MULTIMEDIA PROGRAMMING IN JAVA

Prof.Asoc. Alda Kika
Department of Informatics
Faculty of Natural Sciences
University of Tirana

Objectives

- Applets in Java
- Getting, displaying and scaling the image
- How to create animations from a sequence of images
- How to create image maps
- How to get, play and stop the music using AudioClip
- How to play video using Player interface

Introduction

- Multimedia—using sound, images, graphics, animation and video—makes applications- “come alive.”

- Several examples to show:
  - the basics of manipulating images.
  - creating smooth animations.
  - playing audio files with the AudioClip interface.
  - creating image maps that can sense when the cursor is over them, even without a mouse click.
  - playing video files using the Player interface.
- Also JNLP features that, with the user’s permission, enable an applet or application to access files on the user’s local computer.

Apletet

- An **applet** is a special kind of **Java program** that a browser enabled with Java technology can download from the internet and run.

- An **applet** is typically **embedded inside a web page** and **runs in the context of a browser**.

- When a browser loads a Web page containing an applet, the applet downloads into the Web browser and executes. The **browser** that executes an applet is generically called the **applet container**.

- **Appletviewer** is also an **applet container** that comes with the **JDK**. It can be used for testing applets as you develop them and before embedding them in Web pages.

Construction of Applets

- An applet must be a subclass of the **java.applet.Applet** class.
- The Applet class provides the standard interface between the applet and the browser environment.
- Swing provides a special subclass of the Applet class called **javax.swing.JApplet**.

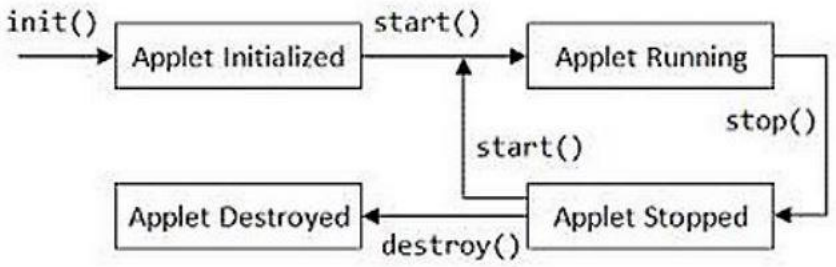
- The **JApplet** class should be used for all applets that use **Swing components** to construct their graphical user interfaces (**GUI**s).

- The browser's **Java Plug-in software** manages the lifecycle of an applet.

Applet lifecycle

Various **methods** are called by the browser **JVM** at different times:

- **init()** initializes the applet
- **start()** starts the applet
- **paint(), repaint(), update()** involved in drawing the applet
- **stop()** stops the applets
- **destroy()** closes the applet



- By **overriding these methods** in your applet subclass, you decide what these methods do.

Applet security

- **Applets cannot**

- Read/write to or from the file system
- Access your network
- Launch applications
- Launch new windows without a warning message
- Go to other websites and download files to your machine

Sandbox Security Model

Client protection from malicious applets

- The Java platform uses the sandbox security model to prevent code that is downloaded to your local computer from accessing local system resources, such as files.

- Code executing in the sandbox is not allowed to “play outside the sandbox”

Sandbox Security Model

Client protection from malicious applets

• Applets can be **signed** with a **digital signature (security certificate)** to indicate that its from a trusted source) and certified to relax security. These applets have extensive capabilities to access the client, but only if the user accepts the applet's security certificate.

• On the other hand, **unsigned applets** operate within a **security sandbox** that allows only a set of safe operations.

Sandbox Security Model

Client protection from malicious applets

Note:

• When a **signed applet** is accessed from **JavaScript** code in an **HTML page**, the applet is executed within the **security sandbox**.

• This implies that the signed applet essentially behaves likes an unsigned applet.

Java Web plug-in

- Version conflicts is one disadvantage of using applets
- To avoid version problems, associated with the Java applets, there is the Java plugin
- This is an up-to-date virtual machine that can be installed on your machine

With recent improvements to the Java Plug-in software, unsigned applets launched using **Java Network Launch Protocol (JNLP)** can safely access the client with the user's permission.

Applet Deployment

- **Applets can be launched in two ways:**

1. You can launch an applet by specifying the applet's launch properties directly in the **<applet>** tag. This old way of deploying applets imposes severe security restrictions on the applet.

2. Alternatively, you can launch your applet by using **Java Network Launch Protocol (JNLP)**. Applets launched by using JNLP have access to powerful JNLP APIs and extensions.

Loading, Displaying and Scaling Images

- Java Web Start and the JNLP FileOpenService (package javax.jnlp) allow the user to select an image, then displays that image and allows the user to scale it.
- After the user selects an image, the applet gets the bytes from the file, then passes them to the ImageIcon (package javax.swing) constructor to create the image that will be displayed.
- Class ImageIcon's constructors can receive arguments of several different formats, including a byte array containing the bytes of an image, an Image (package java.awt) already loaded in memory, or a String or a URL representing the image's location.
- Java supports various image formats, including Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG) and Portable Network Graphics (PNG).

```
1  // Fig. 24.1: LoadImageAndScale.java
2  // Loading, displaying and scaling an image in an applet
3  import java.awt.BorderLayout;
4  import java.awt.Graphics;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7  import javax.jnlp.FileContents;
8  import javax.jnlp.FileOpenService;
9  import javax.jnlp.ServiceManager;
10 import javax.swing.ImageIcon;
11 import javax.swing.JApplet;
12 import javax.swing.JButton;
13 import javax.swing.JFrame;
14 import javax.swing.JLabel;
15 import javax.swing.JOptionPane;
16 import javax.swing.JPanel;
17 import javax.swing.JTextField;
18
19 public class LoadImageAndScale extends JApplet
20 {
21    private ImageIcon image; // references image to display
22    private JPanel scaleJPanel; // JPanel containing the scale-selector
23    private JLabel percentJLabel; // label for JTextField
```

