***Chapter 24***

## *Image Fundamentals*

## There are three common operations that occur when you work with images:

* creating an image
* loading an image, and
* displaying an image

In Java, the **Image** class is used to refer to images in memory and to images that must be loaded from external sources.

*Creating an Image Object*

The **createImage( )** method has the following two forms:

Image createImage(ImageProducer *imgProd*)

Image createImage(int *width*, int *height*)

The 1st form returns an image produced by *imgProd*, which is an object of a class that implements the **ImageProducer** interface (there's not enough time this semester to talk about image producers). The 2nd form returns a blank (empty) image that has the specified width and height. Here is an example:

Canvas c = new Canvas( );

Image test = c.create.Image(200, 100)

This creates an instance of **Canvas**, then calls the **createImage( )** method to make an **Image** object. At this point, the image is blank.

*Loading an Image*

The other way to obtain an image is to load one using **getImage( )**. When **getImage** is invoked, it launches a separate thread of execution in which the image is loaded (or downloaded from the Internet):

Image getImage(URL *url*)

Image getImage(URL *url*, String *imageName*)

The 1st version returns an **Image** object that encapsulates the image found at the location specified by url. The 2nd version returns an **Image** object that encapsulates the image found at the location specified by *url* and having the name specified by *imageName*.

An **Image** is where the actual pixels to be displayed are stored. Every image has an associated graphics context (i.e., an object of class **Graphics** that enables drawing to be performed). Images are first manipulated internally within your applets, then painted to the screen subsequently with **drawImage( )**.

*Displaying an Image*

Once you have an image, you can display it by using **drawImage( )**, which is a member of the **Graphics** class:

boolean drawImage(Image *imgObj*, int *left*, int *top*, ImageObserver *imgOb*)

This displays the image passed in *imgObj* with its upper-left corner specified by *left*, *top*. *imgOb* is a reference to a class that implements the **ImageObserver** interface. An *image observer* is an object that can monitor an image while it loads. It will be discuss later.

Here is a sample applet that loads and displays a single image:

import java.awt.\*;

import java.applet.\*;

import java.net.\*;

public class SimpleImageLoad extends Applet

{ Image img;

URL coach;

public void init( ) {

try {coach = new URL("http://www.cs.sbcc.cc.ca.us/~rhd/");

}

catch (MalformedURLException e) {

showStatus("Exception: " + e.toString());

}

img = getImage(coach, "coach2c.gif");

}

public void paint(Graphics g) {

g.drawImage(img, 0, 0, this);

}

}

*ImageObserver*

When the applet begins to run you can see the image gradually load. This is because the **ImageObserver** interface calls **paint( )** as it receives image data. With **ImageObserver**, you can monitor the loading of an image. An applet can then use the time it takes to load the image to do other things in parallel. When the image has been fully loaded, the entire image is then painted to the screen.

Using an image observer allows you to perform other actions such as showing a progress indicator as you are informed of the progress of the download. This kind of notification is useful when loading an image over a network with a slow modem.

**ImageObserver** defines only one method:

boolean imageUpdate(Image *imgObj*, int *flags*, int *left*, int *top*, int *width*, int *height*)

The default implementation of **imageUpdate( )** in **Applet** has several problems:

* First, it repaints the entire image each time any new data arrives. This causes flashing between the background color and the image.
* Second, it uses a feature of **Applet.repaint( )** to cause the system to only repaint the image every tenth of a second. This causes a jerky, uneven feel as the image is painting.
* Finally, the default implementation knows nothing about images that may fail to load properly.

**Note: getImage( ) always succeeds even when the image specified does not exist**

The example that follows fixes all of these problems in ten lines of code. First, it eliminates the flickering with two small changes:

* It overrides **update( )** so that it calls **paint( )** without painting the background color first (the background is instead painted once in **init**)
* It uses a version of **repaint( )** that paints only a portion of the rectangle is painted

import java.applet.\*;

import java.awt.\*;

import java.awt.image.\*;

import java.net.\*;

public class ObservedImageLoad extends Applet {

Image img;

URL coach;

boolean error = false;

