AP® Computer Science A Picture Lab Student Guide

The AP Program wishes to acknowledge and thank Barbara Ericson of the Georgia Institute of Technology, who developed this lab and the accompanying documentation.

Picture Lab: Student Guide

Introduction
In this lab you will be writing methods that modify digital pictures. In writing these methods you will learn how to traverse a two-dimensional array of integers or objects. You will also be introduced to nested loops, binary numbers, interfaces, and inheritance.

Activities
You will be working through a set of activities. These activities will help you learn about how:

- digital pictures are represented on a computer;
- the binary number system is used to represent values;
- to create colors using light;
- Java handles two-dimensional arrays;
- data from a picture is stored; and
- to modify a digital picture.

Set-up
You will need the `pixLab` folder and a Java Development Kit, also known as a JDK (see http://www.oracle.com/technetwork/java/javase/downloads/index.html). A development environment is also useful. DrJava is a free development environment for Java that allows students to try out code in an interactions pane. It also has a debugger, and can be downloaded from http://drjava.org. However, you can use any development environment with this lab. Just open the files in the `classes` folder and compile them. Please note that there are two small pictures in the `classes` folder that need to remain there: `leftArrow.gif` and `rightArrow.gif`. If you copy the Java source files to another folder you must copy these gif files as well.

Keep the `images` folder and the `classes` folder together in the `pixLab` folder. The `FileChooser` expects the images to be in a folder called `images`, at the same level as the `classes` folder. If it does not find the images there it also looks in the same folder as the class files that are executing. If you wish to modify this, change the `FileChooser.java` class to specify the folder where the pictures are stored. For example, if you want to store the images in "`r://student/images/`," change the following line in the method `getMediaDirectory()` in `FileChooser.java`:

```
URL fileURL = new URL(classURL,"../images/");
```

And modify it to

```
URL fileURL = new URL("r://student/images/");
```

Then recompile.

A1: Introduction to digital pictures and color

If you look at an advertisement for a digital camera, it will tell you how many *megapixels* the camera can record. What is a megapixel? A digital camera has sensors that record color at millions of points arranged in rows and columns (Figure 1). Each point is a *pixel* or *picture (abbreviated **pix**) **el**ement*. A *megapixel* is one million pixels. A 16.2 megapixel camera can store the color at over 16 million pixels. That's a lot of pixels! Do you really need all of them? If you are sending a small version of your picture to a friend's phone, then just a few megapixels will be plenty. But, if you are printing a huge poster from a picture or you want to zoom in on part of the picture, then more pixels will give you more detail.

How is the color of a pixel recorded? It can be represented using the RGB (Red, Green, Blue) color model, which stores values for red, green, and blue, each ranging from 0 to 255. You can make yellow by combining red and green. That probably sounds strange, but combining pixels isn't the same as mixing paint to make a color. The computer uses light to display color, not paint. Tilt the bottom of a CD in white light and you will see lots of colors. The CD acts as a prism and lets you see all the colors in white light. The RGB color model sometimes also stores an alpha value as well as the red, green, and blue values. The alpha value indicates how transparent or opaque the color is. A color that is transparent will let you see some of the color beneath it.

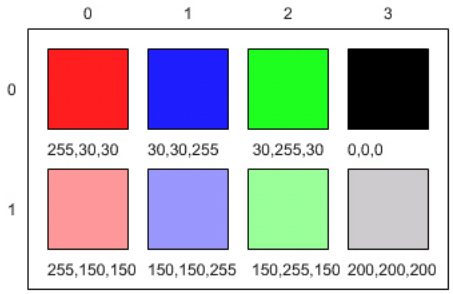

Figure 1: RGB values and the resulting colors displayed in rows and columns

How does the computer represent the values from 0 to 255? A decimal number uses the digits 0 to 9 and powers of 10 to represent values. The decimal number 325 means 5 ones (10^0) plus 2 tens (10^1) plus 3 hundreds (10^2) for a total of three hundred and twenty-five. Computers use *binary numbers*, which use the digits 0 and 1 and powers of 2 to represent values using groups of bits. A *bit* is a **b**inary dig**it,** which can be either 0 or 1. A group of 8 bits is called a *byte*. The binary number 110 means 0 ones (2^0) plus 1 two (2^1) plus 1 four (2^2), for a total of 6.

Questions

1. How many bits does it take to represent the values from 0 to 255?
2. How many bytes does it take to represent a color in the RGB color model?
3. How many pixels are in a picture that is 640 pixels wide and 480 pixels high?

A2: Picking a color

Run the `main` method in `ColorChooser.java`. This will pop up a window (Figure 2) asking you to pick a color. Click on the RGB tab and move the sliders to make different colors.

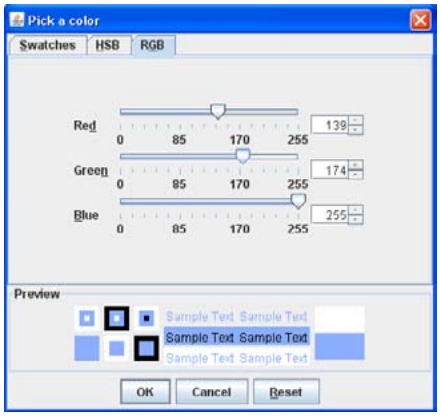

Figure 2: The Color Chooser (This is the version from Java 6.)

