Java OOP: Darkening, Brightening, and Tinting the Colors in a Picture[*]

R.G. (Dick) Baldwin

Abstract

Learn how to darken, brighten, and tint the colors in a Picture object.

1 Table of Contents

2 Preface

This module is one of a series of modules designed to teach you about Object-Oriented Programming (OOP) using Java.

The program described in this module requires the use of the Guzdial-Ericson multimedia class library. You will find download, installation, and usage instructions for the library at Java OOP: The Guzdial-Ericson Multimedia Class Library [1] .

2.1 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.

*Version 1.3: Nov 14, 2012 11:41 am +0000

†http://creativecommons.org/licenses/by/3.0/

1http://cnx.org/content/m44148/latest/

2.1.1 Figures

2.1.2 Listings

3 Preview

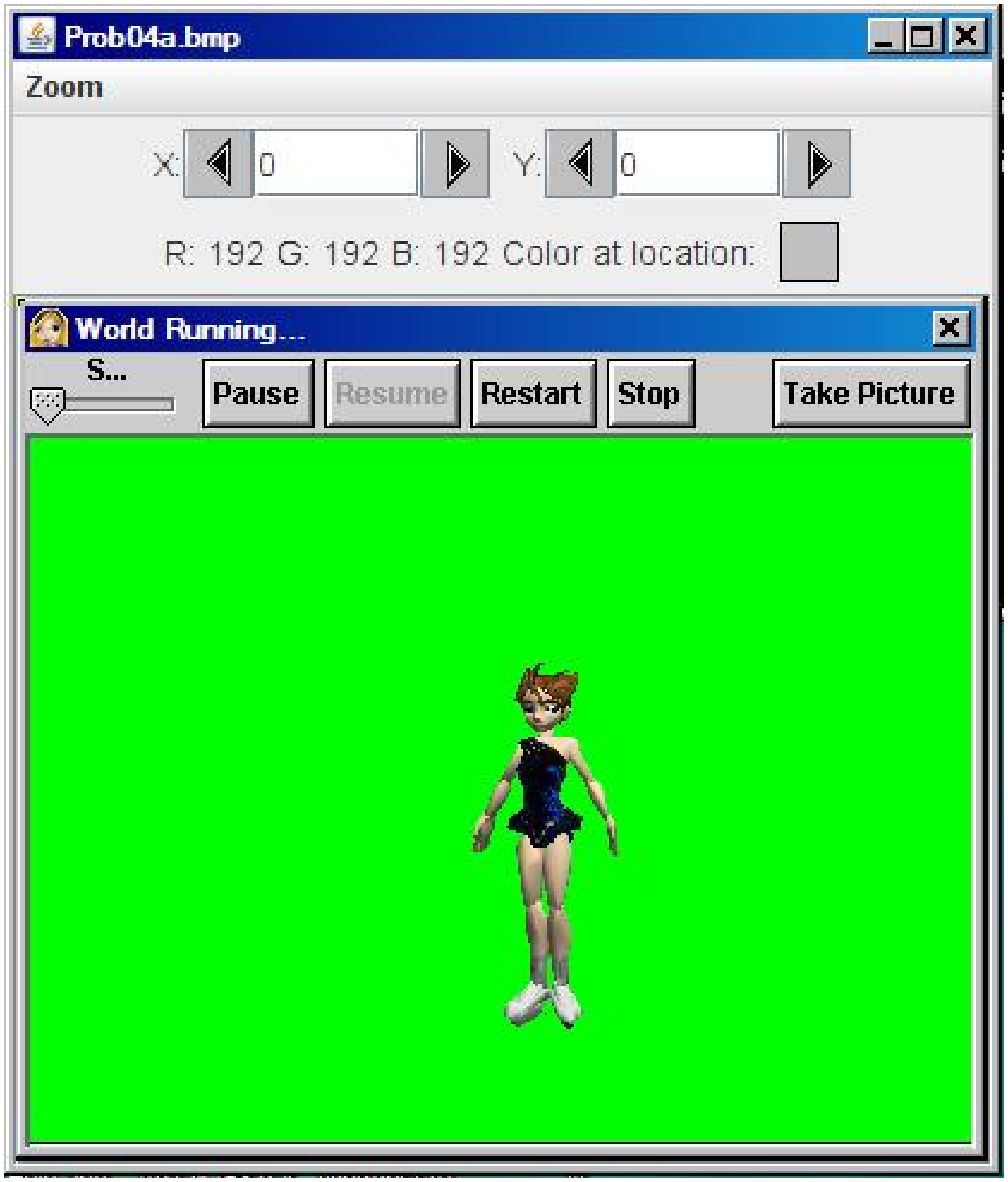
In this module, you will learn how to darken, brighten, and tint the colors in a **Picture** object.

Program specifications

Write a program named **Prob04** that uses the class definition shown in Listing 1 (p. 7) and Ericson's media library along with the image files in the following list to produce the four graphic output images shown in Figure 1 (p. 3) through Figure 4 (p. 6) .

- Prob04a.bmp
- Prob04b.bmp
- Prob04c.jpg

Input file Prob04a.bmp.



R: 192 G: 192 B: 192 Color at location:

Figure 1: Input file Prob04a.bmp.

Input file Prob04b.bmp.