String imgname = "coach2c.gif";

public void init() {

setBackground(Color.blue);

try {coach = new URL("http://www.cs.sbcc.cc.ca.us/~rhd/");

}

catch (MalformedURLException e) {

showStatus("Exception: " + e.toString());

}

img = getImage(coach, imgname);

}

public void paint(Graphics g) {

if (error) {

Dimension d = size();

g.setColor(Color.red);

g.fillRect(0,0,d.width,d.height);

g.setColor(Color.black);

g.drawString("Image not found: " + imgname, 10, d.height/2);

}

returns **false** if image has completed loading or error

returns **true** if image is still loading

Checks flags parameter for:

ABORT bit - the image load was aborted

ERROR bit - the image load encountered an error

SOMEBITS - new image pixels have been received

ALLBITS - the image is now complete

else g.drawImage(img, 0, 0, this);

}

public void update(Graphics g) {

paint(g);

}

public boolean imageUpdate(Image img, int flags, int x, int y, int w, int h) {

if ((flags & SOMEBITS) != 0) { // new partial data

repaint(x, y, w, h); // paint new pixels

}

else if ((flags & (ABORT | ERROR)) != 0) {

error = true; // file not found

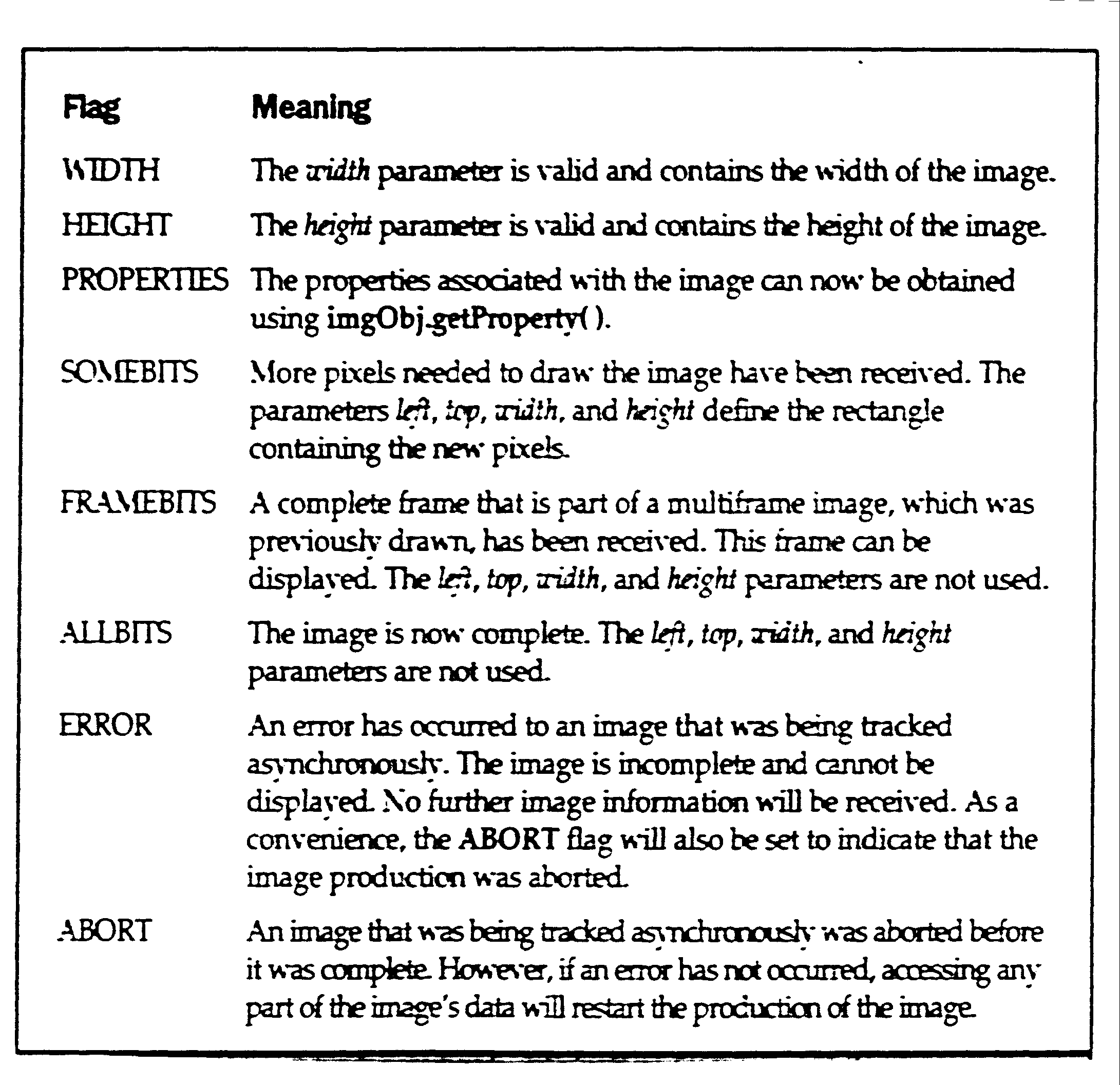
repaint( ); // paint whole applet

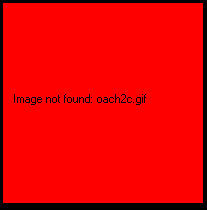
}

return (flags & (ALLBITS | ABORT | ERROR)) = = 0;

}}

The following is a list of constants that can be or’d with flags, to determine the status of a image download:



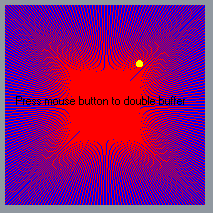


*Buffering*

**Images** are not only useful for storing pictures, but also for rendering drawings. An image is an offscreen drawing surface. This allows you to render an image to a buffer that you can display at a later time. The advantage of doing this is that the image is seen only when it is complete. Drawing a complicated image could take several milliseconds, which can be seen by the user as flashing or flickering. With a buffered image, the applet can instead render the image in the buffer. When completed, it can then be displayed as a unit thereby eliminating the flicker.

Displaying the image requires a **Graphics** object in order to use any of Java's rendering methods. Conveniently, the **Graphics** object that you can use to draw on an **Image** is available via the **getGraphics( )** method.

