The show Method and the PictureFrame Class

Learn how the show() method of the Picture class causes the image contained in a Picture object to be displayed on the screen in a JFrame object. Also learn about the PictureFrame class, which serves as an intermediary between the Picture object and the JFrame object.

Published: March 18, 2009
By Richard G. Baldwin

Java Programming Notes # 356

Preface

General

This lesson is the next in a series *(see Resources)* designed to teach you how to write Java programs to do things like:

- Remove *redeye* from a photographic image.
- Distort the human voice *(as in secret witness interviews on TV)*.
- Display one image inside another image.
- Do edge detection, blurring, and other filtering operations on images.
- Insert animated cartoon characters into videos of live humans.

If you have ever wondered how to do these things, you've come to the right place.

What you have learned so far

If you have studied all of the earlier lessons in this series *(see Resources)*, you have learned about the **Turtle** class, its superclass named **SimpleTurtle**, and the classes from which a turtle's contained objects are instantiated *(**Pen** and **PathSegment**)*. You have learned how to instantiate new **Turtle** objects, placing them in either a **World** object or a **Picture** object. You have learned how to manipulate the **Turtle** objects once you place them in their environment. You have also learned about the **World** class and objects of that class.

Methods of the DigitalPicture interface

In the previous lesson titled *The DigitalPicture Interface: Multimedia Programming with Java (see Resources)*, you learned about the thirteen methods of the **Picture** class and its superclass named **SimplePicture** that are declared in the **DigitalPicture** interface. You also learned about other methods of the **SimplePicture** class that are called by those thirteen methods.

SimplePicture is a large and complex class

The **SimplePicture** class is a large and complex class that defines almost forty methods and several constructors. You have some distance to go before you will understand all of the methods and constructors that are defined in the **SimplePicture** class.

Dispose of the easy methods and constructors

Before getting into the main thrust of this lesson, I am going to dispose of some of the constructors and methods of the **SimplePicture** class that should be easy for you to understand without an explanation from me.

The following is a list of constructors and methods which do not contain complicated code, and which you should have no difficulty understanding if you understood the explanations of constructors and methods in the previous lesson *(see Resources)*. Because of their simplicity, I won't bore you by providing a detailed explanation. You can view the source code for all of these constructors and methods in Listing 13 near the end of the lesson.

- **SimplePicture()** - A constructor that takes no parameters and constructs a **SimplePicture** object with dimensions of 200x100 pixels.
- **SimplePicture(int width,int height,Color theColor)** - Constructs a **SimplePicture** object with the specified dimensions and color.
- **String getExtension()** - Returns the file-name extension of the file from which the **SimplePicture** object's image was extracted.
- **Graphics getGraphics()** - Returns the *graphics context* for the **BufferedImage** object *(as type **Graphics**)* owned by this **SimplePicture** object.
- **Graphics2D createGraphics()** - Returns a reference to a **Graphics2D** object that can be used to call methods of the **Graphics2D** class on the **BufferedImage** object that belongs to the **SimplePicture** object.
- **void setFileName(String name)** - Stores a reference to a **String** object in the **SimplePicture** object's **fileName** variable.
- **PictureFrame getPictureFrame()** - Returns a reference to the **PictureFrame** object belonging to the **SimplePicture** object.
- **void setPictureFrame(PictureFrame pictureFrame)** - Stores a **PictureFrame** object's reference in the **SimplePicture** variable named **pictureFrame**.
- **void hide()** - Sets the visible property belonging to the **PictureFrame** object to *false* causing the picture to disappear from the screen.
- **void setVisible(boolean flag)** - Causes the **SimplePicture** object to be visible or invisible depending on the value of the parameter.
- **void repaint()** - Forces the **SimplePicture** object to repaint itself on the screen.
- **boolean loadImage(String fileName)** - Simply calls the **load** method explained in the previous lesson.
- **static void setMediaPath(String directory)** - Sets the name and path of the directory from which media files will be read.
- **static String getMediaPath(String fileName)** - Returns the name and path of the directory from which media files are currently being read.
- **String toString()** - Overridden **toString** method that returns information about the **SimplePicture** object. Note that this method is overridden again in the **Picture** class.

Methods that I probably will explain later

Click here for a list of constructors and methods of the **SimplePicture** class that are sufficiently interesting or complicated that I will probably explain them in detail in future lessons.

What you will learn in this lesson

Two of the most frequently used methods of the **SimplePicture** class are the methods named **show** and **explore**. These two methods are used to display **Picture** objects in the formats shown in Figure 1 and Figure 2 respectively.

Figure 1. Output format from the show method.

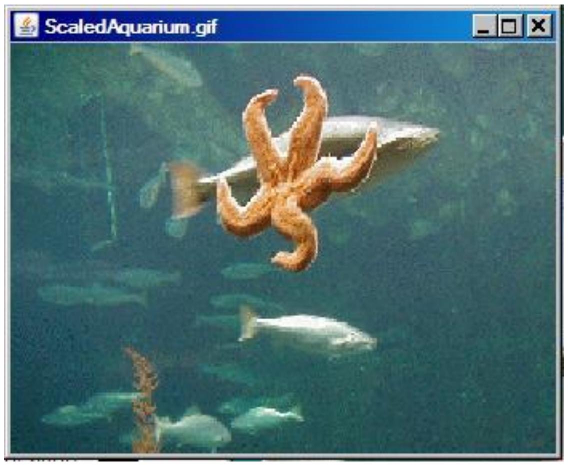
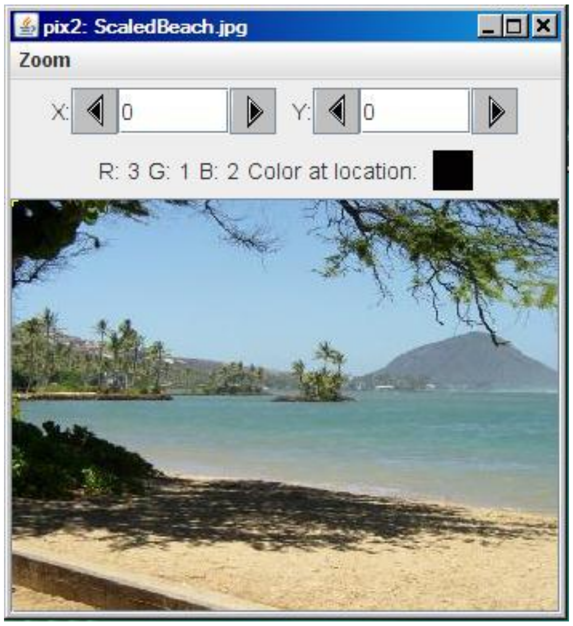


Figure 2. Output format from the explore method.



In this lesson, you will learn about the **show** method of the **Picture** class, along with a related class named **PictureFrame**, *(which you must understand before you can fully understand the* ***show*** *method)*.

A sample program

I will also present and explain a sample program that illustrates one way to take a photograph of a physical object and then superimpose it on another photograph. I confess that this doesn't have a much to do with the **show** method. However, I didn't want to pass up the opportunity to provide another interesting example of image manipulation using Ericson's multimedia library.

The explore method

You will learn about the **explore** method, along with a related class named **PictureExplorer**, *(which you must understand before you can understand the **explore** method)*, in a future lesson.

Source code listings

A complete listing of Ericson's **Picture** class is provided in Listing 12 near the end of the lesson, and a listing of Ericson's **SimplePicture** class is provided in Listing 13. A listing of Ericson's **DigitalPicture** interface is provided in Listing 14, and a listing of Ericson's **PictureFrame** class is provided in Listing 15. A listing of the sample program named Java356a is provided in Listing 16.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

Listings

Supplementary material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com.

General background information

A multimedia class library

In this series of lessons, I will present and explain many of the classes and methods in a multimedia class library that was developed and released under a **Creative Commons Attribution 3.0 United States License** *(see Resources)* by Mark Guzdial and Barbara Ericson at Georgia Institute of Technology. In doing this, I will also present some interesting sample programs that use the library.

Software installation and testing

I explained how to download, install, and test the multimedia class library in an earlier lesson titled *Multimedia Programming with Java, Getting Started (see Resources)*.

Preview

As I mentioned earlier, I will explain the **show** method of the **Picture** class in this lesson. In addition, I will explain the methods of a related class named **PictureFrame**, which you must understand before you can fully understand the **show** method.

