setValue(value);
    }

    // WindowListener
    class MyWindowAdapter extends WindowAdapter
    {
        public void windowClosing(WindowEvent event)
        {
            JFrame frame = (JFrame) event.getSource();
            frame.dispose();
            quit();
```

```
72            }
73        }
74
75        private void quit()
76        {
77            System.out.println("Gracefully exiting the application");
78            System.exit(0);
79        }
80
81        public static void main(String args[])
82        {
83            Frame frame = new MovingUpAndDown();
84            frame.pack();
85            frame.setVisible(true);
86        }
87    }
```

OK, now it is going to be a little more difficult. But as soon as you understand the example above, you will be ready to create your own graphical applications. Additionally, by understanding the example, you will be able to understand how to use the other GUI classes available in Swing (and AWT) as well.

Lines 3-6 import all necessary GUI and event classes needed in this example. Line 9 states that our frame implements the *ActionListener* and *ChangeListener* interface to be able to receive button and slider events. Because we have to add text to the text area and modify the value of the progress bar, we must remember the objects as private member variables (lines 11-12)[3].

The constructor of our frame constructs all the widgets we need. Additionally, it takes care of adding the listeners to the event generators. Lines 16 creates a horizontally orientated slider with a minimum of 0, a maximum of 200 and an initial value of 50. Lines 17-20 configure the slider by setting the tick spacing and enabling the display of ticks and labels. The tool tip, specified in line 21, is a kind of post-it note that appears when the mouse is over the GUI object for a short period of time. It can be used to give the user of the application brief, context-sensitive help. The `MovingUpAndDown` frame is added as a listener to events generated by the slider (each time the slider is moved) in line 22.

Lines 24-28 contain the creation of the vertically assigned progress bar. The used code is similar to the code used for creating the slider object, although we store the object in the private member variable `m_bar`.

Lines 30-31 create a scrollable text area. This is done by first creating a text area by itself (we remember the object, because we want to add text later on). After that, we create a `JScrollPane` that contains the text area by supplying the text area as a parameter in the constructor. This implies that we only have to add the scroll pane to the frame instead of the text area itself (see line 45).

Lines 33-36 should be easy to understand by now. The only thing new is line 35, which specifies that the event caused by this button has a command string “quit”. This can be useful when you are listening to a number of buttons using the same listener, such as the parent frame. Using the command strings, it is very easy to determine which button has been activated by the user.

Line 38 is somewhat more difficult again. This line, in combination with the inner class defined in lines 64-73 is the third way of handling events as described above. The object that must handle the events, which are generated by the frame in case of moving, resizing, closing and minimizing the window, is defined by the inner class `MyWindowAdapter` (lines 64-73). The inner class is inherited from the standard `WindowAdapter` class, which implements the `WindowListener` interface with default empty implementations. In our case, we only

[3]Usage of `m_` to indicate private member variables can be very useful to distinguish member variables from local variables.

want to redefine the `windowClosing` event which is called when the user wants to close the frame. A new instance of this class is used to listen to our `MovingUpAndDown` frame in line 38. When the user closes the frame, we delete the frame from the screen in line 70 and stop the application by calling the `private` member function `quit()`. It is important to realize that if `MyWindowAdapter` would have been a separate class, instead of an inner class, this would have been impossible.

The title and contents of the frame are specified in lines 39-45. How this is being done, should be familiar by now.

Lines 48-53 implement the method defined in the `ActionListener` interface. It is called when the quit button has been pressed. In lines 51-52, we check whether we really pressed the quit button by checking the action command string and calling the `quit()` method in case it was true. Lines 55-62 implement the `stateChanged` method defined in the `ChangeListener` interface. Whenever we move the slider, we retrieve the value of the slider by requesting the source of the event (line 58) and getting the current value (line 59). Additionally, in lines 60-61, we print the value in the text area and adapt the value of the progress bar.

6 Using GUI components

In addition to creating all GUI components yourself, Swing has a number of standard, often used, components available. Among these are some dialog classes and a filechooser as the following example illustrates (see Figure 6).