The following example illustrates the effect of buffering. It uses the **createImage,** **drawImage** and **getGraphics** methods:



import java.awt.\*;

import java.applet.\*;

public class DoubleBuffer extends Applet {

int w, h, mx, my, gap = 3;

boolean flicker = true;

Image buffer = null;

public void init( ) {

Dimension d = size( );

w = d.width;

h = d.height;

buffer = createImage(w, h);

}

public void paint(Graphics g) {

Graphics screengc = null;

if (!flicker) {

screengc = g;

g = buffer.getGraphics( );

}

g.setColor(Color.blue);

g.fillRect(0,0,w,h);

g.setColor(Color.red);

for (int i=0; i<w; i+=gap)

g.drawLine(i,0,w-i,h);

for (int i=0; i<h; i+=gap)

g.drawLine(0,i,w,h-i);

g.setColor(Color.black);

g.drawString("Press mouse button to double buffer", 10, h/2);

g.setColor(Color.yellow);

g.fillOval(mx-gap, my-gap, gap\*2+1, gap\*2+1);

if (!flicker) screengc.drawImage(buffer, 0, 0, null);

}

public void update(Graphics g) {

paint(g);

}

public boolean mouseMove(Event e, int x, int y) {

mx = x;

my = y;

flicker = true;

repaint( );

return true;

}

public boolean mouseDrag(Event e, int x, int y) {

mx = x;

my = y;

flicker = false;

repaint( );

return true;

} }

*MediaTracker*

**ImageObserver** provides a difficult interface when handling multiple images. For simplicity, Sun Microsystems recently added a class called **MediaTracker**. A **MediaTracker** is an object that will check the status of an arbitrary number of images in parallel. In future releases it will also track other media types such as audio.

To use **MediaTracker**, you create an instance and then use **addImage( )** to track the loading status of an image. **addImage( )** has the following forms:

void addImage(Image *imgObj*, int *imgID*)

void addImage(Image *imgObj*, int *imgID*, int *width*, int *height*)

*imgObj* is the image being tracked. *imgID* is the ID number of the image. *width* and *height* contain the dimensions of the object when displayed.

