The DigitalPicture Interface

Learn how to write programs using the methods defined in the Picture and SimplePicture classes that are declared in the DigitalPicture interface.

Published: March 18, 2009
By Richard G. Baldwin

Java Programming Notes # 354

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.
- 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.

The Picture class and the DigitalPicture Interface

If you have studied the earlier lessons in this series *(see Resources)*, you have learned all 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 also need to know about the environment in which a turtle lives. You learned all about the **World** class in the previous lesson *(see Resources)*. In this lesson, you will begin learning about the **Picture** class and its superclass named **SimplePicture**.

The Picture class is relatively simple

In reality, there isn't much to the **Picture** class. It is simply a skeleton class that overrides the **toString** method and provides five different constructors that serve as proxies for the constructors in the superclass named **SimplePicture**. Each **Picture** constructor simply calls a **SimplePicture** constructor, passing the constructor parameters to the **SimplePicture** constructor.

The real functionality lies in SimplePicture

All of the real functionality of a **Picture** object lies in the superclass named **SimplePicture**. Therefore, the class named **SimplePicture** will be the target of this and the next several lessons. However, I have provided a source listing for the **Picture** class in Listing 30 near the end of the lesson for your examination. *(The only changes made to the listing were minor format changes necessary to force the source code to fit into this narrow publication format.)*

A large and complex class

The **SimplePicture** class is a large and complex class containing almost forty different methods. That is obviously too much material for a single lesson, so I will break the class down and explain it in parts.

A complete listing of Ericson's **SimplePicture** class is provided in Listing 31 near the end of the lesson.

The DigitalPicture interface

The **SimplePicture** class implements the **DigitalPicture** interface, which declares the following thirteen methods:

- **String getFileName()**; // get the file name that the picture came from
- **String getTitle()**; // get the title of the picture
- **void setTitle(String title)**; // set the title of the picture
- **int getWidth()**; // get the width of the picture in pixels
- **int getHeight()**; // get the height of the picture in pixels
- **Image getImage()**; // get the image from the picture
- **BufferedImage getBufferedImage()**; // get the buffered image from the picture
- **int getBasicPixel(int x, int y)**; // get the pixel information as an **int** value
- **void setBasicPixel(int x, int y, int rgb)**; // set the pixel information as an **int** value
- **Pixel getPixel(int x, int y)**; // get the pixel information as an object of type **Pixel**
- **void load(Image image)**; // load the image into the picture
- **boolean load(String fileName)**; // load the picture from a file
- **void show()**; // show the picture

My first attempt to compartmentalize...

As my first attempt to compartmentalize and explain the **SimplePicture** class, this lesson will illustrate and explain the thirteen methods in the above list in terms of how they are implemented in the **Picture** and **SimplePicture** classes. In addition, the thirteen methods in the above list often call other methods belonging to the **SimplePicture** class, so I will explain those methods in this lesson as well.

There are also numerous methods defined in the **SimplePicture** class that are not declared in the **DigitalPicture** interface. I will explain those methods in future lessons.

A complete listing of Ericson's **DigitalPicture** interface is provided in Listing 32 near the end of the lesson.

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 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

I will explain the methods in the above list and some additional methods as well, in the context of three sample programs. As usual, I will explain the code in fragments. Because I will be switching back and forth between code fragments extracted from Ericson's **SimplePicture** class and code fragments extracted from my sample programs, things can get confusing.

Reducing the confusion

In an attempt 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 my sample programs against the background color shown in Listing 2.

Listing 2. Background color for Baldwin's code.

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

On the rare occasion that I need to display a code fragment from the **Picture** class, I will present the code fragments against the gray background that you see in Listing 20.

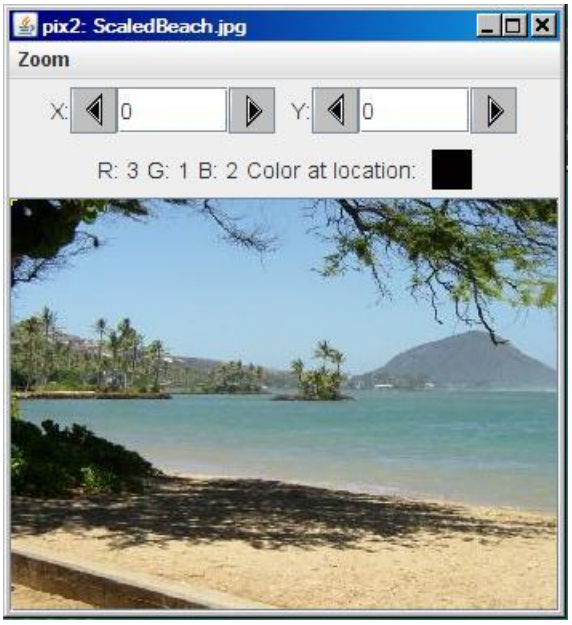
Image file locations
For simplicity, both image files were placed in the same location as the class files for the program.

A preview of the images that will be used

As you might expect from the names of the classes and interfaces that I will be explaining *(Picture, SimplePicture, and DigitalPicture)*, the sample programs in this lesson will deal in one way or another with pictures and images. Two different image files named **ScaledAquarium.gif** and **ScaledBeach.jpg** will be used in these programs. The images contained in the two files are shown in Figure 1 and Figure 2.

Figure 1. Image from the file named ScaledAquarium.gif.

Figure 2. Image from the file named ScaledBeach.jpg.



Different display formats

In some cases, the sample program output will be displayed by calling Ericson's **show** method, which produces the display format shown in Figure 1. In other cases, the sample program output will be displayed by calling Ericson's **explore** method, which produces the display format shown in Figure 2. *(The **show** method and the **explore** method are both methods of the **SimplePicture** class. They will be explained in a future lesson.)*

The images were scaled in advance

For reasons that will become apparent later, both images were scaled in advance to have a height of 256 rows of pixels. Through pure coincidence, that also caused each image to have a width of 341 pixels. *(Although I didn't plan it that way, the two raw images were the same size.)*

Discussion and sample code

The program named Java354a

A complete listing of this program is provided in Listing 33 near the end of the lesson.

The purpose of this program is to illustrate and explain most of the methods that are declared in the **DigitalPicture** interface and implemented in the **SimplePicture** class, along with the methods called by those methods.

Instantiate a Picture object

One **Picture** object is instantiated in this program by using a **Picture** constructor that accepts the name of an image file as a parameter and uses the image from that file as the image in the **Picture** object. That picture is displayed by calling the **show** method on the **Picture** object, producing the screen output shown in Figure 1.