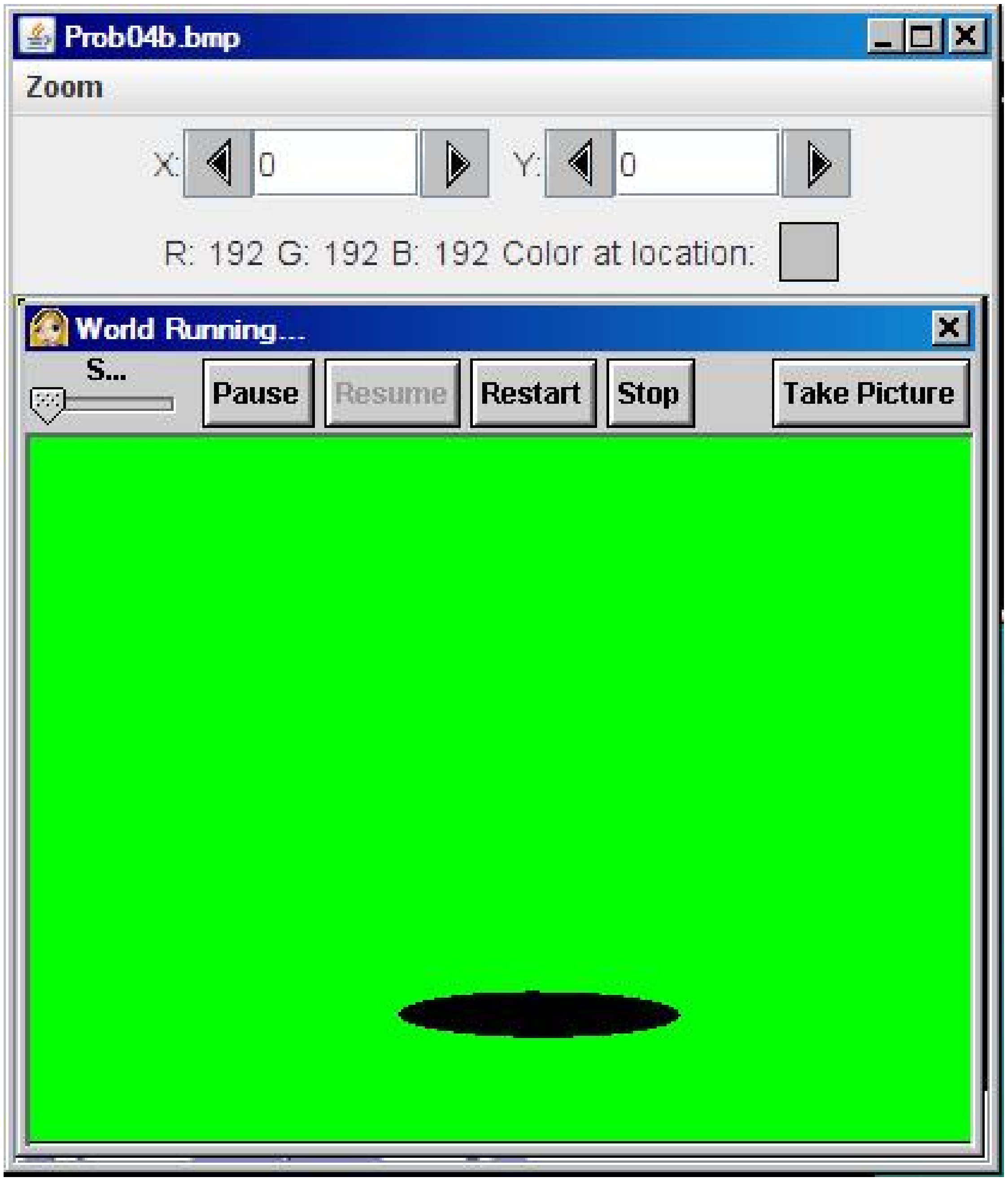
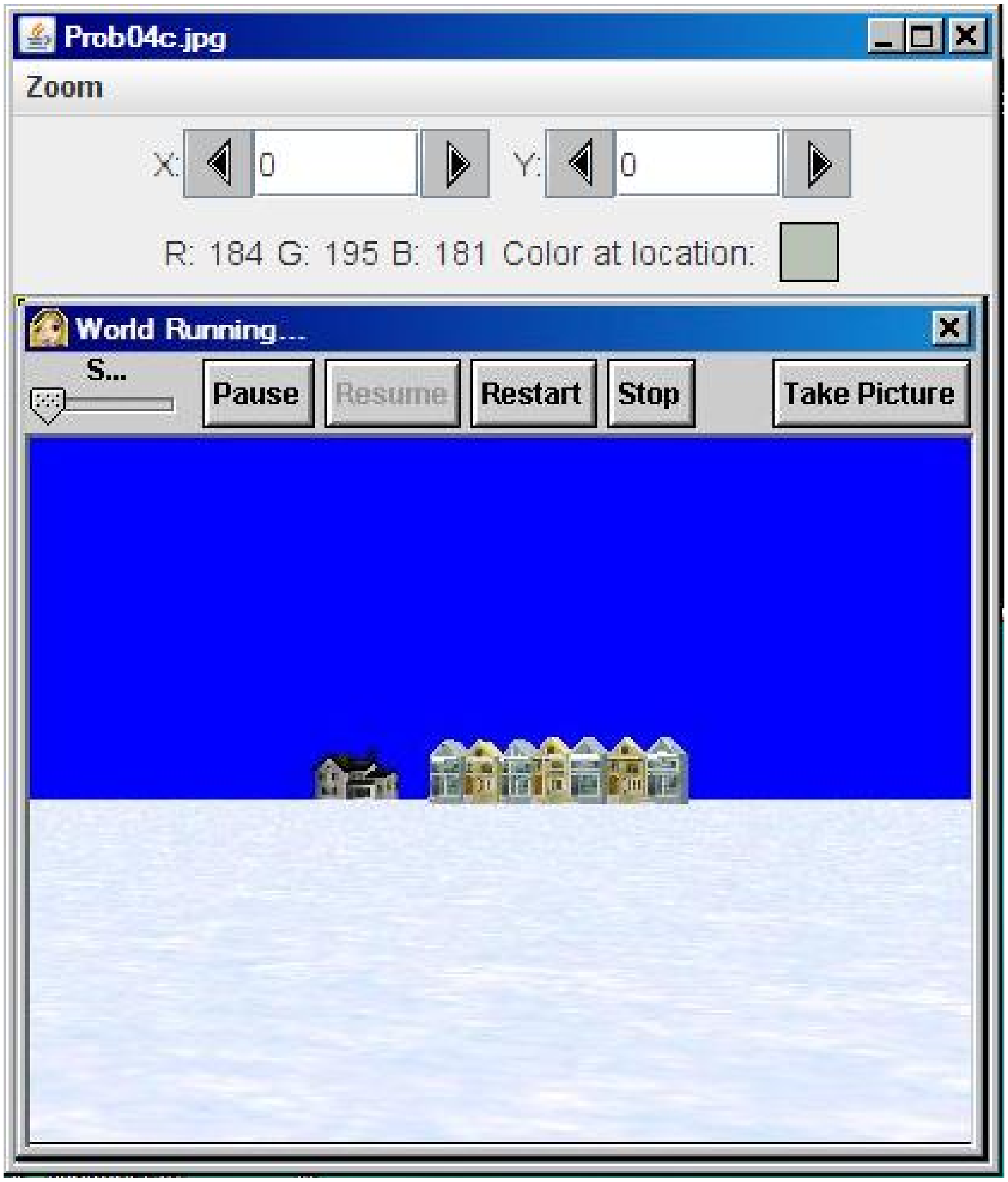


