Richard G Baldwin (512) 223-4758, baldwin@austin.cc.tx.us, http://www2.austin.cc.tx.us/baldwin/

The AWT Package, Graphics - Overview of Advanced Image Processing Capabilities

Java Programming, Lecture Notes # 174, Revised 11/30/97.

Preface

Students in Prof. Baldwin's **Advanced Java Programming** classes at ACC are responsible for knowing and understanding all of the material in this lesson.

The material in this lesson is extremely important. However, there is simply too much material to be covered in detail during lecture periods. Therefore, students in Prof. Baldwin's **Advanced Java Programming** classes at ACC will be responsible for studying this material on their own, and bringing any questions regarding the material to class for discussion.

This lesson was originally written on November 30, 1997 using the software and documentation in the JDK 1.1.3 download package.

Introduction

Previous lessons have introduced you to many aspects of working with images in Java. To an image-processing novice like myself, the advanced image processing capabilities of Java are somewhat mind-boggling. It would seem that a group of image-processing enthusiasts could easily spend an entire semester digging into the many capabilities offered in this area.

Of necessity then, this lesson will present a cursory overview of those capabilities. For a more in-depth discussion, see the book titled Java AWT Reference by John Zukowski.

Most of the advanced image processing capabilities of Java are in the package named **java.awt.image**. Note that this is not the **Image** class of the package named **java.awt**. This is another package containing three interfaces and eleven classes.

The three interfaces are:

- ImageConsumer
- ImageObserver
- ImageProducer

The eleven classes in hierarchical arrangement are:

- Object
    - ColorModel
        - IndexColorModel
        - DirectColorModel
    - FilteredImageSource
    - ImageFilter
        - RGBImageFilter
        - CropImageFilter
        - ReplicateScaleFilter
            - AreaAveragingScaleFilter
    - MemoryImageSource
    - PixelGrabber

A brief discussion of each of the interfaces and each of the classes follows.

ImageProducer Interface

Classes that implement this interface serve as sources for pixel data. The methods of such a class may generate the pixels from scratch, or may interpret data from another source, such as a GIF file. No matter how it generates the data, the primary purpose of an image producer is to deliver pixels to an image consumer.

Image producers operate somewhat like event sources, and the image producer/consumer model is somewhat analogous to the event source/listener model.

In particular, the methods declared in the **ImageProducer** interface make it possible for one or more **ImageConsumer** objects to register their interest in an image. The image producer then invokes methods declared in the **ImageConsumer** interface to deliver the pixels to the image consumers.

Note that an image producer can register many consumers for its pixels in much the same way that an event source can register many listeners for its events. Several methods are declared in the **ImageProducer** interface, including methods for adding interested consumers to a list of interested consumers, and removing consumers from the list of interested consumers.

Thus, in our analogy, the image *producer* is analogous to the event *source*, and the image *consumers* are analogous to the event *listeners*.

ImageConsumer Interface

The **ImageConsumer** interface declares methods that must be implemented by classes which are to receive data from an image producer.

Several methods are declared in the **ImageConsumer** interface which are used to support the delivery of pixels from the producer to the consumer. The primary method used for delivery is

void **setPixels**(int *x*, int *y*, int *width*, int *height*, ColorModel *model*, byte *pixels[]*, int *offset*, int *scansize*)

The JDK 1.1.3 documentation provides the following explanation of some of the parameters to this method.

The pixels of the image are delivered using one or more calls to the **setPixels** method. Each call specifies the location and size of the rectangle of source pixels that are contained in the array of pixels. The specified **ColorModel** object should be used to convert the pixels into their corresponding color and alpha components.

Pixel (m,n) is stored in the pixels array at index (n * scansize + m + offset). The pixels delivered using this method are all stored as bytes.

ImageObserver Interface

The **ImageObserver** interface was discussed in an earlier lesson and therefore won't be discussed further in this lesson.

ColorModel Class

Pictures on a computer screen are composed of individual dots. Each of the dots is commonly referred to as a *pixel* which I believe is an abbreviation for the term *Picture Element*.

Depending on how you want to represent an image on the screen, a single pixel can be required to contain a lot of information, or very little information.

For example, if all you want to display is a series of lines all of exactly the same color against a background of a different color, then the only information that must be contained in an individual pixel is whether it is turned on or off. In this case, only one bit of information is required to represent a pixel.

