15.5 More on Events and Listeners

- The **`WindowListener`** Interface
  - » By implementing the `WindowListener` interface, a window can be its own listener.
    - – ex) public class ButtonDemo extends Jframe implements WindowListener
  - » The advantage of making a window its own listener is that it is easy to call methods from the listener since they are in the same object.

The `WindowListener` Interface

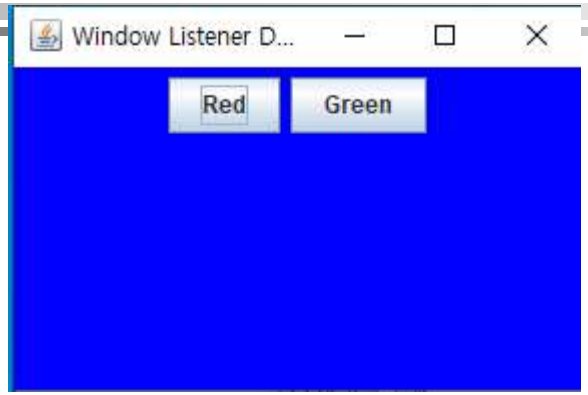
- Implementation of the `WindowListener` interface requires these seven methods to be defined:
  - `public void windowOpened(WindowEvent e)`
  - `public void windowClosing(WindowEvent e)`
  - `public void windowClosed(WindowEvent e)`
  - `public void windowIconified(WindowEvent e)`
  - `public void windowDeiconified(WindowEvent e)`
  - `public void windowActivated(WindowEvent e)`
  - `public void windowDeactivated(WindowEvent e)`
- If a method will be not be used, it should be defined with an empty body
- `WindowAdapter` is a class that implements all seven methods of the `WindowListener` with empty bodies.

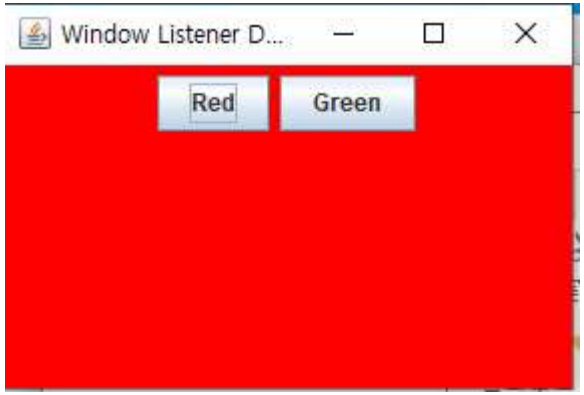
Listing 15.9 A Window Listener - WindowListenerDemo.java

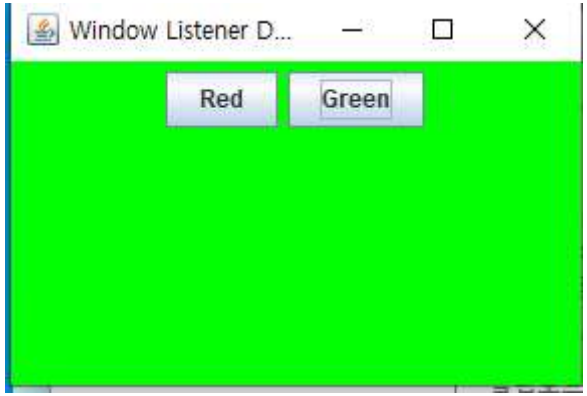
```
// Listing 15.9 A Window Listener

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class WindowListenerDemo extends JFrame
             implements ActionListener, WindowListener
{
   public static final int WIDTH = 300;
   public static final int HEIGHT = 200;

   public static void main(String[] args)
   {
      WindowListenerDemo demoWindow = new WindowListenerDemo( );
      demoWindow.setVisible(true);
   }
```

```
public WindowListenerDemo( )
  {
     setSize(WIDTH, HEIGHT);

     addWindowListener(this);
     setTitle("Window Listener Demonstration");
     Container content = getContentPane( );
     content.setBackground(Color.BLUE);

     content.setLayout(new FlowLayout( ));

     JButton stopButton = new JButton("Red");
     stopButton.addActionListener(this);
     content.add(stopButton);

     JButton goButton = new JButton("Green");
     goButton.addActionListener(this);
     content.add(goButton);
  }

  public void actionPerformed(ActionEvent e)
  {
    Container content = getContentPane( );
    if (e.getActionCommand( ).equals("Red"))
       content.setBackground(Color.RED);
    else if (e.getActionCommand( ).equals("Green"))
       content.setBackground(Color.GREEN);
    else
       System.out.println("Error in WindowListenerDemo.");
  }
```

```
public void windowOpened(WindowEvent e)
  {}

  public void windowClosing(WindowEvent e)
  {
     this.dispose( );
     System.exit(0);
  }

  public void windowClosed(WindowEvent e)
  {}

  public void windowIconified(WindowEvent e)
  {}

  public void windowDeiconified(WindowEvent e)
  {}

  public void windowActivated(WindowEvent e)
  {}

  public void windowDeactivated(WindowEvent e)
  {}
}
```


