Learn Java

2D Arrays: Image Manipulation Project

In this project, you will be creating an application which is able to modify images as well as create new images using 2D arrays! The first section covers stretching the image horizontally, shrinking the image vertically, negating the color, applying a color filter, and inverting the image. The second section covers creating an image consisting of random pixels, placing a rectangle in the image, and using the method to randomly place many rectangles in the image.

Here is an overview about how images work in Java:

Images consist of pixels which are the individual points in the image containing some color. Each pixel has some red, green, blue, and alpha value which represents the amount of each of those colors in the pixel. The red, green, and blue values can be mixed to create all of the visible colors on your screen. The alpha value represents the transparency of the pixel (or how close the color of the pixel is to the background color of the image). A higher resolution image means that there are more pixels contained within it.

In Java, a loaded image is stored into a BufferedImage object. From this object, we can extract each pixel value and store it into a 2D array which we can manipulate. The pixel values are stored as ints because each pixel value in the BufferedImage object is represented by a hexadecimal value which contains the red, green, blue, and alpha components. The maximum value of any of the RGBA values is 255 and the minimum is 0. There are some methods provided for you in this project which handle the conversion between images and 2D arrays as well as extracting the R, G, B, and A values from a pixel. You will only need to implement methods which work with the 2D arrays.

Each of these methods is executable independently so you can select which methods you want to complete or you can complete all of them. You also may want to adjust the window sizes in the workspace to better see the code and the image. You can drag the window borders to adjust the size.

There are a lot of different parts to this project, so you may feel overwhelmed when first looking at it. Remember, the goal here is to practice using 2D arrays. We’ve included the solution code as a file named **Solution.java** in case you get stuck.

**Tasks**

64/64 Complete

Mark the tasks as complete by checking them off

**Provided Method Details**

1.

The first two sections of this project have you investigating the code that we have provided for you. You won’t need to write any code, but it’s important to understand the input and output of each of these methods.

There are four utility methods provided for you. The first method called imgToTwoD() accepts a String which can be a file path or image URL. It returns a 2D array of integers that contains every pixel from the image stored as int hexadecimal values containing the RGBA values for the pixel. Take a look at how the method works in the code editor. In the main() method, we can load image data into a 2D array of ints using the imgToTwoD() method. Note that you can use one of the provided images or load one from a URL.

We’ve provided "./apple.jpg", "./flower.jpg", and "./kitten.jpg".

2.

The second provided method is called twoDToImage(). It accepts a 2D array of integers and a String for the file name. It converts the 2D array of int pixel data into an image and saves it.

We’ll use this method to create images after we modify the 2D array of ints.

Take a look at how the method works in the code editor.

3.

The third provided method is the getRGBAFromPixel() method. This method accepts an int value representing the pixel hexadecimal value and returns a 4 element int array consisting of the R, G, B, and A values (between 0 and 255). Take a look at how the method works in the code editor. In Java, pixels are stored as hexadecimal values. This method is used to extract the color components from the hexadecimal value for the pixel.

4.

The fourth provided method is the getColorIntValFromRGBA() method. This method accepts an array of integers that represent the RGBA values and convert it into a single int value representing the pixel hexadecimal value. Take a look at how the method works in the code editor. Since pixels in Java are stored as hexadecimal values, this method converts the array of red, green, blue, and alpha values back into a hexadecimal value.

5.

In addition to these utility methods, there is a method used for extracting a 3x3 section from the top left of the image called viewImageData. This method is used to view the structure of the image data in both the raw pixel form and the extracted RGBA form. In the main() method, uncomment the call to viewImageData after the image is loaded (This should be line 14 if you haven’t edited the code yet). Save the code and take a look at the console output.

After looking at the output, feel free to re-comment this line of code to help clean up your output for the rest of the project.

**Example Method Walkthrough**

6.

To start this project, we will look at an example of how to load, manipulate, and save an image step-by-step with provided code. We will look at the trimBorders() method. This method accepts a 2D array of int pixel data and the number of pixels to trim off of the borders of the image. It returns the modified image 2D array. Start by scrolling to where this method is defined in the project code.

7.

Try looking through the method and following along with the code. Look at how a portion of the original image is copied over to the new modified image. The input of this method is a 2D array containing the input image data and the output of this method is a new 2D array representing the modified image. A detailed description of this method is in the hint for this task.

