1-Point Penalty

- (w) Extraneous code that causes a side effect or prevents earning points in the rubric (e.g., information written to output)
- (x) Local variables used but none declared
- (y) Destruction of persistent data (e.g., changing value referenced by parameter)
- (z) Void method or constructor that returns a value

No Penalty

- Extraneous code that causes no side effect
- Extraneous code that is unreachable and would not have earned points in rubric
- Spelling/case discrepancies where there is no ambiguity*
- Local variable not declared, provided that other variables are declared in some part
- private qualifier on local variable
- Missing public qualifier on class or constructor header
- Keyword used as an identifier
- Common mathematical symbols used for operators (x • ÷ ≥ < > =)
- [] vs. () vs. <>
- = instead of == and vice versa
- Array/collection element access confusion ([] vs. get for r-values)
- Array/collection element modification confusion ([] vs. set for l-values)
- length/size confusion for array, String, and ArrayList, with or without ()
- Extraneous [] when referencing entire array
- [i, j] instead of [i][j]
- Extraneous size in array declaration, (e.g., int[size] nums = new int[size];)
- Missing ; provided that line breaks and indentation clearly convey intent
- Missing () where indentation clearly conveys intent and () are used elsewhere
- Missing () on parameter-less method or constructor invocations
- Missing () around if/while conditions
- Use of local variable outside declared scope (must be within same method body)
- Failure to cast object retrieved from nongeneric collection

* Spelling and case discrepancies for identifiers fall under the “No Penalty” category only if the correction can be unambiguously inferred from context; for example, “ArrayList” instead of “ArrayList”. As a counterexample, note that if the code declares “Bug bug;” and then uses “Bug.move()” instead of “bug.move()”, the context does not allow for the reader to assume the object instead of the class.
Question 4: GrayImage

Part (a)  countWhitePixels  4 points

Intent: Return the number of white pixels in the image

+1 Accesses all entries in pixelValues (no bounds errors)
+1 Compares an entry of array with WHITE or with 255 in context of iteration
+1 Initializes and increments a counter
+1 Returns correct count of number of white pixels in pixelValues

Part (b)  processImage  5 points

Intent: Process elements of pixelValues and apply specified formula

+1 Accesses all necessary elements in at least one row or one column of pixelValues
+1 Accesses all necessary elements of pixelValues (no bounds errors)
+1 Decrements element at index [a][b] by the original value found in element at index [a+2][b+2]
+1 Modifies all and only appropriate elements of pixelValues (changes must not affect last two rows and columns)
+1 Assigns BLACK or 0 to elements of pixelValues that would otherwise have a value less than BLACK (negative value)

Question-Specific Penalties

-1 (z) Attempts to return a value from processImage
Question 4: GrayImage

Part (a):
```java
public int countWhitePixels() {
    int whitePixelCount = 0;
    for (int[] row : this.pixelValues) {
        for (int pv : row) {
            if (pv == this.WHITE) {
                whitePixelCount++;
            }
        }
    }
    return whitePixelCount;
}
```

Part (a): Alternative solution
```java
public int countWhitePixels() {
    int whitePixelCount = 0;
    for (int row = 0; row < pixelValues.length; row++) {
        for (int col = 0; col < pixelValues[0].length; col++) {
            if (pixelValues[row][col] == this.WHITE) {
                whitePixelCount++;
            }
        }
    }
    return whitePixelCount;
}
```

Part (b):
```java
public void processImage() {
    for (int row = 0; row < this.pixelValues.length-2; row++) {
        for (int col = 0; col < this.pixelValues[0].length-2; col++) {
            this.pixelValues[row][col] -= this.pixelValues[row+2][col+2];
            if (this.pixelValues[row][col] < BLACK) {
                this.pixelValues[row][col] = BLACK;
            }
        }
    }
}
```

Part (b): Alternative solution
```java
public void processImage() {
    for (int row = 0; row < this.pixelValues.length; row++) {
        for (int col = 0; col < this.pixelValues[0].length; col++) {
            if (row + 2 < this.pixelValues.length &&
                col + 2 < this.pixelValues[row].length) {
                this.pixelValues[row][col] -= this.pixelValues[row+2][col+2];
                if (this.pixelValues[row][col] < BLACK) {
                    this.pixelValues[row][col] = BLACK;
                }
            }
        }
    }
}
```