Window Listener D...
Red
Green


Window Listener D...
Red
Green


Window Listener D...
Red
Green

?? extends WindowAdapter

```
// Listing 15.9 A Window Listener

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class WindowListenerDemo extends Jframe, WindowAdapter
             implements ActionListener
{
   public static final int WIDTH = 300;
   public static final int HEIGHT = 200;

   public static void main(String[] args)
   {
      WindowListenerDemo demoWindow = new WindowListenerDemo( );
      demoWindow.setVisible(true);
   }
```

Dispose()

- Method in the class Jframe (← Frame ←Windows)
- Release any resources used by the window
- (Releases all of the native screen resources used by this Window, its subcomponents, and all of its owned children. That is, the resources for these Components will be destroyed, any memory they consume will be returned to the OS, and they will be marked as undisplayable. The Window and its subcomponents can be made displayable again by rebuilding the native resources with a subsequent call to pack or show. The states of the recreated Window and its subcomponents will be identical to the states of these objects at the point where the Window was disposed (not accounting for additional modifications between those actions).

Changing Components

- A program can add or remove components after a GUI has been displayed, but that is beyond the scope of the book.
- Making components visible or not visible gives a similar effect.
- The `setVisible` method is used in the `VisibleDemo` program to make only one of the red and green labels visible at a time.

  (code on next slide)

Changing Components

The `actionPerformed` method from the `VisibleDemo` program

```
public void actionPerformed(ActionEvent e)
{
  if (e.getActionCommand().equals("Red"))
  {
    colorPanel.setBackground(Color.red);
    stopLabel.setVisible(false);
    goLabel.setVisible(true);
    validate();
  }
. . .
}
```

Visibility changes won't occur until the `validate` method is called.

There is similar code for when the `Green` button is pressed, which turns the background green and hides the go label.

Validate()

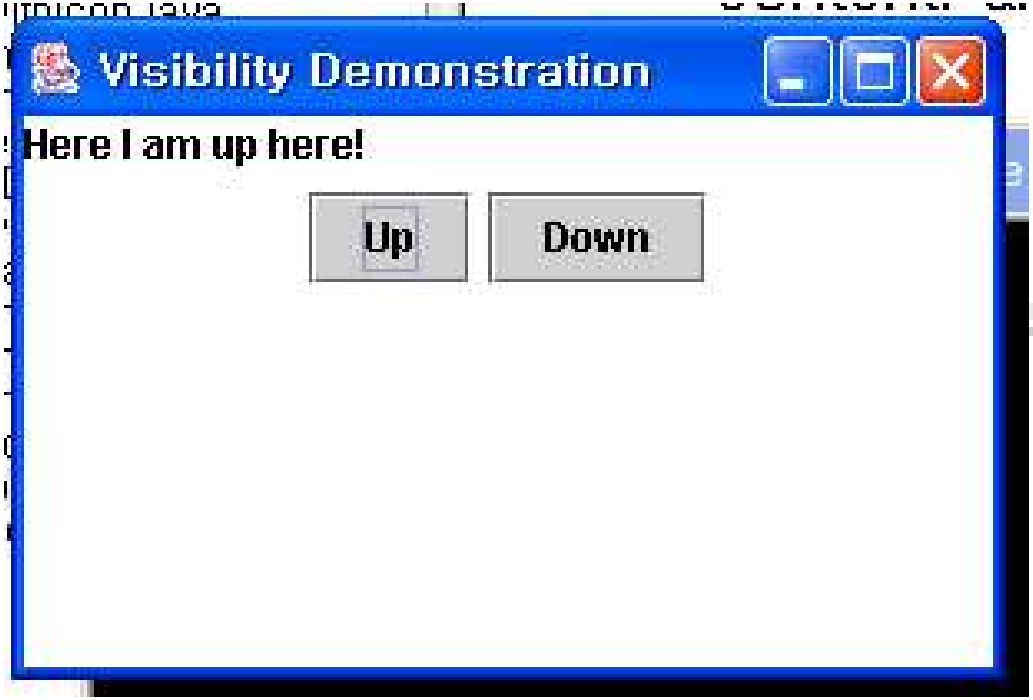
- Every Container class has the method validate()
- An invocation of validate causes the container to lay out its components again.