When you click the OK button, the red, green, and blue values for the color you picked will be displayed as shown below. The `Color` class has a `toString` method that displays the class name followed by the red, green, and blue values. The `toString` method is automatically called when you print an object.

```
java.awt.Color[r=139,g=174,b=255]
```

Java represents color using the `java.awt.Color` class. This is the *full name* for the `Color` class, which includes the *package* name of `java.awt` followed by a period and then the class name `Color`. Java groups related classes into *packages*. The *awt* stands for Abstract Windowing Toolkit, which is the package that contains the original Graphical User Interface (GUI) classes developed for Java. You can use just the short name for a class, like `Color`, as long as you include an import statement at the beginning of a class source file, as shown below. The `Picture` class contains the following import statement.

```
import java.awt.Color;
```

Use the `ColorChooser` class (run the `main` method) to answer the following questions.

Questions

1. How can you make pink?
2. How can you make yellow?
3. How can you make purple?
4. How can you make white?

5. How can you make dark gray?

A3: Exploring a picture

Run the `main` method in `PictureExplorer.java`. This will load a picture of a beach from a file, make a copy of that picture in memory, and show it in the explorer tool (Figure 3). It makes a copy of the picture to make it easier to explore a picture both before and after any changes. You can use the explorer tool to explore the pixels in a picture. Click any location (pixel) in the picture and it will display the row index, column index, and red, green, and blue values for that location. The location will be highlighted with yellow crosshairs. You can click on the arrow keys or even type in values and hit the enter button to update the display. You can also use the menu to change the zoom level.

Figure 3: The Picture Explorer

Questions

1. What is the row index for the top left corner of the picture?
2. What is the column index for the top left corner of the picture?
3. The width of this picture is 640. What is the right most column index?
4. The height of this picture is 480. What is the bottom most row index?
5. Does the row index increase from left to right or top to bottom?
6. Does the column index increase from left to right or top to bottom?
7. Set the zoom to 500%. Can you see squares of color? This is called *pixelation*. Pixelation means displaying a picture so magnified that the individual pixels look like small squares.

Creating and exploring other pictures

Here is the `main` method in the class `PictureExplorer`. Every class in Java can have a `main` method, and it is where execution starts when you execute the command `java ClassName`.

```
public static void main( String args[])
{
  Picture pix = new Picture("beach.jpg");
  pix.explore();
}
```

The body of the `main` method declares a reference to a `Picture` object named `pix` and sets that variable to refer to a `Picture` object created from the data stored in a JPEG file named "`beach.jpg`" in the `images` folder. A JPEG file is one that follows an international standard for storing picture data using *lossy compression*. *Lossy compression* means that the amount of data that is stored is much smaller than the available data, but the part that is not stored is data we won't miss.

Exercises

1. Modify the `main` method in the `PictureExplorer` class to create and explore a different picture from the `images` folder.
2. Add a picture to the `images` folder and then create and explore that picture in the `main` method. If the picture is very large (for instance, one from a digital camera), you can scale it using the `scale` method in the `Picture` class.
   For example, you can make a new picture ("`smallMyPicture.jpg`" in the `images` folder) one-fourth the size of the original ("`myPicture.jpg`") using:

   ```
   Picture p = new Picture("myPicture.jpg");
   Picture smallP = p.scale(0.25,0.25);
   smallP.write("smallMyPicture.jpg");
   ```

A4: Two-dimensional arrays in Java

In this activity you will work with integer data stored in a two-dimensional array. Some programming languages use a one-dimensional (1D) array to represent a two-dimensional (2D) array with the data in either *row-major* or *column-major order*. *Row-major order* in a 1D array means that all the data for the first row is stored before the data for the next row in the 1D array. *Column-major order* in a 1D array means that all the data for the first column is stored before the data for the next column in the 1D array. The order matters, because you need to calculate the position in the 1D array based on the order, the number of rows and columns, and the current column and row numbers (indices). The rows and columns are numbered (indexed) and often that numbering starts at 0 as it does in Java. The top left row has an index of 0 and the top left column has an index of 0. The row number (index) increases from top to bottom and the column number (index) increases from left to right as shown below.

	0	1	2
0	1	2	3
1	4	5	6

If the above 2D array is stored in a 1D array in row-major order it would be:

0	1	2	3	4	5
1	2	3	4	5	6

If the above 2D array is stored in a 1D array in column-major order it would be:

0	1	2	3	4	5
1	4	2	5	3	6

Java actually uses arrays of arrays to represent 2D arrays. This means that each element in the outer array is a reference to another array. The data can be in either row-major or column-major order (Figure 4). The AP Computer Science A course specification tells you to assume that all 2D arrays are row-major, which means that the outer array in Java represents the rows and the inner arrays represent the columns.

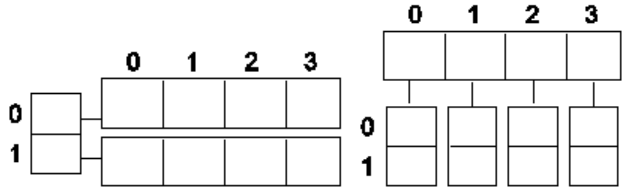

Figure 4: A row-major 2D array (left) and a column-major 2D array (right)

The following table shows the Java syntax and examples for tasks with 2D arrays. Java supports 2D arrays of primitive and object types.

Task	Java Syntax	Examples
Declare a 2D array	`type[][] name`	`int[][] matrix` `Pixel[][] pixels`
Create a 2D array	`new type[nRows][nCols]`	`new int[5][8]` `new Pixel[numRows][numCols]`
Access an element	`name[row][col]`	`int value = matrix[3][2];` `Pixel pixel = pixels[r][c];`
Set the value of an element	`name[row][col] = value`	`matrix[3][2] = 8;` `pixels[r][c] = aPixel;`
Get the number of rows	`name.length`	`matrix.length` `pixels.length`
Get the number of columns	`name[0].length`	`matrix[0].length` `pixels[0].length`

To loop through the values in a 2D array you must have two indexes. One index is used to change the row index and one is used to change the column index. You can use *nested loops,* which is one `for` loop inside of another, to loop through all the values in a 2D array.