```
// UseIt.java

import com.sun.java.swing.*;
import com.sun.java.swing.preview.*;
import java.io.*;

class UseIt
{
    public static void main(String args[])
    {
        JFrame parent = new JFrame();

        int res = JOptionPane.showConfirmDialog(parent,
            "Do you really want to continue", "Dialog",
            JOptionPane.YES_NO_OPTION);
        if (res != JOptionPane.YES_OPTION) System.exit(0);

        JFileChooser filechooser = new JFileChooser();
        String str;
        if(filechooser.showOpenDialog(parent) ==
                        JFileChooser.APPROVE_OPTION)
        {
            File f = filechooser.getSelectedFile();
            str = "You selected: " + f;
        } else str = "Nothing selected";

        JOptionPane.showMessageDialog(parent, str);

        System.exit(0);
    }
}
```

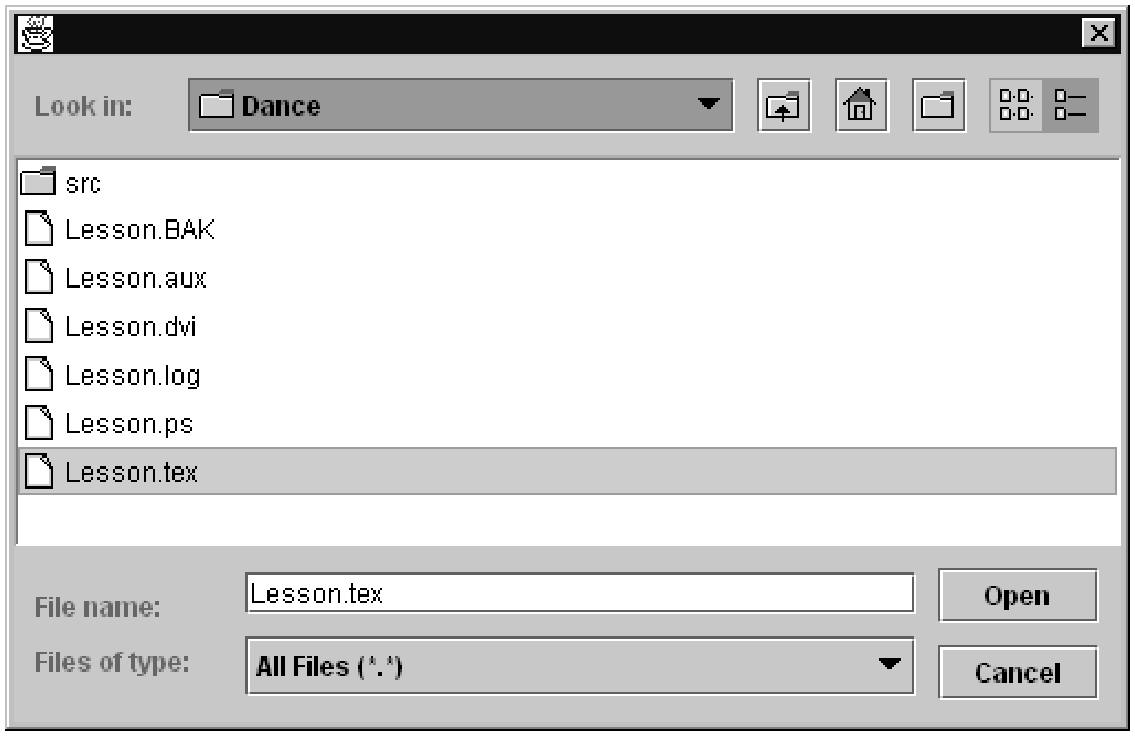


Figure 6: The standard Swing filechooser

The example demonstrates the use of three standard Swing GUI components: the `ConfirmDialog`, the `MessageDialog` and the `FileChooser`. Because the filechooser is not yet final at the time this document has been written, we have to import the file chooser from the preview package in line 4. The program starts by creating a confirm dialog in lines 13-15 by calling the static member function `showConfirmDialog`. The first argument specifies the frame this dialog belongs to; and because we do not have a frame for our application we create an invisible dummy frame in line 11. The second parameter specifies the title of the dialog window. The third parameter contains the text displayed in the dialog and the fourth parameter specifies that we want a 'yes' and 'no' button in the confirm dialog. In addition to the `showConfirmDialog` method used here, a number of variants having different numbers of parameters exist. For information about these variant see the references in Section 8. Only when the user answers 'yes' to the question, will the program continue (line 16).

Lines 18-19 display the file chooser by creating a `JFileChooser` object and opening the dialog box. If the user selects a file the variable `str` will contain the selected file (lines 20-23), otherwise the string "Nothing selected" will be assigned to `str` in line 24.

Finally, line 26 displays a message dialog containing the selected file or a message indicating that nothing has been selected.

7 Dancing on the Web

Java programs can be used in Web browsers by means of an applet. To enable your Swing applications on the Web, you must create an object based on the Swing `JApplet` class. The following example illustrates how this works.

Line 6 specifies that we are creating a class derived from the `JApplet` class. The class has to be public because it is created by a separate application (usually a Web browser). Remember that an AWT `Applet` class exists as well, but when using Swing you must use the `JApplet` class. When creating our own applets, we can redefine three methods: `init`, `start` and `stop` which are called whenever the applet is initialized, started or stopped respectively. This applet consists of a label (line 13) and a combobox (lines 15-23). Note that the calls to standard output (lines 12, 34, 40) will usually print the output in the Java console of the used Web browser.

To use the applet, a JDK1.1 compliant Web browser is necessary[4]. The following HTML page will show how the `WebDance` example can be used.

