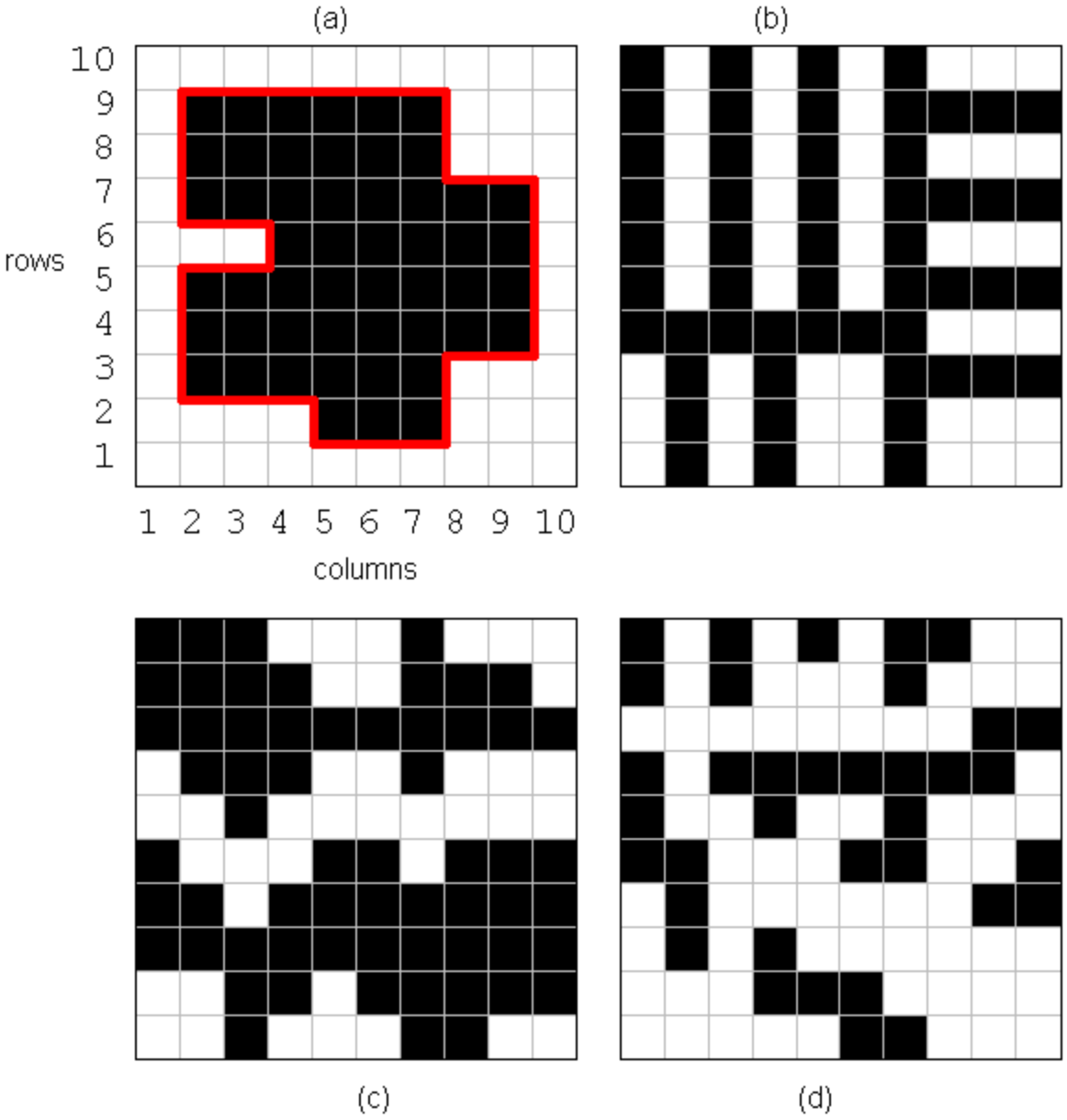
# COMP1406 - Assignment #7

(Due: **Mon. Apr 9th @ 12 noon**)

In this assignment, you will practice using recursion.

Assume that we have an assembly line that can take a picture of a machine part which moves along a conveyor belt. The picture (or image) is represented as a 2D grid of pixels which are either black or white. The pixels can be accessed by specifying the row and column of the pixel where rows and columns are specified by an integer value.

The machine examines the images and attempts to determine whether or not the parts are *broken*. A **broken** part will appear as a set of black pixels which are not all connected together (i.e., there is a separation between one or more sets of black pixel groups. Here are some examples of four possible images. Note that (c) and (d) represent images of ***broken*** parts. The red border represents the perimeter of a part composed of black pixels.



