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

The AWT Package, Graphics- Introduction to Images

Java Programming, Lecture Notes # 170, Revised 09/23/98.

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.

Introduction

This lesson provides an introduction to the handling of images in Java, including sample programs that illustrates some of the methods used to display an image in a **Frame** object. Subsequent lessons will provide additional information including animation with double buffering and some of the more complex image processing techniques.

The drawImage() Method

Methods from several different classes are used in the handling of images in Java. The first thing we need to do is go back and review the **drawImage()** method from the **Graphics** class, because it is the method used to display an image.

The JDK 1.1.3 documentation lists six different overloaded versions of the **drawImage()** method with the brief descriptions given below. More detailed information is available in the JDK documentation package.

drawImage(Image, int, int, Color, ImageObserver) - Draws as much of the specified image as is currently available.

drawImage(Image, int, int, ImageObserver) - Draws as much of the specified image as is currently available.

drawImage(Image, int, int, int, int, Color, ImageObserver) - Draws as much of the specified image as has already been scaled to fit inside the specified rectangle.

drawImage(Image, int, int, int, int, ImageObserver) - Draws as much of the specified image as has already been scaled to fit inside the specified rectangle.

drawImage(Image, int, int, int, int, int, int, int, int, Color, ImageObserver) - Draws as much of the specified area of the specified image as is currently available, scaling it on the fly to fit inside the specified area of the destination drawable surface.

drawImage(Image, int, int, int, int, int, int, int, int, ImageObserver) - Draws as much of the specified area of the specified image as is currently available, scaling it on the fly to fit inside the specified area of the destination drawable surface.

As you can see, there is a lot of flexibility available when actually drawing the image. A more detailed description of one of the methods follows:

```
public abstract boolean drawImage(Image img,
                                  int x,
                                  int y,
                                  int width,
                                  int height,
                                  ImageObserver observer)
    Parameters:
         img - the specified image to be drawn.
         x - the x coordinate.
         y - the y coordinate.
```

```
          width - the width of the rectangle.
          height - the height of the rectangle.
          observer - object to be notified as more of the
image is
          converted.
```

With this version of the method, the image is drawn inside the specified rectangle and is scaled if necessary. When invoked, this method draws as much of the specified image as has already been scaled to fit inside the specified rectangle.

Transparent pixels do not affect whatever pixels are already there.

This method returns immediately, even if the entire image has not yet been scaled, and converted for the current output device. If the current output representation is not yet complete, then **drawImage** returns false. As more of the image becomes available, the process that draws the image notifies the image observer by calling its **imageUpdate** method and the image continues to be drawn in pieces.

This is how the process is explained by Peter van der Linden in his book Just Java 1.1 and Beyond:

"The **Component** class contains an **imageUpdate()** method. When it is called, it schedules a **repaint**, allowing a little more of the image to be drawn without interfering at all with your thread of control."

He goes on to explain that a system property with a key value of **awt.image.incrementaldraw** determines whether or not the image will be rendered in pieces as described above. The default value of this property is **true**. When true, it causes the system to draw parts of an image as they arrive. When **false**, it causes the system to wait until all the image is loaded before drawing.

A second system property with a key value of **awt.imager.redrawrate** determines the minimum period in milliseconds between calls to **repaint()** for images. The default value is 100 which only applies if the other system property described above is **true**.

System properties can be modified. Such modifications persist only for the current invocation of the program.

Three sample programs in this lesson illustrate the use of the **drawImage()** method to draw an image. The first sample program illustrates the effect of drawing individual portions of the image as the scaled portions become available. This program produces a visual output that is not very satisfactory because it tends to flash as successive calls to **imageUpdate** are made in the drawing process.

The second sample program eliminates this flashing problem by using a **MediaTracker** object to block the thread, and delay the invocation of **drawImage()** until the entire scaled image is available for drawing.

According to the JavaSoft documentation, a scaled version of an image will not necessarily be available immediately just because an unscaled version of the image has been constructed for the output device. However, this effect is not apparent in the second sample program.