A sample program

As mentioned earlier, I will also present and explain a sample program that illustrates one way to take a photograph of a physical object and then superimpose it on another photograph.

Reducing the confusion

Because I will be switching back and forth among code fragments extracted from Ericson's **SimplePicture** class, code fragments extracted from Ericson's **PictureFrame** class, and code fragments extracted from my sample program, things can get confusing.

In an effort to reduce the confusion, I will present code fragments from Ericson's **SimplePicture** class against the background color shown in Listing 1.

Listing 1. Background color for the SimplePicture class.

```
I will present code fragments from the
SimplePicture class
against this background color.
```

Similarly, I will present code fragments from Ericson's **PictureFrame** class against the background color shown in Listing 2.

Listing 2. Background color for the PictureFrame class.

```
I will present code fragments from the
PictureFrame class
against this background color.
```

Finally, I will present code fragments from my sample program against the background color shown in Listing 3.

Listing 3. Background color for Baldwin's code.

```
I will present code fragments from my sample
programs
with this background color.
```

Discussion and sample code

The sample program named Java356a

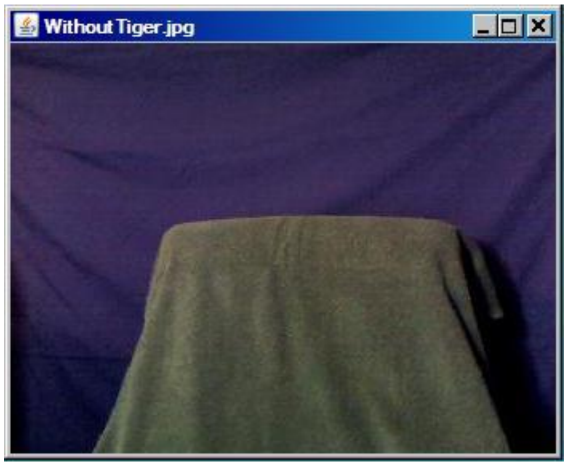
The purpose of this program is to illustrate one way to take a digital photograph of a physical object and then superimpose it on another digital photograph.

The physical setup

A desk chair was placed in front of a bookcase and a blue bed sheet was hung on the bookcase to provide a relatively solid color background. A green towel was placed on the chair to hide the texture in the upholstery. The lighting consisted of ambient room light, a fluorescent desk lamp tilted upward to shine directly on the chair and the tiger, and the light emitted by the dual screens on my computer.

A digital photograph of the chair was taken, using the camera that is built into my laptop computer, producing the image shown in Figure 3.

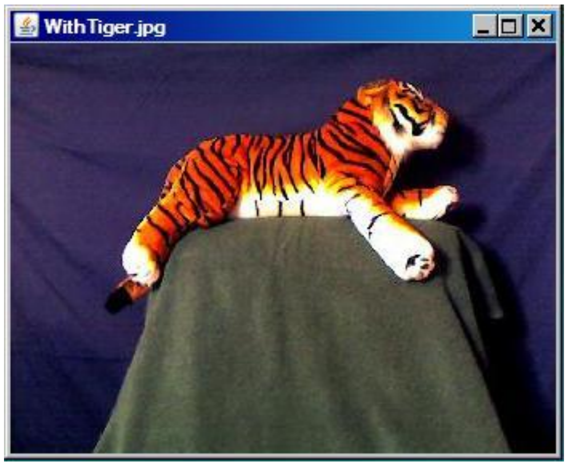
Figure 3. Picture of the chair covered by a towel.



Now add the tiger

Then a stuffed tiger was placed on the back of the chair *(being careful to avoid moving the chair)* and another digital photograph was taken, producing the image shown in Figure 4. *(I just now realized that it might have worked better to also cover the chair with the sheet instead of covering it with a towel of a different color.)*

Figure 4. Picture of the tiger on the chair.

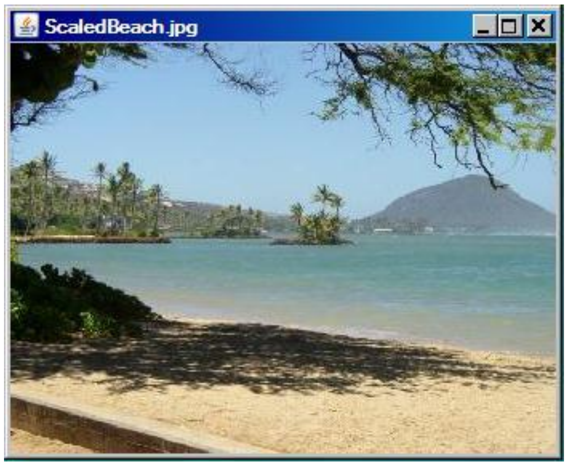


Note the shadow of the tiger on the blue background just to the right of the tiger's head. This will become important later.

Instantiate four Picture objects

Picture objects were instantiated from each of the digital photographs described above. Another **Picture** object was instantiated from an image file *(taken from Ericson's media library)* showing a beach scene. This image has the same dimensions as each of the two digital photographs. The beach scene is shown in Figure 5.

Figure 5. The beach scene.



An all-white Picture object

The overall objective was to process the images in such a way as to superimpose the tiger on the beach scene.

A fourth **Picture** object was instantiated with the same dimensions and an all-white image to accommodate this objective. This **Picture** object was used to produce the final image shown in Figure 4.

Process the images

Color distance
I explained the concept of *color distance* in the earlier lesson titled *3D Displays, Color Distance, and Edge Detection: Multimedia Programming with Java* (see Resources).

Methods of the **SimplePicture** class and the **Pixel** class were used in a pair of nested **for** loops to compute the *color distance* between corresponding pixels in the two photographs and to compare that distance to a specified distance threshold.

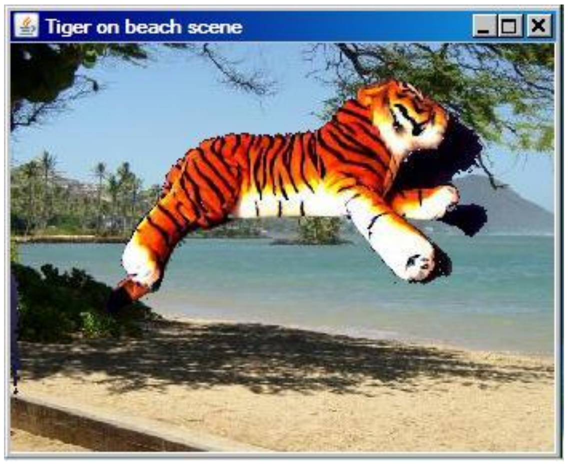
When the color distance between two corresponding pixels, *(one from each photograph)*, exceeded a specified threshold, the color of the pixel from the **Picture** object containing the tiger was copied to the all-white **Picture** object, replacing a white pixel.

Otherwise, the color of the pixel from the **Picture** object containing the beach scene was copied to the all-white **Picture** object. This had the effect of replacing the bed sheet and towel background behind the tiger with the image from the beach scene as a new background .

The final product

The final product is shown in Figure 6. As you can see, the results were reasonably good.

Figure 6. The tiger superimposed on the beach scene.



Factors that affect the quality

The quality of the final product depends heavily on the value of the threshold mentioned above.

Lighting is also very critical. I make no claims of being a photographer and I didn't do anything special to control the lighting. As a result, the shadow of the tiger that was barely noticeable on the dark blue bed sheet in Figure 4 is very noticeable against the light blue background in the final product shown in Figure 6.

Attribution
The idea and some of the methodology for this program came directly from the book titled *Introduction to Computing and Programming with Java: A Multimedia Approach (see Resources)* by Guzdial and Ericson.

Enough talk, let's see some code

A complete listing of this program is provided in Listing 16 near the end of the lesson. As is my custom, I will present and explain this program in fragments. The program begins in Listing 4. *(Remember, the background color in Listing 4 indicates that the code fragment was extracted from my sample program.)*

Listing 4. Beginning of the program named Java356a.

```
import java.awt.Color;

public class Main{
  public static void main(String[] args){
    new Runner().run();
  }//end main method
}//end class Main
```

The **main** method in Listing 4 instantiates a new object of the **Runner** class and calls the **run** method on the object. When the **run** method returns, the **main** method and the program terminate.