The main takeaways are that given a 2D array of ints, we make a new 2D array, and fill that array with some values from the given array. There’s some fancy math to make sure we fill the new array with the correct values, but the essential thing you should focus on is the fact that given a 2D array, we return a new 2D array.

We begin the method by checking that it is possible to trim the number of pixels provided in the input parameters. Since there are the top and bottom borders, we check that the number of rows (height) in the 2D array is greater than the trimmed amount times 2. Since there are the left and right borders, we also check that the number of columns (width) of the image is greater than 2 times the trimmed amount. If it is not valid, we print a message and return the unedited image data.

If the trim amount is valid, then we create a new 2D array of ints to store our modified image. We set the size to equal the original image’s height minus the total trimmed amount for the top and bottom borders and the original image’s width minus the trimmed amount for the left and right borders.

Next, we iterate through every position in the new 2D array. This array is smaller than the original (assuming the amount of trimmed pixels is greater than 0) so we know that we cannot copy every element from the original image or we will get an out of bounds exception. Because of this, we need to adjust the position of the original image we are copying over by the trimmed pixel amount. This means we need to add the number of pixels we are trimming to the row and column accessors.

8.

Once we have completed iterating, we return the modified image data. Now scroll back to the main() method in the code.

In the main() method, we load image data into a 2D array of ints using the imgToTwoD() method. Note that you can use the provided image or load one from a URL. After this, we can simply call our method we created by passing in the loaded image data and storing the returned modified image data into a new 2D array of ints. Finally, we can call the twoDToImage() method using the modified image data to save the new image with a provided file name.

9.

Save the code in the editor which will run the application. Now we can view this modified image by entering the image file name which we saved into the browser on the side of the workspace. The project defaults to <http://localhost:8000/apple.jpg> so change the last part of the URL to match the image name which was saved. Hit enter and the image will appear in the browser window. You can also try changing the file name and loading the image in the browser.

In this example, we have named the file trimmed\_apple.jpg so try entering <http://localhost:8000/trmmed_apple.jpg> in the workspace browser.

**Create a Negative Version of the Image**

10.

Now let’s fill in the negativeColor() method. This method will replace the color of each pixel in the image with the negative version of the pixel. This means that each color component of the pixel (R, G, and B) will be replaced with 255 minus the current value. We’ll be aiming to create an image that looks like this:



Scroll down to where this method is defined.

11.

Begin by creating a new 2D array of ints which is the same size as the input image and setting up nested for loops that iterate through every pixel in the input image. It’s probably easiest to iterate through the pixels in row-major order.

Remember we’re building our new two-dimensional array based on the input array imageTwoD. So when you’re creating your new array, there should be imageTwoD.length rows and imageTwoD.length[0] columns.

12.

Within the inner for loop, we need to get the R, G, B, and A values from each pixel. You can use the provided method getRGBAFromPixel() to accomplish this. This returns an array where each of the four elements corresponds to the RGBA values. Elements 0 to 2 represent red, green, and blue while element 3 represents alpha. Using the method will look something like: int[] rgba = getRGBAFromPixel(imageTwoD[i][j]); depending on your iterator names.

When using: int[] rgba = getRGBAFromPixel(imageTwoD[i][j]);

rgba[0] contains the red value rgba[1] contains the green value rgba[2] contains the blue value rgba[3] contains the alpha value

13.

Now that you have the array containing the RGBA values, you need three lines of code which set the first three elements to 255 minus itself.

Here is an example of what that looks like:

rgba[0] = 255 - rgba[0];  
rgba[1] = 255 - rgba[1];  
rgba[2] = 255 - rgba[2];

14.

Afterwards add another line of code which gets the int hexadecimal pixel data from the RGBA array. you can use the provided getColorIntValFromRGBA() method. You then need to store this value in the new image you created.

This will look something like: manipulatedImg[i][j] = getColorIntValFromRGBA(rgba);

15.

Once the for loops are complete, you need to return the modified image. In the main() method, load an image and pass it into the negativeColor() method. Save the modified image and view it in the browser.

**Stretch the Image Horizontally**

16.