All of the parameters to the **drawImage()** method are pretty obvious except for the last one which is described as follows:

> observer - object to be notified as more of the image is converted.

Here is part of what John Zukowski has to say about this topic on page 35 in his book: Java AWT Reference.

> "For the time being, it's enough to say that the call to **drawImage()** starts a new thread that loads the requested image. An image observer monitors the process of loading an image; the thread that is loading the image notifies the image observer whenever new data has arrived. ... it's safe to use **this** as the image observer in a call to **drawImage()**. More simply, we could say that any component can serve as an image observer for images that are drawn on it."

Zukowski has a lot more to say on the topic later on in his book.

The **drawImage()** method has two parameters which are not primitive int variables as highlighted in **boldface** below:

```
(Image img,
 int x,
 int y,
 int width,
 int height,
 ImageObserver observer)
```

This suggests that we need to know more about the **Image** class and the **ImageObserver** interface in order to be able to understand these parameters.

The Image Class

As usual, the **Image** class consists of constants or variables, constructors, and methods.

Constants of the Image Class

One of the methods of the **Image** class is a method named **getScaledInstance()** which returns a scaled version of an image. (We will use this method in the third sample program in this lesson) One of the parameters to this method is an **int** value that specifies which of several available scaling algorithms is to be used in performing the scaling. All but one of the following constants

is used for the purpose of specifying the scaling algorithm. The names and brief descriptions of the constants are generally self-explanatory. Additional detail can be obtained from the JDK documentation package.

SCALE_AREA_AVERAGING - Use the Area Averaging image scaling algorithm.

SCALE_DEFAULT - Use the default image-scaling algorithm.

SCALE_FAST - Choose an image-scaling algorithm that gives higher priority to scaling speed than smoothness of the scaled image.

SCALE_REPLICATE - Use the image scaling algorithm embodied in the **ReplicateScaleFilter** class.

SCALE_SMOOTH - Choose an image-scaling algorithm that gives higher priority to image smoothness than scaling speed.

UndefinedProperty - The **UndefinedProperty** object should be returned whenever a property which was not defined for a particular image is fetched.

The last constant in this list is not used for scaling. Rather, it is used as a return value from the method named **getProperties()** to indicated that the requested property is not available.

Constructors of the Image Class

Although the **Image** class has a constructor, it is an **abstract** class and an object of the class cannot be instantiated directly by invoking the constructor.

You can obtain an **Image** object indirectly by invoking the **getImage()** method of either the **Applet** class or the **Toolkit** class. **getImage()** uses a separate thread to fetch the image. The practical result of invoking **getImage()** is to associate an **Image** reference with a file located somewhere that contains the image of interest. In this lesson, we will confine ourselves to image files on the local hard disk, but they could be on a server somewhere on the web.

The thread starts execution when you call a method that requires image information such as **drawImage()**.

You can also obtain an **Image** object by invoking the **createImage()** method of either the **Component** class or the **Toolkit** class.

As we will see in two of the sample programs in this lesson, you can also use the **MediaTracker** class to force an image to be loaded before you invoke a method that requires the image information.

Methods of the Image Class

The seven methods of the **Image** class, along with brief descriptions are listed below. You are referred to the JDK documentation for complete descriptions of these methods.

flush() - Flushes all resources being used by this **Image** object.

getGraphics() - Creates a graphics context for drawing to an off-screen image.

getHeight(ImageObserver) - Determines the height of the image.

getProperty(String, ImageObserver) - Gets a property of this image by name.

getScaledInstance(int, int, int) - Creates a scaled version of this image.

getSource() - Gets the object that produces the pixels for the image.

getWidth(ImageObserver) - Determines the width of the image.

We will use two of these methods in the sample programs in this lesson and some of the others in subsequent lessons.

As usual, we are interested not only in the **Image** class, but also in the other classes required to instantiate parameters for the methods, classes required to instantiate the objects returned from the methods, and other classes which provide general support for the methods.

