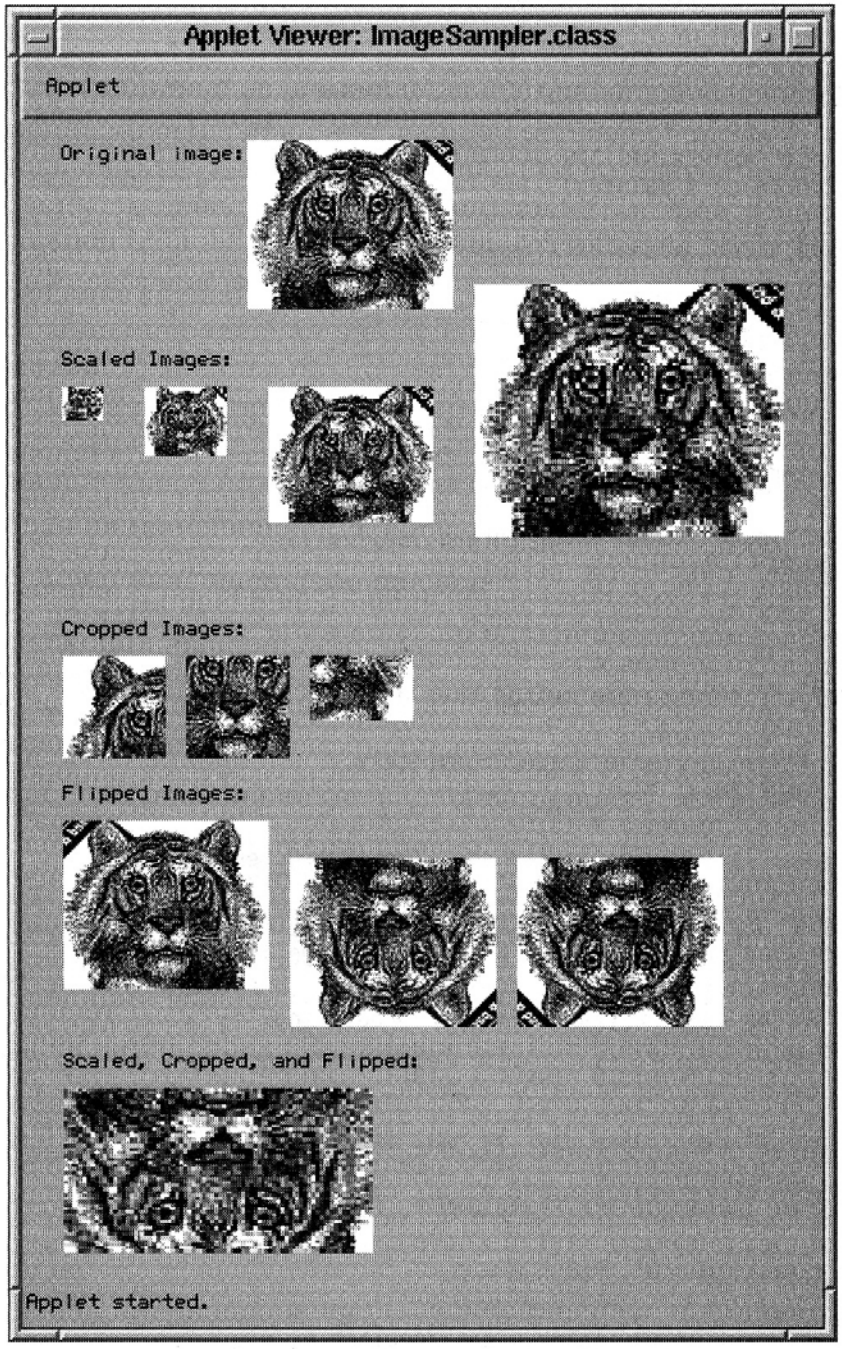
Görüntülerin ölçeklenmesi ve yerleştirilmesi


Applet Viewer: ImageSampler.class
Applet
Original image:
Scaled Images:
Cropped Images:
Flipped Images:
Scaled, Cropped, and Flipped:
Applet started.