For this task, you will be implementing the method stretchHorizontally(). This method will double the width of the provided image data. For every pixel in the original image, you will copy it and place two duplicate pixels side-by-side into the new modified image. You’ll be looking to create an image like this: 

In the code editor, scroll to this empty method.

17.

The first thing you should do in the method is to create a new 2D array of ints to hold the modified image data. The number of rows (or height) should be the same, but the number of columns (or width) should be the twice the number of columns from the original image. You can set the new 2D array to equal: new int[imageTwoD.length][imageTwoD[0].length\*2]

Here is an example: int[][] manipulatedImg = new int[imageTwoD.length][imageTwoD[0].length\*2];

18.

Next, set up nested for loops to iterate through every pixel in the input image using row-major order. To make things easier for us in the future, we can create an additional variable which will keep track of which position we are in for the modified image (since it is double the width). This variable will equal double the column index in the inner for loop. For example, if our inner loop is using columnIndex as the iterator. We can set our variable to equal: it = columnIndex\*2;

Remember the structure for nested for loops. Here is an example with the new variable included.

int it = 0;  
for(int i = 0; i < imageTwoD.length; i++) {  
    for(int j = 0; j < imageTwoD[i].length; j++) {  
        it = j\*2;  
    }  
}

19.

After we have created that additional iterator, we can now copy and duplicate each pixel from the input image. We need two lines of code. One to copy the current pixel to the modifiedImage[rowIndex][it] position and one to copy the current pixel to the modifiedImage[rowIndex][it+1] position. it is the doubled iterator from the last step and the current pixel from the input image is accessed using the outer and inner loop iterators. In the example we can assume that those are rowIndex and columnIndex, but it depends on the variable names you use.

Here is an example of what the copy statements can look like:

manipulatedImg[i][it] = imageTwoD[i][j];  
manipulatedImg[i][it+1] = imageTwoD[i][j];

20.

Once the for loops are complete, you need to return the modified image. Now in the main() method, you can load whatever image you want (either the provided image or one from a URL) as long as the resolution is not extremely high since the storage space of the workspace is limited. Pass the loaded image into the stretchHorizontally() method then click on the Save button in the workspace in order to run the code and save the image. Finally, enter the file name in the browser to view your modified image!

Here is an example of how to call those methods in the main() method:

int[][] imageData =  imgToTwoD("./apple.jpg");  
   
int[][] stretchedHImg = stretchHorizontally(imageData);  
   
twoDToImage(stretchedHImg, "./stretched\_apple.jpg");

**Shrink the Image Vertically**

21.

In order to shrink the image vertically, you will be halfing the height of the image and selecting every other pixel down each column to place in the modified image. We’re trying to create an image that looks something like this (note that this ends up looking fairly similar to the stretched image!): 

This logic should be added in the shrinkVertically() method.

22.

Begin by creating a new 2D array of ints which has half the number of rows of the input image and the same number of columns.

This will look like: int[][] manipulatedImg = new int[imageTwoD.length/2][imageTwoD[0].length];

23.

Next use nested for loops to iterate through each pixel in the input image using column-major order. Make the inner loop iterator increment by 2. We also need to end one pixel early in the vertical direction. Since we divided the height of the image by two and we could be working with an odd number of pixels in the vertical direction, we want to make sure that we do not try and access an out of bounds element.

Remember the structure of column-major order nested for loops. In this case, the inner loop iterator increases by 2 for every iteration:

for(int i = 0; i < imageTwoD[0].length; i++) {  
    for(int j = 0; j < imageTwoD.length-1; j+=2) {  
    }  
}

24.

Inside the inner for loop, place the current pixel from the input image at the row index divided by 2 and the current column index. For example: manipulatedImg[j/2][i] = imageTwoD[j][i];.

The nested loops and copy statement will look something like this:

for(int i = 0; i < imageTwoD[0].length; i++) {  
    for(int j = 0; j < imageTwoD.length; j+=2) {  
        manipulatedImg[j/2][i] = imageTwoD[j][i];  
    }  
}

25.

Once the for loops are complete, you need to return the modified image. In the main() method, load an image and pass it into the shrinkVertically() method. Save the modified image and view it in the browser.

**Invert the Image**

26.