A brief review of the above list, the JDK documentation, and several books suggests that we should take a look at the following classes and interfaces that support the methods of the **Image** class.

- String
- ImageObserver
- ImageProducer
- Graphics
- Object
- MediaTracker

Classes and Interfaces that Support the Image Class

Of the support classes listed in the previous section, you should already know about **String, ImageObserver, Graphics**, and **Object**. We will take a further look at **ImageProducer** and **MediaTracker**.

ImageProducer

This is the interface for objects which can produce the image data for Images.

Each image contains an **ImageProducer** which is used to reconstruct the image whenever it is needed. Examples of the need for reconstruction might be when a new size of the **Image** is scaled, or when the width or height of the Image is being requested.

This interface declares several methods which generally have to do with a concept involving *image producers* and *image consumers*. We will have more to say about this in a subsequent lesson.

MediaTracker

This is a utility class designed to track the status of media objects. In theory, media objects could include audio clips and other media as well as images. However, JDK 1.1.3 is only capable of tracking the status of images.

You can use a media tracker object by instantiating an instance of **MediaTracker** for the **Component** that you want have monitored, and invoking its **addImage()** method for each image that you want to track. Each image can be assigned a unique identifier, or groups of images can be assigned the same identifier.

According to the JDK 1.1.3 documentation, this identifier controls the priority order in which the images are loaded. Images with a lower ID are loaded in preference to those with a higher ID number.

The identifier can also be used to identify unique subsets of images. In other words, by assigning the same identifier to several images, you can track them as a group.

You can determine the status of an image (or group of images) by invoking one of several methods on the **MediaTracker** object and passing the identifier as a parameter to the method.

You can also cause **MediaTracker** to block and wait until a specified image (or group of images) completes loading. We will use this approach in two of the sample programs to make certain that the image has completed loading before we attempt to draw it.

Constants of the MediaTracker Class

This class provides four constants as described below. Full descriptions of these constants can be found in the JDK documentation. As the names suggest, each of these constants is used as a return value for one or more of the methods (such as the method named **statusID()**) that can be invoked to inquire as to the status of a given image.

ABORTED - Flag indicating that the downloading of some media was aborted.

COMPLETE - Flag indicating that the downloading of media was completed successfully.

ERRORED - Flag indicating that the downloading of some media encountered an error.

LOADING - Flag indicating some media is currently being loaded.

Constructors of the MediaTracker Class

MediaTracker has a single constructor that can be invoked to instantiate an object to track the status of some of all of the images on a particular component. A brief description of the constructor follows:

MediaTracker(Component) - Creates a media tracker to track images for a given component.

Methods of the MediaTrackerClass

The **MediaTracker** class has a relatively large number of methods that can be invoked on an object of the class for various purposes. The following boxes provide a listing of the JDK 1.1.3 methods along with a brief description of each. The methods have been grouped into several categories in an attempt to make them easier to understand.

A more complete description of each of the methods can be obtained from the JDK documentation.

Building and Maintaining the List

A **MediaTracker** object has the ability to track the status of some or all of the images being loaded for the particular **Component** that was specified when the **MediaTracker** object was instantiated. The following methods are use to build and maintain that list of images. When you add an image to the list, you also specify a numeric identifier for that image that is later used with the other methods to extract status information about the image.

addImage(Image, int) - Adds an image to the list of images being tracked by this media tracker.

addImage(Image, int, int, int) - Adds a scaled image to the list of images being tracked by this media tracker.

removeImage(Image) - Remove the specified image from this media tracker.

removeImage(Image, int) - Remove the specified image from the specified tracking ID of this media tracker.

removeImage(Image, int, int, int) - Remove the specified image with the specified width, height, and ID from this media tracker.