Scaled,croped and flipped images

```
import java.applet.*;
import java.awt.*;

/** An applet that demonstrates image scaling, cropping, and flipping */
public class ImageSampler extends Applet {
  Image i;

  /** Load the image */
  public void init() {  i = getImage(this.getDocumentBase(), "tiger.gif"); }

  /** Display the image in a variety of ways */
  public void paint(Graphics g) {
    g.drawString("Original image:", 20, 20);     // Display original image
    g.drawImage(i, 110, 10, this);               // Old version of drawImage()

    g.drawString("Scaled Images:", 20, 120);     // Display scaled images
    g.drawImage(i, 20, 130, 40, 150, 0, 0, 100, 100, this);  // New version
    g.drawImage(i, 60, 130, 100, 170, 0, 0, 100, 100, this);
    g.drawImage(i, 120, 130, 200, 210, 0, 0, 100, 100, this);
    g.drawImage(i, 220, 80, 370, 230, 0, 0, 100, 100, this);
```

Scaled,croped and flipped images

```
    g.drawString("Cropped Images:", 20, 250);    // Display cropped images
    g.drawImage(i, 20, 260, 70, 310, 0, 0, 50, 50, this);
    g.drawImage(i, 80, 260, 130, 310, 25, 25, 75, 75, this);
    g.drawImage(i, 140, 260, 190, 310, 50, 50, 100, 100, this);

    g.drawString("Flipped Images:", 20, 330);    // Display flipped images
    g.drawImage(i, 20, 340, 120, 440, 100, 0, 0, 100, this);
    g.drawImage(i, 130, 340, 230, 440, 0, 100, 100, 0, this);
    g.drawImage(i, 240, 340, 340, 440, 100, 100, 0, 0, this);

    g.drawString("Scaled, Cropped, and Flipped:", 20, 460);  // Do all three
    g.drawImage(i, 20, 470, 170, 550, 90, 70, 10, 20, this);
  }
}
```

Olaylar(Events)