As an aside, an early use of the word *bit* was in information theory (Shannon and contemporaries) where one bit of information was deemed to be that amount of information required to determine how an event would fall if, lacking additional information, it had an equal probability of falling in either of two ways. The word was then adopted by computer scientists to represent the fundamental unit of information stored in a computer.

At the other extreme in the pixel data format arena is the format used to represent a pixel in many modern computers where 32-bits are used to represent an individual pixel.

These 32 bits are commonly partitioned into four 8-bit groups or bytes. Three of the bytes are used to represent the contribution of the primary colors: *red*, *green*, and *blue*. The fourth byte (commonly known as the *alpha* byte) is used to represent the degree of transparency with a value of zero representing total transparency and a value of 255 representing total opacity.

In theory, this format can ascribe some 16-million colors to an individual pixel (but your monitor may not be capable of displaying that many individual colors electronically).

Each of these extreme cases could be considered to represent a color model. In Java, a color model determines how colors are represented within the AWT. The **ColorModel** class is an abstract class that you can extend to specify your own representation for colors.

The description of this class from the JDK 1.1.3 documentation reads as follows:

A class that encapsulates the methods for translating from pixel values to alpha, red, green, and blue color components for an image. This class is abstract.

So, why do we need a color model anyway?

Because many of the methods that receive an array of bytes and convert those bytes into actual pixels for rendering on the screen need to know how to interpret the bytes in the array, i.e., how are the bytes to be converted into visual pixels on the screen.

For example, with a simple direct color model, each group of four bytes would probably be interpreted as representing the color value of a single pixel.

For a simple indexed color model, each byte in the array would probably be interpreted as an index into a table of 32-bit integers where each integer represents the actual color to be ascribed to a pixel.

For a crude display which requires only that each pixel be turned on or off, one byte could represent eight pixels on the screen.

The AWT provides two subclasses of the **ColorModel** class: **IndexedColorModel** and **DirectColorModel**. You can also define your own subclasses if you have a mind to.

IndexColorModel Class

You can consume a lot of memory by creating high-resolution images with lots of colors. For example, a 1024 x 768 display contains 786,432 individual pixels. If you represent each of those pixels with four bytes of memory, you will consume 3,145,728 bytes of memory for only one image.

It doesn't take very many images like this to consume more memory than you may have available.

As a result, various schemes have evolved over the years to make it possible to represent images in a reasonable fashion using far less memory to represent each individual image.

A very common scheme is to allocate a smaller number of bits to each pixel (8 bits for example) and to use the pixel values as an index into a table containing the 32-bit representations for some subset of all the possible colors.

For example, if eight bits are used to represent an individual pixel, the pixel value can be used to index into a table containing 256 actual colors. These 256 colors can be selected beforehand, and are often referred to as a palette.

Given the same 1024 x 768 display, this scheme would require 1024 bytes to represent the palette of 256 colors, and 786,432 bytes to represent an image. If there are many images involved, this may be a more paractical approach than representing every pixel by four bytes.

According to the book Java AWT Reference by John Zukowski, in his discussion of the **IndexColorModel**, the says:

"... specifies a **ColorModel** that uses a color map lookup table (with a maximum size of 256), rather than storing color information in the pixels themselves. Pixels are represented by an index into the color map, which is at most an 8-bit quantity. Each entry in the color map gives the alpha, red, green, and blue components of some color. One entry in the map can be designated "transparent." This is called the "transparent" pixel"; the alpha compnent of this map entry is ignored."

The explanation of the same thing in the JDK 1.1.3 documentation reads as follows:

A **ColorModel** class that specifies a translation from pixel values to alpha, red, green, and blue color components for pixels which represent indices into a fixed colormap. An optional transparent pixel value can be supplied which indicates a completely transparent pixel, regardless of any alpha value recorded for that pixel value.

DirectColorModel Class

This class implements the full 32-bit color format in which each pixel consists of four bytes, representing *alpha*, *red*, *green*, and *blue*. The description in the JDK 1.1.3 documentation reads as follows:

A **ColorModel** class that specifies a translation from pixel values to alpha, red, green, and blue color components for pixels which have the color components embedded directly in the bits of the pixel itself.

John Zukowski provides the following description of the **DirectColorModel** class:

"... specifies a color model in which each pixel contains all the color information (alpha, red, green, and blue values) explicitly. Pixels are represented by 32-bit (int) quantities; the constructor lets you change which bits are allotted to each component."