JNLP services for accessing local files if the user gives permission

Fig. 24.1 | Loading, displaying and scaling an image in an applet. (Part 1 of 7.)

```
24      private JTextField scaleInputJTextField; // obtains user's input
25      private JButton scaleChangeJButton; // initiates scaling of image
26      private double scaleValue = 1.0;  //scale percentage for image
27
28      // load image when applet is loaded
29      public void init()
30      {
31         scaleJPanel = new JPanel();
32         percentJLabel = new JLabel( "scale percent:" );
33         scaleInputJTextField = new JTextField( "100" );
34         scaleChangeJButton = new JButton( "Set Scale" );
35
36         // add components and place scaleJPanel in applet's NORTH region
37         scaleJPanel.add( percentJLabel );
38         scaleJPanel.add( scaleInputJTextField );
39         scaleJPanel.add( scaleChangeJButton );
40         add( scaleJPanel, BorderLayout.NORTH );
41
42         // register event handler for scaleChangeJButton
43         scaleChangeJButton.addActionListener(
44            new ActionListener()
45            {
```

Fig. 24.1 | Loading, displaying and scaling an image in an applet. (Part 2 of 7.)

```
46              // when the JButton is pressed, set scaleValue and repaint
47              public void actionPerformed( ActionEvent e )
48              {
49                 scaleValue = Double.parseDouble(
50                    scaleInputJTextField.getText() ) / 100.0;
51                 repaint(); // causes image to be redisplyed at new scale
52              } // end method actionPerformed
53           } // end anonymous inner class
54        ); // end call to addActionListener
55
56        // use JNLP services to open an image file that the user selects
57        try
58        {
59           // get a reference to the FileOpenService
60           FileOpenService fileOpenService =
61              (FileOpenService) ServiceManager.lookup(
62                 "javax.jnlp.FileOpenService" );
63
64           // get file's contents from the FileOpenService
65           FileContents contents =
66              fileOpenService.openFileDialog( null, null );
67
```

This service provides limited access to the local file system

This object enables the user to select a file

Fig. 24.1 | Loading, displaying and scaling an image in an applet. (Part 3 of 7.)

```
68            // byte array to store image's data
69            byte[] imageData = new byte[ (int) contents.getLength() ];
70            contents.getInputStream().read( imageData ); // read image bytes
71            image = new ImageIcon( imageData ); // create the image
72
73            // if image successfully loaded, create and add DrawJPanel
74            add( new DrawJPanel(), BorderLayout.CENTER );
75         } // end try
76         catch( Exception e )
77         {
78            e.printStackTrace();
79         } // end catch
80      } // end method init
81
82      // DrawJPanel used to display loaded image
83      private class DrawJPanel extends JPanel
84      {
85         // display image
86         public void paintComponent( Graphics g )
87         {
88            super.paintComponent( g );
89
```

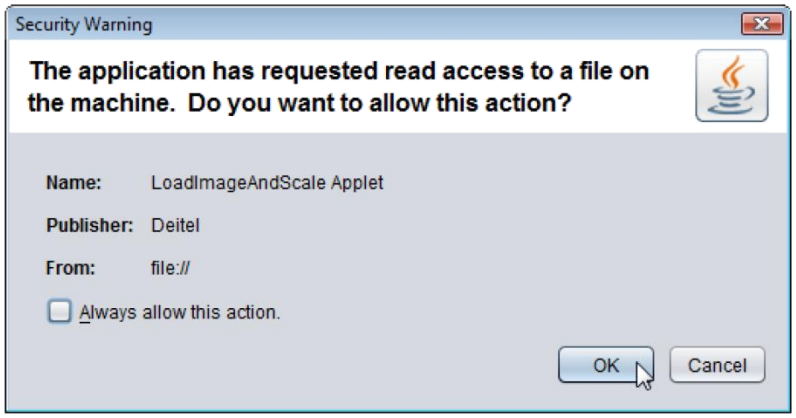
Obtains the bytes of the image and creates an `ImageIcon` to display

Fig. 24.1 | Loading, displaying and scaling an image in an applet. (Part 4 of 7.)

```
90             // the following values are used to center the image
91             double spareWidth =
92                getWidth() - scaleValue * image.getIconWidth();
93             double spareHeight =
94                getHeight() - scaleValue * image.getIconHeight();
95
96             // draw image with scaled width and height
97             g.drawImage( image.getImage(),
98                (int) ( spareWidth ) / 2, (int) ( spareHeight ) / 2,
99                (int) ( image.getIconWidth() * scaleValue ),
100               (int) ( image.getIconHeight() * scaleValue ), this );
101         } // end method paint
102      } // end class DrawJPanel
103   } // end class LoadImageAndScale
```

Fig. 24.1 | Loading, displaying and scaling an image in an applet. (Part 5 of 7.)

(a) Java Web Start security dialog that appears because this applet is requesting access to a file on the local computer.



(b) **Open** dialog that appears if the user clicks **OK** in the security dialog.

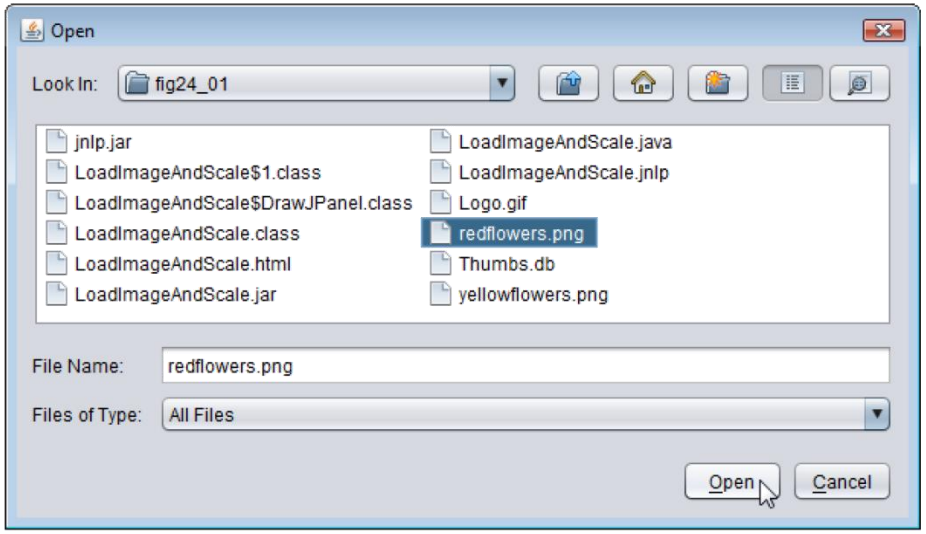


Fig. 24.1 | Loading, displaying and scaling an image in an applet. (Part 6 of 7.)