EVENTS

```
import java.applet.*;
import java.awt.*;

/** A simple applet that uses the Java 1.0 event handling model */
public class Scribble1 extends Applet {
  private int lastx, lasty;    // remember last mouse coordinates
  Button clear_button;         // the Clear button
  Graphics g;                  // A Graphics object for drawing

  /** Initialize the button and the Graphics object */
  public void init() {
    clear_button = new Button("Clear");
    this.add(clear_button);
    g = this.getGraphics();
  }
  /** Respond to mouse clicks */
  public boolean mouseDown(Event e, int x, int y) {
    lastx = x; lasty = y;
    return true;
  }
```

```
 /** Respond to mouse drags */
  public boolean mouseDrag(Event e, int x, int y) {
    g.setColor(Color.black);
    g.drawLine(lastx, lasty, x, y);
    lastx = x; lasty = y;
    return true;
  }
  /** Respond to key presses */
  public boolean keyDown(Event e, int key) {
    if ((e.id == Event.KEY_PRESS) && (key == 'c')) {
      clear();
      return true;
    }
    else return false;
  }
  /** Respond to Button clicks */
  public boolean action(Event e, Object arg) {
    if (e.target == clear_button) {
      clear();
      return true;
    }
    else return false;
  }
  /** convenience method to erase the scribble */
  public void clear() {
    g.setColor(this.getBackground());
    g.fillRect(0, 0, bounds().width, bounds().height);
  }
}
```

```
import java.applet.*;
import java.awt.*;
import java.util.*;

/** An applet that gives details about Java 1.0 events */
public class EventTester1 extends Applet {
 // Handle mouse events
 public boolean mouseDown(Event e, int x, int y)  {
  showLine(mods(e.modifiers) +  "Mouse Down: [" + x + "," + y + "]");
  return true;
 }
 public boolean mouseUp(Event e, int x, int y)  {
  showLine(mods(e.modifiers) + "Mouse Up: [" + x + "," + y + "]");
  return true;
 }
 public boolean mouseDrag(Event e, int x, int y)  {
  showLine(mods(e.modifiers) + "Mouse Drag: [" + x + "," + y + "]");
  return true;
 }
 public boolean mouseMove(Event e, int x, int y) {
  showLine(mods(e.modifiers) + "Mouse Move: [" + x + "," + y + "]");
  return true;
 }
 public boolean mouseEnter(Event e, int x, int y)  {
  showLine("Mouse Enter: [" + x + "," + y + "]"); return true;
 }
 public boolean mouseExit(Event e, int x, int y)  {
  showLine("Mouse Exit: [" + x + "," + y + "]"); return true;
 }
```

```
// Handle focus events
 public boolean gotFocus(Event e, Object what)  {
  showLine("Got Focus"); return true;
 }
 public boolean lostFocus(Event e, Object what)  {
  showLine("Lost Focus"); return true;
 }

 // Handle key down and key up events
 // This gets more confusing because there are two types of key events
 public boolean keyDown(Event e, int key)  {
  int flags = e.modifiers;
  if (e.id == Event.KEY_PRESS)               // a regular key
    showLine("Key Down: " + mods(flags) + key_name(e));
  else if (e.id == Event.KEY_ACTION)         // a function key
    showLine("Function Key Down: " + mods(flags) + function_key_name(key));
  return true;
 }
 public boolean keyUp(Event e, int key)  {
  int flags = e.modifiers;
  if (e.id == Event.KEY_RELEASE)             // a regular key
    showLine("Key Up: " + mods(flags) + key_name(e));
  else if (e.id == Event.KEY_ACTION_RELEASE)   // a function key
    showLine("Function Key Up: " + mods(flags) + function_key_name(key));
  return true;
 }
```

```
// Return the current list of modifier keys
 private String mods(int flags) {
  String s = "[ ";
  if (flags == 0) return "";
  if ((flags & Event.SHIFT_MASK) != 0) s += "Shift ";
  if ((flags & Event.CTRL_MASK) != 0) s += "Control ";
  if ((flags & Event.META_MASK) != 0) s += "Meta ";
  if ((flags & Event.ALT_MASK) != 0) s += "Alt ";
  s += "] ";
  return s;
 }

 // Return the name of a regular (non-function) key.
 private String key_name(Event e) {
  char c = (char) e.key;
  if (e.controlDown()) {    // If CTRL flag is set, handle control chars.
    if (c < ' ') {
      c += '@';
      return "^" + c;
    }
  }
  else {                  // If CTRL flag is not set, then certain ASCII
    switch (c) {          // control characters have special meaning.
      case '\n': return "Return";
      case '\t': return "Tab";
      case '\033': return "Escape";
      case '\010': return "Backspace";
    }
  }
```

```
 // Handle the remaining possibilities.
   if (c == '\177') return "Delete";
   else if (c == ' ') return "Space";
   else return String.valueOf(c);
 }
 // Return the name of a function key.  Just compare the key to the
 // constants defined in the Event class.
 private String function_key_name(int key) {
   switch(key) {
     case Event.HOME: return "Home";     case Event.END: return "End";
     case Event.PGUP: return "Page Up";  case Event.PGDN: return "Page Down";
     case Event.UP: return "Up";         case Event.DOWN: return "Down";
     case Event.LEFT: return "Left";     case Event.RIGHT: return "Right";
     case Event.F1: return "F1";         case Event.F2: return "F2";
     case Event.F3: return "F3";         case Event.F4: return "F4";
     case Event.F5: return "F5";         case Event.F6: return "F6";
     case Event.F7: return "F7";         case Event.F8: return "F8";
     case Event.F9: return "F9";         case Event.F10: return "F10";
     case Event.F11: return "F11";       case Event.F12: return "F12";
   }
   return "Unknown Function Key";
 }

 /** A list of lines to display in the window */
 protected Vector lines = new Vector();
 /** Add a new line to the list of lines, and redisplay */
 protected void showLine(String s) {
   if (lines.size() == 20) lines.removeElementAt(0);
   lines.addElement(s);
   repaint();
 }
 /** This method repaints the text in the window */
 public void paint(Graphics g) {
   for(int i = 0; i < lines.size(); i++)
     g.drawString((String)lines.elementAt(i), 20, i*16 + 50);
 }
}
```

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

/** A simple applet that uses the Java 1.1 event handling model */
public class Scribble2 extends Applet
                implements MouseListener,  MouseMotionListener {
 private int last_x, last_y;

 public void init() {
  // Tell this applet what MouseListener and MouseMotionListener
  // objects to notify when mouse and mouse motion events occur.
  // Since we implement the interfaces ourself, our own methods are called.
  this.addMouseListener(this);
  this.addMouseMotionListener(this);
 }

 // A method from the MouseListener interface.  Invoked when the
 // user presses a mouse button.
 public void mousePressed(MouseEvent e) {
  last_x = e.getX();
  last_y = e.getY();
 }

 // A method from the MouseMotionListener interface.  Invoked when the
 // user drags the mouse with a button pressed.
 public void mouseDragged(MouseEvent e) {
  Graphics g = this.getGraphics();
  int x = e.getX(), y = e.getY();
  g.drawLine(last_x, last_y, x, y);
  last_x = x; last_y = y;
 }
```