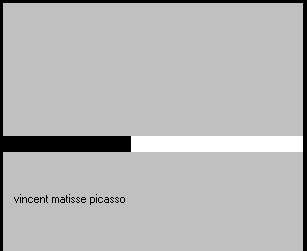
boolean checkID(int *imgID*)

To check the status of an image use:

**checkID** returns a true if all images associated with *imgID* have been loaded.

Use **MediaTracker** when loading a group of images. It allows you to monitor the process enabling you to display something else to entertain the user until they arrive.

The following program creates a **MediaTracker** then adds a collection of images to be tracked using **addImage( )**. In the **paint( )** method, **checkID( )** is called on each tracked image. If all of the images are loaded, they are displayed. Otherwise, a bar chart displays the loading status. Note that the applet has been extended to implement interface **Runnable** so that the animation can be run as a separate thread and have control over the execution of the thread.



/\* <applet code="TrackedImageLoad" width=300 height=400>

\* <param name="img"

\* value="vincent+leonardo+matisse+picasso+renoir+seurat+vermeer">

\* </applet>

\*/

import java.util.\*;

import java.applet.\*;

import java.awt.\*;

public class TrackedImageLoad extends Applet implements **Runnable** {

MediaTracker tracker;

int tracked;

int frame\_rate = 5;

int current\_img = 0;

**Thread** motor;

static final int MAXIMAGES = 10;

Image img[] = new Image[MAXIMAGES];

String name[] = new String[MAXIMAGES];

public void init() {

tracker = new **MediaTracker**(this);

StringTokenizer st = new StringTokenizer(getParameter("img"), "+");

while(st.hasMoreTokens() && tracked <= MAXIMAGES) {

name[tracked] = st.nextToken( );

img[tracked] = **getImage**(getDocumentBase(), name[tracked] + ".jpeg");

tracker.**addImage**(img[tracked], tracked);

tracked++;

}

}

public void paint(Graphics g) {

String loaded = "";

int donecount = 0;

for(int i=0; i<tracked; i++) {

if (tracker.**checkID**(i, true)) {

donecount++;

loaded += name[i] + " ";

}

}

Dimension d = size();

int w = d.width;

int h = d.height;

if (donecount = = tracked) {

frame\_rate = 1;

Image i = img[current\_img++];

int iw = i.getWidth(null);

int ih = i.getHeight(null);

g.drawImage(i, (w - iw)/2, (h - ih)/2, null);

if (current\_img >= tracked)

current\_img = 0;

}

else { int x = w \* donecount / tracked;

g.setColor(Color.black);

g.fillRect(0, h/3, x, 16);

Creates a new thread and invokes the

threads **start** method. The applet's **run**

method is invoked to control the animation.

Everything that used to be accomplished

by the **paint** method is now accomplished

by the infinite loop in **run** with the exception

of the actual display of the image on the

applet - this is still performed by **paint**.

g.setColor(Color.white);

g.fillRect(x, h/3, w-x, 16);

g.setColor(Color.black);

g.drawString(loaded, 10, h/2);

}

}

public void start() {

motor = new **Thread**(this);

motor.start( );

}

public void stop() {

motor.stop();

}

public void run() {

motor.setPriority(Thread.MIN\_PRIORITY);

while (true) {

repaint();

try { Thread.sleep(1000/frame\_rate);

} catch (InterruptedException e) { }

}

}

}

The following code uses **MediaTracker** to enable the program to determine when all images are loaded in conjunction with an animation.

* **MediaTracker** is instantiated with the statement:

imageTracker = new **MediaTracke**r(this);

* The images are then loaded with **getImage**.
* Each image is then registered with **MediaTracker** with the statement:

imageTracker.**addImage**(earth[i], i);

* Once all the images have been registered with the **imageTracker**, the program is forced to wait until the imaged identified as 0 is completely loaded using the following statement:

imageTracker.**waitForId**( 0 );

* **MediaTracker** also provides a **waitForAll** method that blocks until all the images registered are completely loaded.

