**(M4) Write a Java program (Optical Barcode Readers) (4 hrs)**

**Optical Barcode Readers and Writers**

Even though you learned to do multi-file projects, please continue to submit ***only one file*** with a full, working program and run.

**Understand the Problem**

This assignment combines ***2D arrays***, ***interfaces*** (including ***Cloneable)***, and a very active industrial application, ***optical scanning and pattern recognition***.  This will add some elegant code to your resume.

Let's look at a ***Datamatrix***.  What do you notice about it?  The more recognizable feature, perhaps, is the solid black on the***left and bottom***.  This is called the ***Closed Limitation Line***.   Look a little more closely and you'll notice that the ***right and top*** consist of an alternating black and white pattern, so that the odd numbered pixels on the far right (and top) are black, while the even numbered pixels are white.  This is called the***Open Borderline***.  The ***Closed Limitation Line*** and ***Open Borderline*** help the algorithms because they:

* Help situate the code in a standard position.
* Determine the minimum size of each pixel.
* Determine the height and width of the ***Datamatrix*** (which as you can see from the examples can vary).

An example of our datamatrix is this:

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*  
\*                                 \*  
\*\*\*\*\* \*\* \* \*\*\*\* \*\*\*\*\*\* \*\* \*\*\*\* \*\*    
\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*      \*\*\*\*\*\*\*\*\*\*\*\*\*  
\*\*  \*  \*        \*  \*   \*        \*    
\* \*\*  \*     \*\*    \* \*   \* \*\*\*\*   \*\*  
\*\*         \*\*\*\*   \* \*\* \*\* \*\*\*   \*\*   
\*   \*  \*   \*\*\*  \*       \*  \*\*\*   \*\*  
\*  \*\* \*\* \* \*\*\*  \*\*\*  \*  \*  \*\*\* \*     
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

As you can see, I am using \*s to indicate black pixels, and blanks to indicate white ones.  Also, the bottom and left edge are "solid" black (you have to use your imagination a little) and the top and right are alternating black-white pixels.  In order to read this message we need to start at the far left and look at each column.  We throw away the left column which is part of the ***Closed Limitation Line*** and also throw away the bottom and top.  The first column is, then:

128s

\* 64s

32s

\* 16s

Eights

\* Fours

Twos

Ones

This tells us exactly how do read the code:  It is 4 + 16 + 64 = 84 = 'T'  (capital T).  We then move to the next column, and decode that one the same way.

This is pretty much punched paper tape from the 1970s (which was a great start on this concept).

In summary, we are going to use the solid ***Closed Limitation Line***and the ***Open Borderline*** simply to identify the size and extent of the code.  After that, we'll look at (or print if we are creating the label) each column from left-to-right, converting the ASCII codes into a sequence of 8 characters, or visa-versa.

It turns out, the structure of the classes, objects and algorithms we'll need to write will be adequate as a framework for the more complex and real ***Datamatrix***, even though we will only program a faint suggestion of the real deal.

Here is a complete text and the code to go with it, for you to use as you write your program:

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*  
\*                                 \*  
\*\*\*\*\* \*\* \* \*\*\*\* \*\*\*\*\*\* \*\* \*\*\*\* \*\*    
\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*      \*\*\*\*\*\*\*\*\*\*\*\*\*  
\*\*  \*  \*        \*  \*   \*        \*    
\* \*\*  \*     \*\*    \* \*   \* \*\*\*\*   \*\*  
\*\*         \*\*\*\*   \* \*\* \*\* \*\*\*   \*\*   
\*   \*  \*   \*\*\*  \*       \*  \*\*\*   \*\*  
\*  \*\* \*\* \* \*\*\*  \*\*\*  \*  \*  \*\*\* \*     
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

This is a good SAMPLE to look at.

There are three components (other than **main()**) to this assignment.

* **interface BarcodeIO**.  An ***Interface*** (which you define) called **BarcodeIO** that defines the I/O and basic methods of any barcode class which might implement it.
* **class BarcodeImage implements Cloneable**.  An object of this **BarcodeImage** class will be one of the main member-objects of the class that comes next.  **BarcodeImage** will describe the 2D dot-matrix pattern, or "image".  It will contain some methods for storing, modifying and retrieving the data in a 2D image.  *The interpretation of the data is not part of this class.* Its job is only to manage the optical data.  It will implement **Cloneable** interface because it contains deep data.
* **class DataMatrix implements BarcodeIO**.  The class that will contain both a **BarcodeImage** member object and a **text String** member that represents the message encoded in the embedded image.  This class has all the fun.  This is not a true *Datamatrix* because, for one thing, there is no Reed-Solomon error correction.