Title matches the file name

Note that in this case, the title is automatically set to the name of the image file.

Instantiate another Picture object

A second Picture object is constructed by using a **Picture** constructor that accepts the dimensions of the **Picture** object only and constructs a **Picture** object with a default all-white image. The size of the **Picture** object that is constructed is 1x1 pixels.

Then the **load** method that takes the name of an image file is called to load the image from an image file into the small **Picture** object. The size of the **Picture** object changes automatically to accommodate the size of the image.

In this case, the default title is "None". In other words, unlike the previous case, the title is not automatically set to the name of the image file.

The **setTitle** and **getFileName** methods are called to set the title for this picture. Then the **explore** method is called to display the **Picture** object with its new image and title, producing the screen output shown in Figure 2.

A caution regarding the repetitive calling of the show method

If you call the **show** method on a picture, then modify the picture, and then call the show method on the picture again, only one copy of the picture will be displayed. Furthermore, the results may not be what you expect to see. However, displaying the picture in the **explore** format, modifying it, and then displaying it again in the **show** format seems to work OK.

Processing the Picture objects

A pair of nested **for** loops is used in conjunction with the **getBasicPixel** and **setBasicPixel** methods to copy the right half of the image in Figure 1 into the left half of the image in Figure 2, leaving the right half of the image in Figure 2 undisturbed. Then the **show** method is called on the modified second **Picture** object to display it, producing the screen output shown in Figure 3.

Figure 3. Screen output from the program named Java354a.

Text output

At various points along the way, the program calls methods dealing with the file name and the title and eventually prints that information on the system console as shown in Figure 4.

Figure 4. Text output from the program named Java354a.

```
Picture, filename ScaledAquarium.gif height
256 width 341
pix1 Filename: ScaledAquarium.gif
Picture, filename ScaledBeach.jpg height 256
width 341
pix2 FileName: ScaledBeach.jpg
pix1 Title: ScaledAquarium.gif
pix2 Title: pix2: ScaledBeach.jpg
```

Methods illustrated by the program named Java354a

The following methods from the **DigitalPicture** interface are illustrated by this program.

- String getFileName()
- String getTitle()
- void setTitle(String title)
- int getWidth()
- int getHeight()
- int getBasicPixel(int x, int y)
- void setBasicPixel(int x, int y, int rgb)
- boolean load(String fileName)
- void show()

The following methods that are declared in the **DigitalPicture** interface are not illustrated by this program.

- Image getImage()
- BufferedImage getBufferedImage()
- Pixel getPixel(int x, int y)
- void load(Image image)

These methods will be illustrated by the programs named Java354b and Java354c later in this lesson.

Beginning of the program named Java354a

All three of the programs in this lesson begin with the code shown in Listing 3, so I will show this code fragment only once.

Listing 3. Beginning of the program named Java354a.

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

The **main** method for each program is defined in a class named **Main**. The **main** method instantiates an object of the **Runner** class and calls a method named **run** on that object. When the **run** method returns, the **main** method terminates and the program terminates.

Beginning of the class named Runner

Listing 4 shows the beginning of the class named **Runner** and the beginning of the method named **run** for the program named Java354a.

Listing 4. Beginning of the class named Runner.

```
class Runner{
  void run(){
    Picture pix1 = new
Picture("ScaledAquarium.gif");
    pix1.show();//display the picture in the
show format
```

Listing 4 calls one of the overloaded constructors of the **Picture** class to construct a new 341x256 **Picture** object passing the name of an image file as a parameter. As mentioned earlier, for simplicity, the image file was placed in the same directory as the class files for the program.

Picture constructors

The particular constructor called in Listing 4 requires the name of an image file as an incoming **String** parameter. As I mentioned earlier, the constructors for the **Picture** class simply call the corresponding constructors for the superclass named **SimplePicture** passing the incoming parameter to the superclass constructor. You can view the code for the **Picture** constructors in Listing 30 near the end of the lesson.

The SimplePicture constructor

Listing 5 shows the code for the corresponding constructor in the superclass named **SimplePicture**. *(Remember, the background color shown in Listing 5 indicates that the code fragment was extracted from the class named **SimplePicture**.)*

Listing 5. An overloaded constructor from the SimplePicture class.

```
 /**
  * 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);
 }
```

As you can see, the code in Listing 5 simply calls one of the overloaded **load** methods of the **SimplePicture** class to extract the image from the specified image file and load it into a **BufferedImage** object. The **BufferedImage** object is referred to by a private instance variable belonging to the **Picture** object being constructed.

Instance variables of the SimplePicture class

No instance variables are defined in the **Picture** class. The instance variables that are defined in the **SimplePicture** class are listed below:

- private String **fileName**;
- private String **title**;
- private BufferedImage **bufferedImage**;
- private PictureFrame **pictureFrame**;
- private String **extension**;

The overloaded load method

The overloaded load method called in Listing 5 is shown in Listing 6. *(Note that the explanatory comments for this method appear to be incorrect, so I did not include them in Listing 6. You can view those comments in Listing 31.)*

Listing 6. The overloaded load method that requires the name of an image file.

```
 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;
     }
 }
```

Listing 6 calls the **loadOrFail** method *(shown in Listing 7)* of the **SimplePicture** class in an attempt to read the file and extract the image from the file.

If the call to loadOrFail fails...

If the **loadOrFail** method is unsuccessful in finding the specified image file, it throws an **IOException**. Therefore, the call to the **loadOrFail** method in Listing 6 is placed inside a **try** block. In the case of a failure, the **catch** block in Listing 6 is executed. The code in the **catch** block:

- Prints an error message on the standard output device.
- Instantiates a default **BufferedImage** object, storing its reference in the private instance variable named **bufferedImage** that belongs to the **Picture** object being constructed.
- Calls the **addMessage** method to display an error message on the default **BufferedImage** object.
- Returns false.

Screen output for a missing image file

Figure 5 shows a reduced version of the screen output that occurs as a result of a failure by the **loadOrFail** method to find the specified file. *(It was necessary for me to reduce the size of this image to force it to fit in this narrow publication format.)*

Figure 5. Screen output for a missing image file.

If the loadOrFail method finds the file - but no image

If the **loadOrFail** method is successful in finding the specified image file, but is not successful in extracting an image from the file, a runtime error will be thrown without the frame shown in Figure 5 necessarily appearing on the screen.

If the loadOrFail method succeeds in reading an image...