Beginning of the Runner class and the run method

The **runner** class and its **run** method begin in Listing 5.

Listing 5. Beginning of the Runner class and the run method.

```
class Runner{
  void run(){
    //Construct three new 341x256 Picture
objects by
    // providing the names of image files as
parameters
    // to the Picture constructor.
    Picture pic1 = new
Picture("ScaledBeach.jpg");
    Picture pic2 = new
Picture("WithTiger.jpg");
    Picture pic3 = new
Picture("WithoutTiger.jpg");

    //Construct an all-white 341x256 Picture
object.
    Picture pic4 = new Picture(341,256);

    //Display all three Picture objects in the
show
    // format.
    pic1.show();
    pic2.show();
    pic3.show();
```

None of the code in Listing 5 should be new to you by now. The last three statements in Listing 5 call the **show** method of the **Picture** class to provide screen displays in the format shown in Figure 3. *(I will have a great deal more to say about the **show** method later in this lesson.)*

The remainder of the program code

The remainder of the program code is shown in Listing 6.

Listing 6. The remainder of the program code for the program named Java356a.

```
    //Replace pixel colors in the all-white
Picture object
```

```
    // with the colors from either the beach
image or the
    // tiger image.
    Pixel pixA;
    Pixel pixB;
    Pixel pixC;
    Pixel pixD;
    for(int row = 0;row < pic1.getHeight() -
1;row++){
      for(int col = 0;col < pic1.getWidth() -
1;col++){
        pixA = pic1.getPixel(col,row);
        pixB = pic2.getPixel(col,row);
        pixC = pic3.getPixel(col,row);
        pixD = pic4.getPixel(col,row);

        if(pixB.colorDistance(pixC.getColor())
> 50){
          //Replace white pixel with the pixel
color from
          // the tiger image.
          pixD.setColor(pixB.getColor());
        }else{
          //Replace the white pixel with pixel
color from
          // the beach image.
          pixD.setColor(pixA.getColor());
        }//end else
      }//end inner for loop
    }//end outer for loop

    //Display the final product using the show
format.
    pic4.setTitle("Tiger on beach scene");
    pic4.show();
  }//end run
}//end class Runner
```

None of this code is new

Once again, none of the code in Listing 6 should be new to you by now. The code in Listing 6 compares corresponding pixels from the two digital pictures and modifies the colors of the pixels in the all-white **pic4** as described earlier.

Then Listing 6 sets the title for **pic4** to that shown in Figure 6 and calls the **show** method on **pic4** producing the screen output shown in Figure 6. *(You may not want to go swimming on that beach with a tiger on the loose.)*

The show method of the SimplePicture class

The show method is shown in its entirety in Listing 7. *(Remember, the background color in Listing 7 indicates that the code fragment was extracted from Ericson's **SimplePicture** class.)*

Listing 7. The show method of the SimplePicture class.

```
 /**
  * Method to show the picture in a picture
frame
  */
 public void show(){
    // if there is a current picture frame
then use it
   if (pictureFrame != null)
     pictureFrame.updateImageAndShowIt();

   // else create a new picture frame with
this picture
   else
     pictureFrame = new PictureFrame(this);
 }//end show method
```

Instantiating objects of the PictureFrame class

There are only two locations in the **SimplePicture** class where a new object of the **PictureFrame** class is instantiated. One of those locations is in the **show** method shown in Listing 7. The other location is in the repaint method. Therefore, there is a strong possibility that the contents of the variable named **pictureFrame** will be null *(indicating there is no existing **PictureFrame** object)* the first time that the **show** method is called.

Listing 7 begins by checking to see if the instance variable named **pictureFrame** contains null. If so a new **PictureFrame** object that encapsulates a reference to the current **SimplePicture** object is instantiated and its reference is assigned to the instance variable named **pictureFrame**.

The PictureFrame class

That brings us to the need to understand the class named **PictureFrame**. I will begin by walking you through the code that is executed for the case where the **show** method is called and there is no existing **PictureFrame** object assigned to the variable named **pictureFrame**. Then I will come back and walk you through the code that is executed if the **PictureFrame** object already exists when the **show** method is called.

Instance variables of the PictureFrame class

Before getting into that, however, I will list and briefly describe the instance variables belonging to objects of the **PictureFrame** class. I will be referring back to these instance variables later.

- JFrame **frame** = new JFrame() - *Main window used to display the image from the Picture object.*
- ImageIcon **imageIcon** = new ImageIcon() - *ImageIcon object used to display the picture in a label.*
- JLabel **label** = new JLabel(imageIcon) - *Label used to display the picture.*
- DigitalPicture **picture** - *The Picture object to display.*

A new JFrame object

Note that when a new **PictureFrame** object is constructed, a new **JFrame** object is also constructed and its reference is stored in the instance variable named **frame**. This **JFrame** object provides the visual container in which the **Picture** object's image is displayed.

A complete listing of the **PictureFrame** class is provided in Listing 15 near the end of the lesson.

Case with no existing PictureFrame object

When the **show** method in Listing 7 is called and the **pictureFrame** variable contains null, the **PictureFrame** constructor shown in Listing 8 is called to construct a new **PictureFrame** object. *(Remember, the background color in Listing 8 indicates that the code fragment was extracted from the **PictureFrame** class.)*

Listing 8. One of two overloaded constructors for the PictureFrame class.

```
  /**
   * A constructor that takes a picture to
display
   * @param picture  the digital picture to
display in the
   * picture frame
   */
  public PictureFrame(DigitalPicture picture){
    // set the current object's picture to the
passed in
    // picture
    this.picture = picture;

    // set up the frame
    initFrame();
  }//end constructor
```

This constructor saves the incoming reference to the **SimplePicture** object in an instance variable named **picture**. *(Recall that the **SimplePicture** class implements the **DigitalPicture** interface.)*

Then the constructor calls the method named **initFrame** *(see Listing 9)* on the new **PictureFrame** object that is being constructed.

The initFrame method of the PictureFrame class

The **initFrame** method is shown in its entirety in Listing 9.

Listing 9. The initFrame method of the PictureFrame class.

```
  /**
   * A method to initialize the picture frame
   */
  private void initFrame(){
    // set the image for the picture frame
    updateImage();

    // add the label to the frame
    frame.getContentPane().add(label);

    // pack the frame (set the size to as big
as it needs
    // to be)
    frame.pack();

    // make the frame visible
    frame.setVisible(true);
  }//end initFrame method
```

The **initFrame** method immediately calls the **updateImage** method shown in Listing 10. *(I will put the explanation of the **initFrame** method on the back burner for now and return to it later.)*

The updateImage method of the PictureFrame class

Code in the **updateImage** method is executed only if a reference to a **Picture** object is stored in the instance variable named **picture**.

Listing 10. The updateImage method of the PictureFrame class.

```
  /**
   * A method to update the picture frame
image with the
   * image in the picture
   */
  public void updateImage(){
    // only do this if there is a picture
    if (picture != null){
      // set the image for the image icon from
the picture
      imageIcon.setImage(picture.getImage());

      // set the title of the frame to the
title of the
      // picture
      frame.setTitle(picture.getTitle());
    }//end if
  }//end updateImage method
```

An ImageIcon object

As you saw earlier in the list of instance variables, a new object of the **ImageIcon** class is instantiated when a new object of the **PictureFrame** class is constructed. The object's reference is stored in the instance variable named **imageIcon**.

Get the image from the Picture object

Listing 10 calls the **getImage** method on the reference to the **Picture** object to get a reference to the **BufferedImage** object that belongs to the **Picture** object. *(The* ***getImage*** *method contains a single line of code that returns the value of an instance variable named* ***bufferedImage*** *so I won't show that code here.)*

Set the image in the ImageIcon object

Then Listing 10 calls the **setImage** method on the **ImageIcon** object to *"Set the image displayed by the icon," passing the reference to the Picture object's image as a parameter.*

At this point, an **ImageIcon** object has been instantiated that holds a reference to the **BufferedImage** object that belongs to the **Picture** object. Whenever the **ImageIcon** object is displayed, the image belonging to the **Picture** object will also be displayed.

Because the **ImageIcon** object and the **Picture** object refer to the same **BufferedImage** object, modifications to the pixels in the **Picture** object's image will be reflected on the screen the next time the **ImageIcon** object is displayed.