**Phase 1: BarcodeIO**

Define an interface, **BarcodeIO**, that contains these method signatures.  Any class that ***implements*** **BarcodeIO** is expected to store some version of an image and some version of the text associated with that image.

* **public boolean scan(BarcodeImage bc);**
* **public boolean readText(String text);**
* **public boolean generateImageFromText();**
* **public boolean translateImageToText();**
* **public void displayTextToConsole()**;
* **public void displayImageToConsole();**

Now, as I said, this is an ***interface***.  So you should be able to do this part of the assignment in less than 60 seconds.  I'll time you.  Go.

Here are the descriptions of what these will do when implemented in the DataMatrix class, however, descriptions in an interface don't pack any punch in practice.

* **public boolean scan( BarcodeImage bc )** - accepts some image, represented as a **BarcodeImage** object to be described below, and stores a copy of this image.  Depending on the sophistication of the implementing class, the internally stored image might be an exact clone of the parameter, or a refined, cleaned and processed image.  Technically, there is no requirement that an implementing class use a **BarcodeImage** object internally, although we will do so.  For the basic**DataMatrix** option, it will be an exact clone.  Also, no ***translation*** is done here - i.e., any text string that might be part of an implementing class is not touched, updated or defined during the scan.
* **public boolean readText( String text )** - accepts a text string to be eventually encoded in an image. No translation is done here - i.e., any **BarcodeImage** that might be part of an implementing class is not touched, updated or defined during the reading of the text.
* **public boolean generateImageFromText()** - Not technically an I/O operation, this method looks at the internal text stored in the implementing class and produces a companion **BarcodeImage**, internally (or an image in whatever format the implementing class uses).  After this is called, we expect the implementing object to contain a fully-defined image and text that are in agreement with each other.
* **public boolean translateImageToText()** - Not technically an I/O operation, this method looks at the internal image stored in the implementing class, and produces a companion text string, internally.  After this is called, we expect the implementing object to contain a fully defined image and text that are in agreement with each other.
* **public void displayTextToConsole()** - prints out the text string to the console.
* **void displayImageToConsole()**- prints out the image to the console.  In our implementation, we will do this in the form of a dot-matrix of blanks and asterisks, e.g.,

**Phase 2: BarcodeImage**

This class will realize all the essential data and methods associated with a 2D pattern, thought of conceptually as an image of a square or rectangular bar code.  Here are the essential ingredients.  This class has very little "smarts" in it, except for the parameterized constructor.  It mostly just stores and retrieves 2D data.

Remember: **BarcodeImage** implements **Cloneable**.

***DATA***

* **public static final int MAX\_HEIGHT = 30;    public static final int MAX\_WIDTH = 65**;   The ***exact*** internal dimensions of 2D data.
* **private boolean[][] image\_data**This is where to store your image.  If the incoming data is too large, instantiate memory anyway, but leave it blank (white). This data will be **false** for elements that are white, and **true** for elements that are black.

***METHODS***

* Constructors.  Two minimum, but you could have others:
  + **Default Constructor** -  instantiates a 2D array (**MAX\_HEIGHT** x **MAX\_WIDTH**) and stuffs it all with blanks (**false**).
  + **BarcodeImage(String[] str\_data)** -takes a 1D array of **Strings** and converts it to the internal 2D array of **booleans**.
  + HINT  -  This constructor is a little tricky because the incoming image is not the necessarily same size as the internal matrix.  So, you have to pack it into the lower-left corner, causing a bit of stress if you don't like 2D counting.  This is good 2D array exercise.
* ***Accessor*** and ***mutator*** for each bit in the image:  **boolean getPixel(int row, int col)** and **boolean setPixel(int row, int col, boolean value)**;   For the **getPixel()**, you can use the return value for both the actual data and also the error condition -- so that we don't "create a scene" if there is an error; we just return **false**.
* *Optional -* A ***private*** utility method is highly recommended, but not required:  **checkSize(String[] data)**  It does the job of checking the incoming data for every conceivable size or null error.  Smaller is okay.  Bigger or null is not.
* *Optional -* A **displayToConsole()** method that is useful for debugging this class, but not very useful for the assignment at large.
* A **clone()** method that overrides the method of that name in **Cloneable** interface.

**Phase 3: DataMatrix**

This class is a pseudo *Datamatrix* data structure, not a true *Datamatrix*, because it does not contain any error correction or encoding.  However, it does have the 2D array format and a left and bottom BLACK "spine" as well as an alternating right and top **BLACK-WHITE** pattern as seen in the image.

Remember: **DataMatrix** implements **BarcodeIO**.