Next, you will invert the image. You will edit the invertImage() method in order to flip the image vertically and horizontally. Your final result should look something like this:



27.

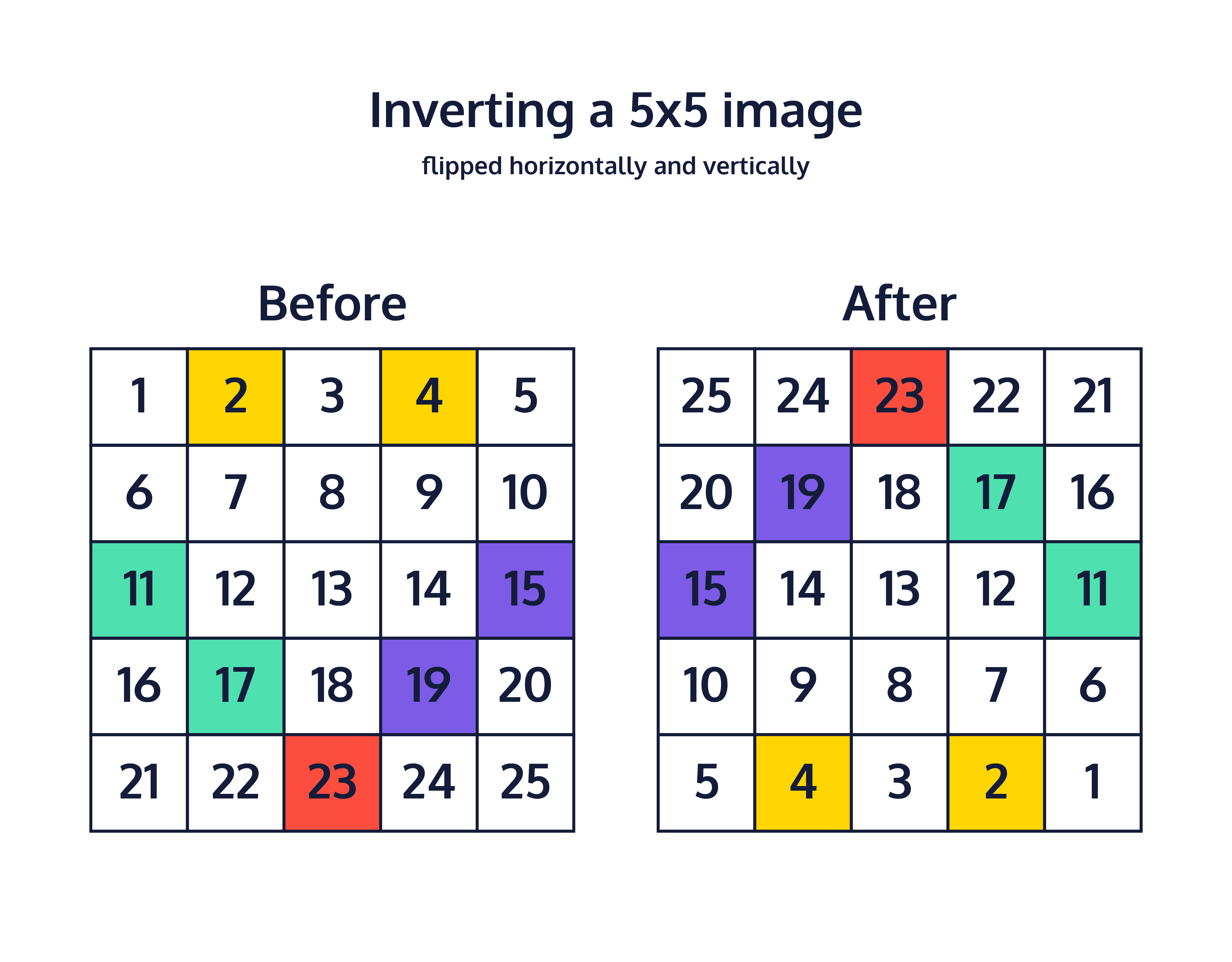
Begin by creating a new 2D array which is the same size as the input image.

28.

Next, iterate through each pixel in the input image using nested for loops.

29.