```
 // The other, unused methods of the MouseListener interface.
  public void mouseReleased(MouseEvent e) {;}
  public void mouseClicked(MouseEvent e) {;}
  public void mouseEntered(MouseEvent e) {;}
  public void mouseExited(MouseEvent e) {;}

  // The other method of the MouseMotionListener interface.
  public void mouseMoved(MouseEvent e) {;}
}
```

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

/**
 * A simple applet that uses external classes to implement
 * the Java 1.1 event handling model
 **/
public class Scribble3 extends Applet {
  int last_x;
  int last_y;

  public void init() {
    MouseListener ml = new MyMouseListener(this);
    MouseMotionListener mml = new MyMouseMotionListener(this);

    // Tell this component what MouseListener and MouseMotionListener
    // objects to notify when mouse and mouse motion events occur.
    this.addMouseListener(ml);
    this.addMouseMotionListener(mml);
  }
}

class MyMouseListener extends MouseAdapter {
  private Scribble3 scribble;
  public MyMouseListener(Scribble3 s) { scribble = s; }
  public void mousePressed(MouseEvent e)  {
    scribble.last_x = e.getX();
    scribble.last_y = e.getY();
  }
}
```

```
class MyMouseMotionListener extends MouseMotionAdapter {
  private Scribble3 scribble;
  public MyMouseMotionListener(Scribble3 s) { scribble = s; }
  public void mouseDragged(MouseEvent e) {
    Graphics g = scribble.getGraphics();
    int x = e.getX(), y = e.getY();
    g.drawLine(scribble.last_x, scribble.last_y, x, y);
    scribble.last_x = x; scribble.last_y = y;
  }
}
```

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

/**
 * A simple applet that uses anonymous inner classes to implement
 * the Java 1.1 event handling model
 **/
public class Scribble4 extends Applet {
  int last_x, last_y;

  public void init() {
    // Define, instantiate and register a MouseListener object
    this.addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        last_x = e.getX();
        last_y = e.getY();
      }
    });

    // Define, instantiate and register a MouseMotionListener object
    this.addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        Graphics g = getGraphics();
        int x = e.getX(), y = e.getY();
        g.setColor(Color.black);
        g.drawLine(last_x, last_y, x, y);
        last_x = x; last_y = y;
      }
    });
```

```
 // Create a clear button
    Button b = new Button("Clear");
    // Define, instantiate, and register a listener to handle button presses
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {  // clear the scribble
        Graphics g = getGraphics();
        g.setColor(getBackground());
        g.fillRect(0, 0, getSize().width, getSize().height);
      }
    });
    // And add the button to the applet
    this.add(b);
  }
}
```

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

/** The application class.  Processes high-level commands sent by GUI */
public class Scribble5 {
 /** main entry point.  Just create an instance of this application class */
 public static void main(String[] args) { new Scribble5(); }

 /** Application constructor:  create an instance of our GUI class */
 public Scribble5() { window = new ScribbleGUI(this); }
 protected Frame window;

 /** This is the application method that processes commands sent by the GUI */
 public void doCommand(String command) {
  if (command.equals("clear")) {         // clear the GUI window
    // It would be more modular to include this functionality in the GUI
    // class itself.  But for demonstration purposes, we do it here.
    Graphics g = window.getGraphics();
    g.setColor(window.getBackground());
    g.fillRect(0, 0, window.getSize().width, window.getSize().height);
  }
  else if (command.equals("print")) {}    // not yet implemented
  else if (command.equals("quit")) {      // quit the application
    window.dispose();                       // close the GUI
    System.exit(0);                       // and exit.
  }
 }
}
```

```
/** This class implements the GUI for our application */
class ScribbleGUI extends Frame {
  int lastx, lasty;   // remember last mouse click
  Scribble5 app;      // A reference to the application, to send commands to.

  /**
   * The GUI constructor does all the work of creating the GUI and setting
   * up event listeners.  Note the use of local and anonymous classes.
   */
  public ScribbleGUI(Scribble5 application) {
    super("Scribble");       // Create the window
    app = application;       // Remember the application reference

    // Create three buttons
    Button clear = new Button("Clear");
    Button print = new Button("Print");
    Button quit = new Button("Quit");

    // Set a LayoutManager, and add the buttons to the window.
    this.setLayout(new FlowLayout(FlowLayout.RIGHT, 10, 5));
    this.add(clear); this.add(print);  this.add(quit);

    // Here's a local class used for action listeners for the buttons
    class ScribbleActionListener implements ActionListener {
      private String command;
      public ScribbleActionListener(String cmd) { command = cmd; }
      public void actionPerformed(ActionEvent e) { app.doCommand(command); }
    }
```

```
// Define action listener adapters that connect the  buttons to the app
   clear.addActionListener(new ScribbleActionListener("clear"));
   print.addActionListener(new ScribbleActionListener("print"));
   quit.addActionListener(new ScribbleActionListener("quit"));

   // Handle the window close request similarly
   this.addWindowListener(new WindowAdapter() {
     public void windowClosing(WindowEvent e) { app.doCommand("quit"); }
   });

   // High-level action events are passed to the application, but we
   // still handle scribbling right here.  Register a MouseListener object.
   this.addMouseListener(new MouseAdapter() {
     public void mousePressed(MouseEvent e) {
       lastx = e.getX(); lasty = e.getY();
     }
   });

   // Define, instantiate and register a MouseMotionListener object
   this.addMouseMotionListener(new MouseMotionAdapter() {
     public void mouseDragged(MouseEvent e) {
       Graphics g = getGraphics();
       int x = e.getX(), y = e.getY();
       g.setColor(Color.black);
       g.drawLine(lastx, lasty, x, y);
       lastx = x; lasty = y;
     }
   });

   // Finally, set the size of the window, and pop it up
   this.setSize(400, 400);
   this.show();
 }
}
```