Figure 2: Input file Prob04b.bmp.

Input file Prob04c.jpg.



R: 184 G: 195 B: 181 Color at location:

Figure 3: Input file Prob04c.jpg.

Required output image.

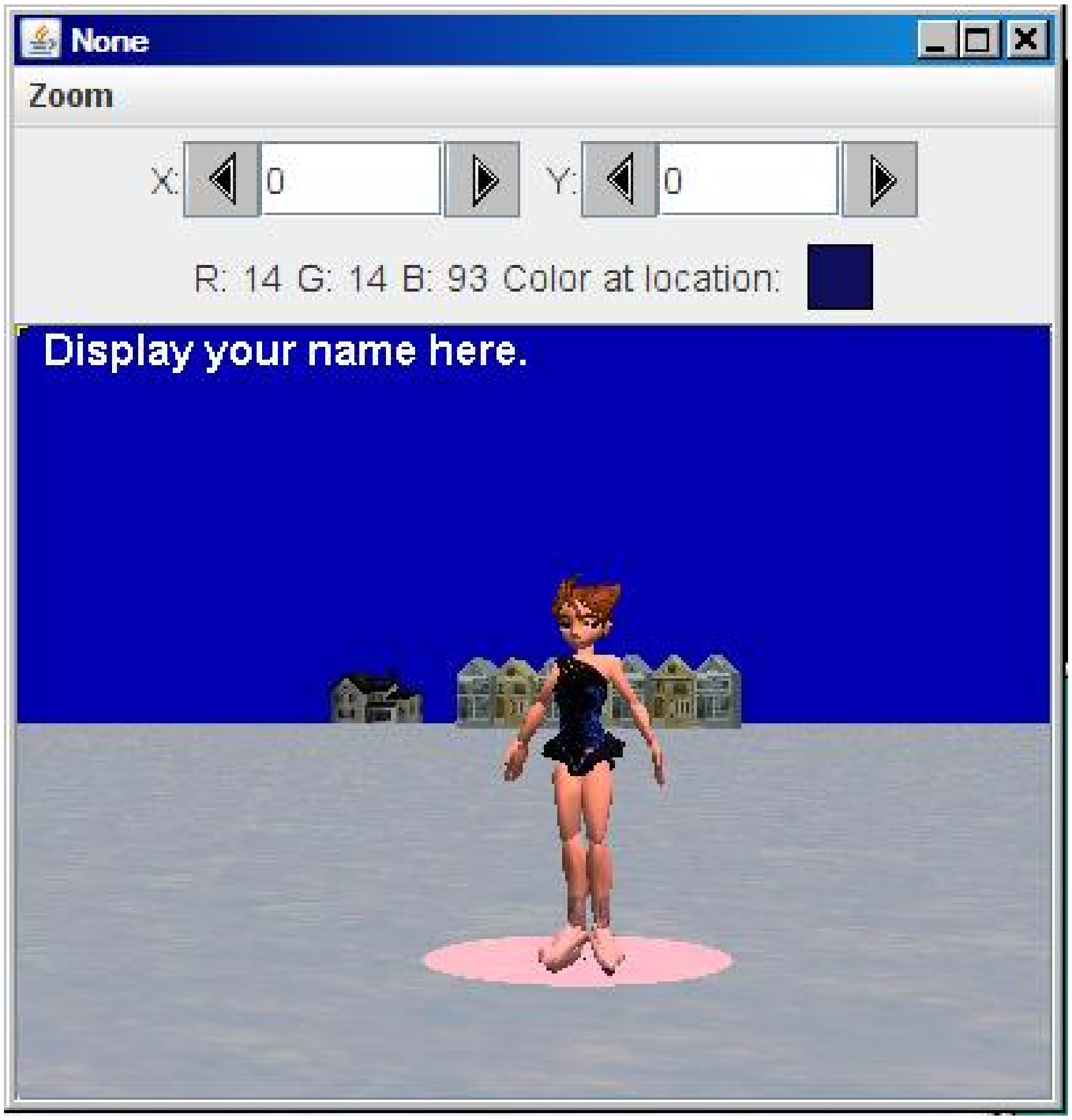


Figure 4: Required output image.