If the **loadOrFail** method succeeds in reading the image from the file, it will encapsulate that image in a **BufferedImage** object and will store that object's reference in the private instance variable named **bufferedImage** belonging to the **Picture** object being constructed.

The loadOrFail method

The source code for the **loadOrFail** method is shown in Listing 7.

Listing 7. The loadOrFail method.

```
 /**
  * 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);
 }
```

File IO is a major topic

File IO is a major topic in its own right. Because this tutorial is not intended to teach you about file IO, I won't explain the code in Listing 7 in detail. If you don't understand that code, you might want to study up on file IO in Java.

Behavior of the loadOrFail method

Basically, the **loadOrFail** method either throws an **IOException** or:

- Finds the specified image file.
- Extracts the image from the image file.
- Encapsulates the image in an object of the **BufferedImage** class.
- Stores the object's reference in a private instance variable named **bufferedImage** belonging to the **Picture** object being constructed.

Another possibility is that the method finds the specified file but is unable to extract an image from it *(the file may be corrupt)* in which case an error or exception will ultimately be thrown.

Instantiate a small Picture object

Having instantiated the **Picture** object based on an existing image file *(see Figure 1)*, the program named Java354a instantiates a new small **Picture** object with a default all-white image. *(Note that the size must be at least 1x1 pixels or a runtime error will occur.)* This **Picture** object is instantiated by calling the **Picture** constructor shown in Listing 8.

Listing 8. Instantiate a small Picture object.

```
    Picture pix2 = new Picture(1,1);
```

Constructor for a Picture object with an all-white image

The source code for the **SimplePicture** constructor that is executed as a result of the code in Listing 8 is shown in Listing 9.

Listing 9. Constructor for a Picture with an all-white image.

```
 /**
  * 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);
 }
```

Listing 9 begins by instantiating a new **BufferedImage** object of a specified size and a particular type.

What is a BufferedImage object?

To make a long story short, a **BufferedImage** object encapsulates an image in a sophisticated and accessible way. *(Many methods are provided to access the data in the object.)*

How is the image data represented?

There are many different ways in which image data can be represented. The code in Listing 9 specifies one of those ways. The parameter value **BufferedImage.TYPE_INT_RGB** is a constant that causes the image to be represented with three 8-bit color components *(red, green, and blue)* packed into pixels of type **int**.

Setting default property values

After instantiating the **BufferedImage** object and storing its reference in the private instance variable named **bufferedImage**, Listing 9 sets default values for the **title**, **fileName**, and **extension** properties.

Set all pixels to white

Then Listing 9 calls the method named **setAllPixelsToAColor** to set all of the pixels in the new image to the color white. The source code for this method is shown in Listing 10.

Listing 10. The method named setAllPixelsToAColor.

```
 /**
  * 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);
     }//end inner loop
   }//end outer loop
 }//end method named setAllPixelsToAColor
```

With the exception of the call to the **getPixel** method, there should be nothing in Listing 10 that causes you any difficulty.

Gain access to each pixel and color it white

Listing 10 uses a pair of nested **for** loops and a call to the **getPixel** method to gain access to every pixel in the image. I will explain the **getPixel** method in detail later in this lesson. For now suffice it to say that the **getPixel** method encapsulates a pixel whose location in the image is specified by a pair of horizontal and vertical coordinates into an object of the class **Pixel**. Then the method returns a reference to the **Pixel** object.

Set the color of the pixel to white

The **Pixel** class provides many methods that can be used to manipulate the pixel. The code in Listing 10 calls the **setColor** method on the **Pixel** object to set the color of each pixel to white.

Load an image from a jpg file and set the title.

Having created the small *(1x1)* **Picture** object, Listing 11 calls the **load** method of the **SimplePicture** class to load the image from an image file into the **BufferedImage** object that belongs to the **Picture** object. Note that this is the same overloaded **load** method that I explained in conjunction with Listing 6 earlier in this lesson.

Listing 11. Load an image from a jpg file and set the title.

```
    pix2.load("ScaledBeach.jpg");

    //Set the title of the picture.
    pix2.setTitle("pix2: " +
pix2.getFileName());
```

*(Note that the size of the **Picture** object increases or decreases automatically to accommodate the size of the image.)*

Listing 11 also calls the **getFileName** method and the **setTitle** method to set the title of the **Picture** object to that shown in the top banner in Figure 2.

The getFileName method

As you can see in Listing 12, the **getFileName** method simply returns the current value of the **fileName** property stored in an instance variable having the same name.

Listing 12. The getFileName method.

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

The setTitle method

The **setTitle** method, which is shown in its entirety in Listing 13, is only slightly more complicated.

Listing 13. The setTitle method.

```
 /**
  * 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);
 }
```

The **setTitle** method begins by storing the incoming **String** parameter in an instance variable named **title**.

A PictureFrame object

The **SimplePicture** class has a private instance variable of type **PictureFrame** named **pictureFrame**. I don't want to dwell on this topic in this lesson, because I will explain the **PictureFrame** class in some detail in a future lesson. For now, suffice it to say that an object of the **PictureFrame** class holds a reference to an object of the **JFrame** class. It is the **JFrame** object that provides the visual manifestation of a **Picture** object as shown in Figure 1.

The code in Listing 13 checks to confirm that such a **PictureFrame** object exists, and if so it calls the **setTitle** method on the reference to the **PictureFrame** object. That call, in turn, calls the **setTitle** method on the **JFrame** object, producing the visual manifestation of a title that you see in the banner at the top of Figure 1.

Display the Picture object in the explore format

Listing 14 calls the **explore** method to display the **Picture** object in the format shown in Figure 2.

Listing 14. Display the Picture object in the explore format.

```
    pix2.explore();
```

The **SimplePicture** class provides two different methods that can be used to display a **Picture** object:

- **show** - produces the output format shown in Figure 1.

- **explore** - produces the output format shown in Figure 2.

I will explain both of these methods in detail in a future lesson. For now, just observe the differences between the format of Figure 1 and the format of Figure 2.

Copy the right half of pix1 to the left half of pix2

Listing 15 uses a pair of nested **for** loops along with calls to the **getHeight**, **getWidth**, **getBasicPixel**, and **setBasicPixel** methods to copy the right half of the image from **pix1** into the left half of **pix2**, leaving the right half of **pix2** undisturbed.