Set the title on the JFrame object

Then Listing 10 gets a reference to the **Picture** object's title, and calls the **setTitle** method on the **JFrame** object, causing the **JFrame** object and the **Picture** object to both refer to the same **String** object as a title.

Modifications to the Picture object's image or title

If the **BufferedImage** or the **String** title belonging to the **Picture** object is modified and then the **PictureFrame** object's **updateImage** method is called, the image referred to by the **ImageIcon** object and the title referred to by the **JFrame** object will reflect the changes.

That's a wrap on the updateImage method

That concludes the explanation of the **updateImage** method shown in Listing 10. Now we will return to the explanation of the **initFrame** method shown in Listing 9.

A new JLabel object

As you saw earlier in the list of instance variables, whenever a new **PictureFrame** object is constructed, a new object of the **JLabel** class is instantiated and its reference is stored in the instance variable named **label**. Furthermore, a reference to the **ImageIcon** object is passed to the constructor when the **JLabel** object is instantiated.

Partial description of a JLabel from Sun

Figure 7 contains a partial description of the **JLabel** class from the Sun documentation.

Figure 7. Partial description of a JLabel from Sun.

A **JLabel** object can display either text, an image, or both... By default, labels are vertically centered in their display area. Text-only labels are leading edge aligned, by default; image-only labels are horizontally centered, by default.

Instantiate a JLabel object with an ImageIcon object's reference

If you instantiate a new **JLabel** object, passing only a reference to an **ImageIcon** object as a parameter to the constructor, this will create *"a **JLabel** instance with the specified image. The label is centered vertically and horizontally in its display area."*

Where is this all going?

By now, you should be starting to see where this is all going. The objective is to display the image belonging to the **Picture** object in a **JFrame** object. One way to do this is by encapsulating a reference to the image in a **JLabel** object and placing the **JLabel** object in the center of the **JFrame** object.

Only two types of encapsulated references are allowed

However, only two types of object references can be encapsulated in a **JLabel** object:

- **String** objects
- **ImageIcon** objects

There is no constructor for a **JLabel** object that will accept a reference to a **BufferedImage** object as a parameter.

Therefore, in order to cause a reference to the **Picture** object's **BufferedImage** to be encapsulated in the **JLabel** object, we must first encapsulate the **BufferedImage** object's reference in an **ImageIcon** object and then encapsulate that object in the **JLabel** object.

Add the JLabel object to the content pane

When the call to the **updateImage** method returns, the code in Listing 9 adds the **JLabel** object to the content pane of the **JFrame** object.

The content pane
If you are unfamiliar with the content pane, see the lesson titled *Swing from A to Z, Some Simple Components* in Resources.

Pack the JFrame object

Then Listing 9 calls the **pack** method on the **JFrame** object to set the size of the **JFrame** object.

What does the pack method do?

To make a long story short, the call to the **pack** method causes the size of the display area of the **JFrame** object *(see Figure 1)* to match the size of the picture's **BufferedImage** object. The overall size of the resulting **JFrame** object will be somewhat larger *(depending on the pluggable look and feel in use)* due to the border around the image and the banner at the top.

Pluggable Look and Feel
If you are unfamiliar with Swing's Pluggable Look and Feel, see the lesson titled *The Swing Package, A Preview of Pluggable Look and Feel* in Resources.

Make the JFrame object visible

Finally, the code in Listing 9 sets the **visible** property belonging to the **JFrame** object to true. This causes the **JFrame** object, along with all of the components *(including the **JLabel** object and the **ImageIcon** object)* contained within the **JFrame** object to become visible on the computer screen.

When the **ImageIcon** object becomes visible, it is really the **Picture** object's **BufferedImage** object, whose reference is held by the **ImageIcon** object, that becomes visible inside the **JFrame** object. *(See Figure 1.)*

Case with an existing PictureFrame object

Please return your attention to the **if-else** statement in the **show** method in Listing 7. If a **PictureFrame** object already exists, the **show** method calls the **updateImageAndShowIt** method on the **PictureFrame** object.

The updateImageAndShowIt method of the PictureFrame class

This method is shown in its entirety in Listing 11.

Listing 11. The updateImageAndShowIt method of the PictureFrame class.

```
  /**
   * A method to update the picture frame
image with the
   * image in the picture and show it
   */
  public void updateImageAndShowIt(){
    // first update the image
    updateImage();

    // now make sure it is shown
    frame.setVisible(true);
  }//end updateImageAndShowIt method
```

Listing 11 begins by calling the **updateImage** method that I explained in conjunction with Listing 10. As I explained at that time, if the **BufferedImage** pixels or the **String** title text belonging to the **Picture** object have been modified, the call to the **updateImage** method will cause those changes to be reflected in a subsequent screen display of the **Picture** object.

Display the updated picture

Then Listing 11 calls the **setVisible** method on the **JFrame** object, forcing the object to repaint itself *(and all of its children)* on the screen. The new screen image will reflect any changes that may have been made to the pixels in the image or the text in the title.

Return of the show method

When the **setVisible** method returns, the **updateImageAndShowIt** method terminates, returning control to the **show** method in Listing 7.

The **show** method has nothing else to do, so it terminates and returns control to the code from which it was originally called, such as the code in Listing 6 for example.

The Picture object will have been displayed

When the **show** method terminates, the **Picture** object on which it was called will have been displayed in a **JFrame** object as shown in Figure 6.

If the **show** method is called on more than one **Picture** object, the resulting images will overlay one another in the upper-left corner of the screen.

That concludes the explanation of the **show** method of the **Picture** class.

More methods of the PictureFrame class

The **PictureFrame** class provides several other methods that could prove to be useful in more complex programs. The signatures and the behaviors of each of those methods are described below:

- **void setPicture(Picture picture)** - sets the picture that will be displayed in the **JFrame** object *(the frame)*.
- **void displayImage()** - Makes sure that the frame is displayed.
- **void hide()** - Hides the frame.
- **void setVisible(boolean flag)** - Sets the visible flag on the frame.
- **void close()** - Closes and disposes of the **JFrame** object *(more permanent than simply hiding the frame)*.
- **void setTitle(String title)** - Sets the title for the frame.
- **void repaint()** - Forces the frame to repaint itself.

The **PictureFrame** class also provides a constructor that takes no parameters.

None of the code is complicated

Neither the constructor nor any of the methods in the above list contain complicated code. If you understood the earlier explanations of the **initFrame** and **updateImage** methods, you should have no difficulty understand the code in the additional constructor and methods. You can view the code for the constructor and the methods in Listing 15 near the end of the lesson.

Therefore, that also concludes the explanation of the **PictureFrame** class.

Run the programs

I encourage you to copy the code from Listing 16, compile the code, and execute it. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Summary

In this lesson, I explained the **show** method of the **Picture** class that causes the image contained in a **Picture** object to be displayed on the screen in a **JFrame** object as shown in Figure 1.

Along the way, I also explained the **PictureFrame** class, which serves as an intermediary between the **Picture** object and the **JFrame** object.

What's next?

In the next lesson, you will learn how to use three different methods, which in turn use affine transforms, to *scale*, *rotate*, and *translate* **Picture** objects.

I will explain and illustrate the following three methods and one overloaded constructor from the **SimplePicture** class:

- Picture **scale**(double xFactor, double yFactor)
- Rectangle2D **getTransformEnclosingRect**(AffineTransform trans
- void **copyPicture**(SimplePicture sourcePicture)
- **SimplePicture**(SimplePicture copyPicture)

The first two methods in the above list involve the application of affine transforms to **Picture** objects.

I will also develop and explain two additional methods that are patterned after the **scale** method. These two methods apply rotation and translation transforms to **Picture** objects.