# (1) The **PartImage** class

Start with the class defined below that represents one of these images using a 2-dimensional array of **booleans**.

```
import javafx.geometry.Point2D;

public class PartImage {
    private boolean[][]     pixels;
    private boolean[][]     visited;
    private int             rows;
    private int             cols;

    public PartImage(int r, int c) {
        rows = r;
        cols = c;
        visited = new boolean[r][c];
        pixels = new boolean[r][c];
    }

     public PartImage(int rw, int cl, byte[][] data) {
         this(rw,cl);
         for (int r=0; r<10; r++) {
             for (int c=0; c<10; c++) {
                 if (data[r][c] == 1)
                     pixels[r][c] = true;
                 else
                     pixels[r][c]= false;
             }
         }
     }

     public int getRows() { return rows; }
     public int getCols() { return cols; }
     public boolean getPixel(int r, int c) { return pixels[r][c]; }

     // You will re-write the 5 methods below
     public void print() {}
     public Point2D findStart() { return null; }
     public int partSize() { return 0; }
     private void expandFrom(int r, int c) { }
     private int perimeterOf(int r, int c) { return 0; }


     public boolean isBroken(){
         Point2D p = findStart();
         expandFrom((int)p.getX(), (int)p.getY());
         return (partSize() != 0);
     }

     public int perimeter() {
         Point2D p = findStart();
         return perimeterOf((int)p.getX(), (int)p.getY());
     }
}
```

Add the following **static** methods to this class, which represents the examples shown earlier:

```
public static PartImage exampleA() {
    byte[][] pix = {{0,0,0,0,0,0,0,0,0,0},
                    {0,1,1,1,1,1,1,0,0,0},
                    {0,1,1,1,1,1,1,0,0,0},
                    {0,1,1,1,1,1,1,1,1,0},
                    {0,0,0,1,1,1,1,1,1,0},
                    {0,1,1,1,1,1,1,1,1,0},
                    {0,1,1,1,1,1,1,1,1,0},
                    {0,1,1,1,1,1,1,0,0,0},
                    {0,0,0,0,1,1,1,0,0,0},
                    {0,0,0,0,0,0,0,0,0,0}};
    return new PartImage(10,10, pix);
}
public static PartImage exampleB() {
    byte[][] pix = {{1,0,1,0,1,0,1,0,0,0},
                    {1,0,1,0,1,0,1,1,1,1},
                    {1,0,1,0,1,0,1,0,0,0},
                    {1,0,1,0,1,0,1,1,1,1},
                    {1,0,1,0,1,0,1,0,0,0},
                    {1,0,1,0,1,0,1,1,1,1},
                    {1,1,1,1,1,1,1,0,0,0},
                    {0,1,0,1,0,0,1,1,1,1},
                    {0,1,0,1,0,0,1,0,0,0},
                    {0,1,0,1,0,0,1,0,0,0}};
    return new PartImage(10,10, pix);
}
public static PartImage exampleC() {
    byte[][] pix = {{1,1,1,0,0,0,1,0,0,0},
                    {1,1,1,1,0,0,1,1,1,0},
                    {1,1,1,1,1,1,1,1,1,1},
                    {0,1,1,1,0,0,1,0,0,0},
                    {0,0,1,0,0,0,0,0,0,0},
                    {1,0,0,0,1,1,0,1,1,1},
                    {1,1,0,1,1,1,1,1,1,1},
                    {1,1,1,1,1,1,1,1,1,1},
                    {0,0,1,1,0,1,1,1,1,1},
                    {0,0,1,0,0,0,1,1,0,0}};
    return new PartImage(10,10, pix);
}
public static PartImage exampleD() {
    byte[][] pix = {{1,0,1,0,1,0,1,1,0,0},
                    {1,0,1,0,0,0,1,0,0,0},
                    {0,0,0,0,0,0,0,0,1,1},
                    {1,0,1,1,1,1,1,1,1,0},
                    {1,0,0,1,0,0,1,0,0,0},
                    {1,1,0,0,0,1,1,0,0,1},
                    {0,1,0,0,0,0,0,0,1,1},
                    {0,1,0,1,0,0,0,0,0,0},
                    {0,0,0,1,1,1,0,0,0,0},
                    {0,0,0,0,0,1,1,0,0,0}};
    return new PartImage(10,10, pix);
}
```

Now complete the following methods:

- Complete the **print()** method that will print out the image in the console window which may print something like what is shown below for the image in picture (a) above.

```
----------
-******---
-******---
-********-
---******-
-********-
-********-
-******---
----***---
----------
```

- Complete the **findStart()** method that returns the location (row/column) of any black pixel in the image. It uses the **Point2D** class (from the **javafx.geometry** package) as the return value (you will have to look up this class in the Java API documentation to see how to use it, if you have not used it before). If no black pixel is found, the method should return **null**.

- Complete the **partSize()** method so that it that returns an **int** representing the number of black pixels in the image. This can be done using a double FOR loop.