Here is a method in the `IntArrayWorker` class that totals all the values in a 2D array of integers in a private instance variable (field in the class) named `matrix`. Notice the nested `for` loop and how it uses `matrix.length` to get the number of rows and `matrix[0].length` to get the number of columns. Since `matrix[0]` returns the inner array in a 2D array, you can use `matrix[0].length` to get the number of columns.

```
public int getTotal()
{
  int total = 0;
  for (int row = 0; row < matrix.length; row++)
  {
    for (int col = 0; col < matrix[0].length; col++)
    {
      total = total + matrix[row][col];
    }
  }
  return total;
}
```

Because Java two-dimensional arrays are actually arrays of arrays, you can also get the total using nested `for-each` loops as shown in `getTotalNested` below. The outer loop will loop through the outer array (each of the rows) and the inner loop will loop through the inner array (columns in that row). You can use a nested `for-each` loop whenever you want to loop through all items in a 2D array and you don't need to know the row index or column index.

```
public int getTotalNested()
{
  int total = 0;
  for (int[] rowArray : matrix)
  {
    for (int item : rowArray)
    {
      total = total + item;
    }
  }
  return total;
}
```

Exercises

1. Write a `getCount` method in the `IntArrayWorker` class that returns the count of the number of times a passed integer value is found in the matrix. There is already a method to test this in `IntArrayWorkerTester`. Just uncomment the method `testGetCount()` and the call to it in the `main` method of `IntArrayWorkerTester`.
2. Write a `getLargest` method in the `IntArrayWorker` class that returns the largest value in the matrix. There is already a method to test this in `IntArrayWorkerTester`. Just uncomment the method `testGetLargest()` and the call to it in the `main` method of `IntArrayWorkerTester`.
3. Write a `getColTotal` method in the `IntArrayWorker` class that returns the total of all integers in a specified column. There is already a method to test this in `IntArrayWorkerTester`. Just uncomment the method `testGetColTotal()` and the call to it in the `main` method of `IntArrayWorkerTester`.

A5: Modifying a picture

Even though digital pictures have millions of pixels, modern computers are so fast that they can process all of them quickly. You will write methods in the `Picture` class that modify digital pictures. The `Picture` class inherits from the `SimplePicture` class and the `SimplePicture` class implements the `DigitalPicture` interface as shown in the Unified Modeling Language (UML) class diagram in Figure 5.

A UML class diagram shows classes and the relationships between the classes. Each class is shown in a box with the class name at the top. The middle area shows attributes (instance or class variables) and the bottom area shows methods. The open triangle points to the class that the connected class inherits from. The straight line links show associations between classes. Association is also called a "has-a" relationship. The numbers at the end of the association links give the number of objects associated with an object at the other end. For example, in Figure 5 it shows that one `Pixel` object has one `Color` object associated with it and that a `Color` object can have zero to many `Pixel` objects associated with it. You may notice that the UML class diagram doesn't look exactly like Java code. UML isn't language specific.