***DATA***

* **public static final char BLACK\_CHAR = '\*';**
* **public static final char WHITE\_CHAR = ' '**;
* **private BarcodeImage image** - a single internal copy of any image *scanned-in* OR *passed-into the constructor* OR *created by BarcodeIO's****generateImageFromText().***
* **private String text** - a single internal copy of any text *read-in* OR *passed-into the constructor* OR *created by****BarcodeIO's translateImageToText().***
* **private int actualWidth** and **actualHeight** - two **ints** that are typically ***less than*** **BarcodeImage.MAX\_WIDTH** and **BarcodeImage.MAX\_HEIGHT** which represent the actual portion of the **BarcodeImage** that has the real *signal*.  This is dependent on the data in the image, and can change as the image changes through mutators.  It can be computed from the "spine" of the image.

***METHODS***

* ***Constructors***.  Three minimum, but you could have more:
  + **Default Constructor** -  constructs an empty, but non-null, image and text value.  The initial image should be all white, however, **actualWidth** and **actualHeight** should start at **0**, so it won't really matter what's in this default image, in practice.  The **text** can be set to blank, "", or something like "undefined".
  + **DataMatrix(BarcodeImage image)** - sets the **image** but leaves the **text** at its default value.  Call **scan()** and avoid duplication of code here.
  + **DataMatrix(String text)** - sets the **text** but leaves the **image** at its default value. Call **readText()** and avoid duplication of code here.
* **readText(String text)** - a ***mutator*** for text.  Like the constructor;  in fact it is called *by* the constructor.
* **scan(BarcodeImage image)** - a ***mutator*** for image.  Like the constructor;  in fact it is called *by* the constructor.  Besides calling the **clone()** method of the **BarcodeImage** class, this method will do a couple of things including set the**actualWidth** and **actualHeight** and call **cleanImage().**  Because **scan()** calls **clone()**, it should deal with the **CloneNotSupportedException** by ***embeddingthe clone() call within a try/catch block***.  Don't attempt to hand-off the exception using a "throws" clause in the function header since that will not be compatible with the underlying **BarcodeIO** interface.  The **catches(...)** clause can have an empty body that does nothing.
* ***Accessors*** for **actualWidth** and **actualHeight** but ***no mutators!*** (why?)
* **private int computeSignalWidth()** and **private int computeSignalHeight()** - Assuming that the image is correctly situated in the lower-left corner of the larger **boolean** array, these methods use the "spine" of the array (left and bottom BLACK) to determine the actual size.
* **Implementation of all BarcodeIO methods.**

Private method:

* **private void cleanImage()** - This private method will make no assumption about the placement of the "signal" within a passed-in **BarcodeImage**.  In other words, the in-coming **BarcodeImage** may not be lower-left justified.  Here's an example of  the placement of such an un-standardized image:

--------------------------------------------------------------

| |

| |

| \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* |

| \* \* |

| \*\*\*\* \*\*\* \*\* \*\*\*\*\* \*\*\*\* \*\*\*\*\*\*\*\*\* |

| \* \*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\* |

| \*\* \* \* \* \* \* \* \* \* |

| \*\*\* \* \* \* \*\* \* \*\* |

| \* \*\* \* \* \* \* \* \* \*\* \* \*\*\* \*\*\* |

| \* \* \*\* \*\*\*\*\* \* \*\* \*\* |

| \*\*\*\* \* \* \* \* \*\* \*\* \* \*\* \* \* \* |

| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |

| |

| |

| |

| |

| |

| |

--------------------------------------------------------------

The **cleanImage()** method would be called from within **scan()** and would ***move the signal to the lower-left of the larger 2D array***.  And, since **scan()** is called by the constructor, that implies that the image gets adjusted upon construction.  This kind of standardization represents the many other image processing tasks that would be implemented in the **scan()** method.  Error correction would be done at this point in a real class design.  After **cleanImage()** the internal representation would look like this:

--------------------------------------------------------------

| |

| |

| |

| |

| |

| |

| |

|\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* |

|\* \* |

|\*\*\*\* \*\*\* \*\* \*\*\*\*\* \*\*\*\* \*\*\*\*\*\*\*\*\* |

|\* \*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\* |

|\*\* \* \* \* \* \* \* \* \* |

|\*\*\* \* \* \* \*\* \* \*\* |

|\* \*\* \* \* \* \* \* \* \*\* \* \*\*\* \*\*\* |

|\* \* \*\* \*\*\*\*\* \* \*\* \*\* |

|\*\*\*\* \* \* \* \* \*\* \*\* \* \*\* \* \* \* |

|\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |

--------------------------------------------------------------

This is not hard to do, and it represents the kind of manipulation you would be expected to do in a real job, but it does require some helper methods (which are optional, meaning you can create your own if you don't like the sound of these):  **private void moveImageToLowerLeft()**, **private void shiftImageDown(int offset),private void shiftImageLeft(int offset).**