(c) Scaling the image.

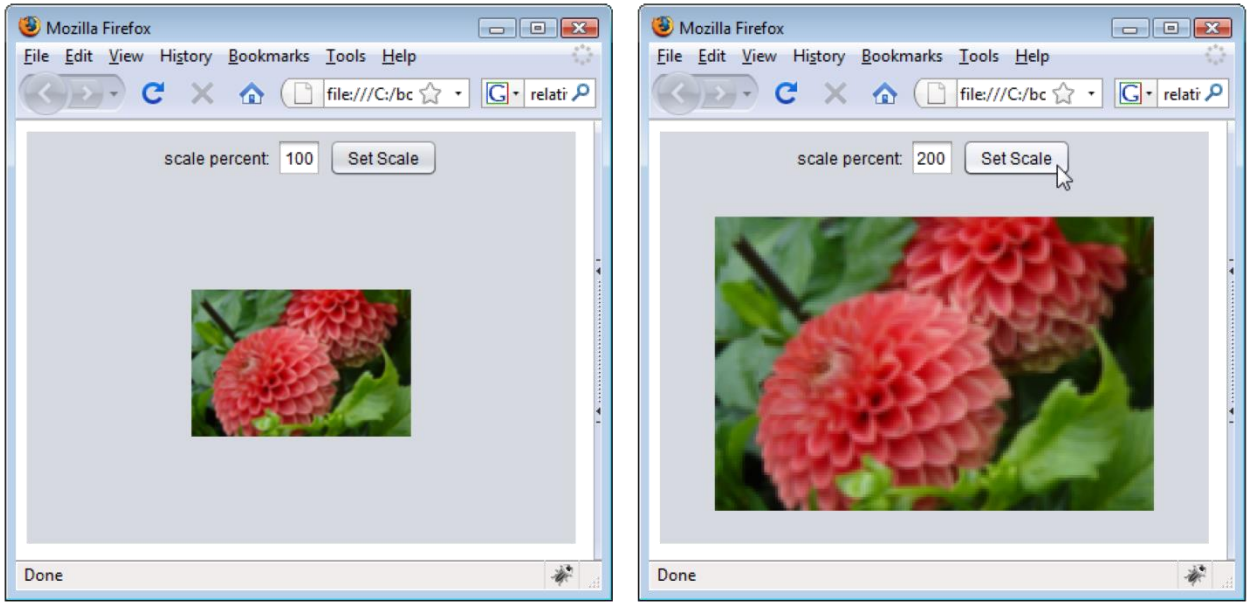


Fig. 24.1 | Loading, displaying and scaling an image in an applet. (Part 7 of 7.)

Loading, Displaying and Scaling Images

- With the user's permission, Java Web Start programs can access the local file system via the JNLP APIs of package javax.jnlp.
- The JNLP ServiceManager class's static lookup method obtain a reference to the File-OpenService.
  - JNLP provides several services, so this method returns an Object reference, which you must cast to the appropriate type.
- FileOpenService's openFileDialog method displaya a file-selection dialog.
  - Java Web Start prompts the user to approve the applet's request for local file-system access.
  - If the user gives permission, the Open dialog is displayed.

Loading, Displaying and Scaling Images

- The openFileDialog method has two parameters

  a String to suggest a directory to open and a String array of acceptable file extensions (such as "png" and "jpg").
  - Program does not make use of either parameter, instead passing null values, which displays an open dialog open to the user's default directory and allows any file type to be selected.
- When the user selects an image file and clicks the Open button in the dialog, method openFileDialog returns a FileContents object, which for security reasons does not give the program access to the file's exact location on disk.
- The program can get an InputStream and read the file's bytes.
- FileContents method getLength returns the number of bytes (as a long) in the file.

Loading, Displaying and Scaling Images

- `DrawJPanel`'s `getWidth` and `getHeight` methods (inherited indirectly from class `Component`) return the `DrawJPanel`'s width and height, respectively.
- The `ImageIcon`'s `getIconWidth` and `getIconHeight` methods return the image's width and height, respectively.
- The `Graphics` class's overloaded `drawImage` methods display a scaled version of the `ImageIcon`.
- `ImageIcon`'s `getImage` method returns an `Image`.

Loading, Displaying and Scaling Images

- The last argument is a reference to an ImageObserver—an interface implemented by class Component.
  - Since class DrawJPanel indirectly extends Component, a DrawJPanel *is an* ImageObserver.
  - This argument is important when displaying large images that require a long time to load (or download from the Internet).
  - As the Image loads, the ImageObserver receives notifications and updates the image on the screen as necessary.

Loading, Displaying and Scaling Images

- Compiling and running this applet requires the jnlp.jar file that contains the JNLP APIs. Found in your JDK installation directory under the directories
  - sample
    jnlp
    servlet
- To compile the applet, use the following command:
  - javac -classpath *PathToJnlpJarFile*
    LoadImageAndScale.java
- where *PathToJnlpJarFile* includes both the path and the file name jnlp.jar.