FilteredImageSource Class

Here is what the JDK 1.1.3 documentation has to say about this class:

This class is an implementation of the **ImageProducer** interface which takes an existing image and a filter object and uses them to produce image data for a new filtered version of the original image.

The constructor for this class requires two parameters:

- a source image of type **ImageProducer** and
- a filter object of type **ImageFilter**.

It constructs an object of type **ImageProducer** which is the result of applying the filter object to the original source image object. The filter can perform a variety of operations such as color shifting, image rotation, cropping, etc.

The methods of the class are generally the methods declared in the **ImageProducer** interface.

ImageFilter Class

According to the JDK 1.1.3 documentation:

This class implements a filter for the set of interface methods that are used to deliver data from an **ImageProducer** to an **ImageConsumer**. It is meant to be used in conjunction with a **FilteredImageSource** object to produce filtered versions of existing images.

It is a base class that provides the calls needed to implement a "Null filter" which has no effect on the data being passed through. Filters should subclass this class and override the methods which deal with the data that needs to be filtered and modify it as necessary.

This class implements the **ImageConsumer** interface. There are a number of methods defined in this class, including those declared in the interface, and those that are used to implement the "null filter" capability described above.

There is only one constructor and it takes no parameters.

To review, this class extends **Object** and is subclassed as shown in the following hierarchy diagram from the JDK 1.1.3 AWT.

- ImageFilter
    - RGBImageFilter
    - CropImageFilter
    - ReplicateScaleFilter
        - AreaAveragingScaleFilter

In practice, objects of this class and its subclasses are instantiated to be passed as one of the two parameters to the constructor of a **FilteredInputSource** object, where it is combined with a source image object to produce a new filtered image object.

The **FilteredInputSource** object that is produced can become the source image object for another image consumer.

RGBImageFilter Class

The JDK 1.1.3 documentation describes this class as follows:

This class provides an easy way to create an **ImageFilter** which modifies the pixels of an image in the default **RGB** ColorModel. It is meant to be used in conjunction with a **FilteredImageSource** object to produce filtered versions of existing images.

It is an abstract class that provides the calls needed to channel all of the pixel data through a single method which converts pixels one at a time in the default **RGB** *ColorModel* regardless of the **ColorModel** being used by the **ImageProducer**.

The only method which needs to be defined to create a useable image filter is the **filterRGB** method.

CropImageFilter Class

The JDK 1.1.3 documentation describes this class as:

An **ImageFilter** class for cropping images. This class extends the basic **ImageFilter** Class to extract a given rectangular region of an existing **Image** and provide a source for a new image containing just the extracted region. It is meant to be used in conjunction with a **FilteredImageSource** object to produce cropped versions of existing images.

ReplicateScaleFilter Class

According to the JDK 1.1.3 documentation, this class is:

An **ImageFilter** class for scaling images using the simplest algorithm. This class extends the basic **ImageFilter** Class to scale an existing image and provide a source for a new image containing the resampled image.

The pixels in the source image are sampled to produce pixels for an image of the specified size by replicating rows and columns of pixels to scale up or omitting rows and columns of pixels to scale down. It is meant to be used in conjunction with a **FilteredImageSource** object to produce scaled versions of existing images.

AreaAveragingScaleFilter Class

The following description of this class was taken from the JDK 1.1.3 documentation:

An **ImageFilter** class for scaling images using a simple area averaging algorithm that produces smoother results than the nearest neighbor algorithm.

This class extends the basic **ImageFilter** Class to scale an existing image and provide a source for a new image containing the resampled image. The pixels in the source image are blended to produce pixels for an image of the specified size. The blending process is analogous to scaling up the source image to a multiple of the destination size using pixel replication and then scaling it back down to the destination size by simply averaging all the pixels in the supersized image that fall within a given pixel of the destination image.

MemoryImageSource Class

The following brief description of this class is from the JDK 1.1.3 documentation.

This class is an implementation of the **ImageProducer** interface which uses an array to produce pixel values for an **Image**.