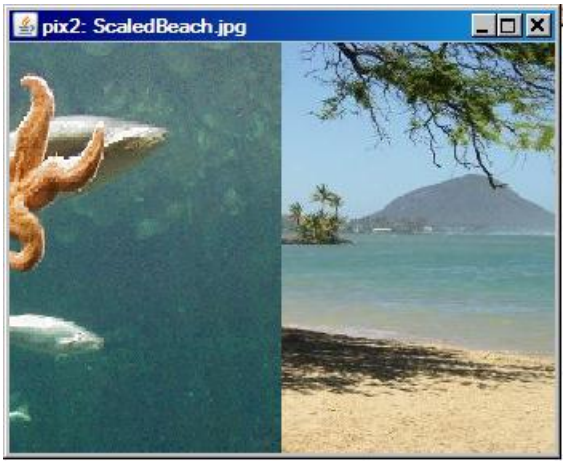
Listing 15. Copy the right half of pix1 to the left half of pix2.

```
    for(int row = 0;row <
pix1.getHeight();row++){
      for(int col = 0;col <
pix2.getWidth()/2;col++){

pix2.setBasicPixel(col,row,pix1.getBasicPixel(
                           col +
pix1.getWidth()/2,row));
      }//end inner for loop
    }//end outer for loop

    //Display the final result.
    pix2.show();
```

Final Picture output from program Java354a

Then Listing 15 calls the **show** method on **pix2** producing the screen output shown in Figure 6.

Figure 6. Final Picture output from the program named Java354a.



As you can see in Figure 6, the right half of the aquarium image has been copied into the left half of the beach image.

The getBasicPixel method

The **getBasicPixel** method is shown in its entirety in Listing 16. This method receives a pair of x,y coordinate values and returns the color contents of the pixel at that location packed in a single value of type **int**.

Listing 16. The getBasicPixel method.

```
 /**
  * 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);
 }
```

The hard work is done by getRGB

The code in Listing 16 is straightforward due to the fact that all of the hard work is handled by a call to the **getRGB** method of the **BufferedImage** class. Sun's description of the **getRGB** method is provided in Figure 7.

Figure 7. Sun's description of the getRGB method.

> Returns an integer pixel in the default RGB color model (TYPE_INT_ARGB) and default sRGB colorspace. Color conversion takes place if this default model does not match the image ColorModel. There are only 8-bits of precision for each color component in the returned data when using this method.

I will leave it as an exercise for the student to study up on the *color model* and the *color space* aspects of a **BufferedImage** object in order to fully understand the description in Figure 7.

The setBasicPixel method

The **setBasicPixel** method is shown in Listing 17.

Listing 17. The setBasicPixel method.

```
 /**
  * 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);
 }
```

Once again, the **setBasicPixel** method delegates the hard work to the **setRGB** method of the **BufferedImage** class. Sun's description of the **setRGB** method is provided in Figure 8.

Figure 8. Sun's description of the setRGB method.

> Sets a pixel in this BufferedImage to the specified RGB value. The pixel is assumed to be in the default RGB color model, TYPE_INT_ARGB, and default sRGB color space. For images with an IndexColorModel, the index with the nearest color is chosen.

The getWidth and getHeight methods of the SimplePicture class

Listing 15 also calls the **getWidth** and **getHeight** methods of the **SimplePicture** class to control the **for** loops. The **getWidth** and **getHeight** methods of the **SimplePicture** class are shown in Listing 18.

Listing 18. The getWidth and getHeight methods.

```
 /**
  * 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();
 }
```

Call corresponding methods on the BufferedImage object

The **getWidth** and **getHeight** methods call methods having the same names on the **BufferedImage** object to get and return the width and the height of the **BufferedImage** object.

Width and height of the image, not the JFrame

It is important to note that the width and height values for a **Picture** object obtained in this manner correspond to the dimensions of the image inside the **JFrame** shown in Figure 1. They do not correspond to the outer dimensions of the **JFrame** object.

The outer dimensions of the **JFrame** object shown in Figure 1 are 350x285 pixels whereas the dimensions of the image are 341x256 pixels. The extra space is consumed by the borders and the banner at the top of the **JFrame** object.

Display some text on the system console

The code in Listing 19 calls the **getFileName** method and the **getTitle** method on the **Picture** objects to print the text shown in Figure 5.

Listing 19. Display some text on the system console.

```
    System.out.println(pix1);
    System.out.println("pix1 Filename: "
                                    +
pix1.getFileName());
    System.out.println(pix2);
    System.out.println("pix2 FileName: "
                                    +
pix2.getFileName());
    System.out.println("pix1 Title: " +
pix1.getTitle());
    System.out.println("pix2 Title: " +
pix2.getTitle());

  }//end run
}//end class Runner
```

I explained the **getFileName** method in conjunction with Listing 12 earlier in this lesson.

Although I haven't shown you the code for the **getTitle** method of the **SimplePicture** class, suffice it to say that this method simply returns the value stored in the private instance variable named **title**.

Overridden toString method of the Picture class

When you pass a Java object's reference to the **println** method, *(as in the first statement in Listing 19)*, code in the **printing** method calls the **toString** method on the incoming object reference to get a **String** object for printing. As I mentioned earlier, the **Picture** class overrides the **toString** method. The overridden version of the **toString** method for the **Picture** class is shown in Listing 20.

Listing 20. Overridden toString method of the Picture class.

```
  /**
   * 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;
  }
```

Given what you have already learned, you should have no difficulty understanding how the code in Listing 20 produces the string shown as the first line of text output in Figure 4.

The end of the program named Java354a

Listing 19 signals the end of the **run** method, the end of the **Runner** class, and the end of the program named Java354a.

The program named Java354b

A complete listing of this program is provided in Listing 34 near the end of the lesson.

The purpose of this program is to illustrate and explain most of the remaining methods that are declared in the **DigitalPicture** interface as implemented in the **Picture** and **SimplePicture** classes, along with methods called by those methods.

The earlier program named Java354a illustrated the use of all but the following four methods that are declared in the **DigitalPicture** interface

- Image **getImage()**
- BufferedImage **getBufferedImage()**
- void **load(Image image)**
- Pixel **getPixel(int x, int y)**

This program creates and displays four **Picture** objects illustrating the first three methods in the above list. This leaves only the **getPixel** method to be illustrated later in this lesson to satisfy the initial contract of the lesson.

This program begins just like the previous program with code that is identical to that shown in Listing 3.

Beginning of the Runner class and the run method

The **Runner** class and the **run** method begin in Listing 21.