Within the inner for loop, you will copy the final row position minus the current row index and the final column position minus the column index. This is similar logic to the negativeColor() method, but taking the negative of the pixel positions instead of the color values. It will look something like this: invertedImg[i][j] = imageTwoD[(imageTwoD.length-1)-i][(imageTwoD[i].length-1)-j];. See the image for an example of what takes place.



In this image, the pixels are numbered to help show how they change position when the image is inverted.

30.

Once the for loops are complete, you need to return the modified image. In the main() method, load an image and pass it into the invertImage() method. Save the modified image and view it in the browser.

**Applying a Color Filter**

31.

For this task you will implement the colorFilter() method. This method modifies every pixel in the image by provided R, G, and B values as input parameters. You must make sure that each color value does not leave the range of 0-255.

This is an example of a filter where the filter was -75, 30, -30 (Meaning the red values were decreased by 75, the green values were increased by 30, and the blue values were decreased by 30)



32.

Begin by creating a new 2D array which is the same size as the input image.

33.

Loop through each pixel in the input image using nested for loops.

34.

For every pixel in the input image, extract the RGBA color values using the provided method getRGBAFromPixel().

Here is an example of what that could look like: int[] rgba = getRGBAFromPixel(imageTwoD[i][j]);

35.

After retrieving the array of RGBA data, store the values of each color plus the modifier value (which can be positive or negative).

Here is an example of what that can look like:

int newRed = rgba[0] + redChangeValue;  
int newGreen = rgba[1] + greenChangeValue;  
int newBlue = rgba[2] + blueChangeValue;

36.

For each of the new color values, test that it does not go outside of the range 0 to 255. If it is less than 0, set it equal to 0. If it is greater than 255, set it equal to 255.

Here is an example check for the red color:

if(newRed>255) {  
    newRed = 255;  
}  
else if(newRed<0) {  
    newRed = 0;  
}

Remember to test all three of the colors.

37.

Set the values in the RGBA array to equal the new color values which you calculated.

Here is an example of what can look like:

rgba[0] = newRed;  
rgba[1] = newGreen;  
rgba[2] = newBlue;

38.

Convert the RGBA array to a single int containing the hexadecimal pixel data using the provided method getColorIntValFromRGBA() and store it in the new image.

Here is an example of what that can look like: manipulatedImg[i][j] = getColorIntValFromRGBA(rgba);

39.

Once the for loops are complete, you need to return the modified image. In the main() method, load an image and pass it into the colorFilter() method along with modifier values for red, green, and blue. Save the modified image and view it in the browser.

**Further Challenges: Painting an Image of Random Colors**

40.

Good Job! You have finished the image processing application! You have seen how images can be converted to 2D arrays of pixels in order to modify them. You got to write methods that manipulate images. Feel free to keep adding to, and experimenting with, your image processing application!

In the next sections, we dive into how to use the Random class to paint random shapes on your images. Consider these sections to be further challenges — if you’re looking for more practice with 2D arrays, these may prove to be interesting challenges, but if you’re feeling worn out, this is a good time to take a break or move on to the next lesson!

The next method you will be implementing is the paintRandomImage() method. It will modify the image passed in by replacing every pixel with a randomly colored pixel.

41.

Begin by creating a new object of the Random class. This class will be used to generate random numbers for us: Random rand = new Random();.

42.

Next, iterate through each pixel in the provided image using nested for loops.

43.

Within the inner for loop, we need to generate a randomly colored pixel. In order to do this, we can use rand.nextInt(256). This will generate a random integer between 0 and 255. Store three of these random ints by calling rand.nextInt(256) three times.

You can store three separate values using the rand.nextInt(num) like so:

int randRed = rand.nextInt(256);  
int randGreen = rand.nextInt(256);  
int randBlue = rand.nextInt(256);

You could also try generating them all in one line with an initializer list.

44.

The three randomly generated values will need to be the first, second, and third elements in a new int array. The fourth element should be 255 since that represents the alpha value (which controls transparency).

For example: int[] rgbaValues = {randRed, randGreen, randBlue, 255};

45.

After storing those values in an int array, pass it into the provided getColorIntValFromRGBA() method and store it into the input image. Note that this is accessing the input image instead of a newly created image like in the image manipulation methods.

It will look something like this: canvas[i][j] = getColorIntValFromRGBA(rgba);

46.

Once the for loops are complete, return the modified input image.