- Complete the method called **expandFrom(int r, int c)** that recursively examines the image by traversing its pixels examining (i.e., "visiting") black pixels in all directions that are connected to a starting point black pixel (specified by the parameters). The idea is that this method will determine all black pixels that are connected to the starting pixel. *(hints: when a white pixel is encountered during the recursion, this is an indication that you don't need to keep recursively checking in that direction. Also, you can mark any encountered black pixels as white so that you don't need to process them again)*. This method **MUST be done recursively without loops, otherwise you will receive 0 marks for this part**.

- Complete the method called **perimeterOf(int r, int c)** which should return an integer representing the perimeter of the part which starts at the given (r,c) point. The method **MUST be done recursively without loops, otherwise you will receive 0 marks for this part**. For example, in the image (A) shown above, the perimeter is **36** (which is the red border length, not the number of black cells on the border) and in image (B) it is **106**. As for the other two images, it depends where you started expanding from. *(hints: You will need to mark pixels as being visited (perhaps use another 2D array). Make sure not to re-visit these pixels. Also, make sure to handle borders correctly. A black pixel on a the top/left border corner, for example, will add 2 to the perimeter count for the top and the left.)*

Here is the test code to test everything:

```
public class PartImageTester {
    public static void main(String[] args) {
        PartImage piA = PartImage.exampleA();
        PartImage piB = PartImage.exampleB();
        PartImage piC = PartImage.exampleC();
        PartImage piD = PartImage.exampleD();
```

```
        System.out.println("\nPart A:");
        System.out.println("  starts at: " + PartImage.exampleA().findStart());
        System.out.println("  size:      " + PartImage.exampleA().partSize());
        System.out.println("  broken:    " + PartImage.exampleA().isBroken());
        System.out.println("  perimeter: " + PartImage.exampleA().perimeter()+ "\n");
        piA.print();
        System.out.println("\nPart B:");
        System.out.println("  starts at: " + PartImage.exampleB().findStart());
        System.out.println("  size:      " + PartImage.exampleB().partSize());
        System.out.println("  broken:    " + PartImage.exampleB().isBroken());
        System.out.println("  perimeter: " + PartImage.exampleB().perimeter()+ "\n");
        piB.print();
        System.out.println("\nPart C:");
        System.out.println("  starts at: " + PartImage.exampleC().findStart());
        System.out.println("  size:      " + PartImage.exampleC().partSize());
        System.out.println("  broken:    " + PartImage.exampleC().isBroken());
        System.out.println("  perimeter: " + PartImage.exampleC().perimeter()+ "\n");
        piC.print();
        System.out.println("\nPart D:");
        System.out.println("  starts at: " + PartImage.exampleD().findStart());
        System.out.println("  size:      " + PartImage.exampleD().partSize());
        System.out.println("  broken:    " + PartImage.exampleD().isBroken());
        System.out.println("  perimeter: " + PartImage.exampleD().perimeter()+ "\n");
        piD.print();

    }
}
```

And here is the expected output:

```
Part A:
  starts at: Point2D [x = 1.0, y = 1.0]
  size:      51
  broken:    false
  perimeter: 36

----------
-*******---
-*******---
-********-
---******-
-********-
-********-
-*******---
----***---
----------


Part B:
  starts at: Point2D [x = 0.0, y = 0.0]
  size:      52
  broken:    false
  perimeter: 106

*-*-*-*---
*-*-*-****
*-*-*-*---
*-*-*-****
*-*-*-*---
*-*-*-****
*******---
-*-*--****
-*-*--*---
-*-*--*---
```

```
Part C:
  starts at: Point2D [x = 0.0, y = 0.0]
  size:      61
  broken:    true
  perimeter: 36

***---*---
****--***-
**********
-***--*---
--*-------
*---**-***
**-*******
**********
--**-*****
--*---**--


Part D:
  starts at: Point2D [x = 0.0, y = 0.0]
  size:      36
  broken:    true
  perimeter: 6

*-*-*-**--
*-*---*---
--------**
*-*******-
*--*--*---
**---**--*
-*------**
-*-*------
---***----
-----**---
```

---

IMPORTANT SUBMISSION INSTRUCTIONS:

Submit your ZIPPED **IntelliJ project file** as you did during the first tutorial for assignment 0.

- YOU WILL LOSE MARKS IF YOU ATTEMPT TO USE ANY OTHER COMPRESSION FORMATS SUCH AS **.RAR**, **.ARC, .TGZ**, **.JAR**, **.PKG**, **.PZIP**.
- If your internet connection at home is down or does not work, we will not accept this as a reason for handing in an assignment late ... so make sure to submit the assignment WELL BEFORE it is due !
- You WILL lose marks on this assignment if any of your files are missing. So, make sure that you hand in the correct files and version of your assignment. You will also lose marks if your code is not written neatly with proper indentation. See examples in the notes for proper style.

---