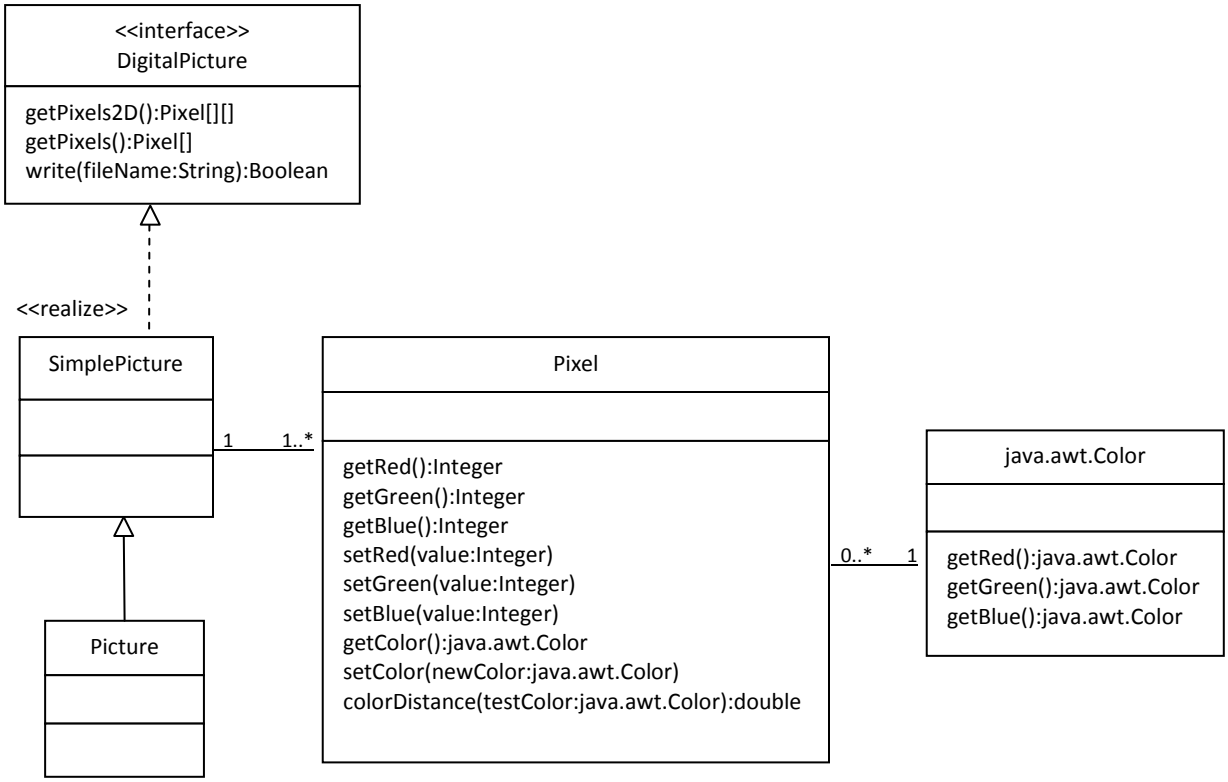

Figure 5: A UML Class Diagram

Questions

1. Open `Picture.java` and look for the method `getPixels2D`. Is it there?
2. Open `SimplePicture.java` and look for the method `getPixels2D`. Is it there?
3. Does the following code compile?
```
DigitalPicture p = new DigitalPicture();
```
4. Assuming that a no-argument constructor exists for `SimplePicture`, would the following code compile?
```
DigitalPicture p = new SimplePicture();
```
5. Assuming that a no-argument constructor exists for `Picture`, does the following code compile?
```
DigitalPicture p = new Picture();
```
6. Assuming that a no-argument constructor exists for `Picture`, does the following code compile?
```
SimplePicture p = new Picture();
```
7. Assuming that a no-argument constructor exists for `SimplePicture`, does the following code compile?
```
Picture p = new SimplePicture();
```

`DigitalPicture` is an *interface*. An *interface* most often only has public abstract methods. An *abstract method* is not allowed to have a body. Notice that none of the methods declared in `DigitalPicture` have a body. If a method can't have a body, what good is it?

Interfaces are useful for separating **what** from **how**. An interface specifies **what** an object of that type needs to be able to do but not **how** it does it. You cannot create an object using an interface type. A class can *implement* (*realize*) an interface as `SimplePicture` does. A non-abstract class provides bodies for all the methods declared in the interface, either directly or through inheritance. You can declare a variable to be of an interface type and then set that variable to refer to an object of any class that implements that interface. For example, Java has a `List` interface that declares the methods that a list should have such as `add`, `remove`, and `get`, etc. But, if you want to create a `List` object you will create an `ArrayList` object. It is recommended that you declare a variable to be of type `List`, not `ArrayList`, as shown below (for a list of names).

```
List<String> nameList = new ArrayList<String>();
```

Why wouldn't you just declare `nameList` to be of the type `ArrayList<String>`? There are other classes in Java that implement the `List` interface. By declaring `nameList` to be of the type `List<String>` instead of `ArrayList<String>`, it is easy to change your mind in the future and use another class that implements the same interface. Interfaces give you some flexibility and reduce the number of changes you might need to make in the future, as long as your code only uses the functionality defined by the interface.

Because `DigitalPicture` declares a `getPixels2D` method that returns a two-dimensional array of `Pixel` objects, `SimplePicture` implements that interface, and `Picture` inherits

from SimplePicture, you can use the getPixels2D method on a Picture object. You can loop through all the Pixel objects in the two-dimensional array to modify the picture. You can get and set the red, green, and/or blue value for a Pixel object. You can also get and/or set the Color value for a Pixel object. You can create a new Color object using a constructor that takes the red, green, and blue values as integers as shown below.

```
Color myColor = new Color(255,30,120);
```

What do you think you will see if you modify the beach picture in the images folder to set all the blue values to zero? Do you think you will still see a beach? Run the main method in the Picture class. The body of the main method will create a Picture object named beach from the "beach.jpg" file, open an explorer on a copy of the picture (in memory), call the method that sets the blue values at all pixels to zero, and then open an explorer on a copy of the resulting picture.

The following code is the main method from the Picture class.

```
public static void main(String[] args)
{
  Picture beach = new Picture("beach.jpg");
  beach.explore();
  beach.zeroBlue();
  beach.explore();
}
```

Exercises

1. Open PictureTester.java and run its main method. You should get the same results as running the main method in the Picture class. The PictureTester class contains class (static) methods for testing the methods that are in the Picture class.
2. Uncomment the appropriate test method in the main method of PictureTester to test any of the other methods in Picture.java. You can comment out the tests you don't want to run. You can also add new test methods to PictureTester to test any methods you create in the Picture class.

The method zeroBlue in the Picture class gets a two-dimensional array of Pixel objects from the current picture (the picture the method was called on). It then declares a variable that will refer to a Pixel object named pixelObj. It uses a nested for-each loop to loop through all the pixels in the picture. Inside the body of the nested for-each loop it sets the blue value for the current pixel to zero. Note that you cannot change the elements of an array when you use a for-each loop. If, however, the array elements are references to objects that have methods that allow changes, you can change the internal state of objects referenced in the array (pixels).