47.

In the main() method, create a new 2D array of ints which will be our blank canvas for the painting methods. 500x500 is a good size for it. Pass the new 2D array into the paintRandomImage() method. Save the image and view it in the browser.

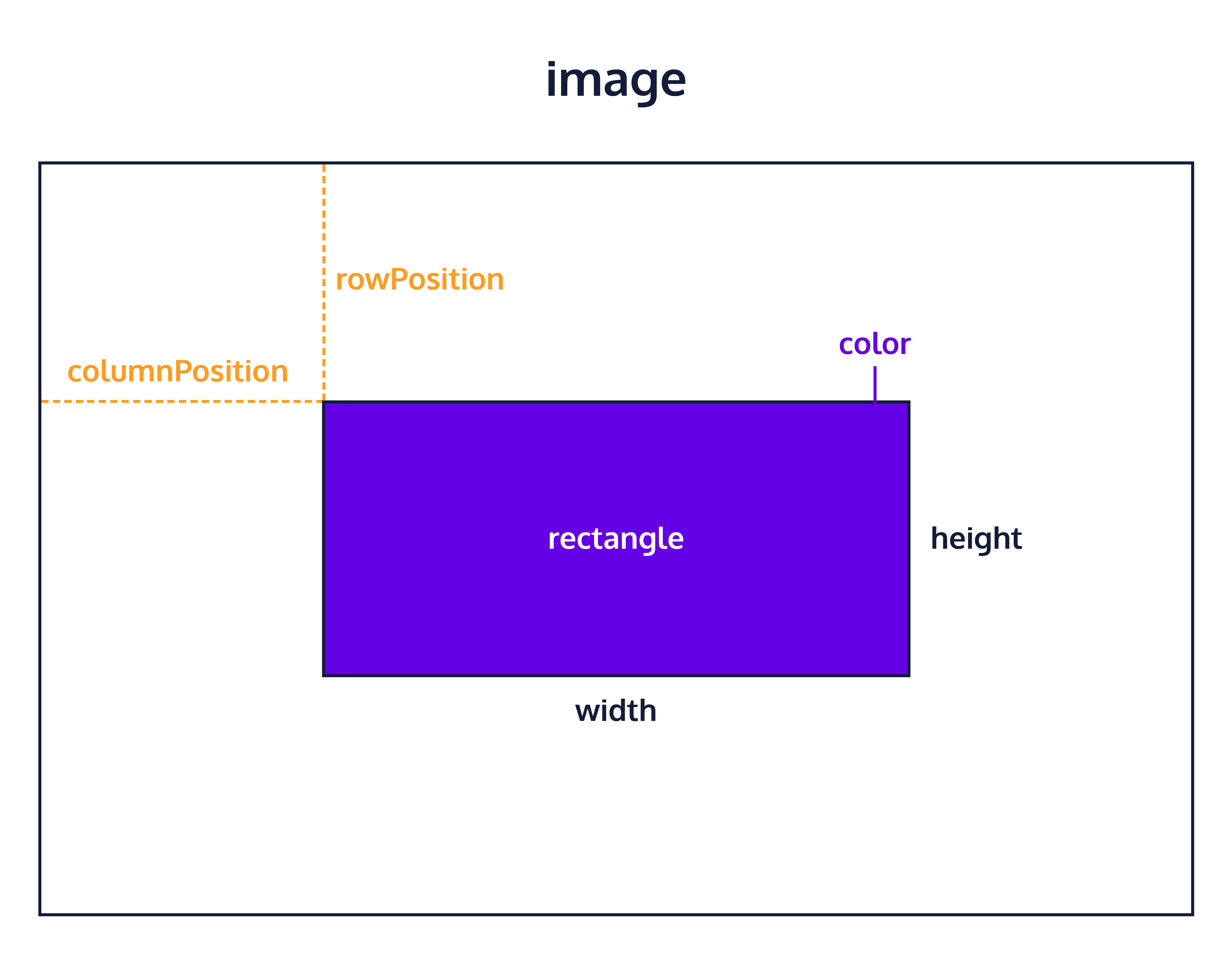
Here is an example of using this method in the main() method:

int[][] blankImg = new int[500][500];  
   
int[][] randomImg = paintRandomImage(blankImg);  
   
twoDToImage(randomImg, "./random\_img.jpg");

**Drawing a Rectangle on an Image**

48.

