Picture Lab

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.

Learning Objectives

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 development environment such as Dr. Java. 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.

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 stores an additional 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.

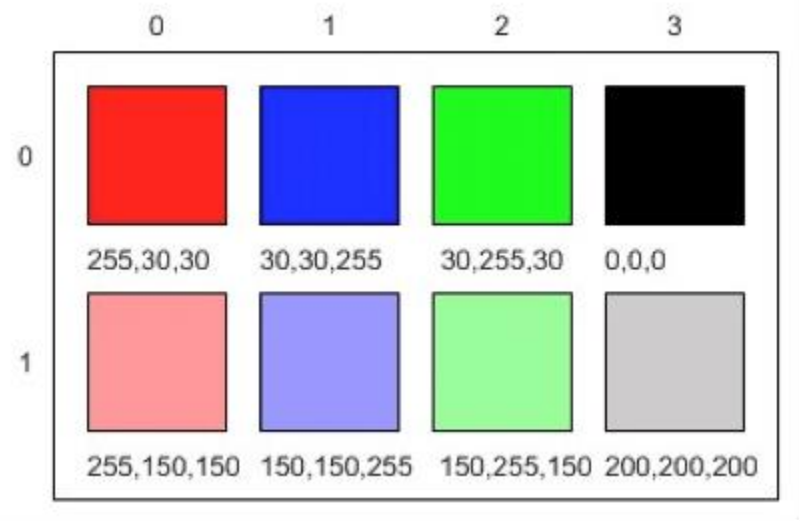


Figure 1: RGB values and the resulting colors displayed in rows and columns

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.