The following code is the `zeroBlue` method in the `Picture` class.

```
public void zeroBlue()
{
  Pixel[][] pixels = this.getPixels2D();
  for (Pixel[] rowArray : pixels)
  {
    for (Pixel pixelObj : rowArray)
    {
      pixelObj.setBlue(0);
    }
  }
}
```

Exercises

3. Using the `zeroBlue` method as a starting point, write the method `keepOnlyBlue` that will keep only the blue values, that is, it will set the red and green values to zero. Create a class (static) method to test this new method in the class `PictureTester`. Be sure to call the new test method in the `main` method in `PictureTester`.
4. Write the `negate` method to negate all the pixels in a picture. To negate a picture, set the red value to 255 minus the current red value, the green value to 255 minus the current green value and the blue value to 255 minus the current blue value. Create a class (static) method to test this new method in the class `PictureTester`. Be sure to call the new test method in the `main` method in `PictureTester`.
5. Write the `grayscale` method to turn the picture into shades of gray. Set the red, green, and blue values to the average of the current red, green, and blue values (add all three values and divide by 3). Create a class (static) method to test this new method in the class `PictureTester`. Be sure to call the new test method in the `main` method in `PictureTester`.
6. Challenge — Explore the "`water.jpg`" picture in the `images` folder. Write a method `fixUnderwater()` to modify the pixel colors to make the fish easier to see. Create a class (static) method to test this new method in the class `PictureTester`. Be sure to call the new test method in the `main` method in `PictureTester`.

A6: Mirroring pictures

Car designers at General Motors Research Labs only sculpt half of a car out of clay and then use a vertical mirror to reflect that half to see the whole car. What if we want to see what a picture would look like if we placed a mirror on a vertical line in the center of the width of the picture to reflect the left side (Figure 6)?

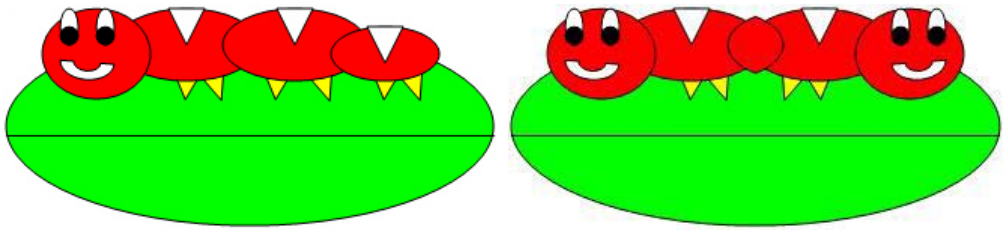

Figure 6: Original picture (left) and picture after mirroring (right)

How can we write a method to mirror a picture in this way? One way to figure out the *algorithm,* which is a description of the steps for solving a problem, is to try it on smaller and simpler data. Figure 7 shows the result of mirroring a two-dimensional array of numbers from left to right vertically.

	0	1	2	3	4
0	1	2	3	4	5
1	6	7	8	9	10
2	11	12	13	14	15

1	2	3	2	1
6	7	8	7	6
11	12	13	12	11

Figure 7: Two-Dimensional array of numbers (left) and mirrored result (right)

Can you figure out the *algorithm* for this process? Test your algorithm on different sizes of two-dimensional arrays of integers. Will it work for 2D arrays with an odd number of columns? Will it work for 2D arrays with an even number of columns?

One algorithm is to loop through all the rows and half the columns. You need to get a pixel from the left side of the picture and a pixel from the right side of the picture, which is the same distance from the right end as the left pixel is from the left end. Set the color of the right pixel to the color of the left pixel. The column number at the right end is the number of columns, also known as the width, minus one. So assuming there are at least 3 pixels in a row, the first left pixel will be at row=0, col=0 and the first right pixel will be at row=0, col=width-1. The second left pixel will be at row=0, col=1 and the corresponding right pixel will be at row=0, col=width-1-1. The third left pixel will be at row=0, col=2 and its right pixel will be at row=0, col=width-1-2. Each time the left pixel is at (current row value, current column value), the corresponding right pixel is at (current row value, width - 1 - (current column value)).

The following method implements this algorithm. Note that, because the method is not looping through all the pixels, it cannot use a nested `for-each` loop.

```
public void mirrorVertical()
{
  Pixel[][] pixels = this.getPixels2D();
  Pixel leftPixel = null;
  Pixel rightPixel = null;
  int width = pixels[0].length;
  for (int row = 0; row < pixels.length; row++)
  {
    for (int col = 0; col < width / 2; col++)
    {
      leftPixel = pixels[row][col];
      rightPixel = pixels[row][width - 1 - col];
      rightPixel.setColor(leftPixel.getColor());
    }
  }
}
```

You can test this with the `testMirrorVertical` method in `PictureTester`.

Exercises

1. Write the method `mirrorVerticalRightToLeft` that mirrors a picture around a mirror placed vertically from right to left. Hint: you can copy the body of `mirrorVertical` and only change one line in the body of the method to accomplish this. Write a class (static) test method called `testMirrorVerticalRightToLeft` in `PictureTester` to test this new method and call it in the `main` method.
2. Write the method `mirrorHorizontal` that mirrors a picture around a mirror placed horizontally at the middle of the height of the picture. Mirror from top to bottom as shown in the pictures below (Figure 8). Write a class (static) test method in `PictureTester` to test this new method and call it in the `main` method.

Figure 8: Original picture (left) and mirrored from top to bottom (right)

3. Write the method `mirrorHorizontalBotToTop` that mirrors the picture around a mirror placed horizontally from bottom to top. Hint: you can copy the body of `mirrorHorizontal` and only change one line to accomplish this. Write a class (static) test method in `PictureTester` to test this new method and call it in the `main` method.
4. Challenge — Work in groups to figure out the algorithm for the method `mirrorDiagonal` that mirrors just a square part of the picture from bottom left to top right around a mirror placed on the diagonal line (the diagonal line is the one where the row index equals the column index). This will copy the triangular area to the left and below the diagonal line as shown below. This is like folding a square piece of paper from the bottom left to the top right, painting just the bottom left triangle and then (while the paint is still wet) folding the paper up to the top right again. The paint would be copied from the bottom left to the top right as shown in the pictures below (Figure 9). Write a class (static) test method in `PictureTester` to test this new method and call it in the `main` method.