Loading, Displaying and Scaling Images

- To package the applet for use with Java Web Start, you must create a JAR file that contains the applet's code and the jnlp.jar file.
- To do so, use the command
  - jar cvf LoadImageAndScale.jar *.class *PathToJnlpJarFile*
- where *PathToJnlpJarFile* includes both the path and the file name jnlp.jar.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <jnlp codebase="file:." href="LoadImageAndScale.jnlp">
3
4     <information>
5        <title>LoadImageAndScale Applet</title>
6        <vendor>Deitel</vendor>
7        <offline-allowed/>
8     </information>
9
10    <resources>
11       <java version="1.6+"/>
12       <jar href="LoadImageAndScale.jar" main="true"/>
13       <jar href="jnlp.jar"/>
14    </resources>
15
16    <applet-desc
17       name="LoadImageAndScale"
18       main-class="LoadImageAndScale"
19       width="400"
20       height="300">
21    </applet-desc>
22 </jnlp>
```

Contains the classes that implement JNLP services

Fig. 24.2 | JNLP document for the `LoadImageAndScale` applet.

24.2 Loading, Displaying and Scaling Images (cont.)

- In this example, we use an `applet` element to specify the applet's class and provide two `param` elements between the `applet` element's tags.
- The first specifies that this applet should be draggable.
  - The user can hold the *Alt* key and use the mouse to drag the applet outside the browser window.
  - Clicking the close box on the applet when it is executing outside the browser causes the applet to move back into the browser window if it is still open, or to terminate otherwise.
- The second `param` element shows an alternate way to specify the JNLP file that launches an applet.

```
1  <html>
2  <body>
3  <applet code="LoadImageAndScale.class" width="400" height="300">
4      <param name="draggable" value="true">
5      <param name="jnlp_href" value="LoadImageAndScale.jnlp">
6  </applet>
7  </body>
8  </html>
```

This applet can be dragged outside the browser to install on the desktop

Fig. 24.3 | XHTML document to load the `LoadImageAndScale` applet and make it draggable outside the browser window.

Animating a Series of Images

- Next, we animate a series of images that are stored in an array of ImageIcons.
- Class LogoAnimator declares a main method (lines 8–20 of) to execute the animation as an application.

```
1  // Fig. 24.4: LogoAnimatorJPanel.java
2  // Animating a series of images.
3  import java.awt.Dimension;
4  import java.awt.event.ActionEvent;
5  import java.awt.event.ActionListener;
6  import java.awt.Graphics;
7  import javax.jnlp.FileContents;
8  import javax.jnlp.FileOpenService;
9  import javax.jnlp.ServiceManager;
10 import javax.swing.ImageIcon;
11 import javax.swing.JPanel;
12 import javax.swing.Timer;
13
14 public class LogoAnimatorJPanel extends JPanel
15 {
16    protected ImageIcon images[]; // array of images
17    private int currentImage = 0; // current image index
18    private final int ANIMATION_DELAY = 50; // millisecond delay
19    private int width; // image width
20    private int height; // image height
21
22    private Timer animationTimer; // Timer drives animation
23
```

Fig. 24.4 | Animating a series of images. (Part 1 of 6.)

```
24      // constructor initializes LogoAnimatorJPanel by loading images
25      public LogoAnimatorJPanel()
26      {
27         try
28         {
29            // get reference to FileOpenService
30            FileOpenService fileOpenService =
31               (FileOpenService) ServiceManager.lookup(
32                  "javax.jnlp.FileOpenService" );
33
34            // display dialog that allows user to select multiple files
35            FileContents[] contents =
36               fileOpenService.openMultiFileDialog( null, null );
37
38            // create array to store ImageIcon references
39            images = new ImageIcon[ contents.length ];
40
41            // load the selected images
42            for ( int count = 0; count < images.length; count++ )
43            {
44               // create byte array to store an image's data
45               byte[] imageData =
46                   new byte[ (int) contents[ count ].getLength() ];
47
```

Fig. 24.4 | Animating a series of images. (Part 2 of 6.)

```
48              // get image's data and create image
49              contents[ count ].getInputStream().read( imageData );
50              images[ count ] = new ImageIcon( imageData );
51           } // end for
52
53           // this example assumes all images have the same width and height
54           width = images[ 0 ].getIconWidth();   // get icon width
55           height = images[ 0 ].getIconHeight(); // get icon height
56        } // end try
57        catch( Exception e )
58        {
59           e.printStackTrace();
60        } // end catch
61     } // end LogoAnimatorJPanel constructor
62
63     // display current image
64     public void paintComponent( Graphics g )
65     {
66        super.paintComponent( g ); // call superclass paintComponent
67
68        images[ currentImage ].paintIcon( this, g, 0, 0 );
69
```

Fig. 24.4 | Animating a series of images. (Part 3 of 6.)

```
70          // set next image to be drawn only if Timer is running
71          if ( animationTimer.isRunning() )
72             currentImage = ( currentImage + 1 ) % images.length;
73       } // end method paintComponent
74
75       // start animation, or restart if window is redisplayed
76       public void startAnimation()
77       {
78          if ( animationTimer == null )
79          {
80             currentImage = 0; // display first image
81
82             // create timer
83             animationTimer =
84                new Timer( ANIMATION_DELAY, new TimerHandler() );
85
86             animationTimer.start(); // start Timer
87          } // end if
88          else // animationTimer already exists, restart animation
89          {
90             if ( ! animationTimer.isRunning() )
91                animationTimer.restart();
92          } // end else
93       } // end method startAnimation
```

Timers generate ActionEvents

Fig. 24.4 | Animating a series of images. (Part 4 of 6.)

```
94
95      // stop animation Timer
96      public void stopAnimation()
97      {
98         animationTimer.stop();
99      } // end method stopAnimation
100
101     // return minimum size of animation
102     public Dimension getMinimumSize()
103     {
104        return getPreferredSize();
105     } // end method getMinimumSize
106
107     // return preferred size of animation
108     public Dimension getPreferredSize()
109     {
110        return new Dimension( width, height );
111     } // end method getPreferredSize
112
113     // inner class to handle action events from Timer
114     private class TimerHandler implements ActionListener
115     {
```

Used by layout managers to help size the component

Used by layout managers to help size the component

Fig. 24.4 | Animating a series of images. (Part 5 of 6.)