Waiting for Image(s) to finish Loading

A **MediaTracker** object can be used to cause its thread to block until one or more of the images on its list have completed loading. This is accomplished using the following methods. The first and third methods return void. The second and fourth methods return **true** if all images were successfully loaded and **false** otherwise.

waitForAll() - Starts loading all images tracked by this media tracker.

waitForAll(long) - Starts loading all images tracked by this media tracker.

waitForID(int) - Starts loading all images tracked by this media tracker with the specified identifier.

waitForID(int, long) - Starts loading all images tracked by this media tracker with the specified identifier.

Checking the Load Status of Images

It is also possible to use the following methods to check the status of one or more images on the list without blocking the thread. This makes it possible to continue doing other work while the image(s) are loading. These methods return **true** or **false** to indicate if loading is complete.

You will note that there are two overloaded versions of each of these methods. The version with the **boolean** parameter will start loading any images that have not already started loading if the **boolean** parameter is **true**. The other version will not start loading any images. That is the general interpretation of the **boolean** parameter in the other methods of this class that have a **boolean** parameter.

checkAll() - Checks to see if all images being tracked by this media tracker have finished loading.

checkAll(boolean) - Checks to see if all images being tracked by this media tracker have finished loading.

checkID(int) - Checks to see if all images tracked by this media tracker that are tagged with the specified identifier have finished loading.

checkID(int, boolean) - Checks to see if all images tracked by this media tracker that are tagged with the specified identifier have finished loading.

Testing for Successful Load

The fact that one of the above methods indicates that loading is complete is not a guarantee that the load was successful. The following methods can be used to determine if there was a problem loading the images. These methods either return a list or a **boolean**, depending on which is required to satisfy the intent of the method.

getErrorsAny() - Returns a list of all media that have encountered an error.

getErrorsID(int) - Returns a list of media with the specified ID that have encountered an error.

isErrorAny() - Checks the error status of all of the images.

isErrorID(int) - Checks the error status of all of the images tracked by this media tracker with the specified identifier.

Get the Numeric Status Value

The following methods return the bitwise inclusive **OR** of the integer status values of the images being inquired about.

statusAll(boolean) - Calculates and returns the bitwise inclusive **OR** of the status of all media that are tracked by this media tracker.

statusID(int, boolean) - Calculates and returns the bitwise inclusive **OR** of the status of all media with the specified identifier that are tracked by this media tracker.

First Sample Program

This program illustrates loading and drawing an image without using **MediaTracker** to improve the visual effect. The use of **MediaTracker** is illustrated in the second and third sample programs in this lesson.

This program also illustrates:

- Use of the **Toolkit** class and the **getImage()** method to associate an image file on the local hard disk with the name of a reference variable of type **Image**.
- Use of the **drawImage()** method to draw a loaded image onto a **Frame** object.
- Use of the **getWidth()** and **getHeight()** methods to determine the size of the image for drawing purposes.

- Use of the **translate()** method to eliminate the coordinate offset caused by the insets of a **Frame** object.
- Use of an anonymous inner-class to service the "close" button on a Frame object. This program was tested using JDK 1.1.3 under Win95.

Interesting Code Fragments for First Sample Program

The first interesting code fragment is the declaration of an instance variable which is a reference variable of type **Image**. This reference is used later to manipulate the image.

```
class Image01 extends Frame{ //controlling class
  Image image; //reference to an Image object
```

The next interesting code fragment is the statement inside the constructor for the container that uses the **getImage()** method of the **Toolkit** class to associate the image file named **logomain.gif** with the reference variable named **image**.

It is important to note that this statement does not cause the image to appear on the screen when its container becomes visible. Rather, the container is visible for a noticeable period of time before the image is actually drawn on the surface of that container.

```
    image =

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

The next interesting code fragment is the code inside the overridden **paint()** method that invokes one overloaded version of the **drawImage()** method on the graphics context for the container. The parameter list for the version used specifies the image to be drawn by the name of the reference variable that refers to the image file on the local hard disk as its first parameter.

The parameter list also specifies the coordinates where the upper left-hand corner of the image is to be placed when the image is drawn (0,0 in this case).

The parameter list specifies the width and height of the rendered image. Note that this version of the **drawImage()** method automatically scales the image to force it to meet this size requirement.

The **getWidth()** and **getHeight()** methods of the **Image** class were used to determine the original width and height of the image before scaling. These dimensions were then divided by 2 and passed as parameters so that the rendered version of the image would be half its original size.

Note that the **getWidth()** and **getHeight()** methods require the image observer as a parameter. In this case, the container object specified by **this** is the image observer.

Finally, the **drawImage()** method requires the image observer as its last parameter. Again, **this** was the image observer object.