Resources

- Creative Commons Attribution 3.0 United States License
- Media Computation book in Java - numerous downloads available
- Introduction to Computing and Programming with Java: A Multimedia Approach
- DrJava download site
- DrJava, the JavaPLT group at Rice University
- DrJava Open Source License
- The Essence of OOP using Java, The this and super Keywords
- Threads of Control
- Painting in AWT and Swing
- Wikipedia Turtle Graphics
- IsA or HasA
- Vector Cad-Cam XI Lathe Tutorial
- Classification of 3D to 2D projections
- Color model from Wikipedia
- Light and color: an introduction by Norman Koren
- Color Principles - Hue, Saturation, and Value
- 200 Implementing the Model-View-Controller Paradigm using Observer and Observable
- 300 Java 2D Graphics, Nested Top-Level Classes and Interfaces
- 302 Java 2D Graphics, The Point2D Class

- 304 Java 2D Graphics, The Graphics2D Class
- 306 Java 2D Graphics, Simple Affine Transforms
- 308 Java 2D Graphics, The Shape Interface, Part 1
- 310 Java 2D Graphics, The Shape Interface, Part 2
- 312 Java 2D Graphics, Solid Color Fill
- 314 Java 2D Graphics, Gradient Color Fill
- 316 Java 2D Graphics, Texture Fill
- 318 Java 2D Graphics, The Stroke Interface
- 320 Java 2D Graphics, The Composite Interface and Transparency
- 322 Java 2D Graphics, The Composite Interface, GradientPaint, and Transparency
- 324 Java 2D Graphics, The Color Constructors and Transparency
- 400 Processing Image Pixels using Java, Getting Started
  402 Processing Image Pixels using Java, Creating a Spotlight
  404 Processing Image Pixels Using Java: Controlling Contrast and Brightness
  406 Processing Image Pixels, Color Intensity, Color Filtering, and Color Inversion
  408 Processing Image Pixels, Performing Convolution on Images
  410 Processing Image Pixels, Understanding Image Convolution in Java
  412 Processing Image Pixels, Applying Image Convolution in Java, Part 1
  414 Processing Image Pixels, Applying Image Convolution in Java, Part 2
  416 Processing Image Pixels, An Improved Image-Processing Framework in Java
  418 Processing Image Pixels, Creating Visible Watermarks in Java
  450 A Framework for Experimenting with Java 2D Image-Processing Filters
  452 Using the Java 2D LookupOp Filter Class to Process Images
  454 Using the Java 2D AffineTransformOp Filter Class to Process Images
  456 Using the Java 2D LookupOp Filter Class to Scramble and Unscramble Images
  458 Using the Java 2D BandCombineOp Filter Class to Process Images
  460 Using the Java 2D ConvolveOp Filter Class to Process Images
  462 Using the Java 2D ColorConvertOp and RescaleOp Filter Classes to Process Images
- 506 JavaBeans, Introspection
- 2100 Understanding Properties in Java and C#
- 2300 Generics in J2SE, Getting Started
- 340 Multimedia Programming with Java, Getting Started
- 342 Getting Started with the Turtle Class: Multimedia Programming with Java
- 344 Continuing with the SimpleTurtle Class: Multimedia Programming with Java
- 346 Wrapping Up the SimpleTurtle Class: Multimedia Programming with Java
- 348 The Pen and PathSegment Classes: Multimedia Programming with Java
- 349 A Pixel Editor Program in Java: Multimedia Programming with Java
- 350 3D Displays, Color Distance, and Edge Detection
- 351 A Slider-Controlled Softening Program for Digital Photos
- 352 Adding Animated Movement to Your Java Application
- 353 A Slider-Controlled Sharpening Program for Digital Photos
- 354 The DigitalPicture Interface

- 355 The HSB Color Model

Complete program listings

Complete listings of the programs discussed in this lesson are provided in Listing 12 through Listing 16 below.

Listing 12 . Source code for Ericson's Picture class.

```
import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.awt.image.BufferedImage;
import java.text.*;

/**
 * A class that represents a picture.  This
class inherits
 * from SimplePicture and allows the student
to add
 * functionality to the Picture class.
 *
 * Copyright Georgia Institute of Technology
2004-2005
 * @author Barbara Ericson
ericson@cc.gatech.edu
 */
public class Picture extends SimplePicture
{
  ///////////////////// constructors
/////////////////////

  /**
   * Constructor that takes no arguments
   */
  public Picture ()
  {
    /* not needed but use it to show students
the implicit
     * call to super()
     * child constructors always call a parent
constructor
     */
    super();
  }

  /**
   * Constructor that takes a file name and
creates the
   * picture
   * @param fileName the name of the file to
create the
   * picture from
   */
  public Picture(String fileName)
  {
    // let the parent class handle this
fileName
    super(fileName);
  }

  /**
   * Constructor that takes the width and
height
   * @param width the width of the desired
picture
   * @param height the height of the desired
picture
   */
  public Picture(int width, int height)
  {
    // let the parent class handle this width
and height
    super(width,height);
  }

  /**
   * Constructor that takes a picture and
creates a
   * copy of that picture
   */
  public Picture(Picture copyPicture)
  {
    // let the parent class do the copy
    super(copyPicture);
  }

  /**
   * Constructor that takes a buffered image
   * @param image the buffered image to use
   */
  public Picture(BufferedImage image)
  {
    super(image);
  }

  ////////////////////// methods
/////////////////////////

  /**
   * Method to return a string with
information about this
   * picture.
   * @return a string with information about
the picture
   * such as fileName, height and width.
   */
  public String toString()
  {
    String output =
      "Picture, filename " + getFileName() +
      " height " + getHeight()
      + " width " + getWidth();
    return output;

  }

} // this } is the end of class Picture, put
all new
  // methods before this
```

