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 `Pic` class that modify digital pictures. The `Pic` 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 6.

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, fields) 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 6 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. It simply illustrates how related classes interact with one another.

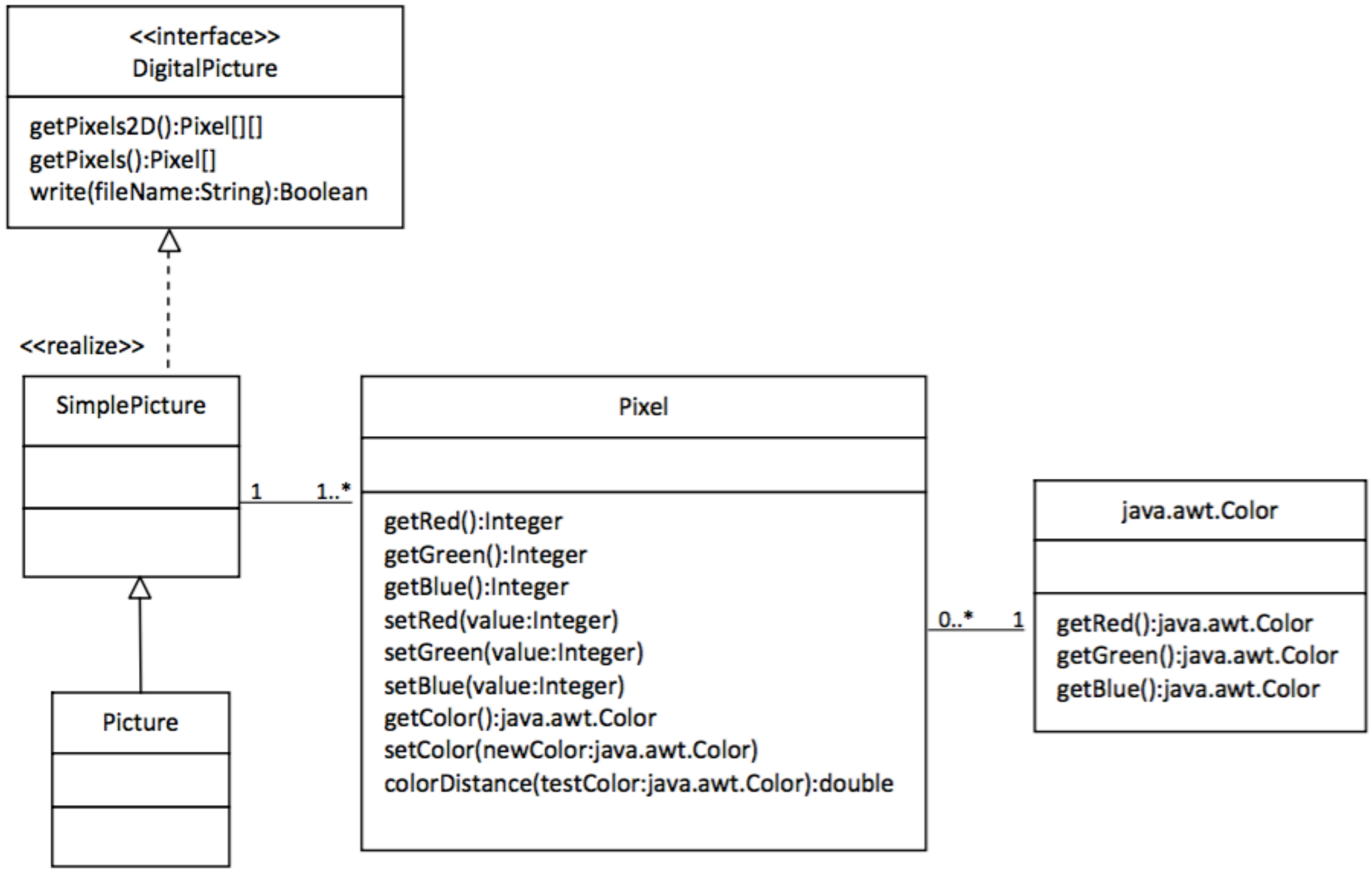

Figure 6 - A UML Class Diagram

We'll formally study UML diagrams later in the year; don't get too worried about their workings for the time being. The focus of this part of the project is to explore the manipulation of pictures. Before we do that, you'll need to remember what a picture is composed of.

1. A picture is a 2D array of _______________ objects.

2. A pixel object is composed of a __________ , __________ , and a __________ value.

3. Therefore, if we are to modify a picture you'll need to traverse through a 2D array and modify _______________ objects by altering their __________ , __________ , and __________ values.

To obtain a 2D array of pixels for any given picture, we'll utilize the `getPixels2D` method that has been written for you in the `DigitalPicture` interface (for now, think of an interface like a class). So when writing methods to manipulate a picture, you'll always need to begin by calling the `getPixels2D` method and saving its return value as a 2D array. Then you can traverse through the 2D array to manipulate the pixels.

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? Will you still see a beach?

Run the `main` method in the `Pic` class. The body of the main method will create a `Pic` 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 code below is the `main` method of the `Pic` class with some additional comments.

```
public static void main(String[] args)
{

   Picture beach = new Picture("beach.jpg"); //make an object
   beach.explore(); //open the picture explorer
   beach.zeroBlue(); //invoke zeroBlue on the picture object, beach
   beach.explore(); //open the picture explorer on the modified pic

}
```

Exercises

4. Open `PicTester.java` and run its `main` method. You should get the same results as running the `main` method in the `Pic` class. The `PicTester` class contains class (`static`) methods for testing the methods that are in the `Pic` class.

5. Uncomment the appropriate test method in the `main` method of `PicTester` to test any of the other methods in `Pic.java`. You can comment out the tests you don't want to run. You can also add new test methods to `PicTester` to test any methods you create in the `Pic` class.

The method `zeroBlue` in the `Pic` 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 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` object to zero.

6. Modify the code in the `zeroBlue` method to use a regular for loop instead of a for-each loop. The first loop should access all of the rows and the second loop should access all of the columns.

```
public void zeroBlue()
{
  Pixel[][] pixels = this.getPixels2D(); //get 2D array of pixels










}
```

7. Using the altered `zeroBlue` method from above, write the method `keepOnlyBlue` in the `Picture` class. It will keep only the blue values as is and it will set the red and green values to zero. Create a `static` (class) method to test this new method in the class `PictureTester`. Be sure to call the new test method in the `main` method in `PictureTester`.

8. 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 `static` (class) method to test this new method in the class `PictureTester`. Be sure to call the new test method in the `main` method in `PictureTester`.

9. 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 of that particular pixel. Create a `static` (class) method to test this new method in the class `PictureTester`. Be sure to call the new test method in the `main` method in `PictureTester`.

10. 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.