New classes

You may define new classes as necessary to cause your program to behave as required, but you may not modify the class definition for the class named **Prob04** shown in Listing 1 (p. 7) .

Required text output

In addition to the four output images mentioned above, your program must display your name and the other line of text shown in Figure 5 (p. 7) on the command-line screen.

Required text output.

```
    Display your name here.
Picture, filename None height 293 width 392
```

Figure 5: Required text output.

4 General background information

This program uses a black ellipse on a green background *(see Figure 2 (p. 4))* as a pattern to cause the pixels in a snow scene *(see Figure 3 (p. 5))* at the location of the ellipse to be given a red tint and causes all other pixels in the snow scene to be darkened *(see Figure 4 (p. 6))* .

The program also causes a red skater in a green background *(see Figure 1 (p. 3))* to be given a red tint and then drawn on the snow scene at the location of the red-tinted ellipse. The effect is that of a spotlight with a red tint shining on the skater *(see Figure 4 (p. 6))* .

5 Discussion and sample code

Will discuss in fragments

I will discuss this program in fragments. A complete listing of the program is provided in Listing 10 (p. 15) near the end of the module.

The driver class named Prob04

The driver class containing the **main** method is shown in Listing 1 (p. 7) .

Listing 1: The driver class named Prob04.

```
    import java.awt.Color;

public class Prob04{
  public static void main(String[] args){
    Prob04Runner obj = new Prob04Runner();
    obj.run();
  }//end main
}//end class Prob04
```

As has been the case in several earlier modules, the code in the **main** method instantiates an object of the class named **Prob04Runner** and calls the **run** method on that object.

When the **run** method returns, the **main** method terminates causing the program to terminate.

Beginning of the class named Prob04Runner

The class named **Prob04Runner** begins in Listing 2 (p. 7) .

Listing 2: Beginning of the class named Prob04Runner.

```
  class Prob04Runner{

public Prob04Runner(){//constructor
  System.out.println("Display your name here.");
}//end constructor
```

Listing 2 (p. 7) shows the constructor for the class, which simply displays the student's name on the command line screen as shown in Figure 5 (p. 7) .

Beginning of the run method

The beginning of the **run** method, which is called in Listing 1 (p. 7) , is shown in Listing 3 (p. 8) .

Listing 3: Beginning of the run method.

```
  public void run(){

Picture skater = new Picture("Prob04a.bmp");
skater.explore();
skater = crop(skater,6,59,392,293);

Picture hole = new Picture("Prob04b.bmp");
hole.explore();
hole = crop(hole,6,59,392,293);


Picture snowScene = new Picture("Prob04c.jpg");
snowScene.explore();
snowScene = crop(snowScene,6,59,392,293);
```

Instantiate and display three Picture objects

The code in Listing 3 (p. 8) instantiates **Picture** objects from the three image files and displays those pictures in Figure 1 (p. 3) , Figure 2 (p. 4) , and Figure 3 (p. 5) .

Crop the pictures

Note that the images in those three pictures contain the **Alice** [2] runtime window controls as a banner that reads **World Running...** and a line of buttons.

The method named crop

Listing 3 (p. 8) calls a method named **crop** on all three **Picture** objects to eliminate the **Alice** runtime controls. Note that the same rectangular area is preserved for all three images. Thus, all three images are the same size after cropping.

The cropped snow scene image

If you were to display the picture whose reference is stored in the variable named **snowScene** after the **crop** method returns, you would see the image shown in Figure 6 (p. 9) with the **Alice** runtime controls no longer visible.

[2]http://www.alice.org/

Cropped version of the snow scene image.