```
<html>
<h1>Swing on the Web</h1>
<hr>
<applet width=300 height=200
   archive="swingall.jar"
   code="WebDance.class">
</applet>
```

The Java-plugin (see `http://java.sun.com/products/plugin/`) is a Swing-enabled plugin for browsers. This has the advantage that the startup time is much less because we do not have to send all Swing classes over the network. When you want that your applets use the Java-plugin, you must first convert your Web pages using the HTML converter provided by Sun.

8 More information

This ends your first babysteps towards programming graphical user interfaces in Java with Swing. Although the examples described above and the references to more information given below will certainly help you getting started, the only way to really learn how to swing is by doing it yourself...

- **Examples**
  A very efficient way to learn Java and Swing is by looking at a number of examples. In addition to the examples given above, a good example showing the use of Swing is the SwingSet example that comes with the Swing package.
  - `http://www.cs.vu.nl/~bastiaan/DancingLesson/`
  - `<swing-directory>/examples/SwingSet/`
- **The API documentation**
  The JavaAPI and SwingAPI documentation is a large collection of HTML documents containing all information about the available classes, interfaces and methods. The JavaAPI can be found at:
  - `http://java.sun.com/products/jdk/1.1/docs/api/packages.html`

  The SwingAPI can be found at:
  - `http://java.sun.com/products/jfc/swingdoc-api/overview-summary.html`
- **The Swing connection**
  A lot of information about Swing can be found at the Swing homepage, the Swing documentation site and the Swing connection site:

[4]These include on Solaris and Windows NT/95: the Sun applet viewer, Netscape Communicator 4.04/4.05 with the JDK1.1 patch and Microsoft Internet Explorer 4.

  - http://java.sun.com/products/jfc/
  - http://java.sun.com/products/jfc/docs.html
  - http://java.sun.com/products/jfc/swingdoc-current/

- **The JFC/Swing tutorial**
  A tutorial containing information about getting started, an overview of all Swing components as well as a description of each Swing component can be found at:
  - http://java.sun.com/docs/books/tutorial/ui/swing/index.html

- **The Software Engineering home page**
  Most of the information sources mentioned above, will be available at the VU via the following URL:
  - http://www.cs.vu.nl/˜se/