Figure 9: Original picture (left) and mirrored around the diagonal line with copying from bottom left to top right (right)

A7: Mirroring part of a picture

Sometimes you only want to mirror part of a picture. For example, Figure 10 shows a temple in Greece that is missing a part of the roof called the pediment. You can use the explorer tool to find the area that you want to mirror to produce the picture on the right. If you do this you will find that you can mirror the rows from 27 to 96 (inclusive) and the columns from 13 to 275 (inclusive). You can change the starting and ending points for the row and column values to mirror just part of the picture.

Figure 10: Greek temple before (left) and after (right) mirroring the pediment

To work with just part of a picture, change the starting and ending values for the nested `for` loops as shown in the following `mirrorTemple` method. This method also calculates the distance the current column is from the `mirrorPoint` and then adds that distance to the `mirrorPoint` to get the column to copy to.

```
public void mirrorTemple()
{
  int mirrorPoint = 276;
  Pixel leftPixel = null;
  Pixel rightPixel = null;
  int count = 0;
  Pixel[][] pixels = this.getPixels2D();

  // loop through the rows
  for (int row = 27; row < 97; row++)
  {
    // loop from 13 to just before the mirror point
    for (int col = 13; col < mirrorPoint; col++)
    {

      leftPixel = pixels[row][col];
      rightPixel = pixels[row]
                       [mirrorPoint - col + mirrorPoint];
      rightPixel.setColor(leftPixel.getColor());
    }
```

```
    }
  }
```

You can test this with the `testMirrorTemple` method in `PictureTester`.

How many times was `leftPixel = pixels[row][col];` executed? The formula for the number of times a nested loop executes is the number of times the *outer loop* executes multiplied by the number of times the *inner loop* executes. The outer loop is the one looping through the rows, because it is outside the other loop. The inner loop is the one looping through the columns, because it is inside the row loop.

How many times does the outer loop execute? The outer loop starts with `row` equal to 27 and ends when it reaches 97, so the last time through the loop `row` is 96. To calculate the number of times a loop executes, subtract the starting value from the ending value and add one. The outer loop executes 96 – 27 + 1 times, which equals 70 times. The inner loop starts with `col` equal to 13 and ends when it reaches 276, so, the last time through the loop, `col` will be 275. It executes 275 – 13 + 1 times, which equals 263 times. The total is 70 * 263, which equals 18,410.

Questions

1. How many times would the body of this nested `for` loop execute?
```
for (int row = 7; row < 17; row++)
  for (int col = 6; col < 15; col++)
```
2. How many times would the body of this nested `for` loop execute?
```
for (int row = 5; row <= 11; row++)
  for (int col = 3; col <= 18; col++)
```

Exercises

1. Check the calculation of the number of times the body of the nested loop executes by adding an integer `count` variable to the `mirrorTemple` method that starts out at 0 and increments inside the body of the loop. Print the value of `count` after the nested loop ends.
2. Write the method `mirrorArms` to mirror the arms on the snowman ("`snowman.jpg`") to make a snowman with 4 arms. Write a class (static) test method in `PictureTester` to test this new method and call it in the `main` method.
3. Write the method `mirrorGull` to mirror the seagull ("`seagull.jpg`") to the right so that there are two seagulls on the beach near each other. Write a class (static) test method in `PictureTester` to test this new method and call it in the `main` method.

A8: Creating a collage

You can copy one picture to another by copying the color from the pixels in one picture to the pixels in the other picture. To do this you will need to keep track of the row and column information for both the picture you are copying from and the picture you are copying to, as shown in the following `copy` method. The easiest way to do this is to declare and initialize both a `fromRow` and `toRow` in the outer `for` loop and increment them both at the end of the loop. A `for` loop can have more than one variable declaration and initialization and/or modification. Just separate the items with commas. Note that the inner loop has both a `fromCol` and a `toCol` declared, initialized, and incremented.

```
public void copy(Picture fromPic,
                 int startRow, int startCol)
{
  Pixel fromPixel = null;
  Pixel toPixel = null;
  Pixel[][] toPixels = this.getPixels2D();
  Pixel[][] fromPixels = fromPic.getPixels2D();
  for (int fromRow = 0, toRow = startRow;
       fromRow < fromPixels.length &&
       toRow < toPixels.length;
       fromRow++, toRow++)
  {
    for (int fromCol = 0, toCol = startCol;
         fromCol < fromPixels[0].length &&
         toCol < toPixels[0].length;
         fromCol++, toCol++)
    {
      fromPixel = fromPixels[fromRow][fromCol];
      toPixel = toPixels[toRow][toCol];
      toPixel.setColor(fromPixel.getColor());
    }
  }
}
```

You can create a collage by copying several small pictures onto a larger picture. You can do some picture manipulations like zero blue before you copy the picture as well. You can even mirror the result to get a nice artistic effect (Figure 11).