Listing 21. Beginning of the Runner class and the run method.

```
class Runner{
  void run(){
    //Construct a new 341x256 Picture object
by providing
    // the name of an image file as a
parameter to the
    // Picture constructor.
    Picture pix1 = new
Picture("ScaledAquarium.gif");
    pix1.setTitle("pix1");
    pix1.show();

    //Construct another new 341x256 Picture
object by
    // providing the name of an image file as
a parameter
    // to the Picture constructor.
    Picture pix2 = new
Picture("ScaledBeach.jpg");
    pix2.setTitle("pix2");
    pix2.show();
```

Listing 21 creates and displays two different **Picture** objects using code that you have seen before. The screen output produced by the code in Listing 21 is shown in Figure 9.

Figure 9. First two screen displays from the program named Java354b.

These are the same images that you saw before. Only the title in the **JFrame** object is different.

Instantiate a Picture object using a different constructor

Listing 22 constructs a third new 341x256 **Picture** object by extracting the **BufferedImage** object reference from **pix1** and passing it as a parameter to a different overloaded constructor for the **Picture** class.

Listing 22. Instantiate a Picture object using a different overloaded constructor.

```
    Picture pix3 = new
Picture(pix1.getBufferedImage());
    pix3.setTitle("pix3");
    pix3.show();
```

The call to the **getBufferedImage** method returns the reference to the **BufferedImage** object stored in the instance variable named **bufferedImage**. You can view the code for that method in Listing 31 near the end of the lesson.

Listing 22 also sets the title for the new **Picture** object and calls the **show** method to display it.

Source code for the overloaded constructor

Listing 23 shows the source code for the overloaded **Picture** constructor that is called to create the new **Picture** object in Listing 22.

Listing 23. Source code for the overloaded constructor.

```
/**
 * 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";
}
```

This constructor receives an incoming parameter that is a reference to a **BufferedImage** object. It stores that reference in its own instance variable named **bufferedImage**, thereby causing the referenced image to become the image for the new **Picture** object being constructed.

Two references to the same BufferedImage object

At this point, we have two **Picture** objects, **pix1** and **pix3** sharing a common **BufferedImage** object. They each contain a reference to the same **BufferedImage** object. This is probably not a good idea, because any changes made to the pixels in the **BufferedImage** object by way of either **Picture** object will show up in both pictures. *(See Listing 24 for a better approach.)*

Screen output produced by Listing 22

The screen output produced by Listing 22 is shown in Figure 10.

Figure 10. Screen output produced by Listing 22.

Identical except for the title

If you compare Figure 10 with the top image in Figure 9, you will see that they are identical except for the title. As described above, the image showing in both figures is a common **BufferedImage** object. Therefore, the two onscreen images are identical.

However, the **JFrame** objects that provide the onscreen visual manifestations of the two **Picture** objects are different, and each **JFrame** object has its own title. Therefore, the titles in the two figures are different.

Call the other overloaded load method

Image versus BufferedImage
Image is the superclass of BufferedImage. Therefore, a BufferedImage object is also an Image object.

The **SimplePicture** class provides two overloaded methods named **load**. One of them, which was shown and discussed in Listing 6, requires a file name as an incoming parameter. The other overloaded **load** method, which is shown in Listing 25, requires a reference to an object of type **Image** as an incoming parameter.

Construct and display one more Picture object

Listing 24 calls this version of the **load** method to construct and display a fourth new 341x256 **Picture** object. Instead of constructing a new **Picture** object by passing an image reference to an overloaded constructor *(as in Listing 22)*, this code starts with an all-white **Picture** object and then loads an image extracted from **pix2**.

Listing 24. Use the other overloaded load method.

```
    Image image = pix2.getImage();

    //Get the size of the image and pass those
dimensions
```

```
    // to the constructor for the Picture
object.
    Picture pix4 = new
Picture(image.getWidth(null),

image.getHeight(null));
    //Now load the image into the Picture
object and
    // display the picture.
    pix4.load(image);
    pix4.setTitle("pix4");
    pix4.show();

  }//end run
}//end class Runner
```

Size is not set automatically

Note that unlike the load method that takes a file name as a parameter *(shown in Listing 6)*, this version of the load method does not automatically set the size of the **Picture** object to match the size of the image.

Listing 24 begins by calling the **getImage** method on **pix2** to get a reference to the **Image** object belonging to that picture. *(The call actually gets a reference to a **BufferedImage** object and saves it as the superclass type **Image**.)*

Listing 24 calls the **getWidth** and **getHeight** methods on the **Image** object to get the dimensions of the image. These values are passed to the **Picture** constructor to create a new all-white **Picture** object having the same dimensions as the image.

Then Listing 24 calls the **load** method on the **Picture** object to load the image into the picture.

Source code for the other overloaded load method

The source code for the version of the overloaded **load** method that is called in Listing 24 is shown in Listing 25.

Listing 25. Source code for the other overloaded load method.

```
 /**
  * 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();
 }
```

Get a graphics context

Listing 25 begins by calling the **createGraphics** method on the all-white **BufferedImage** object that belongs to the **Picture** object to get a reference to the graphics context belonging to that **BufferedImage** object.

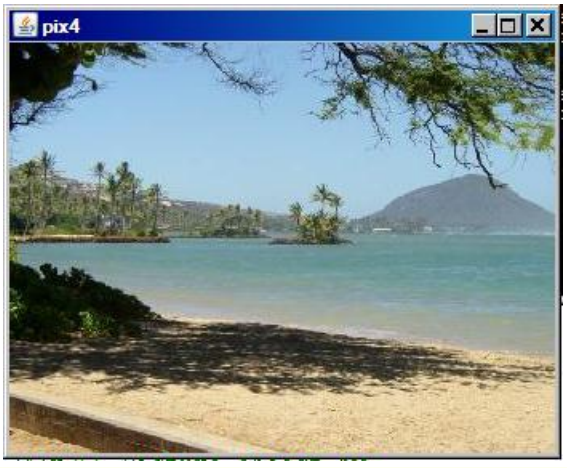
A graphics context
I explained the concept of a *graphics context* in the previous lesson. (See Resources.)

Then Listing 25 calls the **drawImage** method on that graphics context to draw the image received as an incoming parameter on that graphics context. This replaces the all-white pixels with the colored pixels that describe the image.

Display the picture

Finally, Listing 25 calls the **show** method to automatically display the new picture. This causes the call to the **show** method in Listing 24 to be redundant. *(That call to the **show** method could be eliminated and the image shown in Figure 11 would still appear on the screen.)*

Figure 11. Screen output produced by Listing 24.



Looks like the bottom image in Figure 9

Note once again that the image in Figure 11 looks exactly like the bottom image in Figure 9. Only the title is different. This is because the images in the pictures named **pix2** and **pix4** were *derived from* the same image.