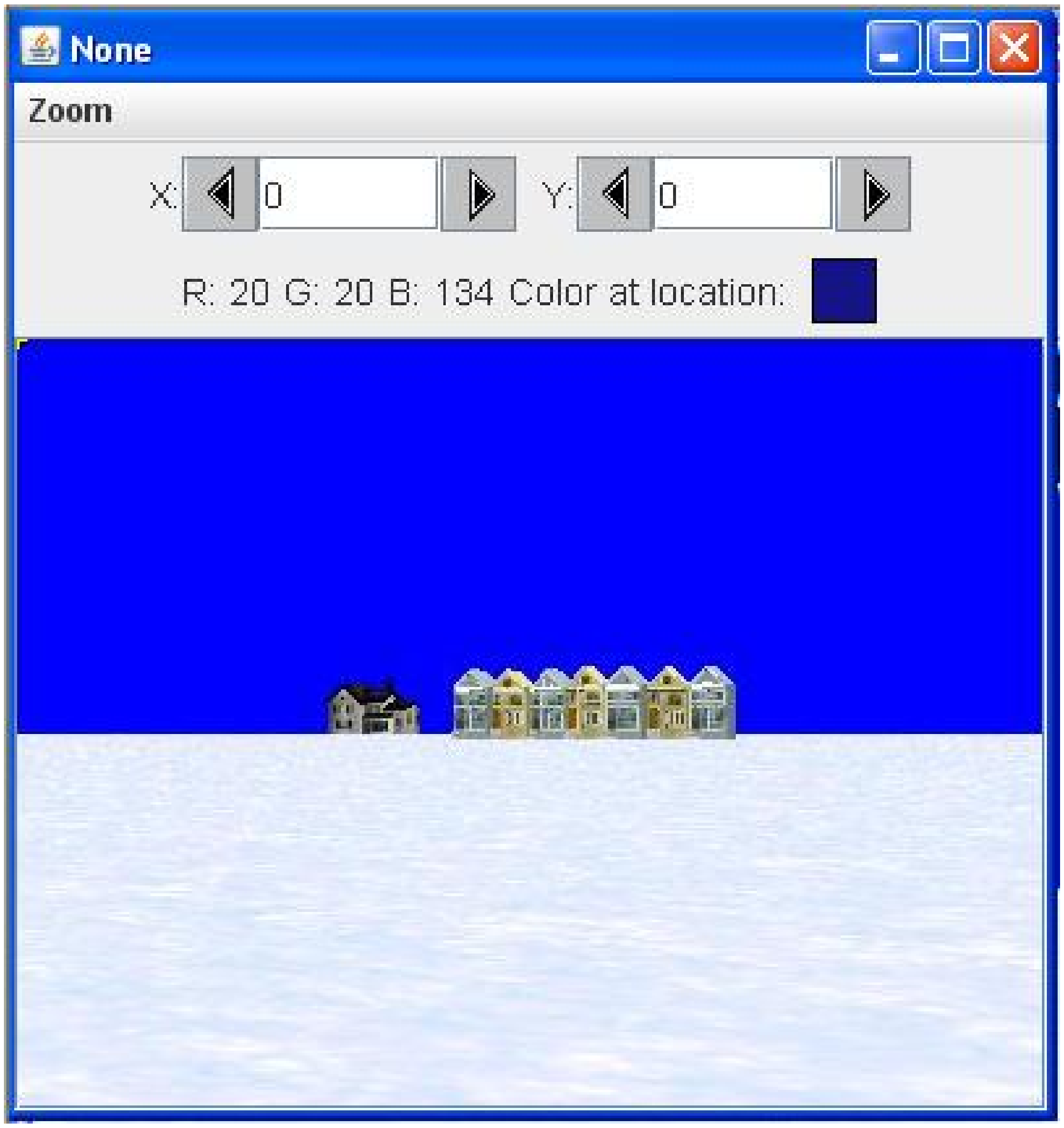


Figure 6: Cropped version of the snow scene image.

The crop method

The method named **crop** that was used to crop the three pictures is essentially the same as the cropping methods that I explained in earlier modules. Therefore, I won't repeat that explanation here. You can view the **crop** method in its entirety in Listing 10 (p. 15) near the end of the module.

Darken the background of the snow scene

Listing 4 (p. 10) calls the method named **darkenBackground** to make all of the pixels darker in the snow scene except for those in the location of the ellipse as shown in Figure 4 (p. 6) . The pixels at the location of the ellipse are given a red tint.

Listing 4: Darken the background of the snow scene.

```
    darkenBackground(hole,snowScene);
```

Put the run method on temporary hold

I will put the explanation of the **run** method on hold at this point and explain the method named **darkenBackground** .

The method named darkenBackground

The method named **darkenBackground** receives references to two **Picture** objects as parameters. It uses the first picture as a *pattern* from which it determines which pixels in the second picture *(destination)* should be darkened.

The *pattern* and *destination* images

In this case, a cropped version of the image of the black ellipse shown in Figure 2 (p. 4) is the pattern. The cropped image of the snow scene shown in Figure 6 (p. 9) is the destination image whose pixels will be darkened.

Assumptions

The **darkenBackground** method assumes that the pattern image has a pure green background as shown in Figure 2 (p. 4) . It also assumes that the pattern and the destination have the same dimensions.

Behavior of the method

The **darkenBackground** method darkens every pixel in the destination that is at the location of a green pixel in the pattern.

The method applies a red tint to every pixel in the destination that is at the location of a non-green pixel in the pattern

Beginning of the darkenBackground method

The **darkenBackground** method begins in Listing 5 (p. 10) .

Listing 5: Beginning of the darkenBackground method.

```
  private void darkenBackground(Picture pattern,
                                 Picture dest){

Pixel[] patternPixels = pattern.getPixels();
Pixel[] destPixels = dest.getPixels();
Color color = null;
int red = 0;
int green = 0;
int blue = 0;
```

Get two arrays of pixel data

The **darkenBackground** method begins by calling the **getPixels** method on each of the picture objects to create a pair of array objects containing pixel data.

You learned how to use the **getPixels** method in an earlier module. Recall that this approach is useful when you don't need to be concerned about the locations of the pixels in an x-y coordinate system.

Arrays have the same length

Because the two pictures have the same dimensions, the two arrays have the same length.

A given array index specifies pixel data from the same location in both pictures.

Beginning of the processing loop

The method uses a **for** loop to traverse the two arrays of pixel data in parallel, using information from the *pattern* picture to make the color changes to the *destination* picture described above. The **for** loop begins in Listing 6 (p. 11) .

Listing 6: Beginning of the processing loop.

```
    for(int cnt = 0;cnt < patternPixels.length;cnt++){
  color = patternPixels[cnt].getColor();
  if(color.equals(Color.GREEN)){
    //Darken corresponding pixel in the destination.
    color = destPixels[cnt].getColor();
    destPixels[cnt].setColor(color.darker());
```

Behavior of the processing loop

The loop begins by getting the color value of the next pixel in the pattern array. If the color of the pattern pixel is green, the code in Listing 6 (p. 11) :

- Gets the color from the corresponding pixel in the destination array.
- Calls the method named **darker** , which is a method of the **Color** class in the standard Sun library, to produce a darker version of the pixel color.
- Replaces the pixel color in the destination array with the darker version of the color.