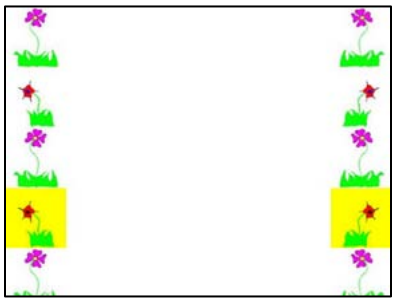

Figure 11: Collage with vertical mirror

The following method shows how to create a simple collage using the `copy` method.

```
public void createCollage()
{
  Picture flower1 = new Picture("flower1.jpg");
  Picture flower2 = new Picture("flower2.jpg");
  this.copy(flower1,0,0);
  this.copy(flower2,100,0);
  this.copy(flower1,200,0);
  Picture flowerNoBlue = new Picture(flower2);
  flowerNoBlue.zeroBlue();
  this.copy(flowerNoBlue,300,0);
  this.copy(flower1,400,0);
  this.copy(flower2,500,0);
  this.mirrorVertical();
  this.write("collage.jpg");
}
```

Notice that the `Picture` method `write` can be used to save a copy of the final collage to your disk as a JPEG picture file. You can also specify the full path name of where to write the picture ("`c:\temp\collage.jpg`"). Be sure to include the extension `(.jpg)` as well so that your computer knows the file type.

You can test this with the `testCollage` method in `PictureTester`.

Exercises

1. Create a second `copy` method that adds parameters to allow you to copy just part of the `fromPic`. You will need to add parameters that specify the start row, end row, start column, and end column to copy from. Write a class (static) test method in `PictureTester` to test this new method and call it in the `main` method.

2. Create a `myCollage` method that has at least three pictures (can be the same picture) copied three times with three different picture manipulations and at least one mirroring. Write a class (static) test method in `PictureTester` to test this new method and call it in the `main` method.

A9: Simple edge detection

Detecting edges is a common image processing problem. For example, digital cameras often feature face detection. Some robotic competitions require the robots to find a ball using a digital camera, so the robot needs to be able to "see" a ball.

One way to look for an edge in a picture is to compare the color at the current pixel with the pixel in the next column to the right. If the colors differ by more than some specified amount, this indicates that an edge has been detected and the current pixel color should be set to black. Otherwise, the current pixel is not part of an edge and its color should be set to white (Figure 12). How do you calculate the difference between two colors? The formula for the difference between two points (x_1,y_1) and (x_2,y_2) is the square root of $((x_2 - x_1)^2 + (y_2 - y_1)^2)$. The difference between two colors $(red_1,green_1,blue_1)$ and $(red_2,green_2,blue_2)$ is the square root of $((red_2 - red_1)^2 + (green_2 - green_1)^2 + (blue_2 - blue_1)^2)$. The `colorDistance` method in the `Pixel` class uses this calculation to return the difference between the current pixel color and a passed color.

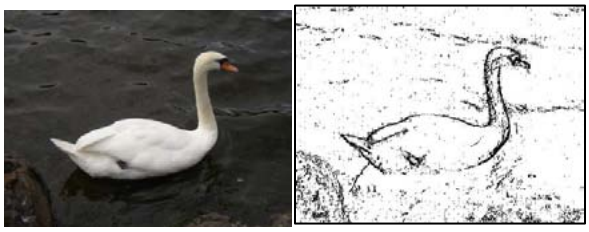

Figure 12: Original picture and after edge detection

The following method implements this simple algorithm. Notice that the nested `for` loop stops earlier than when it reaches the number of columns. That is because in the nested loop the current color is compared to the color at the pixel in the next column. If the loop continued to the last column this would cause an out-of-bounds error.

```
public void edgeDetection(int edgeDist)
{
  Pixel leftPixel = null;
  Pixel rightPixel = null;
  Pixel[][] pixels = this.getPixels2D();
  Color rightColor = null;
  for (int row = 0; row < pixels.length; row++)
  {
    for (int col = 0;
         col < pixels[0].length-1; col++)
    {
      leftPixel = pixels[row][col];
      rightPixel = pixels[row][col+1];
      rightColor = rightPixel.getColor();
      if (leftPixel.colorDistance(rightColor) >
          edgeDist)
        leftPixel.setColor(Color.BLACK);
      else
        leftPixel.setColor(Color.WHITE);
    }
  }
}
```

You can test this with the `testEdgeDetection` method in `PictureTester`.

Exercises

1. Notice that the current edge detection method works best when there are big color changes from left to right but not when the changes are from top to bottom. Add another loop that compares the current pixel with the one below and sets the current pixel color to black as well when the color distance is greater than the specified edge distance.
2. Work in groups to come up with another algorithm for edge detection.

How image processing is related to new scientific breakthroughs

Many of today's important scientific breakthroughs are being made by large, interdisciplinary collaborations of scientists working in geographically widely distributed locations, producing, collecting, and analyzing vast and complex datasets.

One of the computer scientists who works on a large interdisciplinary scientific team is Dr. Cecilia Aragon. She is an associate professor in the Department of Human Centered Design & Engineering and the eScience Institute at the University of Washington, where she directs the Scientific Collaboration and Creativity Lab. Previously, she was a computer scientist in the Computational Research Division at Lawrence Berkeley National Laboratory for six years, after earning her Ph.D. in Computer Science from UC Berkeley in 2004. She earned her B.S. in mathematics from the California Institute of Technology.

Her current research focuses on human-computer interaction (HCI) and computer-supported cooperative work (CSCW) in scientific collaborations, distributed creativity, information visualization, and the visual understanding of very large data sets. She is interested in how social media and new methods of computer-mediated communication are changing scientific practice. She has developed novel visual interfaces for collaborative exploration of very large scientific data sets, and has authored or co-authored many papers in the areas of computer-supported cooperative work, human-computer interaction, visualization, visual analytics, image processing, machine learning, cyberinfrastructure, and astrophysics.