Listing 13. Source code for Ericson's SimplePicture class.

```
import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
import javax.swing.ImageIcon;
import java.awt.*;
import java.io.*;
import java.awt.geom.*;

/**
 * A class that represents a simple picture.  A
simple
 * picture may have an associated file name and a
title.
 * A simple picture has pixels, width, and height.
A
 * simple picture uses a BufferedImage to hold the
pixels.
 * You can show a simple picture in a PictureFrame
(a
 * JFrame).
 *
 * Copyright Georgia Institute of Technology 2004
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public class SimplePicture implements
DigitalPicture
{

  /////////////////////// Fields
/////////////////////////

  /**
   * the file name associated with the simple
picture
   */
  private String fileName;

  /**
   * the title of the simple picture
   */
  private String title;

  /**
   * buffered image to hold pixels for the simple
picture
   */
  private BufferedImage bufferedImage;

  /**
   * frame used to display the simple picture
   */
  private PictureFrame pictureFrame;

  /**
   * extension for this file (jpg or bmp)
   */
  private String extension;


 /////////////////////// Constructors
////////////////////

 /**
  * A Constructor that takes no arguments.  All
fields
  * will be null. A no-argument constructor must
be given
  * in order for a class to be able to be
subclassed.  By
  * default all subclasses will implicitly call
this in
  * their parent's no argument constructor unless
a
  * different call to super() is explicitly made
as the
  * first line of code in a constructor.
  */
 public SimplePicture()
 {this(200,100);}

 /**
  * A Constructor that takes a file name and uses
the
  * file to create a picture
  * @param fileName the file name to use in
creating the
  * picture
  */
 public SimplePicture(String fileName)
 {

   // load the picture into the buffered image
   load(fileName);

 }
 /**
  * A constructor that takes the width and height
desired
  * for a picture and creates a buffered image of
that
  * size.  This constructor doesn't  show the
picture.
  * @param width the desired width
  * @param height the desired height
  */
 public  SimplePicture(int width, int height)
 {
   bufferedImage = new BufferedImage(
               width, height,
BufferedImage.TYPE_INT_RGB);
   title = "None";
   fileName = "None";
   extension = "jpg";
   setAllPixelsToAColor(Color.white);
 }

 /**
  * A constructor that takes the width and height
desired
  * for a picture and creates a buffered image of
that
  * size.  It also takes the color to use for the
  * background of the picture.
  * @param width the desired width
  * @param height the desired height
  * @param theColor the background color for the
picture
  */
 public  SimplePicture(
                    int width, int height, Color
theColor)
 {
   this(width,height);
   setAllPixelsToAColor(theColor);
 }

 /**
  * A Constructor that takes a picture to copy
  * information from
  * @param copyPicture the picture to copy from
  */
 public SimplePicture(SimplePicture copyPicture)
 {
   if (copyPicture.fileName != null)
   {
      this.fileName = new
String(copyPicture.fileName);
      this.extension = copyPicture.extension;
   }
   if (copyPicture.title != null)
```

```
      this.title = new String(copyPicture.title);
   if (copyPicture.bufferedImage != null)
   {
     this.bufferedImage =
                 new
BufferedImage(copyPicture.getWidth(),

copyPicture.getHeight(),

BufferedImage.TYPE_INT_RGB);
     this.copyPicture(copyPicture);
   }
 }

 /**
  * A constructor that takes a buffered image
  * @param image the buffered image
  */
 public SimplePicture(BufferedImage image)
 {
   this.bufferedImage = image;
   title = "None";
   fileName = "None";
   extension = "jpg";
 }

 ////////////////////////// Methods
//////////////////////

 /**
  * Method to get the extension for this picture
  * @return the extendsion (jpg or bmp)
  */
 public String getExtension() { return extension;
}


 /**
  * Method that will copy all of the passed source
  * picture into the current picture object
  * @param sourcePicture  the picture object to
copy
  */
 public void copyPicture(SimplePicture
sourcePicture)
 {
   Pixel sourcePixel = null;
   Pixel targetPixel = null;

   // loop through the columns
   for (int sourceX = 0, targetX = 0;
        sourceX < sourcePicture.getWidth() &&
        targetX < this.getWidth();
        sourceX++, targetX++)
   {
     // loop through the rows
     for (int sourceY = 0, targetY = 0;
          sourceY < sourcePicture.getHeight() &&
          targetY < this.getHeight();
          sourceY++, targetY++)
     {
       sourcePixel =

sourcePicture.getPixel(sourceX,sourceY);
       targetPixel =
this.getPixel(targetX,targetY);

targetPixel.setColor(sourcePixel.getColor());
     }
   }

 }

 /**
  * Method to set the color in the picture to the
passed
  * color
  * @param color the color to set to
  */
 public void setAllPixelsToAColor(Color color)
 {
   // loop through all x
   for (int x = 0; x < this.getWidth(); x++)
   {
     // loop through all y
     for (int y = 0; y < this.getHeight(); y++)
     {
       getPixel(x,y).setColor(color);
     }
   }
 }

 /**
  * Method to get the buffered image
  * @return the buffered image
  */
 public BufferedImage getBufferedImage()
 {
    return bufferedImage;
 }

 /**
  * Method to get a graphics object for this
picture to
  * use to draw on
  * @return a graphics object to use for drawing
  */
 public Graphics getGraphics()
 {
   return bufferedImage.getGraphics();
 }
 /**
  * Method to get a Graphics2D object for this
picture
  * which can be used to do 2D drawing on the
picture
  */
 public Graphics2D createGraphics()
 {
   return bufferedImage.createGraphics();
 }

 /**
  * Method to get the file name associated with
the
  * picture
  * @return  the file name associated with the
picture
  */
 public String getFileName() { return fileName; }

 /**
  * Method to set the file name
  * @param name the full pathname of the file
  */
 public void setFileName(String name)
 {
   fileName = name;
 }

 /**
  * Method to get the title of the picture
  * @return the title of the picture
  */
 public String getTitle()
 { return title; }

 /**
  * Method to set the title for the picture
  * @param title the title to use for the picture
  */
 public void setTitle(String title)
 {
   this.title = title;
   if (pictureFrame != null)
       pictureFrame.setTitle(title);
 }

 /**
  * Method to get the width of the picture in
pixels
  * @return the width of the picture in pixels
  */
 public int getWidth(){ return
bufferedImage.getWidth(); }

 /**
```

```
  * Method to get the height of the picture in
pixels
  * @return  the height of the picture in pixels
  */
 public int getHeight(){
  return bufferedImage.getHeight();
 }

 /**
  * Method to get the picture frame for the
picture
  * @return the picture frame associated with this
  * picture (it may be null)
  */
 public PictureFrame getPictureFrame()
                                  { return
pictureFrame; }

 /**
  * Method to set the picture frame for this
picture
  * @param pictureFrame the picture frame to use
  */
 public void setPictureFrame(PictureFrame
pictureFrame)
 {
   // set this picture objects' picture frame to
the
   // passed one
   this.pictureFrame = pictureFrame;
 }

 /**
  * Method to get an image from the picture
  * @return  the buffered image since it is an
image
  */
 public Image getImage()
 {
   return bufferedImage;
 }

 /**
  * Method to return the pixel value as an int for
the
  * given x and y location
  * @param x the x coordinate of the pixel
  * @param y the y coordinate of the pixel
  * @return the pixel value as an integer (alpha,
red,
  * green, blue)
  */
 public int getBasicPixel(int x, int y)
 {
    return bufferedImage.getRGB(x,y);
 }
 /**
  * Method to set the value of a pixel in the
picture
  * from an int
  * @param x the x coordinate of the pixel
  * @param y the y coordinate of the pixel
  * @param rgb the new rgb value of the pixel
(alpha, red,
  * green, blue)
  */
 public void setBasicPixel(int x, int y, int rgb)
 {
   bufferedImage.setRGB(x,y,rgb);
 }

 /**
  * Method to get a pixel object for the given x
and y
  * location
  * @param x  the x location of the pixel in the
picture
  * @param y  the y location of the pixel in the
picture
  * @return a Pixel object for this location
  */
 public Pixel getPixel(int x, int y)
 {
   // create the pixel object for this picture and
the
   // given x and y location
   Pixel pixel = new Pixel(this,x,y);
   return pixel;
 }

 /**
  * Method to get a one-dimensional array of
Pixels for
  * this simple picture
  * @return a one-dimensional array of Pixel
objects
  * starting with y=0
  * to y=height-1 and x=0 to x=width-1.
  */
 public Pixel[] getPixels()
 {
   int width = getWidth();
   int height = getHeight();
   Pixel[] pixelArray = new Pixel[width * height];

   // loop through height rows from top to bottom
   for (int row = 0; row < height; row++)
     for (int col = 0; col < width; col++)
       pixelArray[row * width + col] =
                                    new
Pixel(this,col,row);
```

```
   return pixelArray;
 }



 /**
  * Method to load the buffered image with the
passed
  * image
  * @param image  the image to use
  */
 public void load(Image image)
 {
   // get a graphics context to use to draw on the
   // buffered image
   Graphics2D graphics2d =
bufferedImage.createGraphics();

   // draw the image on the buffered image
starting
   // at 0,0
   graphics2d.drawImage(image,0,0,null);

   // show the new image
   show();
 }

 /**
  * Method to show the picture in a picture frame
  */
 public void show()
 {
    // if there is a current picture frame then
use it
   if (pictureFrame != null)
     pictureFrame.updateImageAndShowIt();

   // else create a new picture frame with this
picture
   else
     pictureFrame = new PictureFrame(this);
 }

 /**
  * Method to hide the picture
  */
 public void hide()
 {
   if (pictureFrame != null)
     pictureFrame.setVisible(false);
 }

 /**
  * Method to make this picture visible or not
  * @param flag true if you want it visible else
```

```
false
  */
 public void setVisible(boolean flag)
 {
   if (flag)
     this.show();
   else
     this.hide();
 }

 /**
  * Method to open a picture explorer on a copy of
this
  * simple picture
  */
 public void explore()
 {
   // create a copy of the current picture and
explore it
   new PictureExplorer(new SimplePicture(this));
 }

 /**
  * Method to force the picture to redraw itself.
This is
  * very useful after you have changed the pixels
in a
  * picture.
  */
 public void repaint()
 {
   // if there is a picture frame tell it to
repaint
   if (pictureFrame != null)
     pictureFrame.repaint();

   // else create a new picture frame
   else
     pictureFrame = new PictureFrame(this);
 }

 /**
  * Method to load the picture from the passed
file name
  * @param fileName the file name to use to load
the
  * picture from
  */
 public void loadOrFail(
                       String fileName) throws
IOException
 {
    // set the current picture's file name
   this.fileName = fileName;

   // set the extension
   int posDot = fileName.indexOf('.');
   if (posDot >= 0)
     this.extension = fileName.substring(posDot +
1);

   // if the current title is null use the file
name
   if (title == null)
     title = fileName;

   File file = new File(this.fileName);

   if (!file.canRead())
   {
     // try adding the media path
     file = new File(

FileChooser.getMediaPath(this.fileName));
     if (!file.canRead())
     {
       throw new IOException(this.fileName + "
could not"
       + " be opened. Check that you specified the
path");
     }
   }

   bufferedImage = ImageIO.read(file);
 }


 /**
  * Method to write the contents of the picture to
a file
  * with the passed name without throwing errors
  * (THIS MAY NOT BE A VALID DESCRIPTION - RGB)
  * @param fileName the name of the file to write
the
  * picture to
  * @return true if success else false
  */
 public boolean load(String fileName)
 {
     try {
         this.loadOrFail(fileName);
         return true;

     } catch (Exception ex) {
         System.out.println("There was an error
trying"
                                + " to open " +
fileName);
         bufferedImage = new
BufferedImage(600,200,

BufferedImage.TYPE_INT_RGB);
```

```
         addMessage("Couldn't load " +
fileName,5,100);
         return false;
     }

 }


 /**
  * Method to load the picture from the passed
file name
  * this just calls load(fileName) and is for name
  * compatibility
  * @param fileName the file name to use to load
the
  * picture from
  * @return true if success else false
  */
 public boolean loadImage(String fileName)
 {
     return load(fileName);
}

 /**
  * Method to draw a message as a string on the
buffered
  * image
  * @param message the message to draw on the
buffered
  * image
  * @param xPos  the leftmost point of the string
in x
  * @param yPos  the bottom of the string in y
  */
 public void addMessage(
                       String message, int xPos,
int yPos)
 {
   // get a graphics context to use to draw on the
   // buffered image
   Graphics2D graphics2d =
bufferedImage.createGraphics();

   // set the color to white
   graphics2d.setPaint(Color.white);

   // set the font to Helvetica bold style and
size 16
   graphics2d.setFont(new
Font("Helvetica",Font.BOLD,16));

   // draw the message
   graphics2d.drawString(message,xPos,yPos);

 }
 /**
  * Method to draw a string at the given location
on the
  * picture
  * @param text the text to draw
  * @param xPos the left x for the text
  * @param yPos the top y for the text
  */
 public void drawString(String text, int xPos, int
yPos)
 {
   addMessage(text,xPos,yPos);
 }

 /**
   * Method to create a new picture by scaling the
   * current picture by the given x and y factors
   * @param xFactor the amount to scale in x
   * @param yFactor the amount to scale in y
   * @return the resulting picture
   */
  public Picture scale(double xFactor, double
yFactor)
  {
    // set up the scale tranform
    AffineTransform scaleTransform =
                                   new
AffineTransform();
    scaleTransform.scale(xFactor,yFactor);

    // create a new picture object that is the
right size
    Picture result = new Picture(
                           (int) (getWidth() *
xFactor),
                           (int) (getHeight() *
yFactor));

    // get the graphics 2d object to draw on the
result
    Graphics graphics = result.getGraphics();
    Graphics2D g2 = (Graphics2D) graphics;

    // draw the current image onto the result
image
    // scaled

g2.drawImage(this.getImage(),scaleTransform,null);

    return result;
  }

  /**
   * Method to create a new picture of the passed
width.
   * The aspect ratio of the width and height will
```

```
stay
   * the same.
   * @param width the desired width
   * @return the resulting picture
   */
  public Picture getPictureWithWidth(int width)
  {
    // set up the scale tranform
    double xFactor = (double) width /
this.getWidth();
    Picture result = scale(xFactor,xFactor);
    return result;
  }

  /**
   * Method to create a new picture of the passed
height.
   * The aspect ratio of the width and height will
stay
   * the same.
   * @param height the desired height
   * @return the resulting picture
   */
  public Picture getPictureWithHeight(int height)
  {
    // set up the scale tranform
    double yFactor = (double) height /
this.getHeight();
    Picture result = scale(yFactor,yFactor);
    return result;
  }

 /**
  * Method to load a picture from a file name and
show it
  * in a picture frame
  * @param fileName the file name to load the
picture
  * from
  * @return true if success else false
  */
 public boolean loadPictureAndShowIt(String
fileName)
 {
   boolean result = true;// the default is that it
worked

   // try to load the picture into the buffered
image from
   // the file name
   result = load(fileName);

   // show the picture in a picture frame
   show();

   return result;
 }

 /**
  * Method to write the contents of the picture to
a file
  * with the passed name
  * @param fileName the name of the file to write
the
  * picture to
  */
 public void writeOrFail(String fileName)
                                        throws
IOException
 {
   //the default is current
   String extension = this.extension;

   // create the file object
   File file = new File(fileName);
   File fileLoc = file.getParentFile();

   // canWrite is true only when the file exists
   // already! (alexr)
   if (!fileLoc.canWrite()) {
       // System.err.println(
       // "can't write the file but trying anyway?
...");
        throw new IOException(fileName +
        " could not be opened. Check to see if you
can"
        + " write to the directory.");
   }

   // get the extension
   int posDot = fileName.indexOf('.');
   if (posDot >= 0)
       extension = fileName.substring(posDot + 1);

   //write the contents of the buffered image to
the file
   // as jpeg
   ImageIO.write(bufferedImage, extension, file);

 }

 /**
  * Method to write the contents of the picture to
a file
  * with the passed name without throwing errors
  * @param fileName the name of the file to write
the
  * picture to
  * @return true if success else false
  */
 public boolean write(String fileName)
 {
     try {
         this.writeOrFail(fileName);
         return true;
     } catch (Exception ex) {
         System.out.println(
                     "There was an error trying to
write "
                     + fileName);
         return false;
     }

 }

 /**
  * Method to set the media path by setting the
directory
  * to use
  * @param directory the directory to use for the
media
  * path
  */
 public static void setMediaPath(String directory)
{
   FileChooser.setMediaPath(directory);
 }

 /**
  * Method to get the directory for the media
  * @param fileName the base file name to use
  * @return the full path name by appending
  * the file name to the media directory
  */
 public static String getMediaPath(String
fileName) {
   return FileChooser.getMediaPath(fileName);
 }

  /**
   * Method to get the coordinates of the
enclosing
   * rectangle after this transformation is
applied to
   * the current picture
   * @return the enclosing rectangle
   */
  public Rectangle2D getTransformEnclosingRect(

AffineTransform trans)
  {
    int width = getWidth();
    int height = getHeight();
    double maxX = width - 1;
    double maxY = height - 1;
    double minX, minY;
    Point2D.Double p1 = new Point2D.Double(0,0);
    Point2D.Double p2 = new
```

```
Point2D.Double(maxX,0);
    Point2D.Double p3 = new
Point2D.Double(maxX,maxY);
    Point2D.Double p4 = new
Point2D.Double(0,maxY);
    Point2D.Double result = new
Point2D.Double(0,0);
    Rectangle2D.Double rect = null;

    // get the new points and min x and y and max
x and y
    trans.deltaTransform(p1,result);
    minX = result.getX();
    maxX = result.getX();
    minY = result.getY();
    maxY = result.getY();
    trans.deltaTransform(p2,result);
    minX = Math.min(minX,result.getX());
    maxX = Math.max(maxX,result.getX());
    minY = Math.min(minY,result.getY());
    maxY = Math.max(maxY,result.getY());
    trans.deltaTransform(p3,result);
    minX = Math.min(minX,result.getX());
    maxX = Math.max(maxX,result.getX());
    minY = Math.min(minY,result.getY());
    maxY = Math.max(maxY,result.getY());
    trans.deltaTransform(p4,result);
    minX = Math.min(minX,result.getX());
    maxX = Math.max(maxX,result.getX());
    minY = Math.min(minY,result.getY());
    maxY = Math.max(maxY,result.getY());

    // create the bounding rectangle to return
    rect = new Rectangle2D.Double(
              minX,minY,maxX - minX + 1, maxY -
minY + 1);
    return rect;
  }

 /**
  * Method to return a string with information
about this
  * picture
  * @return a string with information about the
picture
  */
 public String toString()
 {
   String output =
     "Simple Picture, filename " + fileName +
     " height " + getHeight() + " width " +
getWidth();
   return output;
 }

} // end of SimplePicture class
```