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

/** A simple applet that uses low-level event handling under Java 1.1 */
public class Scribble6 extends Applet {
 private int lastx, lasty;

 /** Tell the system we're interested in mouse events, mouse motion events,
  *  and keyboard events.  This is a required or events won't be sent.
  */
 public void init() {
  this.enableEvents(AWTEvent.MOUSE_EVENT_MASK |
                    AWTEvent.MOUSE_MOTION_EVENT_MASK |
                    AWTEvent.KEY_EVENT_MASK);
  this.requestFocus();  // Ask for keyboard focus so we get key events
 }

 /** Invoked when a mouse event of some type occurs */
 public void processMouseEvent(MouseEvent e) {
  if (e.getID() == MouseEvent.MOUSE_PRESSED) {  // check the event type
    lastx = e.getX(); lasty = e.getY();
  }
  else super.processMouseEvent(e); // pass unhandled events to our superclass
 }
```

```
/** Invoked when a mouse motion event occurs */
  public void processMouseMotionEvent(MouseEvent e) {
    if (e.getID() == MouseEvent.MOUSE_DRAGGED) {  // check type
      int x = e.getX(), y = e.getY();
      Graphics g = this.getGraphics();
      g.drawLine(lastx, lasty, x, y);
      lastx = x; lasty = y;
    }
    else super.processMouseMotionEvent(e);
  }

  /** Called on key events:  clear the screen when 'c' is typed */
  public void processKeyEvent(KeyEvent e) {
    if ((e.getID() == KeyEvent.KEY_TYPED) && (e.getKeyChar() == 'c')) {
      Graphics g = this.getGraphics();
      g.setColor(this.getBackground());
      g.fillRect(0, 0, this.getSize().width, this.getSize().height);
    }
    else super.processKeyEvent(e);  // pass unhandled events to our superclass
  }
}
```

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

/** A simple applet that uses low-level event handling under Java 1.1 */
public class Scribble7 extends Applet {
 private int lastx, lasty;

 /** Specify the event types we care about, and ask for keyboard focus */
 public void init() {
  this.enableEvents(AWTEvent.MOUSE_EVENT_MASK |
                    AWTEvent.MOUSE_MOTION_EVENT_MASK |
                    AWTEvent.KEY_EVENT_MASK);
  this.requestFocus();  // Ask for keyboard focus so we get key events
 }

 /**
  * Called when an event arrives.  Do the right thing based on the event
  * type.  Pass unhandled events to the superclass for possible processing
  */
```

```
 public void processEvent(AWTEvent e) {
   MouseEvent me;
   Graphics g;
   switch(e.getID()) {
   case MouseEvent.MOUSE_PRESSED:
     me = (MouseEvent)e;
     lastx = me.getX(); lasty = me.getY();
     break;
   case MouseEvent.MOUSE_DRAGGED:
     me = (MouseEvent)e;
     int x = me.getX(), y = me.getY();
     g = this.getGraphics();
     g.drawLine(lastx, lasty, x, y);
     lastx = x; lasty = y;
     break;
   case KeyEvent.KEY_TYPED:
     if (((KeyEvent)e).getKeyChar() == 'c') {
       g = this.getGraphics();
       g.setColor(this.getBackground());
       g.fillRect(0, 0, this.getSize().width, this.getSize().height);
     }
     else super.processEvent(e);
     break;
   default: super.processEvent(e); break;
   }
  }
}
```