```
116         // respond to Timer's event
117         public void actionPerformed( ActionEvent actionEvent )
118         {
119            repaint(); // repaint animator
120         } // end method actionPerformed
121      } // end class TimerHandler
122   } // end class LogoAnimatorJPanel
```

Fig. 24.4 | Animating a series of images. (Part 6 of 6.)

```
1  // Fig. 24.5: LogoAnimator.java
2  // Displaying animated images on a JFrame.
3  import javax.swing.JFrame;
4
5  public class LogoAnimator
6  {
7     // execute animation in a JFrame
8     public static void main( String args[] )
9     {
10       LogoAnimatorJPanel animation = new LogoAnimatorJPanel();
11
12       JFrame window = new JFrame( "Animator test" ); // set up window
13       window.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
14       window.add( animation ); // add panel to frame
15
16       window.pack();  // make window just large enough for its GUI
17       window.setVisible( true );   // display window
18
19       animation.startAnimation();  // begin animation
20    } // end main
21 } // end class LogoAnimator
```

Fig. 24.5 | Displaying animated images on a `JFrame`. (Part 1 of 2.)

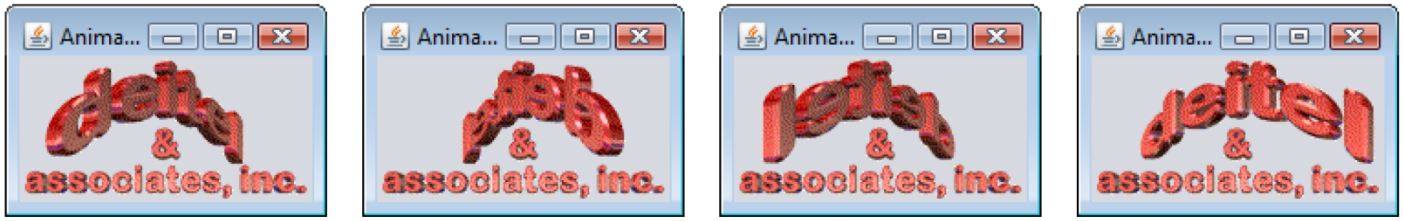


Fig. 24.5 | Displaying animated images on a `JFrame`. (Part 2 of 2.)

Animating a Series of Images

- The JNLP `FileOpenService`'s `openMultiFileDialog` method displays a file-selection dialog that allows the user to select multiple files at once.
- First the user is prompted to give permission, then the Open dialog appears if permission is granted.
- The method returns an array of `FileContents` objects representing the set of files selected by the user.
- The animation is driven by an instance of class `Timer` (from package `javax.swing`).

Animating a Series of Images

- A `Timer` generates `ActionEvent`s at a fixed interval in milliseconds and notifies all its `ActionListener`s each time an `ActionEvent` occurs.
- The `Timer` object's `start` method starts the `Timer`.
- `Timer` method `isRunning` determinea whether the `Timer` is running (i.e., generating events).
- `Timer` method `restart` indicates that the `Timer` should start generating events again.
- `ImageIcon`'s `paintIcon` method displays an `ImageIcon`.
- `Timer` method `stop` indicates that the `Timer` should stop generating events.

Animating a Series of Images

- Layout managers often use a component's `getPreferredSize` method (inherited from class `java.awt.Component`) to determine the component's preferred width and height.
- If a new component has a preferred width and height, it should override method `getPreferredSize` to return that width and height as an object of class `Dimension` (package `java.awt`).
- The `Dimension` class represents the width and height of a GUI component.

Animating a Series of Images

- Method getMinimumSize determines the minimum width and height of the component.
- New components should override method getMinimumSize (also inherited from class Component).
- Method getMinimumSize simply calls getPreferredSize (a common programming practice) to indicate that the minimum size and preferred size are the same.
- Some layout managers ignore the dimensions specified by these methods.

Animating a Series of Images

- Compiling and running this application requires the `jnlp.jar` file that contains the JNLP APIs. To compile the application use the following command:
  - `javac -classpath` *`PathToJnlpJarFile`* `*.java`

  where *PathToJnlpJarFile* includes both the path and the file name `jnlp.jar`.
- To package the application for use with Java Web Start, create a JAR file that contains the applet's code and the `jnlp.jar` file. To do so, use the command
  - `jar cvf LogoAnimator.jar *.class` *`PathToJnlpJarFile`*

  where *PathToJnlpJarFile* includes both the path and the file name `jnlp.jar`.

Animating a Series of Images

- The JNLP document in is similar to the one in previous example .
- The only new feature in this document is the `application-desc` element, which specifies the name of the application and its main class.
- To run this application, use the command
  - `javaws LogoAnimator.jnlp`

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <jnlp codebase="file:." href="LogoAnimator.jnlp">
3
4     <information>
5        <title>LogoAnimator</title>
6        <vendor>Deitel</vendor>
7        <offline-allowed/>
8     </information>
9
10    <resources>
11       <java version="1.6+"/>
12       <jar href="LogoAnimator.jar" main="true"/>
13       <jar href="jnlp.jar"/>
14    </resources>
15
16    <application-desc
17       name="LogoAnimator"
18       main-class="LogoAnimator">
19    </application-desc>
20 </jnlp>
```

Fig. 24.6 | JNLP document for the `LoadImageAndScale` applet.