package earth;

import java.applet.Applet;

import java.awt.\*;

import java.lang.Object;

import java.util.TimerTask;

import java.util.Timer;

public class earthimage extends Applet {

private Image earth[];

private int totalImages = 33, // total number of images

currentImage = 0, // current image subscript

sleepTime = 100; // milliseconds to sleep

MediaTracker imageTracker;

Timer myTimer;

// load the images when the applet begins executing

public void init()

{

myTimer = new Timer(true);

myTimer.schedule(

new TimerTask ( ){

public void run( ) {

repaint( );

}

}

,0, sleepTime);

earth = new Image[ totalImages ];

imageTracker = new MediaTracker( this );

for ( int i = 0; i < earth.length; i++ ) {

earth[ i ] = getImage( getDocumentBase(),

"images/earth" + (i+1) + ".gif" );

// track loading image

imageTracker.addImage( earth[ i ], i );

}

try {

imageTracker.waitForID( 0 ); } catch( InterruptedException e ) { }

}

public void start(Graphics g)

{

g.drawImage(earth[0], 0,0, 300, 300,this);

currentImage = 1;

}

public void paint( Graphics g )

{

if ( imageTracker.checkID( currentImage, true ) )

{

g.drawImage(earth[ currentImage ], 0, 0, 300, 300, this );

if (currentImage== 0)

earth[totalImages-1].flush( );

else earth[currentImage-1].flush( );

currentImage = ++currentImage % totalImages;

}

else

postEvent( new Event( this, Event.MOUSE\_ENTER, "" ) );

}

// override update to eliminate flicker

public void update( Graphics g )

{

paint( g );

}

}

*Loading and Playing Audio Clips*

Java programs can manipulate and play audio clips. This chapter contains a section on how this is done in swing. I will not lecture on this activity, since it closely resembles loading images.

You will need to incorporate sound into your final project. To do so, you’ll need to review the section on *Loading and Playing Audio Clips* in the textbook, or, review the program listed below.

You may use either swing, or non-swing based methods. The textbook incorporates examples using swing. The following program accomplishes the same goal, but in a non-swing based manner:

import java.applet.\*;

import java.awt.\*;

import java.awt.event.\*;

public class LoadAudioAndPlay extends Applet implements ActionListener{

private AudioClip sound;

private Button playSound, loopSound, stopSound;

public void init( )

{ sound = getAudioClip (getDocumentBase( ), "spacemusic.au" );

playSound = new Button("Play");

playSound.addActionListener(this);

add(playSound);

loopSound = new Button("Loop");

loopSound.addActionListener(this);

add(loopSound);

stopSound = new Button("Stop");

stopSound.addActionListener(this);

add(stopSound);

}

public void actionPerformed(ActionEvent e)

{ if (e.getSource( ) == playSound)

sound.play();

else if (e.getSource( ) == loopSound)

sound.loop( );

else if (e.getSource( ) == stopSound)

sound.stop( );

}

}

*Chapter 24 Assignment*

For full credit, your final (Chapter 24) assignment is to write an applet that:

* has an animation with at least 10 images
* launches a frame that does something within it
* plays sound
* is demonstrated in class

Check my website for files that you care about, and additional information about this project: *http://www.cs.sbcc.edu/~rhd/cs120.html*

*Note regarding Sound Files:* With Internet Explorer, my experience continues to indicate that sounds files used in applets must be configured **precisely** as **.au** files - **Freq:8000 MuLAW 8bit**. You may not vary from this format. If you have a .wav file that you’d like to play in IE, you’ll need to convert it into this format. There are many programs available that will support the conversion of *.wav* files to *.au files Freq:80000 MuLAW 8bit*. Use any program that you like! I have provided a link to a program that works fine, and can be downloaded for free. To get the program, go to my java website page, and click on either the *Audacity* or *Audition* hypertext links.

This project is worth 40 points:

* 10 points for animation
* 10 points for sound
* 10 points for frame
* 10 points for classroom demonstration

It is due the last day of scheduled classes (check the bottom of the 2nd page of your syllabus for the date). Demonstrations will occur on the last class meeting of the semester. No late work will be accepted.

There will be significant extra credit if your project reaches

*Coach’s Hall of Fame*

However, your animation must run in Internet Explorer launched from an *.html* file.