There are six different constructors, each of which requires some or all of the following parameters:

- Width and height in pixels of the image being created,
- Color model to be used in the conversion (two of the constructors don't require this and default to the **RGB** color model),
- An array of bytes or integers containing the values to be converted to pixels in accordance with the color model,
- An offset specifying the first pixel used in the array,
- The number of pixels per scan line in the array which may be larger than the number of pixels in the actual scan line in the image, resulting in some of the data in the array being ignored,
- **A Hashtable** object containing the properties associated with the image, if any.

In all cases, the constructor constructs an **ImageProducer** object which uses an array of bytes or integers to produce data for an **Image** object.

This class implements the **ImageProducer** interface and as such defines all of the methods of that interface. Several other methods are defined as well.

Also according to the JDK 1.1.3 documentation:

The MemoryImageSource is also capable of managing a memory image which varies over time to allow animation or custom rendering.

According to Zukowski, in JDK 1.1.x, **MemoryImageSource** can now pass multiple frames to interested consumers, and mimics **GIF89A** multiframe functionality to produce animation. Zukowski provides an example of this feature in his book Java AWT Reference.

PixelGrabber Class

The **PixelGrabber** class is a utility for converting an image into an array of pixel values. The description of the class, according to the JDK 1.1.3 documentation is:

The **PixelGrabber** class implements an **ImageConsumer** which can be attached to an **Image** or **ImageProducer** object to retrieve a subset of the pixels in that image.

This class has three constructors which require different parameters to control how the object is constructed.

Two of the constructors require you to pass an array of **int** which is where the numeric values of the pixels will be stored.

The third constructor doesn't require the array as a parameter. In this case, you invoke the **getPixel()** method on the **PixelGrabber** object to get the buffer where the resulting data is stored.

This class implements the **ImageConsumer** interface. Hence, it defines the methods of that interface as well as other methods which determine its behavior.

Sample Program

As I mentioned early in this lesson, the image-processing capabilities of Java are extensive. Thus, many different sample programs would be required to illustrate all of the features.

We won't attempt to illustrate all the features in this lesson. Rather, we will provide a simple program that illustrates a few of the features in hopes that you can use that as a jumping off place to investigate the other features on your own.

This program illustrates image manipulation by removing the red color from all the pixels in an image and also making the image partially transparent.

The program requires access to an image file named **"logomain.gif"** in the current directory on the hard disk.

Just about any image should do, but it needs to be large enough that you can see it when it is displayed at its normal size and should be small enough to fit on the screen.

If the program is unable to load the image file within ten seconds, it will abort with an error message. You can easily modify the source code to change the name of the image file and change the timeout interval if you wish to do so.

A large portion of the code in this program has been illustrated in previous lessons and won't be discussed in this lesson. The following discussion primarily involves material that has not been covered in a previous lesson.

This program reads an image file from the disk and saves it in memory under the name **rawImage**.

Then it declares an array of int of sufficient size to contain one int value for every pixel in the image. The name of the array is **pix**.

Then it instantiates an object of type **PixelGrabber** which associates the **rawImage** with the array of int named **pix**.

Following this, it invokes the **grabPixels()** method on the object of type **PixelGrabber** to cause the pixels in the **rawImage** to be converted to numeric values of type **int** and stored in the array named **pix**.

Then it uses bitwise operators to mask the *red* byte out of every pixel value and to make the image partially transparent by masking the *alpha* byte with the hex value **C0** (an alpha value of **00** is completely transparent, and an alpha value of **FF** is completely opaque).

Then it uses the **createImage()** method of the **Component** class along with the constructor for the **MemoryImageSource** class to create a new image from the modified pixel data. The name of the new image is **modImage**.

Finally, it overrides the **paint()** method where it uses the **drawImage()** method to display both the raw image and the modified image on the same **Frame** object for comparison.

The colors in the modified image should reflect the lack of red (assuming they contained red in the first place).

Also the modified image is partially transparent, allowing the yellow background to show through on many pixels.

On a Win95 system. transparency seems to be accomplished by causing a regular distribution of pixels to be rendered in the background color. The higher the degree of transparency, the larger the percentage of pixels that are rendered in the background color.

This program was tested using JDK 1.1.3 under Win95.

Interesting Code Fragments

The first interesting code fragment is the statement in the constructor that gets an image from a specified file. We have seen this before, but it is being repeated here for review purposes.

```
    rawImage =

Toolkit.getDefaultToolkit().getImage("logomain.gif");
```

Follwing this, we use the **MediaTracker** class to track the loading of the image file, etc., but you have seen all of that before. After doing some things with height, width, etc., we get down to the topic that is being illustrated in this lesson.