Image Maps

- Image map
  - An image with hot areas that the user can click to accomplish a task.
  - Commonly used to create interactive web pages.
  - When the user positions the mouse pointer over a hot area, normally a descriptive message appears in the status area of the browser or in a tool tip.
- Method `showStatus` of class `Applet` displays a `String` in the status bar of the applet container.

```
1  // Fig. 24.7: ImageMap.java
2  // Image map.
3  import java.awt.event.MouseAdapter;
4  import java.awt.event.MouseEvent;
5  import java.awt.event.MouseMotionAdapter;
6  import java.awt.Graphics;
7  import javax.swing.ImageIcon;
8  import javax.swing.JApplet;
9
10 public class ImageMap extends JApplet
11 {
12    private ImageIcon mapImage;
13
14    private static final String captions[] = { "Common Programming Error",
15       "Good Programming Practice", "Look-and-Feel Observation",
16       "Performance Tip", "Portability Tip",
17       "Software Engineering Observation", "Error-Prevention Tip" };
18
```

Fig. 24.7 | Image map. (Part 1 of 6.)

```
19      // sets up mouse listeners
20      public void init()
21      {
22         addMouseListener(
23
24            new MouseAdapter() // anonymous inner class
25            {
26               // indicate when mouse pointer exits applet area
27               public void mouseExited( MouseEvent event )
28               {
29                  showStatus( "Pointer outside applet" );
30               } // end method mouseExited
31            } // end anonymous inner class
32         ); // end call to addMouseListener
33
34         addMouseMotionListener(
35
36            new MouseMotionAdapter() // anonymous inner class
37            {
38               // determine icon over which mouse appears
39               public void mouseMoved( MouseEvent event )
40               {
```

Fig. 24.7 | Image map. (Part 2 of 6.)

```
41                  showStatus( translateLocation(
42                     event.getX(), event.getY() ) );
43               } // end method mouseMoved
44            } // end anonymous inner class
45         ); // end call to addMouseMotionListener
46
47         mapImage = new ImageIcon( "icons.png" ); // get image
48      } // end method init
49
50      // display mapImage
51      public void paint( Graphics g )
52      {
53         super.paint( g );
54         mapImage.paintIcon( this, g, 0, 0 );
55      } // end method paint
56
57      // return tip caption based on mouse coordinates
58      public String translateLocation( int x, int y )
59      {
60         // if coordinates outside image, return immediately
61         if ( x >= mapImage.getIconWidth() || y >= mapImage.getIconHeight() )
62            return "";
63   ‘
```

Fig. 24.7 | Image map. (Part 3 of 6.)

```
64          // determine icon number (0 - 6)
65          double iconWidth = ( double ) mapImage.getIconWidth() / 7.0;
66          int iconNumber = ( int )( ( double ) x / iconWidth );
67
68          return captions[ iconNumber ]; // return appropriate icon caption
69       } // end method translateLocation
70    } // end class ImageMap
```

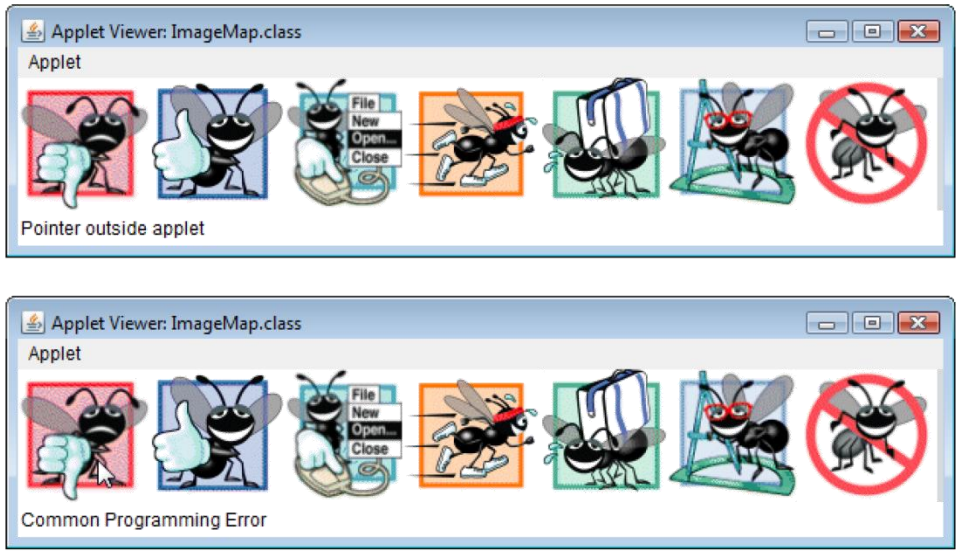


Fig. 24.7 | Image map. (Part 4 of 6.)

Fig. 24.7 | Image map. (Part 5 of 6.)

Fig. 24.7 | Image map. (Part 6 of 6.)

Loading and Playing Audio Clips

- Java programs can manipulate and play audio clips.
- Java provides several mechanisms for playing sounds in an applet.
  - The two simplest are the `Applet`'s `play` method and the `play` method of the `AudioClip` *interface*.
  - Additional audio capabilities are available in the Java Media Framework and Java Sound APIs.

Loading and Playing Audio Clips (cont.)

- `Applet` method `play` loads a sound and plays it once, then the sound can be garbage collected.
- Two versions:
  - ```
    public void play( URL location,
        String soundFileName );
    public void play( URL soundURL );
    ```
- The first version loads the audio clip stored in file `soundFileName` from `location` and plays the sound.
- The second version of method `play` takes a `URL` that contains the location and the file name of the audio clip.
- The statement
  - `play( getDocumentBase(), "hi.au" );`
- loads the audio clip in file `hi.au` and plays the clip once.
    ```

Loading and Playing Audio Clips

- The sound engine that plays the audio clips supports several audio file formats, including Sun Audio file format (`.au` extension), Windows Wave file format (`.wav` extension), Macintosh AIFF file format (`.aif` or `.aiff` extensions) and Musical Instrument Digital Interface (MIDI) file format (`.mid` or `.rmi` extensions).
- An applet can use an `AudioClip` to store audio for repeated use throughout a program's execution.
- `Applet` method `getAudioClip` has two forms that take the same arguments as method `play`.
  - Returns a reference to an `AudioClip`.
- An `AudioClip` has three methods—`play`, `loop` and `stop`.
  - `play` plays the audio clip once.
  - `loop` continuously loops through the audio clip in the background
  - `stop` terminates an audio clip that is currently playing.