```
    //Now draw it half its normal size.
    g.drawImage(image,0,0,image.getWidth(this)/2,
```

```
                                image.getHeight(this)/2,this);
```

A complete program listing for the first sample program follows in the next section.

Program Listing for First Sample Program

Some of the interesting code fragments in the following program listing are highlighted in **boldface**.

```
/*File Image01.java
Copyright 1997, R.G.Baldwin

This program was tested using JDK 1.1.3 under Win95.

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

class Image01 extends Frame{ //controlling class
  Image image; //reference to an Image object

  public Image01(){//constructor
    this.setTitle("Copyright 1997, R.G.Baldwin");
    this.setSize(350,200);

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

    //Make the Frame object visible.  Note that the image
    // is not visible on the Frame object when it first
    // appears on the screen.
    this.setVisible(true);

    //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
  public void paint(Graphics g){
    //Translate origin to avoid insets.
    g.translate(
              this.getInsets().left,this.getInsets().top);

    //Now draw it half its normal size.
    g.drawImage(image,0,0,image.getWidth(this)/2,
                              image.getHeight(this)/2,this);
```

```
  }//end paint()
//=======================================================//
  public static void main(String[] args){
    new Image01();//instantiate this object
  }//end main
}//end Image01 class
//=======================================================//
```

Second Sample Program

The previous program did not provide a pleasing visual effect while the image was being drawn on the screen as it was loading. For the image being used in the test, which was not particularly large, there was a lot of flashing while the image was being drawn on the screen.

This problem was eliminated in the second program by using a **MediaTracker** object to block the thread until the image was fully loaded before attempting to draw it on the screen.

A maximum allowable load time of one second was specified. If the image failed to load successfully within one second, the program would display a message to that effect and terminate. A larger image, or an image loaded from a slower source (such as a web server) might require more time.

Interesting Code Fragments for Second Sample Program

Note that I am skipping those code fragments in this program which duplicate the previous program. The first interesting new code fragment in this program is the statement that instantiates a **MediaTracker** object. The parameter to the constructor is the **Component** whose images are to be tracked, which in this case is **this**.

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

The next interesting code fragment is the statement to add the image to the list of images being tracked by the **MediaTracker** object. In this case, the parameters are:

- a reference to the image file and
- an *identifier* which is used later to inquire about the load status of the image.

Any **int** value can be used as the identifier. Recall that this value also represents the priority for tracking with low values having the higher priority.

```
    tracker.addImage(image,1);//add image to tracker list
```

The next interesting code fragment is the invocation of the **waitForId()** method on the tracker object. In this case, the identifier of the image being tracked is passed as a parameter. Also a value of 1000 milliseconds is passed as a parameter to cause the program to cease blocking and terminate if the load of the image isn't completed within that time interval.

This particular method returns **true** if the image was successfully loaded and returns **false** otherwise. Some of the other methods that can be used for similar purposes return void, in which case other means should be taken to determine if the image was successfully loaded.

In this program, the return value is used to terminate the program if the image fails to load successfully. (Note the "!" operator in front of the word tracker)..

This method also throws an **InterruptedException** object which is a *checked* exception. Therefore, the method call is enclosed in a **try** block followed by a **catch** block.