These canonical solutions serve an expository role, depicting general approaches to solution. Each reflects only one instance from the infinite set of valid solutions. The solutions are presented in a coding style chosen to enhance readability and facilitate understanding.
Complete method countWhitePixels below.

```java
/** @return the total number of white pixels in this image.
 * Postcondition: this image has not been changed.
 */
public int countWhitePixels() {
    int numWhite = 0;
    for (int i = 0; i < pixelValues.length; i++) {
        for (int j = 0; j < pixelValues[i].length; j++) {
            if (pixelValues[i][j] == WHITE) {
                numWhite++;
            }
        }
    }
    return numWhite;
}
```

Part (b) begins on page 22.
Complete method `processImage` below.

```java
/**
 * Processes this image in row-major order and decreases the value of each pixel at
 * position (row, col) by the value of the pixel at position (row + 2, col + 2) if it exists.
 * Resulting values that would be less than BLACK are replaced by BLACK.
 * Pixels for which there is no pixel at position (row + 2, col + 2) are unchanged.
 */
public void processImage() {

    // Indexes to length - 2 to avoid having to worry about bounds.
    for (int i = 0; i < pixelValues.length - 2; i++) {
        for (int j = 0; j < pixelValues[i].length - 2; j++) {
            pixelValue[i][j] -= pixelValue[i + 2][j + 2];
            if (pixelValue[i][j] < BLACK) {
                pixelValue[i][j] = BLACK;
            }
        }
    }
}
```
Complete method `countWhitePixels` below.

```java
/** *
 * @return  the total number of white pixels in this image.
 * Postcondition: this image has not been changed.
 */
public int countWhitePixels()
{
    int count = 0;
    for (int x = 0; x < pixelValues.length; x++)
    {
        for (int y = 0; y < pixelValues[x].length; y++)
        {
            if (pixelValues[x][y] == 255)
                count++;
        }
    }
    return count;
}
```

Part (b) begins on page 22.
Complete method `processImage` below.

```java
/**
 * Processes this image in row-major order and decreases the value of each pixel at
 * position (row, col) by the value of the pixel at position (row + 2, col + 2) if it exists.
 * Resulting values that would be less than `BLACK` are replaced by `BLACK`.
 * Pixels for which there is no pixel at position (row + 2, col + 2) are unchanged.
 */
public void processImage()
{
    for (int x = 0; x < pixelValues.length - 2; x++)
    {
        for (int y = 0; y < pixelValues.length - 2; y++)
        {
            pixelValues[x][y] = pixelValues[x][y] - pixelValues[x+2][y+2];
        }
    }
    return pixelValues;
}
```
Complete method countWhitePixels below.

```java
/**
 * @return the total number of white pixels in this image.
 * Postcondition: this image has not been changed.
 */
public int countWhitePixels()
{
    int count = 0;
    for (K = 0; K < 4; K++)
    {
        for (J = 0; J < 5; J++)
        {
            if (pixelValues[K][J] == 255)
            {
                count++;
            }
        }
    }
    return (count);
}
```

Part (b) begins on page 22.
Complete method `processImage` below.

```java
/**
 * Processes this image in row-major order and decreases the value of each pixel at
 * position (row, col) by the value of the pixel at position (row + 2, col + 2) if it exists.
 * Resulting values that would be less than BLACK are replaced by BLACK.
 * Pixels for which there is no pixel at position (row + 2, col + 2) are unchanged.
 */
public void processImage()
    for (k = 0; k < pixelValues.length; k++)
        for (j = 0; j < pixelValues.length; j++)
            if (pixelValues[k+2][j+2] != null)
                pixelValues[k][j] = pixelValues[k+2][j+2]
            else
                pixelValues[k][j]
```
**Overview**