```
1  // Fig. 24.8: LoadAudioAndPlay.java
2  // Loading and playing an AudioClip.
3  import java.applet.AudioClip;
4  import java.awt.event.ItemListener;
5  import java.awt.event.ItemEvent;
6  import java.awt.event.ActionListener;
7  import java.awt.event.ActionEvent;
8  import java.awt.FlowLayout;
9  import javax.swing.JApplet;
10 import javax.swing.JButton;
11 import javax.swing.JComboBox;
12
13 public class LoadAudioAndPlay extends JApplet
14 {
15    private AudioClip sound1, sound2, currentSound;
16    private JButton playJButton, loopJButton, stopJButton;
17    private JComboBox soundJComboBox;
18
19    // load the audio when the applet begins executing
20    public void init()
21    {
22       setLayout( new FlowLayout() );
23
```

Fig. 24.8 | Loading and playing an `AudioClip`. (Part 1 of 4.)

```
24          String choices[] = { "Welcome", "Hi" };
25          soundJComboBox = new JComboBox( choices ); // create JComboBox
26
27          soundJComboBox.addItemListener(
28
29             new ItemListener() // anonymous inner class
30             {
31                // stop sound and change sound to user's selection
32                public void itemStateChanged( ItemEvent e )
33                {
34                   currentSound.stop();
35                   currentSound = soundJComboBox.getSelectedIndex() == 0 ?
36                      sound1 : sound2;
37                } // end method itemStateChanged
38             } // end anonymous inner class
39          ); // end addItemListener method call
40
41          add( soundJComboBox ); // add JComboBox to applet
42
43          // set up button event handler and buttons
44          ButtonHandler handler = new ButtonHandler();
45
46          // create Play JButton
47          playJButton = new JButton( "Play" );
```

Fig. 24.8 | Loading and playing an `AudioClip`. (Part 2 of 4.)

```
48          playJButton.addActionListener( handler );
49          add( playJButton );
50
51          // create Loop JButton
52          loopJButton = new JButton( "Loop" );
53          loopJButton.addActionListener( handler );
54          add( loopJButton );
55
56          // create Stop JButton
57          stopJButton = new JButton( "Stop" );
58          stopJButton.addActionListener( handler );
59          add( stopJButton );
60
61          // load sounds and set currentSound
62          sound1 = getAudioClip( getDocumentBase(), "welcome.wav" );
63          sound2 = getAudioClip( getDocumentBase(), "hi.au" );
64          currentSound = sound1;
65       } // end method init
66
67       // stop the sound when the user switches web pages
68       public void stop()
69       {
70          currentSound.stop(); // stop AudioClip
71       } // end method stop
```

Fig. 24.8 | Loading and playing an `AudioClip`. (Part 3 of 4.)

```
72
73       // private inner class to handle button events
74       private class ButtonHandler implements ActionListener
75       {
76          // process play, loop and stop button events
77          public void actionPerformed( ActionEvent actionEvent )
78          {
79             if ( actionEvent.getSource() == playJButton )
80                currentSound.play(); // play AudioClip once
81             else if ( actionEvent.getSource() == loopJButton )
82                currentSound.loop(); // play AudioClip continuously
83             else if ( actionEvent.getSource() == stopJButton )
84                currentSound.stop(); // stop AudioClip
85          } // end method actionPerformed
86       } // end class ButtonHandler
87    } // end class LoadAudioAndPlay
```

Fig. 24.8 | Loading and playing an `AudioClip`. (Part 4 of 4.)

Playing Video and Other Media with Java Media Framework

- Sun provides a reference implementation of the JMF specification—JMF 2.1.1e
  - Supports media file types such as Microsoft Audio/Video Interleave (.avi), Macromedia Flash movies (.swf), Future Splash (.spl), MPEG Layer 3 Audio (.mp3), Musical Instrument Digital Interface (MIDI; .mid or .rmi extensions), MPEG-1 videos (.mpeg, .mpg), QuickTime (.mov), Sun Audio file format (.au extension), and Macintosh AIFF file format (.aif or .aiff extensions).
- To compile and run this application, you must include in the class path the jmf.jar file that is installed with the Java Media Framework.

Playing Video and Other Media with Java Media Framework

- The JMF offers several mechanisms for playing media.
- The simplest is using objects that implement interface Player declared in package javax.media.
- To play a media clip
  - Create a URL object that refers to it.
  - Pass the URL as an argument to static method createRealizedPlayer of class Manager to obtain a Player for the media clip.
- Class Manager declares utility methods for accessing system resources to play and to manipulate media.

```
1  // Fig. 24.9: MediaPanel.java
2  // JPanel that plays a media file from a URL.
3  import java.awt.BorderLayout;
4  import java.awt.Component;
5  import java.io.IOException;
6  import java.net.URL;
7  import javax.media.CannotRealizeException;
8  import javax.media.Manager;
9  import javax.media.NoPlayerException;
10 import javax.media.Player;
11 import javax.swing.JPanel;
12
13 public class MediaPanel extends JPanel
14 {
15    public MediaPanel( URL mediaURL )
16    {
17       setLayout( new BorderLayout() ); // use a BorderLayout
18
19       // Use lightweight components for Swing compatibility
20       Manager.setHint( Manager.LIGHTWEIGHT_RENDERER, true );
21
22       try
23       {
```

Fig. 24.9 | `JPanel` that plays a media file from a URL. (Part 1 of 3.)