We declare an array object of type **int** to receive the numeric representation of all the pixels in the image.

```
    int[] pix = new int[rawWidth * rawHeight];
```

Then we instantiate an object of type **PixelGrabber** that deals with some size information, and associates our **rawImage** object with the array named **pix** where we will deposit the numeric values for the pixels. This is accomplished with the following statement. You are referred to the JDK 1.1.3 documentation for an explanation of all the parameters to the constructor.

```
      PixelGrabber pgObj = new PixelGrabber(
                             rawImage,0,0,rawWidth,rawHeight,

pix,0,rawWidth);
```

Once we have the **PixelGrabber** object, we invoke the **grabPixels()** method on that object to perform the actual conversion of the pixels in the image to numeric form.

A more detailed description of this method is definitely worth seeing. The following description was extracted from the JDK 1.1.3 documentation.

public boolean **grabPixels**() throws InterruptedException

Request the **Image** or **ImageProducer** to start delivering pixels and wait for all of the pixels in the rectangle of interest to be delivered.

Returns: **true** if the pixels were successfully grabbed, **false** on abort, error or timeout

Throws: InterruptedException Another thread has interrupted this thread.

We also invoked the **getStatus()** method to check the status of the operation. A description of that method follows.

public synchronized int **getStatus**()

Return the status of the pixels. The **ImageObserver** *flags* representing the available pixel information are returned.

Returns: the bitwise OR of all relevant **ImageObserver** flags

To make much sense out of this, we need information about the **ImageObserver***flags*. A description of the **ImageObserver** flags follows. We will make use of the **ALLBITS** flag in our program.

ABORT - An image which was being tracked asynchronously was aborted before production was complete.

ALLBITS - A static image which was previously drawn is now complete and can be drawn again in its final form.
ERROR - An image which was being tracked asynchronously has encountered an error.
FRAMEBITS - Another complete frame of a multi-frame image which was previously drawn is now available to be drawn again.
HEIGHT - The height of the base image is now available and can be taken from the height argument to the imageUpdate callback method.
PROPERTIES - The properties of the image are now available.
SOMEBITS - More pixels needed for drawing a scaled variation of the image are available.
WIDTH - The width of the base image is now available and can be taken from the width argument to the imageUpdate callback method.

The statement in our sample program that invokes the **grabPixels()** and **getStatus()** methods is shown below.

```
      if(pgObj.grabPixels() && ((pgObj.getStatus() &
                           ImageObserver.ALLBITS) != 0)){
```

If the above statement returns **true**, the next step is to process the numeric pixel values to produce the desired modification of the image. (When examining the above statement, pay close attention to the difference between the use of logical **&&** and bitwise **&**.)

Once we have the pixel data in numeric **int** form (alpha, red, green, blue bytes) it is a simple matter to use a **for** loop and a bitwise **AND** operation to mask out the *red* byte and to modify the *alpha* byte. To accomplish this, we **AND** the *red* byte with hex **00** and **AND** the *alpha* byte with hex **C0**.

```
        for(int cnt = 0; cnt <
(rawWidth*rawHeight);cnt++){
          pix[cnt] = pix[cnt] & 0xC000FFFF;
        }//end for loop
```

The next step is to use the **createImage()** method of the **Component** class to create an image from the numeric array.

To accomplish this, we need to instantiate an object of the **MemoryImageSource** class describing the manner in which we want to convert the array of numeric data to an image and pass that object to the **createImage()** method. (See the JDK 1.1 documentation for a description of the parameters to the **MemoryImageSource** constructor.) You will probably recognize most of those parameters from their names.

The following code creates the desired **Image** object and names it **modImage**.

```
    modImage = this.createImage(
                   new MemoryImageSource(

rawWidth,rawHeight,pix,0,rawWidth));
```

Finally, we override the **paint()** method to display both the raw image and the modified image, one above the other on the same **Frame** object for comparison.

```
      g.drawImage(rawImage,inLeft,inTop,this);
      g.drawImage(modImage,inLeft,inTop+rawHeight,this);
```

A complete listing of the program is provided in the next section.