A better approach than before

However, unlike the earlier case, future changes made to the image in **pix4** will not be reflected in **pix2** and vice versa. Although the image in pix4 *(shown in Figure 11)* is *derived from* the image in p**ix2,** the **BufferedImage** object referenced in **pix4** is a different object than the **BufferedImage** object referenced in **pix2**. *(We do not have two references to the same **BufferedImage** object in this case.)*

The end of the program named Java354b

Listing 24 signals the end of the **run** method, the end of the **Runner** class, and the end of the program named Java354b.

The program named Java354c

A complete listing of this program is provided in Listing 35 near the end of the lesson.

Only the getPixel method remains to be explained

The programs named Java354a and Java354b illustrated all of the methods declared in the **DigitalPicture** interface other than the **getPixel** method.

The purpose of this program is to illustrate the use of the **getPixel** method as implemented in the **SimplePicture** class, and to compare its use with the **getBasicPixel** and **setBasicPixel** methods.

Create two Picture objects from the same image file

This program begins by creating two **Picture** objects containing the same image. *(Note however that each **Picture** object contains a reference to a different **BufferedImage** object. The images are the same because both **Picture** objects are created from the same image file.)*

Modify the green color component for each row of pixels

Then the program modifies the green color component for each row of pixels in one **Picture** object using the **getBasicPixel** and **setBasicPixel** methods. This approach requires a programming knowledge of *bit manipulations* along with knowledge of how the color components are stored in the integer that represents a pixel.

After that, the program makes the same modifications to the green color components in each row of pixels in the other **Picture** object. In this case, the modifications are made using the **getPixel** method and methods of the **Pixel** class.

Compare the two approaches

This makes it possible to compare the two approaches. The comparison illustrates the reduction in complexity achieved by using the **getPixel** method in place of the **getBasicPixel** and **setBasicPixel** methods.

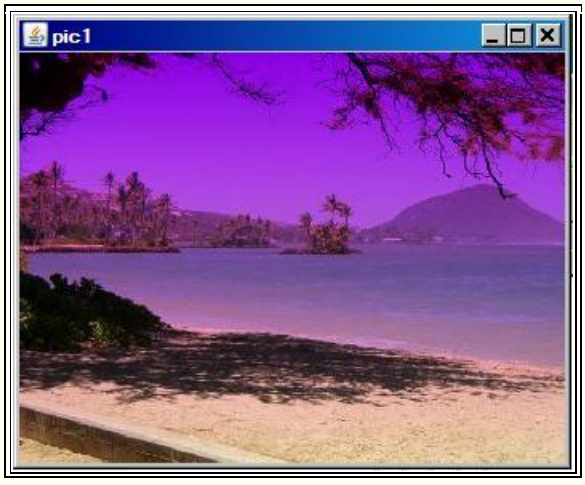
Three statements are required

Both approaches require three statements inside a pair of nested **for** loops, but the three statements involving bit manipulations are much more complex than the statements that call methods on the **Pixel** object.

The same visual results

Each approach produces the same visual result. The two modified images are shown in Figure 12. As you can see, the only differences between the two are the titles. *(The original images are the same as the beach image that I have been using throughout this lesson, so I won't show those images again.)*

Figure 12. Screen output from the program named Java354c.



Modifications to the green color component

The green color component for each image was scaled by zero for every pixel in the first row and was scaled by 1.0 for every pixel in the last row. *(Recall that the overall size of this image was originally scaled so that it would have 256 rows of pixels.)*

The scale factor that was applied to each row between the first and last rows was proportional to the row number.

As you can see, this resulted in an image with a magenta tinge at the top and the correct colors at the bottom. *(Compare the colors in the last row of pixels in the images in Figure 12 with the corresponding pixels in Figure 2.)*

Beginning of the Runner class and the run method

This program begins just like the previous two programs with code that is identical to that shown in Listing 3, so I won't show that code again.

Listing 26 shows the beginning of the **Runner** class and the **run** method.

Listing 26. Beginning of the Runner class and the run method.

```
class Runner{
  void run(){
    //Construct a new 341x256 Picture object
by providing
    // the name of an image file as a
parameter to the
    // Picture constructor.
    Picture pic1 = new
Picture("ScaledBeach.jpg");
    pic1.setTitle("pic1");
    pic1.explore();
```

```
    //Construct another new 341x256 Picture
object by
    // providing the name of an image file as
a parameter
    // to the Picture constructor.
    Picture pic2 = new
Picture("ScaledBeach.jpg");
    pic2.setTitle("pic2");
    pic2.explore();
```

Listing 26 instantiates two new **Picture** objects, having identical images, using code that you have seen before. As you can see, both **Picture** object extract an image from the same image file, so the **BufferedImage** objects in both **Picture** objects contain the same combination of colored pixels.

Scale the green color component using bit manipulations

Listing 27 uses the **getBasicPixel** and **setBasicPixel** methods, in conjunction with a pair of nested **for** loop to scale the green color component in each row as described above.

Listing 27. Scale the green color component using bit manipulations.

```
    //Declare some working constants and
variables.
    final int maskA = 0x0000FF00;//green only
    final int maskB = 0xFFFF00FF;//all but
green
    int pixA = 0;
    int greenByte = 0;

    for(int row = 0;row <
pic1.getHeight();row++){
      for(int col = 0;col <
pic1.getWidth();col++){
        //Working at the bit level, scale the
green byte
        // by 0.0 in the first row and 1.0 in
the last row
        // with proportional scaling in
between.
        pixA = pic1.getBasicPixel(col,row);
        greenByte =
             (int)(((pixA & maskA) >> 8)*
row/255.0) << 8;
        pic1.setBasicPixel(
                       col,row,(pixA & maskB) |
greenByte);
      }//end inner for loop
    }//end outer for loop
    pic1.show();
```

Won't explain the bit manipulation code

It is not my purpose in this lesson to teach you how to do bit manipulations in Java. If you don't understand the code in Listing 27, you should review Java bit manipulations. *(See, for example, my earlier lesson titled The AWT Package, Graphics - Overview of Advanced Image Processing Capabilities in Resources.)*

For the purposes of this lesson, the important thing is to compare the complexity of the code inside the nested **for** loops in Listing 27 with the code inside the nested **for** loops in Listing 28.

Scale the green color component using the getPixel method

Listing 28 scales the green color component in each pixel by repetitively calling the **getPixel** method to get a reference to an object of the **Pixel** class representing each pixel. Then the **getGreen** and **setGreen** methods are called on the **Pixel** object to scale the green color component in the pixel.