```
24            // create a player to play the media specified in the URL
25            Player mediaPlayer = Manager.createRealizedPlayer( mediaURL );
26
27            // get the components for the video and the playback controls
28            Component video = mediaPlayer.getVisualComponent();
29            Component controls = mediaPlayer.getControlPanelComponent();
30
31            if ( video != null )
32               add( video, BorderLayout.CENTER ); // add video component
33
34            if ( controls != null )
35               add( controls, BorderLayout.SOUTH ); // add controls
36
37            mediaPlayer.start(); // start playing the media clip
38         } // end try
39         catch ( NoPlayerException noPlayerException )
40         {
41            System.err.println( "No media player found" );
42         } // end catch
43         catch ( CannotRealizeException cannotRealizeException )
44         {
45            System.err.println( "Could not realize media player" );
46         } // end catch
```

Fig. 24.9 | `JPanel` that plays a media file from a URL. (Part 2 of 3.)

```
47         catch ( IOException iOException )
48         {
49            System.err.println( "Error reading from the source" );
50         } // end catch
51      } // end MediaPanel constructor
52   } // end class MediaPanel
```

Fig. 24.9 | `JPanel` that plays a media file from a URL. (Part 3 of 3.)

Playing Video and Other Media with Java Media Framework

- static Manager method setHint specifies rendering hints to the Manager.
  - e.g., instruct the Manager to use a lightweight renderer that is compatible with lightweight Swing components, as opposed to the default heavyweight renderer.
- static method createRealizedPlayer of class Manager creates and realizes a Player that plays the media file.
  - When a Player realizes, it identifies the system resources it needs to play the media.
  - Depending on the file, realizing can be a resource-consuming and time-consuming process.

Playing Video and Other Media with Java Media Framework

- Throws three checked exceptions, NoPlayerException, CannotRealizeException and IOException.
  - A NoPlayerException indicates that the system could not find a player that can play the file format.
  - A CannotRealizeException indicates that the system could not properly identify the resources a media file needs.
  - An IOException indicates that there was an error while reading the file.

Playing Video and Other Media with Java Media Framework

- Player method getVisualComponent gets a Component that displays the visual (generally video) aspect of the media file.
- Player method getControlPanelComponent gets a Component that provides playback and media controls.
- Player method start begins playing a media file.

```
1  // Fig. 24.10: MediaTest.java
2  // Test application that creates a MediaPanel from a user-selected file.
3  import java.io.File;
4  import java.net.MalformedURLException;
5  import java.net.URL;
6  import javax.swing.JFileChooser;
7  import javax.swing.JFrame;
8
9  public class MediaTest
10 {
11    // launch the application
12    public static void main( String args[] )
13    {
14       // create a file chooser
15       JFileChooser fileChooser = new JFileChooser();
16
17       // show open file dialog
18       int result = fileChooser.showOpenDialog( null );
19
20       if ( result == JFileChooser.APPROVE_OPTION ) // user chose a file
21       {
22          URL mediaURL = null;
23
```

Fig. 24.10 | Test application that creates a MediaPanel from a user-selected file. (Part 1 of 3.)

```
            try
            {
               // get the file as URL
               mediaURL = fileChooser.getSelectedFile().toURI().toURL();
            } // end try
            catch ( MalformedURLException malformedURLException )
            {
               System.err.println( "Could not create URL for the file" );
            } // end catch

            if ( mediaURL != null ) // only display if there is a valid URL
            {
               JFrame mediaTest = new JFrame( "Media Tester" );
               mediaTest.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

               MediaPanel mediaPanel = new MediaPanel( mediaURL );
               mediaTest.add( mediaPanel );

               mediaTest.setSize( 300, 300 );
               mediaTest.setVisible( true );
            } // end inner if
         } // end outer if
      } // end main
   } // end class MediaTest
```

Fig. 24.10 | Test application that creates a MediaPanel from a user-selected file.
(Part 2 of 3.)

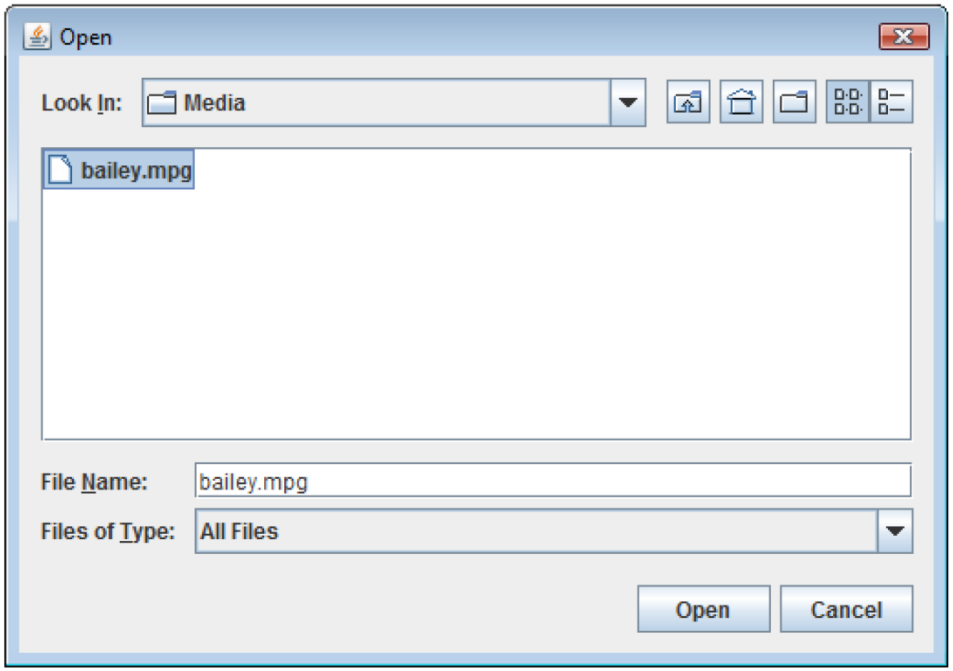


Fig. 24.10 | Test application that creates a MediaPanel from a user-selected file. (Part 3 of 3.)

Playing Video and Other Media with Java Media Framework

- Method `toURI` of class `File` returns a `URI` that points to the `File` on the system.
- Method `toURL` of class `URI` get the file's `URL`.