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

/** A program that displays all the event that occur in its window */
public class EventTester2 extends Frame
{
 /** The main method: create an EventTester frame, and pop it up */
 public static void main(String[] args) {
  EventTester2 et = new EventTester2();
  et.setSize(500, 400);
  et.show();
 }

 /** The constructor: register the event types we are interested in */
 public EventTester2() {
  super("Event Tester");
  this.enableEvents(AWTEvent.MOUSE_EVENT_MASK |
                    AWTEvent.MOUSE_MOTION_EVENT_MASK |
                    AWTEvent.KEY_EVENT_MASK |
                    AWTEvent.FOCUS_EVENT_MASK |
                    AWTEvent.COMPONENT_EVENT_MASK |
                    AWTEvent.WINDOW_EVENT_MASK);
 }
```

```
/**
  * Display mouse events that don't involve mouse motion.
  * The mousemods() method prints modifiers, and is defined below.
  * The other methods return additional information about the mouse event.
  * showLine() displays a line of text in the window.  It is defined
  * at the end of this class, along with the paint() method.
  */
 public void processMouseEvent(MouseEvent e) {
  String type = null;
  switch(e.getID()) {
  case MouseEvent.MOUSE_PRESSED:   type = "MOUSE_PRESSED"; break;
  case MouseEvent.MOUSE_RELEASED:  type = "MOUSE_RELEASED"; break;
  case MouseEvent.MOUSE_CLICKED:   type = "MOUSE_CLICKED"; break;
  case MouseEvent.MOUSE_ENTERED:   type = "MOUSE_ENTERED"; break;
  case MouseEvent.MOUSE_EXITED:    type = "MOUSE_EXITED"; break;
  }
  showLine(mousemods(e) + type + ": [" + e.getX() + "," + e.getY() + "] " +
         "num clicks = " + e.getClickCount() +
         (e.isPopupTrigger()?"; is popup trigger":""));
 }

 /**
  * Display mouse moved and dragged mouse event.  Note that MouseEvent
  * is the only event type that has two methods, two EventListener interfaces
  * and two adapter classes to handle two distinct categories of events.
  * Also, as seen in init(), mouse motion events must be requested
  * separately from other mouse event types.
  */
```

```
public void processMouseMotionEvent(MouseEvent e) {
   String type = null;
   switch(e.getID()) {
   case MouseEvent.MOUSE_MOVED:   type = "MOUSE_MOVED"; break;
   case MouseEvent.MOUSE_DRAGGED: type = "MOUSE_DRAGGED"; break;
   }
   showLine(mousemods(e) + type + ": [" + e.getX() + "," + e.getY() + "] " +
         "num clicks = " + e.getClickCount() +
         (e.isPopupTrigger()?"; is popup trigger":""));
 }
 /** Return a string representation of the modifiers for a MouseEvent.
  *  Note that the methods called here are inherited from InputEvent.
  */
 protected String mousemods(MouseEvent e) {
   int mods = e.getModifiers();
   String s = "";
   if (e.isShiftDown()) s += "Shift ";
   if (e.isControlDown()) s += "Ctrl ";
   if ((mods & InputEvent.BUTTON1_MASK) != 0) s += "Button 1 ";
   if ((mods & InputEvent.BUTTON2_MASK) != 0) s += "Button 2 ";
   if ((mods & InputEvent.BUTTON3_MASK) != 0) s += "Button 3 ";
   return s;
 }
 /**
  * Display keyboard events.
  * Note that there are three distinct types of key events, and that
  * key events are reported by key code and/or Unicode character.
  * KEY_PRESSED and KEY_RELEASED events are generated for all key strokes.
  * KEY_TYPED events are only generated when a key stroke produces a
  * Unicode character; these events do not report a key code.
  * If isActionKey() returns true, then the key event reports only
  * a key code, because the key that was pressed or released (such as a
  * function key) has no corresponding Unicode character.
  * Key codes can be interpreted by using the many VK_ constants defined
  * by the KeyEvent class, or they can be converted to strings using
  * the static getKeyText() method as we do here.
  */
```