In 2008, she received the Presidential Early Career Award for Scientists and Engineers (PECASE) for her work in collaborative data-intensive science. Her research has been recognized with four Best Paper awards since 2004, and she was named one of the Top 25 Women of 2009 by Hispanic Business Magazine. She was the architect of the Sunfall data visualization and workflow management system for the Nearby Supernova Factory, which helped advance the study of supernovae in order to reduce the statistical uncertainties on key cosmological parameters that categorize dark energy, one of the grand challenges in physics today.

Cecilia Aragon is also one of the most skilled aerobatic pilots flying today. A two-time member of the U.S. Aerobatic Team, she was a medalist at the 1993 U.S. National Championships and the 1994 World Aerobatic Championships, and was the California State Aerobatic Champion.

Glossary

1. Abstract class — You cannot create an object of an abstract class type. But, you can create an object of a subclass of an abstract class (as long as the subclass is not also an abstract class).
2. Abstract method — An abstract method cannot have a method body in the class where the method is declared to be abstract.
3. Algorithm — A step-by-step description of how to solve a problem.
4. AWT — The Abstract Windowing Toolkit. It is the package that contains the original Graphical User Interface (GUI) classes developed for Java.
5. Binary number — A binary number contains only the digits 0 and 1. Each place is a power of 2 starting with 2^0 on the right. The decimal number 6 would be 110 in binary. That would be $0 * 2^0 + 1 * 2^1 + 1 * 2^2 = 6$.
6. Bit — A **b**inary dig**it**, which means that it has a value of either 0 or 1.
7. Byte — A consecutive group of 8 bits.
8. Column-major order — An order for storing two-dimensional array data in a one-dimensional array, so that all the data for the first column is stored before all the data for the second column and so on. In a two-dimensional array represented using an array of arrays (like in Java) this means that the outer array represents the columns and the inner arrays represent the rows.
9. Digital camera — A camera that can take digital pictures.
10. Digital picture — A picture that can be stored on a computer.
11. Inheritance — In Java, a class can specify the parent class from which it inherits instance variables (object fields) and object methods. Even though instance variables may be inherited, if they are declared to be private they cannot be directly accessed using dot notation in the inheriting class. Private methods that are inherited can also not be directly called in an inheriting class.
12. Inner loop — In a nested loop (a loop inside of another loop) the loop that is inside of another loop is considered the inner loop.
13. Interface — A special type of class that can only have public abstract methods in it and/or static constants.
14. Lossy compression — Lossy compression means that the amount of data that is stored is much smaller than the available data, but the part that is not stored is data that humans would not miss.
15. Media computation — A method of teaching programming by having students write programs that manipulate media: pictures, sounds, text, movies. This approach was developed by Dr. Mark Guzdial at Georgia Tech.
16. Megapixel — One million pixels.
17. Nested loop — One loop inside of another loop.
18. Outer loop — In a nested loop (a loop inside of another loop) the loop that is outside of another loop is considered the outer loop.
19. Package — A package in Java is a group of related classes.
20. Pixel — A picture (abbreviated **pix**) **el**ement.
21. RGB model — Represents color as amounts of red, green, and blue light. It sometimes also includes alpha, which is the amount of transparency.

22. Row-major order — An order for storing two-dimensional array data in a one-dimensional array, so that all the data for the first row is stored before all the data for the second row, and so on. In a two-dimensional array represented using an array of arrays (like in Java) this means that the outer array represents the rows and the inner arrays represent the columns.
23. Subclass — A class that has inherited from another class.
24. Superclass — A class that another class has inherited from.
25. UML —Unified Modeling Language. It is a general purpose modeling language used in object-oriented software development.

References

Dann, W., Cooper, S., & Ericson, B. (2009) *Exploring Wonderland: Java Programming Using Alice and Media Computation*. Englewood, NJ: Prentice-Hall.

Guzdial, M., & Ericson B. (2006) *Introduction to Computing and Programming in Java: A Multimedia Approach*. Englewood, NJ: Prentice-Hall.

Guzdial, M., & Ericson, B. (2009) *Introduction to Computing and Programming in Python: A Multimedia Approach*. (2nd ed.). Englewood, NJ: Prentice-Hall.

Guzdial, M., & Ericson, B. (2010) *Problem Solving with Data Structures using Java: A Multimedia Approach*. Englewood, NJ: Prentice-Hall.

Quick Reference

DigitalPicture Interface

```
Pixel[][] getPixels2D()         // implemented in SimplePicture
void explore()                  // implemented in SimplePicture
boolean write(String fileName)  // implemented in SimplePicture
```

SimplePicture Class (implements Digital Picture)

```
public SimplePicture()
public SimplePicture(int width, int height)
public SimplePicture(SimplePicture copyPicture)
public SimplePicture(String fileName)
public Pixel[][] getPixels2D()
public void explore()
public boolean write(String fileName)
```

Picture Class (extends SimplePicture)

```
public Picture()
public Picture(int height, int width)
public Picture(Picture copyPicture)
public Picture(String fileName)
public Pixel[][] getPixels2D()         // from SimplePicture
public void explore()                  // from SimplePicture
public boolean write(String fileName)  // from SimplePicture
```

Pixel Class

```
public double colorDistance(Color testColor)
public double getAverage()
public int getRed()
public int getGreen()
public int getBlue()
public Color getColor()
public int getRow()
public int getCol()
public void setRed(int value)
public void setGreen(int value)
public void setBlue(int value)
public void setColor(Color newColor)
```

java.awt.Color Class

```
public Color(int r, int g, int b)
public int getRed()
public int getGreen()
public int getBlue()
```