Compare figures to see the results

If you compare Figure 4 (p. 6) with Figure 3 (p. 5) , you will see that *(ignoring the skater and the ellipse)* , all of the pixels in Figure 4 (p. 6) are darker versions of the colors in Figure 3 (p. 5) .

The brighter method

The **Color** class also provides a method named **brighter** which has the opposite effect. In particular, it can be used to brighten the color of a pixel.

These two methods are very useful for making a pixel darker or brighter without having to know anything about the actual color of the pixel.

The else clause in the processing loop

If the color of the pattern pixel *(tested in Listing 6 (p. 11))* is not green, the code in the **else** clause in Listing 7 (p. 11) is executed.

Listing 7: The else clause in the processing loop.

```
      }else{
    //Apply a red tint to the corresponding pixel in
    // the destination.
    color = destPixels[cnt].getColor();
    red = color.getRed();
    if(red*1.25 < 255){
      red = (int)(red * 1.25);
    }else{
      red = 255;
    }//end else
    green = (int)(color.getGreen() * 0.8);
    blue = (int)(color.getBlue() * 0.8);
    destPixels[cnt].setColor(new Color(red,green,blue));
  }//end else
}//end for loop

}//end darkenBackground
```

The snow scene and the ellipse only...

At this point, only the images from Figure 2 (p. 4) *(the ellipse)* and Figure 3 (p. 5) *(the snow scene)* are being processed. The image of the skater in Figure 1 (p. 3) hasn't entered the picture yet.

Application of the ellipse pattern

The code in Listing 7 (p. 11) is executed only if the pixel from the destination picture is at a location that matches one of the non-green *(black)* pixels in the ellipse in Figure 2 (p. 4) .

(The fact that the pixel is black is of no consequence. The only thing that matters is that it is not green.)

The objective of the else clause

The objective is to modify the pixel color in the destination picture at this location to give it a red tint as shown in Figure 4 (p. 6) .

Get, save, and modify the red color value

Listing 7 (p. 11) begins by getting the color of the pixel from the current location in the destination picture. It extracts and saves the red color value of the pixel. Then, depending on the current value of the red color value, it either:

- Multiplies the red color value by a factor of 1.25, or
- Sets the red color value to the maximum allowable value of 255.

Decrease the color values for blue and green

Following this, it gets the green and blue color values and multiplies each of them by 0.8.

Replace the old pixel color with the new color

Finally, it replaces the pixel color with a new color using the modified values of red, green, and blue.

The texture is preserved

As you can see in Listing 4 (p. 10) , this process causes the pixels in locations that match the ellipse to take on a red tint, but the texture of the image is not destroyed as it would be if the pixels had simply been replaced by pixels that all have the same color of pink.

The end of the darkenBackground method

Listing 7 (p. 11) signals the end of the processing loop and the end of the **darkenBackground** method.

Apply a red tint to the skater

Returning to where we left off in the **run** method in Listing 4 (p. 10) , the code in Listing 8 (p. 12) calls a method named **redTint** , passing a reference to the picture that contains a cropped image of the skater. The method applies a red tint to the skater.

Listing 8: Apply a red tint to the skater.

```
    redTint(skater);
```

The **redTint** method assumes that the image being processed has a pure green background like that shown in Figure 1 (p. 3) . The method applies an algorithm very similar to that shown in Listing 7 (p. 11) to apply a red tint to every pixel that is not pure green.

Because of the similarity of the code in the **redTint** method and the code in Listing 7 (p. 11) , a detailed explanation of the **redTint** method should not be required. You can view the method in its entirety in Listing 10 (p. 15) near the end of the module.

The skater with the red tint applied

If you were to display the picture referred to by **skater** immediately after the **redTint** method returns in Listing 8 (p. 12) , you would see the image shown in Figure 7.

The skater with a red tint applied.