Listing 14. Source code for Ericson's DigitalPicture interface.

```
import java.awt.Image;
import java.awt.image.BufferedImage;

/**
 * Interface to describe a digital picture.  A
digital
 * picture can have a associated file name.
It can have
 * a title.  It has pixels associated with it
and you can
 * get and set the pixels.  You can get an
Image from a
 * picture or a BufferedImage.  You can load
it from a
 * file name or image.  You can show a
picture.  You can
 * create a new image for it.
 *
 * Copyright Georgia Institute of Technology
2004
 * @author Barb Ericson ericson@cc.gatech.edu
 */
public interface DigitalPicture
{
 // get the file name that the picture came
from
 public String getFileName();

 // get the title of the picture
 public String getTitle();

 // set the title of the picture
 public void setTitle(String title);

 // get the width of the picture in pixels
 public int getWidth();

 // get the height of the picture in pixels
 public int getHeight();

 // get the image from the picture
 public Image getImage();

 // get the buffered image
 public BufferedImage getBufferedImage();

 // get the pixel information as an int
 public int getBasicPixel(int x, int y);

 // set the pixel information
 public void setBasicPixel(int x, int y, int
rgb);

 // get the pixel information as an object
 public Pixel getPixel(int x, int y);

 // load the image into the picture
 public void load(Image image);

 // load the picture from a file
 public boolean load(String fileName);

 // show the picture
 public void show();
}
```