Other considerations for **DataMatrix**

* **displayImageToConsole()** should display only the relevant portion of the image, clipping the excess blank/white from the top and right.  Also, show a border as in:

------------------------------------

|\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* |

|\* \*|

|\*\*\*\* \* \*\*\*\*\* \*\*\*\* \*\*\*\* \*\*\*\*\*\*\*\* |

|\* \*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*|

|\* \* \*\* \* \* \* \* \* \*\* |

|\* \* \* \* \*\* \* \* \* \*\*\*\*|

|\* \* \* \*\* \* \* \* \* \*\* \* |

|\*\* \* \*\*\* \*\*\*\*\* \*\* \* \* \*\*|

|\*\*\*\* \* \*\*\*\* \*\* \* \* \* \* \* |

|\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*|

* *Recommendation -* The methods **generateImageFromText()** and **translateImageToText(),** are the tricky parts, and it will help if you have some methods like the following to break up the work:  **private char readCharFromCol(int col)** and**private boolean WriteCharToCol(int col, int code)**.  While you don't have to use these exact methods, you must not turn in huge methods **generateImageFromText()** and **translateImageToText()** that are not broken down to smaller ones.
* *Optional* - **public void displayRawImage()** can be implemented to show the full image data including the blank top and right.  It is a useful debugging tool.
* *Optional* - **private void clearImage()** - a nice utility that sets the image to white =  false.

You may need to digest what you are doing and why you are doing it at each juncture.  If you just focus on each individual method, writing and testing as you go, you will be fine.  You and your team may need to spend time over multiple sittings to do this.  I am here for questions, as usual.

Here is a sample **main()** to run.  You can add to it, but include these bar codes for decoding:

   public static void main(String[] args)  
   {  
      String[] sImageIn =  
      {  
         "                                               ",  
         "                                               ",  
         "                                               ",  
         "     \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* ",  
         "     \*                                       \* ",  
         "     \*\*\*\*\*\* \*\*\*\* \*\*\*\*\*\* \*\*\*\*\*\*\* \*\* \*\*\* \*\*\*\*\*   ",  
         "     \*     \*    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ",  
         "     \* \*\*    \* \*        \*\*  \*    \* \* \*   \*     ",  
         "     \*   \*    \*  \*\*\*\*\*    \*   \* \*   \*  \*\*  \*\*\* ",  
         "     \*  \*\*     \* \*\*\* \*\*   \*\*  \*    \*\*  \*\*\*  \*  ",  
         "     \*\*\*  \* \*\*   \*\*  \*   \*\*\*\*    \*  \*  \*\* \* \*\* ",  
         "     \*\*\*\*\*  \*\*\*  \*  \* \*   \*\* \*\* \*\*  \*   \* \*    ",  
         "     \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ",    
         "                                               ",  
         "                                               ",  
         "                                               "  
  
      };

           
        
      String[] sImageIn\_2 =  
      {  
            "                                          ",  
            "                                          ",  
            "\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*     ",  
            "\*                                    \*    ",  
            "\*\*\*\* \*\*\* \*\*   \*\*\*\*\* \*\*\*\*   \*\*\*\*\*\*\*\*\*      ",  
            "\* \*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*    ",  
            "\*\* \*      \*    \*  \* \* \*         \* \*       ",  
            "\*\*\*   \*  \*           \* \*\*    \*      \*\*    ",  
            "\* \*\* \* \*  \*   \* \* \* \*\*  \*   \*\*\*   \*\*\*     ",  
            "\* \*           \*\*    \*\*\*\*\*  \*   \*\*   \*\*    ",  
            "\*\*\*\*  \*  \* \*  \* \*\*  \*\* \*   \*\* \*  \* \*      ",  
            "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*    ",  
            "                                          ",  
            "                                          ",  
            "                                          ",  
            "                                          "  
  
      };

      BarcodeImage bc = new BarcodeImage(sImageIn);  
      DataMatrix dm = new DataMatrix(bc);  
       
      // First secret message  
      dm.translateImageToText();  
      dm.displayTextToConsole();  
      dm.displayImageToConsole();  
        
      // second secret message  
      bc = new BarcodeImage(sImageIn\_2);  
      dm.scan(bc);  
      dm.translateImageToText();  
      dm.displayTextToConsole();  
      dm.displayImageToConsole();  
        
      // create your own message  
      dm.readText("What a great resume builder this is!");  
      dm.generateImageFromText();  
      dm.displayTextToConsole();  
      dm.displayImageToConsole();  
   }