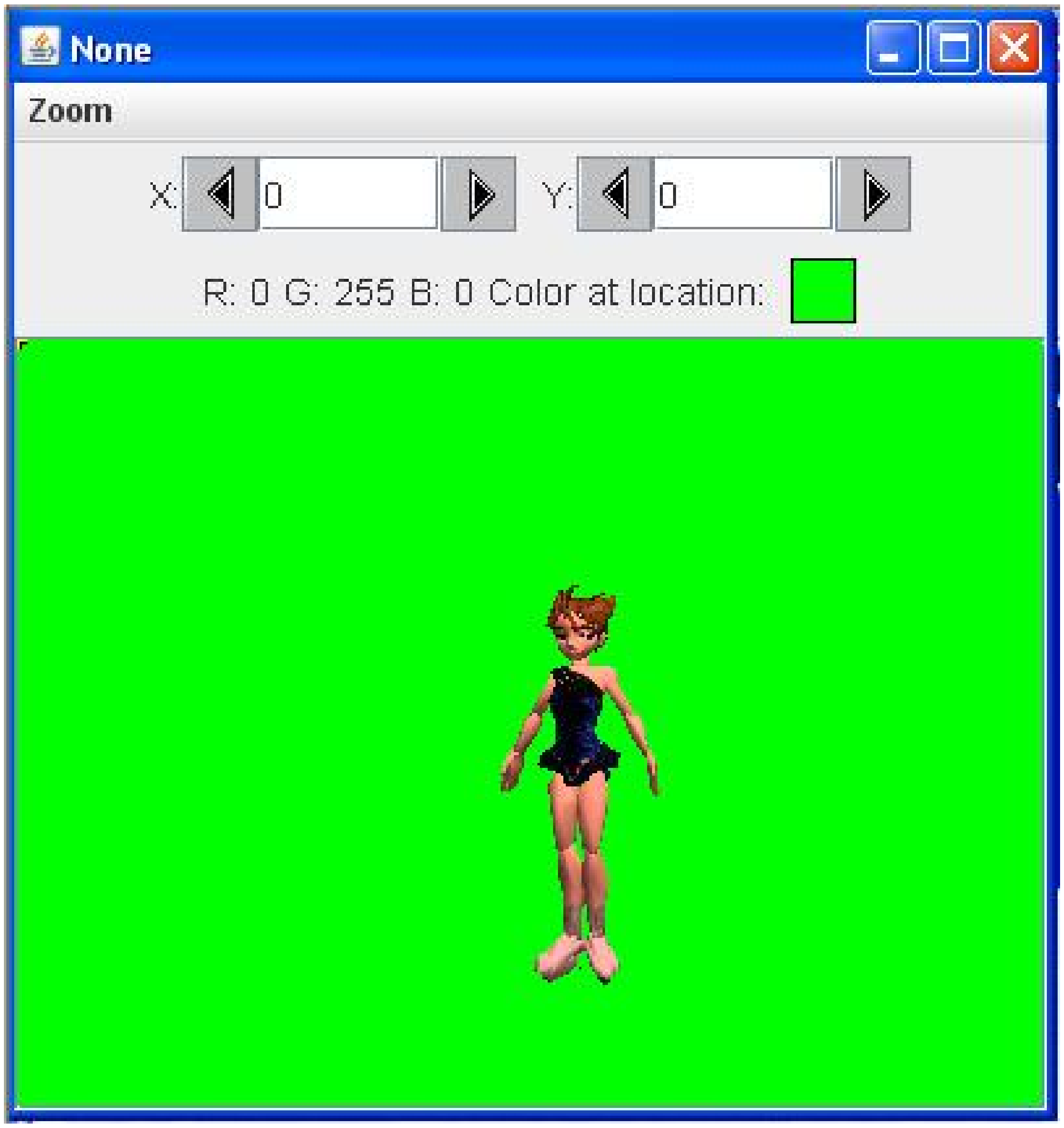


Figure 7: The skater with a red tint applied.

Compare Figure 7 with Figure 1

You can see the effect of applying the red tint process to the skater by comparing Figure 7 (p. 13) with Figure 1 (p. 3) . Note that the process does not change the color of the green pixels.

The remainder of the run method

Continuing with the **run** method, Listing 9 (p. 14) calls a method named **greenScreenDraw** to draw the cropped, red-tinted skater on the snow scene as shown in Figure 4 (p. 6) .

Listing 9: The remainder of the run method.

```
    //Draw the skater on the snowScene.
    greenScreenDraw(skater,snowScene,0,0);

    //Display students name on the final output and
    // display it.
    snowScene.addMessage("Display your name here.",10,15);

    snowScene.explore();
    System.out.println(snowScene);

  }//end run method
```

Behavior of the method

The **greenScreenDraw** method copies all non-green pixels from a source image to a destination image at a specified location. This method is very similar to methods that I have explained in earlier modules. Therefore, an explanation of the method in this module should not be needed.

You can view the **greenScreenDraw** method in its entirety in Listing 10 (p. 15) near the end of the module.

Add text and display the final output image

When the **greenScreenDraw** method returns, Listing 9 (p. 14) calls the **addMessage** method to display the student's name on the snow scene and then calls the **explore** method to produce the output image shown in Figure 4 (p. 6) . None of that should be new to you at this point.

Display text and return

Finally, Listing 9 (p. 14) displays some information on the command line screen as shown in Figure 5 (p. 7) and returns control to the **main** method in Listing 1 (p. 7) .

Terminate the program

Having nothing more to do, the **main** method terminates, causing the program to terminate and return control to the operating system.

6 Run the program

I encourage you to copy the code from Listing 10 (p. 15) . 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.

Click the following links to download the required input files:

- Prob04a.bmp [3]
- Prob04b.bmp [4]
- Prob04c.jpg [5]

7 Summary

In this module, you learned how to darken, brighten, and tint the colors in a **Picture** object.

[3]http://cnx.org/content/m44234/1.3/Prob04a.bmp
[4]http://cnx.org/content/m44234/1.3/Prob04b.bmp
[5]http://cnx.org/content/m44234/1.3/Prob04c.jpg

8 What's next?

You will probably learn more than you already know about interfaces, arrays of type Object, etc., in the next module.

9 Online video links

Select the following links to view online video lectures on the material in this module.

- ITSE 2321 Lecture 09 [6]
  - Part01 [7]
  - Part02 [8]
  - Part03 [9]

10 Miscellaneous

This section contains a variety of miscellaneous information.

NOTE: **Housekeeping material**

- Module name: Java OOP: Darkening, Brightening, and Tinting the Colors in a Picture
- File: Java3018.htm
- Published: August 1, 2012
- Revised: November 14, 2012

NOTE: **Disclaimers:** **Financial** : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.

In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.

Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

11 Complete program listing

A complete listing of the program discussed in this module is shown in Listing 10 (p. 15) below.

Listing 10: Complete program listing.