You will be implementing a method that draws a rectangle on an image using a provided width, height, rowPosition, columnPosition, and color. The image below shows what each of the input parameter does when creating the rectangle.



It’s useful to think of these rectangles being drawn from the top left corner.

49.

Begin by iterating through every pixel in the input image using nested for loops.

50.

For every pixel, check if the current row index is greater than or equal to the provided rowPosition and also that the index is less than or equal to the rowPosition plus the width: i>=rowPosition && i<=rowPosition + width.

51.

Additionally, check if the current column index is greater than or equal to the provided colPosition and also that the index is less than or equal to the colPosition plus the height: j>=colPosition && j<=colPosition + height.

52.

If both of the previous tests are true, then paint the pixel the provided color since the pixel position lies within the rectangle shape.

The last three steps can look like this in your code:

if(i>=rowPosition && i<=rowPosition + width) {  
    if(j>=colPosition && j<=colPosition + height) {  
        canvas[i][j] = color;  
    }  
}

53.

Once the iterations are complete, return the modified input image.

54.

In the main() method, pass an image into the paintRectangle() method and save the image. Enter in the desired values for your rectangle. You can create a color using an array such as: int[] rgba = {255, 255, 0, 255}; and passing it into the getColorIntValFromRGBA() method. Save the image and view it in the browser.

Here is an example of what that could look like in your main() method:

int[] rgba = {255, 255, 0, 255};  
   
int[][] rectangleImg = paintRectangle(randomImg, 200, 200, 100, 100, getColorIntValFromRGBA(rgba));  
   
twoDToImage(rectangleImg, "./rectangle.jpg");

**Create Abstract Geometric Art Utilizing the paintRectangle Method**

55.

Finally, you will implement the generateRectangles() method. This will use the paintRectangles() method you created in order to paint a piece of art. It will generate randomly positioned, sized, and colored rectangles based on the provided number. The input parameter, numRectangles will determine how many randomly generated rectangles will be placed in the image.

56.

Firstly, create an object of the Random class to use.

This is done using: Random rand = new Random();

57.

Next, iterate for the number of rectangles provided using a for loop.

For example:

for(int i = 0; i<numRectangles; i++) {  
}

58.