```
public void processKeyEvent(KeyEvent e) {
   String eventtype, modifiers, code, character;
   switch(e.getID()) {
   case KeyEvent.KEY_PRESSED:  eventtype = "KEY_PRESSED"; break;
   case KeyEvent.KEY_RELEASED: eventtype = "KEY_RELEASED"; break;
   case KeyEvent.KEY_TYPED:    eventtype = "KEY_TYPED"; break;
   default: eventtype = "UNKNOWN";
   }

   // Convert the list of modifier keys to a string
   modifiers = KeyEvent.getKeyModifiersText(e.getModifiers());

   // Get string and numeric versions of the key code, if any.
   if (e.getID() == KeyEvent.KEY_TYPED) code = "";
   else code = "Code=" + KeyEvent.getKeyText(e.getKeyCode()) +
         " (" + e.getKeyCode() + ")";

   // Get string and numeric versions of the Unicode character, if any.
   if (e.isActionKey()) character = "";
   else character = "Character=" + e.getKeyChar() +
         " (Unicode=" + ((int)e.getKeyChar()) + ")";

   // Display it all.
   showLine(eventtype + ": " + modifiers + " " + code + " " + character);
 }

 /** Display keyboard focus events.  Focus can be permanently
  *  gained or lost, or temporarily transferred to or from a component. */
 public void processFocusEvent(FocusEvent e) {
   if (e.getID() == FocusEvent.FOCUS_GAINED)
     showLine("FOCUS_GAINED" + (e.isTemporary()?" (temporary)":""));
   else
     showLine("FOCUS_LOST" + (e.isTemporary()?" (temporary)":""));
 }
```

```
/** Display Component events.  */
 public void processComponentEvent(ComponentEvent e) {
  switch(e.getID()) {
  case ComponentEvent.COMPONENT_MOVED: showLine("COMPONENT_MOVED"); break;
  case ComponentEvent.COMPONENT_RESIZED: showLine("COMPONENT_RESIZED");break;
  case ComponentEvent.COMPONENT_HIDDEN: showLine("COMPONENT_HIDDEN"); break;
  case ComponentEvent.COMPONENT_SHOWN: showLine("COMPONENT_SHOWN"); break;
  }
 }

 /** Display Window events.  Note the special handling of WINDOW_CLOSING */
 public void processWindowEvent(WindowEvent e) {
  switch(e.getID()) {
  case WindowEvent.WINDOW_OPENED: showLine("WINDOW_OPENED"); break;
  case WindowEvent.WINDOW_CLOSED: showLine("WINDOW_CLOSED"); break;
  case WindowEvent.WINDOW_CLOSING: showLine("WINDOW_CLOSING"); break;
  case WindowEvent.WINDOW_ICONIFIED: showLine("WINDOW_ICONIFIED"); break;
  case WindowEvent.WINDOW_DEICONIFIED: showLine("WINDOW_DEICONIFIED"); break;
  case WindowEvent.WINDOW_ACTIVATED: showLine("WINDOW_ACTIVATED"); break;
  case WindowEvent.WINDOW_DEACTIVATED: showLine("WINDOW_DEACTIVATED"); break;
  }

  // If the user requested a window close, quit the program.
  // But first display a message, force it to be visible, and make
  // sure the user has time to read it.
  if (e.getID() == WindowEvent.WINDOW_CLOSING) {
   showLine("WINDOW_CLOSING event received.");
   showLine("Application will exit in 5 seconds");
   update(this.getGraphics());
   try {Thread.sleep(5000);} catch (InterruptedException ie) { ; }
   System.exit(0);
  }
 }
```

```
/** The list of lines to display in the window */
  protected Vector lines = new Vector();

  /** Add a new line to the list of lines, and redisplay */
  protected void showLine(String s) {
    if (lines.size() == 20) lines.removeElementAt(0);
    lines.addElement(s);
    repaint();
  }

  /** This method repaints the text in the window */
  public void paint(Graphics g) {
    for(int i = 0; i < lines.size(); i++)
      g.drawString((String)lines.elementAt(i), 20, i*16 + 50);
  }
}
```