[6] http://www.youtube.com/playlist?list=PL11F9AC688AC89E56
[7] http://www.youtube.com/watch?v=T-pcmz5XmQY
[8] http://www.youtube.com/watch?v=T16JMwpBIUI
[9] http://www.youtube.com/watch?v=aXLKqYA_0Ng

```
    /*File Prob04 Copyright 2008 R.G.Baldwin
*********************************************************/

import java.awt.Color;
public class Prob04{
  public static void main(String[] args){
    Prob04Runner obj = new Prob04Runner();
    obj.run();
  }//end main
}//end class Prob04
//=====================================================//

class Prob04Runner{

  public Prob04Runner(){//constructor
    System.out.println("Display your name here.");
  }//end constructor
  //---------------------------------------------------//

  public void run(){

    Picture skater = new Picture("Prob04a.bmp");
    skater.explore();
    skater = crop(skater,6,59,392,293);

    Picture hole = new Picture("Prob04b.bmp");
    hole.explore();
    hole = crop(hole,6,59,392,293);


    Picture snowScene = new Picture("Prob04c.jpg");
    snowScene.explore();
    snowScene = crop(snowScene,6,59,392,293);

    //Make all the pixels darker in the snow scene except
    // for those in the location of the hole. Make them
    // brighter.
    darkenBackground(hole,snowScene);

    //Apply a red tint to the skater
    redTint(skater);

    //Draw the skater on the snowScene.
    greenScreenDraw(skater,snowScene,0,0);

    //Display students name on the final output and
    // display it.
    snowScene.addMessage("Display your name here.",10,15);

    snowScene.explore();
    System.out.println(snowScene);
  }//end run method
  //----------------------------------------------------//

  //Assumes the source has a pure green background.
  // Applies a red tint to every pixel that is not pure
  // green.
  private void redTint(Picture pic){
    Pixel[] pixels = pic.getPixels();
    Color color = null;
    int red = 0;
    int green = 0;
    int blue = 0;
    for(int cnt = 0;cnt < pixels.length;cnt++){
      color = pixels[cnt].getColor();
      //Apply a red tint to all non-green pixels
      if(!(color.equals(Color.GREEN))){
        //Increase the value of the red component
        red = color.getRed();
        if(red*1.25 < 255){
          red = (int)(red * 1.25);
        }else{
          red = 255;
        }//end else
        //Decrease the value of blue and green
        green = (int)(color.getGreen()*0.8);
        blue = (int)(color.getBlue()*0.8);

        //Apply the new color to the pixel.
        pixels[cnt].setColor(new Color(red,green,blue));
      }//end if
    }//end for loop
  }//end redTint
  //----------------------------------------------------//

  //Assumes the pattern image has a pure green background.
  // Assumes that the pattern and the destination have the
  // same dimensions. Darkens every pixel in the
  // destination that is at the location of a green pixel
  // in the pattern. Applies a red tint to every pixel
  // in the destination that is at the location of a
  // non-green pixel in the pattern
  private void darkenBackground(
                         Picture pattern,
                         Picture dest){

    Pixel[] patternPixels = pattern.getPixels();
    Pixel[] destPixels = dest.getPixels();
    Color color = null;
    int red = 0;
    int green = 0;
    int blue = 0;

    for(int cnt = 0;cnt < patternPixels.length;cnt++){
      color = patternPixels[cnt].getColor();
      if(color.equals(Color.GREEN)){
        //Darken corresponding pixel in the destination.
        color = destPixels[cnt].getColor();
        destPixels[cnt].setColor(color.darker());
      }else{
        //Apply a red tint to the corresponding pixel in
        // the destination.
        color = destPixels[cnt].getColor();
        red = color.getRed();
        if(red*1.25 < 255){
          red = (int)(red * 1.25);
        }else{
          red = 255;
        }//end else
        green = (int)(color.getGreen() * 0.8);
        blue = (int)(color.getBlue() * 0.8);
        destPixels[cnt].setColor(new Color(red,green,blue));
      }//end else
    }//end for loop
  }//end darkenBackground
  //----------------------------------------------------//

  //Assumes a source image with a pure green background.
  // Copies all non-green pixels from the source image to
  // the destination image at the location explained
  // below. Note that JPEG images typically won't have
  // a pure green background even if they had a pure
  // green background before being compressed into the
  // JPEG format.  BMP images work well for this.
  private void greenScreenDraw(
                         Picture source,
                         Picture dest,
                         //Place the upper-left corner
                         // of the source image at the
                         // following location in the
                         // destination image.
                         int destX,
                         int destY){
    int width = source.getWidth();
    int height = source.getHeight();
    Pixel pixel = null;
    Color color = null;

    for(int row = 0;row < height;row++){
      for(int col = 0;col < width;col++){
        color = source.getPixel(col,row).getColor();
        if(!(color.equals(Color.GREEN))){
          pixel = dest.getPixel(destX + col,destY + row);
          pixel.setColor(color);
        }//end if
      }//end inner loop
    }//end outer loop

  }//end greenScreenDraw
  //----------------------------------------------------//

  //Crops a Picture object to the given width and height
  // with the upper-left corner located at startCol and
  // startRow.
  private Picture crop(Picture pic,int startCol,
                                   int startRow,
                                   int width,
                                   int height){
    Picture output = new Picture(width,height);

    int colOut = 0;
    int rowOut = 0;
    int col = 0;
    int row = 0;
    Pixel pixel = null;
    Color color = null;
    for(col = startCol;col < startCol+width;col++){
      for(row = startRow;row < startRow+height;row++){
        color = pic.getPixel(col,row).getColor();
        pixel = output.getPixel(colOut,rowOut);
        pixel.setColor(color);
        rowOut++;
      }//end inner loop
      rowOut = 0;
      colOut++;
    }//end outer loop
    return output;
  }//end crop

}//end class Prob04Runner
```

-end-