**Interfaces Lab**

The second portion of the Picture lab will take a close look at the role of interfaces in a programming project. Take note, the purpose of an interface is to define what behaviors an object will need but not to specify how the object will behave. The implementation of the how is provided in a non-abstract class.

### Set up

Your work with the Picture Lab will pick up where you left off. Take a moment to prepare your files.

* Create a new project called 08.07 Picture Lab in the Mod08 Lessons folder.
* Copy the pixLab folder containing the classes and images folders with all your files into the newly-created project folder.
* To get started, launch BlueJ by double clicking the package.bluej file in the classes folder.

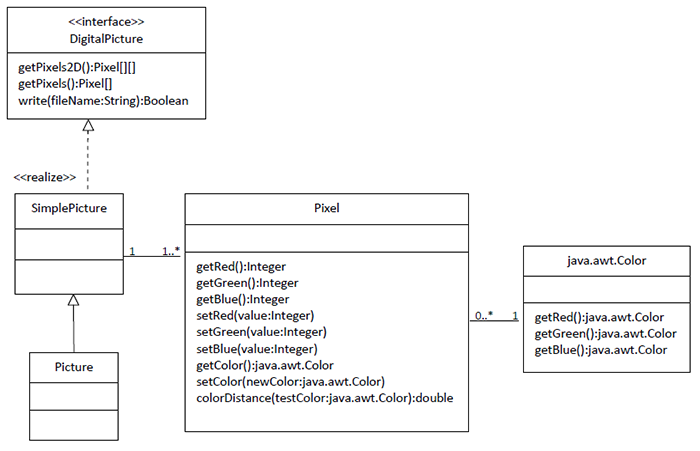
As you complete the Activities, you'll be asked to record your observation answers to questions.

* Open and save the [08.07 Picture Lab Worksheet](https://lti.flvsgl.com/flvs-cat-content/rl22dk0rtkul6je8icl8kkab7u/flvs-cat-session/apcomputersciencea_v20/module08/lesson07/docs/15_07a/08_07_picture_lab_questions.doc) to the project folder. Update the heading to include your name.
* Open and save the [Picture Lab Quick Reference](https://lti.flvsgl.com/flvs-cat-content/rl22dk0rtkul6je8icl8kkab7u/flvs-cat-session/apcomputersciencea_v20/module08/lesson07/docs/15_07a/08_07_picture_lab_quick_reference.doc) to the project folder. Use this reference as you complete the Picture Lab.

### Activity 5: 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.

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, the figure 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.



Open your saved copy of the **08.07 Worksheet** and answer the questions for Activity 5. Use the class diagram and Java programs to help you discover the answers.

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.

### Activity 5: Modifying a picture continued

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 PictureTester class. The body of the main method will call the testZeroBlue method. That method then 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 zeroBlue that calls 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 testZeroBlue method from the PictureTester class:

public static void testZeroBlue()   
{  
Picture beach = new Picture("beach.jpg");   
beach.explore();  
beach.zeroBlue();  
beach.explore();  
}

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

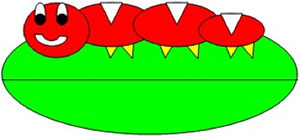
Complete the exercises using the Java files and then make note of the results in the **08.07 Worksheet**.

1. Using the zeroBlue method as a starting point, choose to write one of the following methods: keepOnlyBlue, keepOnlyRed, or keepOnlyGreen. For example, write the method keepOnlyBlue so 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.
2. 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.
3. Write the grayscale method to turn the picture into shades of gray. For each pixel, 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.
4. Challenge! Implementing this method is optional. 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.

Update the **08.07 Worksheet** with the results of the Activity 5 Exercises.

### Activity 6: 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)?

© 2017 CollegeBoard

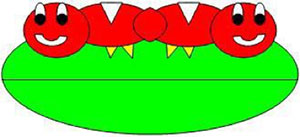
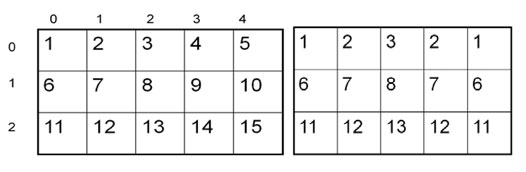
© 2017 CollegeBoard

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.

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

© 2017 CollegeBoard

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

Complete the exercises using the Java files and then make note of the results in the 08.07 Worksheet.

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.

© 2017 CollegeBoard

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

1. 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.

Update the **08.07 Worksheet** with the results of the Activity 6 Exercises.

### Activity 7: 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.

© 2017 CollegeBoard

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.

### Exercises

Complete the exercises using the Java files and then make note of the results in the 08.07 Worksheet.

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 four 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.

Update the **08.07 Worksheet** with the results of the Activity 7 Exercises.