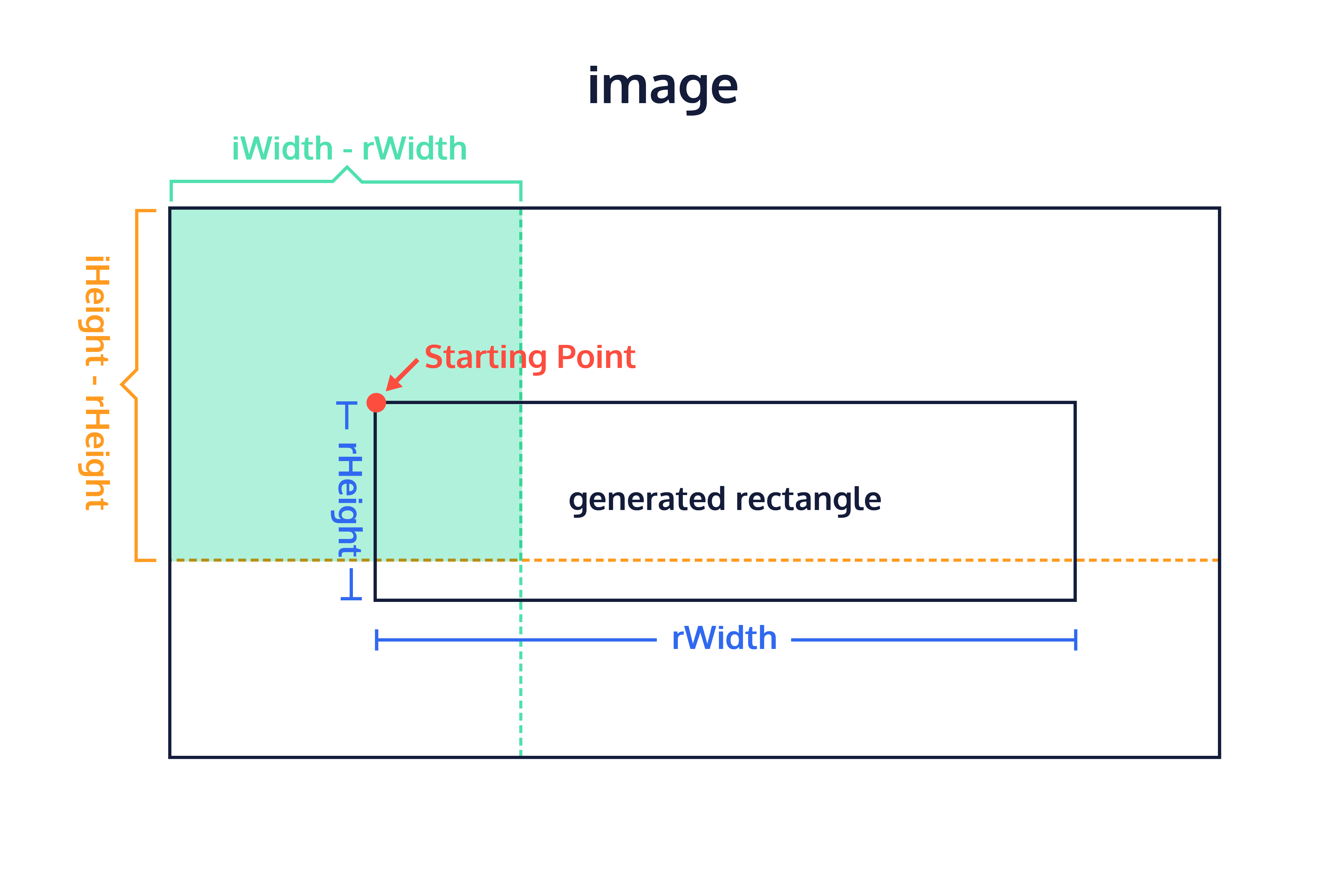
Generate and store two random integers for the width and height of the rectangle. For width, pass in the number of columns in the 2D array to the rand.nextInt(num) method. For height, pass in the number of rows into the rand.nextInt(num) method.

This will look something like:

int randomWidth = rand.nextInt(canvas[0].length);  
int randomHeight = rand.nextInt(canvas.length);

59.

Generate and store two random integers for the row position and column position. For row position, pass in the number of rows in the image minus the randomly generated height from the previous step. This will ensure that the rectangle does not get created outside of the image. For the column position generate a random integer using the number of columns in the image minus the randomly generated width from the last step. The following image shows why we perform this calculation.



In this image, the starting position of the rectangle can only be inside the shaded box. If we were to try to put the top left corner outside the shaded box, part of the rectangle would be drawn outside of the image, and we would get an error.

This will look something like:

int randomRowPos = rand.nextInt(canvas.length-randomHeight);  
int randomColPos = rand.nextInt(canvas[0].length-randomWidth);

60.

Next, create a random color to paint the rectangle. You can do this in a similar way to the method used in paintRandomImage() make sure the three random color values are between 0 and 255. Store them in an array with 255 for the A value and convert it to the int value using getColorIntValFromRGBA.

You can use initializer lists in order to easily create your custom color:

int[] rgba = {rand.nextInt(256), rand.nextInt(256), rand.nextInt(256), 255};  
int randomColor = getColorIntValFromRGBA(rgba);

61.

Finally, call the paintRectangle() method using all of the values and the input image and overwrite the input image.

Here is what that looks like: canvas = paintRectangle(canvas, randomWidth, randomHeight, randomRowPos, randomColPos, randomColor);

62.

After the iterations are complete, return the modified input image.

63.

In the main() method, pass an image into the generateRectangles() method and a number of rectangles to generate, 1000 rectangles is a good number to use. Save the image and view the image in the browser.

**Conclusion**

64.

Wow, you made it through the bonus work. Pat yourself on the back, this was quite a difficult project! Reflect on all the different ways you used loops and 2D arrays to create these images.

Some additional challenges you can do are: adding more parameters to the image manipulation methods (such as a factor to stretch or shrink by), painting with different shapes (triangles, circles), and more complex image filters (saturation).