Program Listing

Some of the interesting code fragments are highlighted in **boldface** in the following program listing. A description of the program was provided in an earlier section.

```
/*File Image05.java
Copyright 1997, R.G.Baldwin

This program illustrates image manipulation by removing
the red color from all the pixels in an image and also
making the image partially transparent.

The program requires access to an image file named
"logomain.gif" in the current directory on the hard disk.

If the program is unable to load the image file within ten
seconds, it will abort with an error message.

This program was tested using JDK 1.1.3 under Win95.

***********************************************************/
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;

class Image05 extends Frame{ //controlling class
  Image rawImage;//ref to raw image file fetched from disk
  int rawWidth;
  int rawHeight;

  Image modImage; //ref to modified image

  //Inset values for the container object
  int inTop;
  int inLeft;

  //=====================================================//

  public static void main(String[] args){
    Image05 obj = new Image05();//instantiate this object
```

```
      obj.repaint();//render the image
   }//end main

  //=====================================================//

  public Image05(){//constructor
    //Get an image from the specified file in the current
    // directory on the local hard disk.
    rawImage =
      Toolkit.getDefaultToolkit().getImage("logomain.gif");

    //Use a MediaTracker object to block until the image
    // is loaded or ten seconds has elapsed.
    MediaTracker tracker = new MediaTracker(this);
    tracker.addImage(rawImage,1);

    try{ //because waitForID throws InterruptedException
      if(!tracker.waitForID(1,10000)){
        System.out.println("Load error.");
        System.exit(1);
      }//end if
    }catch(InterruptedException e){System.out.println(e);}

    //Raw image has been loaded.  Establish width and
    // height of the raw image.
    rawWidth = rawImage.getWidth(this);
    rawHeight = rawImage.getHeight(this);

    this.setVisible(true);//make the Frame visible

    //Get and store inset data for the Frame object so
    // that it can be easily avoided.
    inTop = this.getInsets().top;
    inLeft = this.getInsets().left;

    //Use the insets and the size of the raw image to
    // establish the overall size of the Frame object.
    // Make the Frame object twice the height of the
    // image so that the raw image and the modified image
    // can both be rendered on the Frame object.
    this.setSize(inLeft+rawWidth,inTop+2*rawHeight);
    this.setTitle("Copyright 1997, Baldwin");
    this.setBackground(Color.yellow);

    //Declare an array object to receive the pixel
    // representation of the image
    int[] pix = new int[rawWidth * rawHeight];

    //Convert the rawImage to numeric pixel representation
    try{//because grapPixels() throws InterruptedException
      //Instantiate a PixelGrabber object specifying
      // pix as the array in which to put the numeric
      // pixel data. See JDK 1.1.3 docs for parameters
      PixelGrabber pgObj = new PixelGrabber(
                          rawImage,0,0,rawWidth,rawHeight,
                                         pix,0,rawWidth);
      //Invoke the grabPixels() method on the PixelGrabber
      // object to actually convert the image to an array
      // of numeric pixel data stored in pix. Also test
      // for success in the process.
      if(pgObj.grabPixels() && ((pgObj.getStatus() &
                              ImageObserver.ALLBITS) != 0)){
        //Mask the red byte out of every pixel by ANDing
        // the red byte with 00.  Also make partially
        // transparent by ANDing the alpha byte with C0
        for(int cnt = 0; cnt < (rawWidth*rawHeight);cnt++){
          pix[cnt] = pix[cnt] & 0xC000FFFF;
        }//end for loop
      }//end if statement
      else System.out.println("Pixel grab not successful");
    }catch(InterruptedException e){System.out.println(e);}

    //Use the createImage() method to create a new image
    // from the array of pixel values.
    modImage = this.createImage(
                    new MemoryImageSource(
                        rawWidth,rawHeight,pix,0,rawWidth));

    //Anonymous inner-class listener to terminate program
    this.addWindowListener(
      new WindowAdapter(){//anonymous class definition
        public void windowClosing(WindowEvent e){
          System.exit(0);//terminate the program
        }//end windowClosing()
      }//end WindowAdapter
    );//end addWindowListener
  }//end constructor
//=========================================================//

  //Override the paint method to display both the rawImage
  // and the modImage on the same Frame object.
  public void paint(Graphics g){
    if(modImage != null){
      g.drawImage(rawImage,inLeft,inTop,this);
      g.drawImage(modImage,inLeft,inTop+rawHeight,this);
    }//end if
  }//end paint()
}//end Image05 class
//=========================================================//
```

-end-