Listing 15. Source code for Ericson's PictureFrame class.

```
import javax.swing.*;
import java.awt.*;

/**
 * Class that holds a digital picture and
displays it.
 *
 * Copyright Georgia Institute of Technology
2004
 * @author Barb Ericson
 */
public class PictureFrame
{

  ////////////////// fields
////////////////////////////

  /**
   * Main window used as the frame
   */
  JFrame frame = new JFrame();

  /**
   * ImageIcon used to display the picture in
the label
   */
  ImageIcon imageIcon = new ImageIcon();

  /**
   * Label used to display the picture
   */
  private JLabel label = new
JLabel(imageIcon);

  /**
```

```
   * Digital Picture to display
   */
  private DigitalPicture picture;

  ///////////////// constructors
////////////////////////

  /**
   * A constructor that takes no arguments.
This is
   * needed for subclasses of this class
   */
  public PictureFrame()
  {
    // set up the frame
    initFrame();
  }

  /**
   * A constructor that takes a picture to
display
   * @param picture  the digital picture to
display in the
   * picture frame
   */
  public PictureFrame(DigitalPicture picture)
  {
    // set the current object's picture to the
passed in
    // picture
    this.picture = picture;

    // set up the frame
    initFrame();
  }

  ///////////////////////// methods
//////////////////////

  /**
   * Method to set the picture to show in this
picture
   * frame
   * @param picture the new picture to use
   */
  public void setPicture(Picture picture)
  {
    this.picture = picture;
    imageIcon.setImage(picture.getImage());
    frame.pack();
    frame.repaint();
  }

  /**
   * A method to update the picture frame
image with the
```

```
   * image in the picture
   */
  public void updateImage()
  {
    // only do this if there is a picture
    if (picture != null)
    {
      // set the image for the image icon from
the picture
      imageIcon.setImage(picture.getImage());

      // set the title of the frame to the
title of the
      // picture
      frame.setTitle(picture.getTitle());

    }
  }

  /**
   * A method to update the picture frame
image with the
   * image in the picture and show it
   */
  public void updateImageAndShowIt()
  {
    // first update the image
    updateImage();

    // now make sure it is shown
    frame.setVisible(true);
  }

  /**
   * A method to make sure the frame is
displayed
   */
  public void displayImage()
  {
    frame.setVisible(true);
  }

  /**
   * A method to hide the frame
   */
  public void hide()
  {
    frame.setVisible(false);
  }

  /**
   * A method to set the visible flag on the
frame
   * @param flag the flag to use
   */
  public void setVisible(boolean flag)
  {
    frame.setVisible(flag);
  }

  /**
   * A method to close a picture frame
   */
  public void close()
  {
    frame.setVisible(false);
    frame.dispose();
  }

  /**
   * Method to set the title for the picture
frame
   * @param title the title to use
   */
  public void setTitle(String title)
  {
    frame.setTitle(title);
  }

  /**
   * Method to force the picture frame to
repaint (redraw)
   */
  public void repaint()
  {

    // make it visible
    frame.setVisible(true);

    // update the image from the picture
    updateImage();

    // tell the JFrame to handle the repaint
    frame.repaint();
  }

  /**
   * A method to initialize the picture frame
   */
  private void initFrame()
  {

    // set the image for the picture frame
    updateImage();

    // add the label to the frame
    frame.getContentPane().add(label);

    // pack the frame (set the size to as big
as it needs
    // to be)
    frame.pack();
    // make the frame visible
    frame.setVisible(true);
  }

}
```

Listing 16. Source code for the program named Java356a.

```
/*Program Java356a
Copyright R.G.Baldwin 2009

The purpose of this program is to illustrate one way to
take a photograph of a physical object and then
superimpose it on another photograph.

A desk chair was placed in front of a bookcase. A blue
sheet was hung on the bookcase to provide a relatively
solid color background. A green towel was placed on the
chair to hide the texture in the upholstry. A digital
photograph of the chair was taken. Then a stuffed tiger
was placed on the back of the chair and another digital
photograph was taken.

Picture objects were instantiated from each of the
photographs. Another Picture object was instantiated from
an image file showing a beach scene with the same
dimensions. A fourth Picture object was instantiated with
the same dimensions and an all-white image.

Methods of the SimplePicture class and the Pixel class
were used in a pair of nested for loops to compare the
color distance between corresponding pixels in the two
photographs to within a specified tolerance. When the
color distance between the two pixels exceeded a specified
threshold, the color of the pixel from the photograph
containing the tiger was copied to the all-white Picture
object, replacing a white pixel. Otherwise, the color of
the pixel from the beach image was copied to the all-white
Picture object.

The results were moderately good. However, lighting is
critical and I didn't do anything special to control the
lighting. As a result, a shadow of the tiger that was
barely noticeable on the blue sheet is very noticeable in
the final product.

Note:  The idea for this program came directly from the
book titled Introduction to Computing and Programming with
Java: A Multimedia Approach by Guzdial and Ericson.

Tested using Windows Vista Premium Home edition and
```

```
Ericson's multimedia library.
**********************************************************/
import java.awt.Color;
public class Main{
  public static void main(String[] args){
    new Runner().run();
  }//end main method
}//end class Main
//------------------------------------------------------//

class Runner{
  void run(){
    //Construct three new 341x256 Picture objects by
    // providing the names of image files as parameters
    // to the Pictue constructor.
    Picture pic1 = new Picture("ScaledBeach.jpg");
    Picture pic2 = new Picture("WithTiger.jpg");
    Picture pic3 = new Picture("WithoutTiger.jpg");

    //Construct an all-white 341x256 Picture object.
    Picture pic4 = new Picture(341,256);

    //Display all three Picture objects in the show
    // format.
    pic1.show();
    pic2.show();
    pic3.show();

    //Replace pixel colors in the all-white Picture object
    // with the colors from either the beach image or the
    // tiger image.
    Pixel pixA;
    Pixel pixB;
    Pixel pixC;
    Pixel pixD;
    for(int row = 0;row < pic1.getHeight() - 1;row++){
      for(int col = 0;col < pic1.getWidth() - 1;col++){
        pixA = pic1.getPixel(col,row);
        pixB = pic2.getPixel(col,row);
        pixC = pic3.getPixel(col,row);
        pixD = pic4.getPixel(col,row);

        if(pixB.colorDistance(pixC.getColor()) > 50){
          //Replace white pixel with the pixel color from
          // the tiger image.
          pixD.setColor(pixB.getColor());
        }else{
          //Replace the white pixel with pixel color from
          // the beach image.
          pixD.setColor(pixA.getColor());
        }//end else
      }//end inner for loop
    }//end outer for loop

    //Display the final product using the show format.
    pic4.setTitle("Tiger on beach scene");
    pic4.show();
  }//end run
}//end class Runner
```

Copyright

About the author

Richard Baldwin *is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is object-oriented programming using Java and other OOP languages.*

Richard has participated in numerous consulting projects and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Programming Tutorials, which have gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.

In addition to his programming expertise, Richard has many years of practical experience in Digital Signal Processing (DSP). His first job after he earned his Bachelor's degree was doing DSP in the Seismic Research Department of Texas Instruments. (TI is still a world leader in DSP.) In the following years, he applied his programming and DSP expertise to other interesting areas including sonar and underwater acoustics.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

Baldwin@DickBaldwin.com

-end-