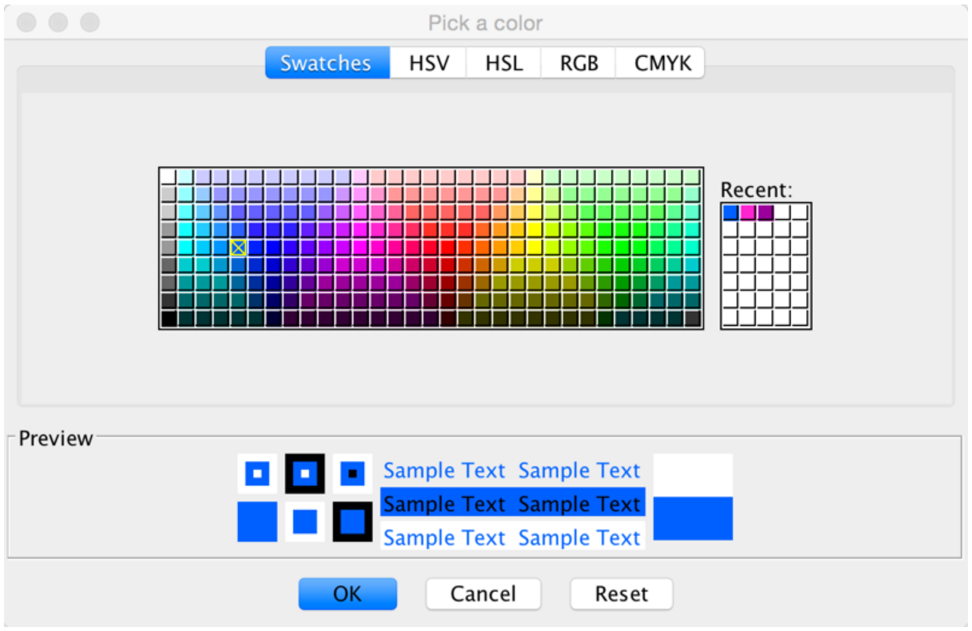


Figure 2: The Color Chooser

If you're using Dr. Java, when you click the **OK** button, the red, green, and blue values for the color you picked will be displayed in the interactions pane as shown below. TheColor 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=0,g=102,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;
```

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. Notice that clicking the left arrow on the Row indicator moves the crosshair + up and clicking the right arrow moves it down. You can also use the menu to change the zoom level to see more detail in the image.

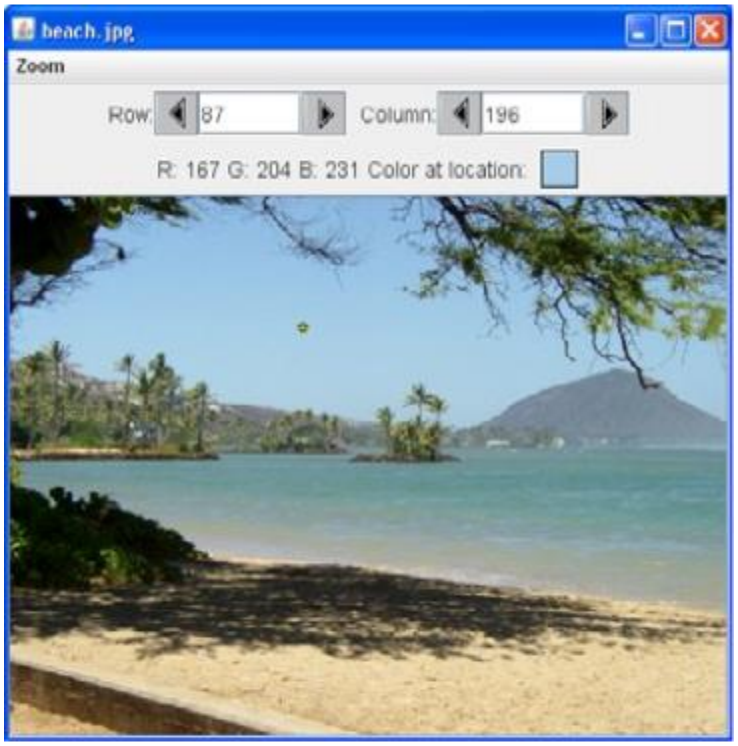


Figure 3: The Picture Explorer

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 (you can find it by scrolling all the way down to line 800 in the Java source code file).

```
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.

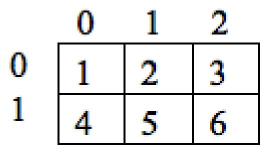
Exercises

1. Modify the main method in the PictureExplorer class to create and explore a different picture from the images folder.
2. Repeat this for a few more pictures.

Two-dimensional arrays in Java

In this activity you will work with integer data stored in a two-dimensional array. Open the **IntArrayWorker.java** class in the **classes** directory of the **pixLab** folder. Line 4 of the class shows the declaration of a global variable, **matrix**, which is a 2-D array of type **int**. As a global variable, it is visible in all of the methods of the class.

The rows and columns are numbered (indexed) and that numbering starts at 0 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

Java 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). It is common to assume that 2D arrays are row-major, which means that the outer array in Java represents the rows and the inner arrays represent the columns.

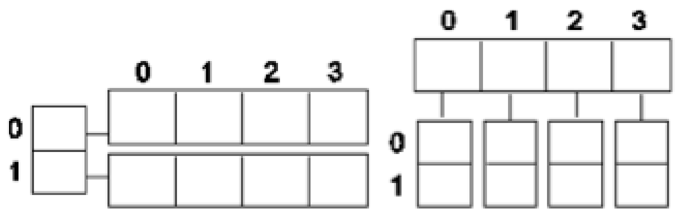


Figure 4: A row-major 2D array (left) and a column-major 2D array (right)

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 pix = 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* to loop through all the values in a 2D array.

Line 18 of the **IntArrayWorker** class begins a method that totals all the values in the 2D array of integers, **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 (this is what we visualize as a row), 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 enhanced for 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 enhanced for 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**
4. Show the revised **IntArrayWorker** source code file to your lab TA and the run of the **IntArrayWorkerTester** file.