This question addressed the use of a rectangular two-dimensional array of primitive values, accessing and modifying array elements, comparing array elements to other values, managing a counter, and understanding the distinction between void methods and those that return a value. Students were asked to implement two methods of the GrayImage class. In part (a) students were required to implement the method countWhitePixels, which determined and returned the number of array elements that contained the value 255, which was also represented by the symbolic constant WHITE. This could be accomplished by traversing the entire array, using nested loops, comparing every element with WHITE, and incrementing a counter. In part (b) students were required to implement the method processImage, which modified array elements on the basis of the values of elements at other indices and assigned the value 0 (symbolic constant BLACK) to elements whose resulting values would be less than 0. Here again a viable approach to the solution involved the use of nested loops, taking care to avoid out-of-bounds indices. (Although the question specified processing in row-major order, that aspect of the solution was not assessed because correct results were achievable via other means.)

**Sample: 4A**  
**Score: 9**

In part (a) the student correctly implements the countWhitePixels method. The student accesses all entries of the array pixelValues with no bounds errors by correctly determining the numbers of rows and columns in the array. The student also compares each entry of the array to the value of WHITE (255) in the context of iteration. The counter used to keep track of the number of white pixels in pixelValues is correctly declared, initialized, and incremented. The value of this counter is correctly returned. The student earned all 4 points in part (a).

In part (b) the student correctly implements the processImage method. The student accesses all necessary elements in at least one row or one column of pixelValues. In fact, the student correctly accesses all necessary elements in the entire array. The student decrements the value of the element at index [a][b] by the original value found in the element at index [a+2][b+2]. The student modifies all required elements, as well as avoids modifying inappropriate elements (those that fall into the last two rows or last two columns). When the result of the modification is less than BLACK, the student assigns the value of BLACK to that particular element. The student earned all 5 points in part (b).

**Sample: 4B**  
**Score: 6**

In part (a) the student correctly implements the countWhitePixels method. The student accesses all entries of the array pixelValues with no bounds errors by correctly determining the numbers of rows and columns in the array. The student also compares each entry of the array to the value of WHITE (255) in the context of iteration. The counter used to keep track of the number of white pixels in pixelValues is correctly declared, initialized, and incremented. The value of this counter is correctly returned. The student earned all 4 points in part (a).

In part (b) the student does not correctly implement the processImage method. The student attempts to determine the number of columns in the array using pixelValues.length, which represents the number of rows in the array. This causes the array to be treated as a square array, meaning the student can access at least one row or one column of the array but may miss some entries in the array. The student decrements the value of the element at index [a][b] by the original value of the element at
index \[a+2 \][b+2\]. The student modifies all required elements, as well as avoids modifying inappropriate elements (those that fall into the last two rows or last two columns). When the result of the modification is less than BLACK, the student does not assign the value of BLACK to that particular element. The method `processImage` is a void method, but the array `pixelValues` is incorrectly returned at the end of the method. Although the student earned 3 of the 5 points in this part, the incorrect return caused the student to lose 1 additional point under the General Scoring Guidelines. In total, the student earned 2 points in part (b).

Sample: 4C  
Score: 2

In part (a) the student does not correctly implement the `countWhitePixels` method and does not access all entries in the array. This is because the student accesses the array as a fixed size array based on the example given in the question. The student does compare some entries of an array to the value of WHITE (255) in the context of iteration. The counter used to keep track of the number of white pixels in `pixelValues` is incremented. But the counter is not declared and initialized before use. The value of this counter is not correctly calculated, because a local variable with the same name is re-declared within the loop, thus preventing the counter from being incremented correctly. The student earned 1 point in part (a).

In part (b) the student does not correctly implement the `processImage` method. The student attempts to determine the number of columns in the array using `pixelValues.length`, which represents the number of rows in the array. This causes the array to be treated as a square array, meaning the student can access at least one row or one column of the array but may miss some entries in the array. The student does not perform any subtraction in the code. The student modifies all required elements but also modifies some elements that should not be modified (those that fall into the last two rows or last two columns), resulting in an IndexOutOfBoundsException. The student does not assign the value of BLACK to an element when appropriate. The student earned 1 point in part (b).