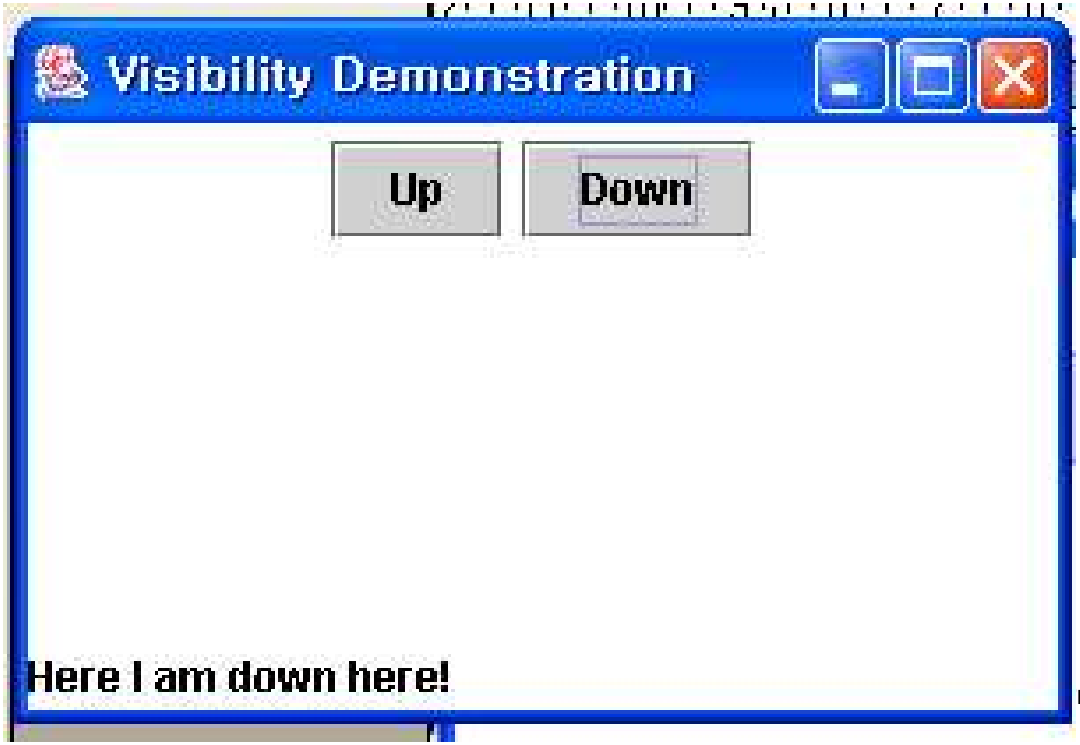
Listing 15.11 Invisible Labels - VisibilityDemo.java

```
//VisibilityDemo.java

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class VisibilityDemo extends JFrame
                        implements ActionListener
{
    public static final int WIDTH = 300;
    public static final int HEIGHT = 200;

    private JLabel upLabel;
    private JLabel downLabel;
```

```
public VisibilityDemo( )
  {
     setSize(WIDTH, HEIGHT);
     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    // Exit the application using the System.exit method
     setTitle("Visibility Demonstration");
     Container contentPane = getContentPane( );
     contentPane.setLayout(new BorderLayout( ));
     contentPane.setBackground(Color.WHITE);

     upLabel = new JLabel("Here I am up here!");
     contentPane.add(upLabel, BorderLayout.NORTH);
     upLabel.setVisible(false);
     downLabel = new JLabel("Here I am down here!");
     contentPane.add(downLabel, BorderLayout.SOUTH);
     downLabel.setVisible(false);

     JPanel buttonPanel = new JPanel( );
     buttonPanel.setBackground(Color.WHITE);
     buttonPanel.setLayout(new FlowLayout( ));
     JButton upButton = new JButton("Up");
     upButton.addActionListener(this);
     buttonPanel.add(upButton);
     JButton downButton = new JButton("Down");
     downButton.addActionListener(this);
     buttonPanel.add(downButton);
     contentPane.add(buttonPanel, BorderLayout.CENTER);
  }
```

```
public void actionPerformed(ActionEvent e)
   {
      if (e.getActionCommand( ).equals("Up"))
      {
         upLabel.setVisible(true);
         downLabel.setVisible(false);
         validate( );
      }
      else if (e.getActionCommand( ).equals("Down"))
      {
         downLabel.setVisible(true);
         upLabel.setVisible(false);
         validate( );
      }
      else
         System.out.println("Error in VisibilityDemo interface.");
   }

   public static void main(String[] args)
   {
      VisibilityDemo demoGui = new VisibilityDemo( );
      demoGui.setVisible(true);
   }
}
```


Visibility Demonstration
Here I am up here!
Up
Down


Visibility Demonstration
Up
Down
Here I am down here!