Listing 28. Scale the green color component using the getPixel method and methods of the Pixel class.

```
    //Do the same thing to the other picture
working at
    // the Pixel level.
    Pixel pixB = null;
    int greenValue = 0;
    for(int row = 0;row <
pic2.getHeight();row++){
      for(int col = 0;col <
pic2.getWidth();col++){
        pixB = pic2.getPixel(col,row);
        greenValue = (int)(pixB.getGreen() *
row/255.0);
        pixB.setGreen(greenValue);
      }//end inner for loop
    }//end outer for loop
    pic2.show();

  }//end run
}//end class Runner
```

Source code for the getPixel method

The source code for the **getPixel** method is shown in Listing 29.

Listing 29. Source code for the getPixel method.

```
 /**
  * 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;
 }
```

Listing 29 instantiates a new object of the **Pixel** class representing the physical pixel at a specified coordinate position in the image and returns a reference to that **Pixel** object.

Get and set methods for color components are available

I will explain the **Pixel** class in detail in a future lesson. For now, suffice it to say that the **Pixel** class defines *get* and *set* methods that make it possible to manipulate each of the three color components in the manner shown in Listing 28. For example, the **getGreen** method returns the value of the green color component as an eight-bit integer value stored in the least significant eight bits of a value of type **int**.

The **setGreen** method requires an incoming parameter of type **int** that contains the eight-bit value of the green color component as the least significant eight bits of the incoming parameter. This method causes that value to be set into the appropriate group of eight bits that represent the green color component in the **int** value that represents the pixel.

Note that no bit shifting is required to use either of these methods. Note also that there is no requirement for knowledge of how the color components are stored in the integer that represents a pixel.

The end of the program named Java354c

Listing 28 signals the end of the **run** method, the end of the **Runner** class, and the end of the program named Java354c.

Run the programs

I encourage you to copy the code from Listing 33 through Listing 35, 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

As my first attempt to explain the **Picture** class and the **SimplePicture** class, this lesson has illustrated and explained the thirteen methods that are declared in the **DigitalPicture** interface in terms of how those methods are defined in the **SimplePicture** class. The lesson also explained methods that are called by those thirteen methods.

What's next?

In the next lesson, you will 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. You will also learn about the the **PictureFrame** class, which serves as an intermediary between the **Picture** object and the **JFrame** object.

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
- 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

Complete program listings

Complete listings of the programs discussed in this lesson are shown in Listing 30 through Listing 35 below.

Listing 30. 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 31. 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 32. 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 33. Source code for the program named Java354a.

```
/*Program Java354a
Copyright R.G.Baldwin 2009

The purpose of this program is to illustrate the use of
several of the methods that are declared in the
DigitalPicture interface as implemented in the Picture
class.

One Picture object is constructed by using a Picture
constructor that accepts the name of an image file as a
parameter and uses the image from that file as the
image in the Picture object. The picture is displayed by
calling the show method on the Picture object.

In this case, the title is automatically set to the name
of the image file.

A second Picture object is constructed by using a Picture
constructor that accepts the dimensions of the Picture
object only and constructs a Picture object with a default
all-white image. The size of the Picture object that is
constructed is only 1x1.

Then the load method that takes the name of an image file
is called to load the image from an image file into the
small Picture object. The size of the picture object
changes to accommodate the size of the image.

In this case, the default title is "None". The setTitle
and getFileName methods are used to set the title for the
picture. Then the explore method is called to display the
Picture object with its new image and title.

Note that if you call the show method on a picture,
modify the picture, and call the show method on the
picture again, only one copy of the picture is displayed
and the results may not be what you expect to see.
However, displaying the picture in the explore format,
modifying it, and then displaying it again in the show
format seems to work OK.

A pair of nested for loops is used in conjunction with the
getBasicPixel and setBasicPixel methods to copy the right
half of the image in the first Picture object into the
left half of the second Picture object, leaving the right
half of the second Picture object undisturbed.

Then the show method is called on the modified second
picture object to display it.

Note that both image files are in the current directory.

Along the way, the program calls methods dealing with the
file name and the title and eventually prints that
```

```
information on the system console.

The following methods from the DigitalPicture interface
are used in this program.
* String getFileName()
* String getTitle()
* void setTitle(String title)
* int getWidth()
* int getHeight()
* int getBasicPixel(int x, int y)
* void setBasicPixel(int x, int y, int rgb)
* boolean load(String fileName)
* void show()

The following methods that are declared in the
DigitalPicture interface are not used in this program.
* Image getImage()
* BufferedImage getBufferedImage()
* Pixel getPixel(int x, int y)
* void load(Image image)

Tested using Windows Vista Premium Home edition and
Ericson's multimedia library.
**********************************************************/

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

class Runner{
  void run(){
    //Construct a new 341x256 Picture object providing the
    // name of an image file as a parameter.
    Picture pix1 = new Picture("ScaledAquarium.gif");
    pix1.show();//display the picture in the show format

    //Create a new small Picture object with a default
    // all-white image. It must be at least 1x1 or a
    // runtime error will occur.
    Picture pix2 = new Picture(1,1);

    //Load a 341x256 image from a jpg file into the small
    // Picture object. Note that the size of the Picture
    // object increases or decreases to accommodate the
    // size of the image.
    pix2.load("ScaledBeach.jpg");

    //Set the title of the picture.
    pix2.setTitle("pix2: " + pix2.getFileName());

    //Note that if you call the show method on a picture,
    // modify the picture, and call the show method on the
    // picture again, only one copy of the picture is
```

```
    // displayed and the results may not be what you
    // expect to see. However, displaying the picture in
    // the explore format, modifying it, and then
    // displaying it again in the show format seems to
    // work OK.
    pix2.explore();

    //Use the getBasicPixel and setBasicPixel methods to
    // copy the right half of the image from pix1 into the
    // left half of pix2, leaving the right half of pix2
    // undisturbed.
    for(int row = 0;row < pix1.getHeight();row++){
      for(int col = 0;col < pix2.getWidth()/2;col++){
        pix2.setBasicPixel(col,row,pix1.getBasicPixel(
                           col + pix1.getWidth()/2,row));
      }//end inner for loop
    }//end outer for loop
    //Display the final result.
    pix2.show();

    //Display some text on the system console.
    System.out.println(pix1);
    System.out.println("pix1 Filename: "
                                    + pix1.getFileName());
    System.out.println(pix2);
    System.out.println("pix2 FileName: "
                                    + pix2.getFileName());
    System.out.println("pix1 Title: " + pix1.getTitle());
    System.out.println("pix2 Title: " + pix2.getTitle());

  }//end run
}//end class Runner
```