```
    try{
      //Block for up to one second while trying to load
      // the image. A larger image may require more time.
      if(!tracker.waitForID(1,1000)){
                System.out.println("Failed to load
image");
                System.exit(0);
        }//end if
    }catch(InterruptedException e){System.out.println(e);}
```

Finally, the **drawImage()** method is used as before to draw the image on its container once it is loaded. This program provides a much more pleasing visual experience than the previous program because only one draw operation is executed to draw the entire image.

```
   //Now draw it half its normal size.
    g.drawImage(image,0,0,image.getWidth(this)/2,
                              image.getHeight(this)/2,this);
```

A complete program listing for this program follows in the next section.

Program Listing for Second Sample Program

Some of the interesting code fragments in this complete program listing are highlighted in **boldface.**

```
/*File Image02.java
Copyright 1997, R.G.Baldwin

This program illustrates the fetch and display of an
image.

In doing so, it also illustrates:

Use of the Toolkit class and the getImage() method to read
an image file from the local hard disk.

Use of a MediaTracker object to monitor the loading of
an image file from the local hard disk and block the
thread until the image is loaded.

Use of the drawImage() method to display a loaded image
onto a Frame object.

Use of the getWidth() and getHeight() methods to determine
```

```
the size of the image for drawing purposes.

Use of the translate() method to eliminate the coordinate
offset caused by the insets of a Frame object.

Use of an anonymous inner-class to service the "close"
button on a Frame object.

This program was tested using JDK 1.1.3 under Win95.

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

class Image02 extends Frame{ //controlling class
  Image image; //reference to an Image object

  public Image02(){//constructor
    this.setTitle("Copyright 1997, R.G.Baldwin");
    this.setSize(350,200);

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

    //Make the Frame object visible.  Note that the image
    // is not visible on the Frame object when it first
    // appears on the screen.
    this.setVisible(true);

    //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
  public void paint(Graphics g){
    //Translate origin to avoid insets.
    g.translate(
              this.getInsets().left,this.getInsets().top);

    //Use a MediaTracker object to block until the image
    // is fully loaded before attempting to draw it on the
    // screen. If the image fails to load successfully
    // within one second, terminate the program.  Without
    // the use of the MediaTracker object, the display of
    // the image is very choppy while it is being loaded
    // for the particular image being used.  A smaller
    // image may load and display more smoothly without
```

```
    // use of MediaTracker.
    MediaTracker tracker = new MediaTracker(this);
    tracker.addImage(image,1);//add image to tracker list
    try{
      //Block for up to one second while trying to load
      // the image. A larger image may require more time.
      if(!tracker.waitForID(1,1000)){
                System.out.println("Failed to load image");
                System.exit(0);
        }//end if
    }catch(InterruptedException e){System.out.println(e);}