Listing 34. Source code for the program named Java354b.

```
/*Program Java354b
Copyright R.G.Baldwin 2009

The purpose of this program is to illustrate the use of
several of the methods that are declared in the
DigitalPicture interface as implemented in the Picture
class.

The earlier program named Java354a illustrated the use of
all but the following four methods that are declared in
the DigitalPicture interface
* Image getImage()
* BufferedImage getBufferedImage()
* void load(Image image)
* Pixel getPixel(int x, int y)

This program creates and displays four Picture objects
```

```
using the first three methods in the above list, leaving
only the following method to be illustrated in another
program.
* Pixel getPixel(int x, int y)

Tested using Windows Vista Premium Home edition and
Ericson's multimedia library.
*******************************************************/
import java.awt.Image;
public class Main{
  public static void main(String[] args){
    new Runner().run();
  }//end main method
}//end class Main
//-------------------------------------------------------//

class Runner{
  void run(){
    //Construct a new 341x256 Picture object by providing
    // the name of an image file as a parameter to the
    // Picture constructor.
    Picture pix1 = new Picture("ScaledAquarium.gif");
    pix1.setTitle("pix1");
    pix1.show();

    //Construct another new 341x256 Picture object by
    // providing the name of an image file as a parameter
    // to the Picture constructor.
    Picture pix2 = new Picture("ScaledBeach.jpg");
    pix2.setTitle("pix2");
    pix2.show();

    //Construct a third new 341x256 Picture object by
    // extracting the BufferedImage object from pix1 and
    // passing it as a parameter to the constructor for
    // the Picture constructor.
    Picture pix3 = new Picture(pix1.getBufferedImage());
    pix3.setTitle("pix3");
    pix3.show();

    //Construct a fourth new 341x256 Picture object by
    // starting with an all-white Picture object and
    // loading an image extracted from pix2.
    Image image = pix2.getImage();

    //Note that unlike the load method that takes a file
    // name as a parameter, this version of the load
    // method does not automatically set the size of the
    // Picture object to match the size of the image.
    //Get the size of the image and pass those dimensions
    // to the constructor for the Picture object.
    Picture pix4 = new Picture(image.getWidth(null),
                               image.getHeight(null));
    //Now load the image into the Picture object and
    // display the picture.
    pix4.load(image);
```

```
    pix4.setTitle("pix4");
    pix4.show();

  }//end run
}//end class Runner
```

Listing 35. Source code for the program named Java354c.

```
/*Program Java354c
Copyright R.G.Baldwin 2009

The programs named Java354a and Java354b illustrated all
of the methods declared in the DigitalPicture interface
other than the getPixel method.

The purpose of this program is to illustrate the use of
the getPixel method as implemented in the Picture class,
and to compare its use with the getBasicPixel and
setBasicPixel methods.

This program begins by creating two Picture objects
containing the same image. Then the program modifies the
green component for each row of one picture using the
getBasicPixel and setBasicPixel methods. This approach
requires a programming knowledge of bit manipulations.

Then the program does the same thing to the other picture
using the getPixel method and methods of the Pixel class.
This illustrates the reduction in complexity achieved
by using the getPixel method in place of the getBasicPixel
method.

Both approaches require three statements inside a pair of
nested for loops, but the three statements involving bit
manipulations are much more complex. Both processes
produce the same visual result.

With regard to the modification of the green color
component in the two pictures, the green color component
is scaled by zero for every pixel in the first row and is
scaled by 1.0 for every pixel in the bottom row with the
scale factors between the first and last row being
proportional to the row number. This results in an image
with a purple tinge at the top and the correct colors at
the bottom.

Tested using Windows Vista Premium Home edition and
Ericson's multimedia library.
**************************************************/
import java.awt.Image;
public class Main{
  public static void main(String[] args){
```

```
    new Runner().run();
  }//end main method
}//end class Main
//-----------------------------------------------------//

class Runner{
  void run(){
    //Construct a new 341x256 Picture object by providing
    // the name of an image file as a parameter to the
    // Picture constructor.
    Picture pic1 = new Picture("ScaledBeach.jpg");
    pic1.setTitle("pic1");
    pic1.explore();

    //Construct another new 341x256 Picture object by
    // providing the name of an image file as a parameter
    // to the Picture constructor.
    Picture pic2 = new Picture("ScaledBeach.jpg");
    pic2.setTitle("pic2");
    pic2.explore();

    //Modify the green component for each row separately
    // using the getBasicPixel and setBasicPixel methods.
    // This approach requires a programming knowledge of
    // bit manipulations.
    //Do the same thing using the getPixel method to
    // illustrate the reduction in complexity achieved
    // by using the getPixel method.
    //Both approaches require three statements inside a
    // pair of nested for loops, but the three statements
    // involving bit manipulations are much more complex.
    //Note that both Picture objects contain the same
    // image and both processes produce the same visual
    // output.

    //The green color component is scaled by zero for
    // every pixel in the first row and is scaled by
    // 1.0 for every pixel in the bottom row with the
    // scale factors between the first and last row being
    // proportional to the row number. This results in an
    // image with a purple tinge at the top and the
    // correct colors at the bottom.

    //Declare some working constants and variables.
    final int maskA = 0x0000FF00;//green only
    final int maskB = 0xFFFF00FF;//all but green
    int pixA = 0;
    int greenByte = 0;



    for(int row = 0;row < pic1.getHeight();row++){
      for(int col = 0;col < pic1.getWidth();col++){
        //Working at the bit level, scale the green byte
        // by 0.0 in the first row and 1.0 in the last row
        // with proportional scaling in between.
```

```
        pixA = pic1.getBasicPixel(col,row);
        greenByte =
             (int)(((pixA & maskA) >> 8)* row/255.0) << 8;
        pic1.setBasicPixel(
                        col,row,(pixA & maskB) | greenByte);
      }//end inner for loop
    }//end outer for loop
    pic1.show();

    //Do the same thing to the other picture working at
    // the Pixel level.
    Pixel pixB = null;
    int greenValue = 0;
    for(int row = 0;row < pic2.getHeight();row++){
      for(int col = 0;col < pic2.getWidth();col++){
        //Working at the Pixel level, do the same thing.
        pixB = pic2.getPixel(col,row);
        greenValue = (int)(pixB.getGreen() * row/255.0);
        pixB.setGreen(greenValue);
      }//end inner for loop
    }//end outer for loop
    pic2.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-