    //Now draw it half its normal size.
    g.drawImage(image,0,0,image.getWidth(this)/2,
                           image.getHeight(this)/2,this);
  }//end paint()
//=====================================================//
  public static void main(String[] args){
    new Image02();//instantiate this object
  }//end main
}//end Image02 class
//=====================================================//
```

Third Sample Program

Although the two previous programs are rather simple, they illustrate a number of important concepts involved in the handling of images in Java. This third sample program illustrates some additional concepts.

In this program, five separate scaled instances of an image are created and drawn. This program illustrates the use of the **getScaledInstance()** method of the Image class for producing multiple instances of an image with different scaling.

The program invokes the **getImage()** method on an object of the **Toolkit** class to associate an **Image** reference named **rawImage** to an image file on the local hard disk.

Then the program invokes the **getScaledInstance()** method on the **rawImage** object five times in succession to produce five new scaled instances of the image. A different scaling algorithm is used for each of the new scaled instances of the image.

A **MediaTracker** object is instantiated in the overridden **paint()** method. The **rawImage** object as well as the five new scaled instances are added to the list being tracked by the **MediaTracker** object. All are added with the same identifier (same priority).

The **waitForAll()** method of the **MediaTracker** class is used to block the thread until all six of the image objects are loaded and properly scaled. A noticeable pause of about 4 to 5 seconds occurs at this point using JDK 1.1.3 under Win95 with a 133mhz Pentium processor. (This will probably vary depending on the complexity and size of the image.)

All five scaled images are then drawn on the screen. There are no obvious pauses at this point. All five images appear on the screen almost simultaneously.

In an earlier program written by this author, but not included in this lesson, when a version of the **drawImage()** method that scales while drawing the images was used, and several scaled versions of the image were drawn, there were noticeable pauses and a noticeable progression from the first to the last image as they were being drawn on the screen.

Thus, the tradeoff is to incur the earlier pause while the thread is blocked and the images are being scaled in order to achieve very fast drawing of the scaled images. This program was tested using JDK 1.1.3 under Win95.

Interesting Code Fragments for Third Sample Program

For the most part, only those code fragments that were not highlighted in one of the previous programs will be highlighted here.

The first interesting code fragment uses the **getImage()** method to associate an **Image** reference variable named **rawImage** to an image file in the current directory on the local hard disk. This is essentially the same as the previous program. However, the reference variable named **rawImage** is then used to create five additional scaled instances of the image using the **getScaledInstance()** method.

Each of the five scaled instances are created using a different scaling algorithm where the algorithm being used is specified by the symbolic constant in the third position of the parameter list (such as **SCALE_AREA_AVERAGING** for example).

The width and the height of the scaled instance are specified by the first two parameters. If either of these parameters is -1, the scaled size in pixels of the scaled instance is based on the other parameter and the same height-to-width ratio as the original is maintained. If non-negative values are passed for both parameters, the image will be scaled to fit the rectangle defined by those parameters.

A parameter value of -1 was used as the second parameter in all five invocations of **getScaledInstance()** in this program. Also, all five scaled instances were scaled to the same size using different scaling algorithms.

```
    rawImage =

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

    image1 = rawImage.getScaledInstance(
                          200,-
1,Image.SCALE_AREA_AVERAGING);
    image2 = rawImage.getScaledInstance(
                          200,-1,Image.SCALE_DEFAULT);
    image3 = rawImage.getScaledInstance(
                          200,-1,Image.SCALE_FAST);
```

```
    image4 = rawImage.getScaledInstance(
                         200,-1,Image.SCALE_REPLICATE);
    image5 = rawImage.getScaledInstance(
                         200,-1,Image.SCALE_SMOOTH);
```

The next interesting code fragment occurs in the overridden **paint()** method where the six **Image** references are added to the list of images being tracked by a **MediaTracker** object. All six references are added at the same priority level.

```
    MediaTracker tracker = new MediaTracker(this);
    //Add images to the tracker list
    tracker.addImage(rawImage,1);
    tracker.addImage(image1,1);
    tracker.addImage(image2,1);
    tracker.addImage(image3,1);
    tracker.addImage(image4,1);
    tracker.addImage(image5,1);
```

The next interesting code fragment is the code to block the thread until all six of the images have completed loading and scaling.

```
    try{
      tracker.waitForAll();
    }catch(InterruptedException e){System.out.println(e);}
```

The final interesting code fragment shows the use of a *non-scaling* version of the **drawImage()** method to draw the five pre-scaled images. Because the method is not required to scale the images as they are being drawn, the images appear on the screen almost simultaneously (in the blink of an eye).

Note that the **isErrorAny()** method was used (with a **not** operator) to confirm that there were no load errors prior to invoking the **drawImage()** method to render the images to the screen.

```
    if(!tracker.isErrorAny()){
      g.drawImage(image1,0,0,this);
      g.drawImage(image2,0,80,this);
      g.drawImage(image3,0,160,this);
      g.drawImage(image4,0,240,this);
      g.drawImage(image5,0,320,this);
    }//end if
    else{
      System.out.println("Load error");
      System.exit(1);
    }//end else
```

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

Program Listing for Third Sample Program

Some of the interesting code fragments are highlighted in **boldface** in the program listing that follows.

```
/*File Image03.java
Copyright 1997, R.G.Baldwin

This program illustrates the use of the getScaledInstance()
method of the Image class.

The program invokes the getImage() method on an object of
the Toolkit class to associate an Image reference named
rawImage to an image file on the local hard disk.

Then the program invokes the getScaledInstance() method
on the rawImage object five times in succession to produce
five new scaled instances of the image.  A different
scaling algorithm is implemented for each of the new
scaled instances of the image.

A MediaTracker object is instantiated in the overridden
paint() method.  The rawImage object as well as the five
new scaled instances are added to the list being tracked
by the MediaTracker object.  All are added with the same
identifier (same priority).

The waitForAll() method of the MediaTracker class is used
to block the thread until all six of the image objects are
loaded and properly scaled.  A noticeable pause of about
4 or 5 seconds occurs at this point using JDK 1.1.3 under
Win95 with a 133mhz Pentium processor.

All five scaled images are then drawn on the screen.  There
are no obvious pauses at this point.  All five images
appear on the screen almost simultaneously.

In an earlier program written by this author, but not
included in the lesson, when a version of the drawImage()
method that scales while drawing the images was used,
and several scaled versions of the image were drawn,
there was a noticeable progression from the first to the
last image as they were being drawn on the screen.

Thus, the tradeoff is to incur the earlier pause in order
to achieve very fast drawing of the scaled images.

This program was tested using JDK 1.1.3 under Win95.

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

class Image03 extends Frame{ //controlling class
  //references to Image objects
  Image rawImage,image1,image2,image3,image4,image5;

  public Image03(){//constructor
    this.setTitle("Copyright 1997, R.G.Baldwin");
```

```
    this.setSize(300,500);

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

    //Create five scaled instances of the image using each
    // of the five different scaling algorithms
    image1 = rawImage.getScaledInstance(
                         200,-
1,Image.SCALE_AREA_AVERAGING);
    image2 = rawImage.getScaledInstance(
                         200,-1,Image.SCALE_DEFAULT);
    image3 = rawImage.getScaledInstance(
                         200,-1,Image.SCALE_FAST);
    image4 = rawImage.getScaledInstance(
                         200,-1,Image.SCALE_REPLICATE);
    image5 = rawImage.getScaledInstance(
                         200,-1,Image.SCALE_SMOOTH);

    //Make the Frame object visible.  Note that the image
    // is not visible on the Frame object when it first
    // appears on the screen.
    this.setVisible(true);

    //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
  public void paint(Graphics g){
    //Translate origin to avoid insets.
    g.translate(
              this.getInsets().left,this.getInsets().top);

    //Use a MediaTracker object to block until all images
    // are scaled and loaded before attempting to draw
    // them. Instantiate the object and add all five
    // scaled images to the list.
    MediaTracker tracker = new MediaTracker(this);
    //Add images to the tracker list
    tracker.addImage(rawImage,1);
    tracker.addImage(image1,1);
    tracker.addImage(image2,1);
    tracker.addImage(image3,1);
    tracker.addImage(image4,1);
    tracker.addImage(image5,1);
```

```
    try{
      //Block until all images on the list are scaled and
      // loaded.  There is a noticeable pause of 4 or 5
      // seconds at this point using JDK 1.1.3 under Win95
      // with a 133mhz Pentium processor.
      tracker.waitForAll();
    }catch(InterruptedException e){System.out.println(e);}

    //Now test for errors and draw all five images if no
    // errors were detected.  All five images appear
    // on the screen almost simultaneously.
    if(!tracker.isErrorAny()){
            g.drawImage(image1,0,0,this);
            g.drawImage(image2,0,80,this);
            g.drawImage(image3,0,160,this);
            g.drawImage(image4,0,240,this);
            g.drawImage(image5,0,320,this);
    }//end if
    else{
      System.out.println("Load error");
      System.exit(1);
    }//end else
  }//end paint()
//=====================================================//
  public static void main(String[] args){
    new Image03();//instantiate this object
  }//end main
}//end Image03 class
//=====================================================//
```

-end-