# Data Structures and Algorithms II
# COMP 2631 (Winter 2015)

# Assignment 2
# Bitmap Hacker GUI

**Due Date:** Thursday, February 19, before 11:55pm (submit to Moodle)

**Overview**

The goal of this assignment is to build a graphical user interface (GUI) for the Bitmap hacker you wrote in Lab 2. You will write two classes: `Bitmap` and `BitmapGUI`. The `Bitmap` class will contain all the functionality of your Lab 2 code, and the `BitmapGUI` class will deal with GUI details. Full descriptions of these classes are below.

## The `Bitmap` class

Unlike the approach in Lab 2, the `Bitmap` class will not contain `static` methods because we need to be able to create `Bitmap` objects. A `Bitmap` object will contain all the important information extracted from a BMP file.

***Instance variables***
Your `Bitmap` class should have the following instance variables (along with appropriate class constants):

```
    private int dataOffset;
    private int width;
    private int height;
    private int numPaddingBytes;
    private int[] header;
    private Color[][] pixels;
```

These are mostly self-explanatory. The `header` instance variable will contain an exact copy of the header information in the file, which you will need later to create a new BMP file. Clearly the size of the `header` array will be the same as the value you will store in `dataOffset`. The 2-D array `pixels` will contain one `Color` object for each pixel in the image. Note that you are no longer using the `Pixel` class you wrote for Lab 2, since the existing `Color` class provides the basic functionality of `Pixel`.

***Constructor***
Include a single `Bitmap` constructor with one `File` parameter. This constructor can simply call the `readBitmap` method described below.

***Methods***
At minimum, provide methods with the following signatures:

```
    public int getWidth()
    public int getHeight()
    public void readBitmap(File f)
    public void writeBitmap(File f)
    public void flip()
    public void blur()
    public void enhanceColor(???)
    public BufferedImage getImage()
```

The methods `getWidth` and `getHeight` are standard accessors. You can provide other accessors if you want, but do not write any mutator methods for the instance variables listed above. The method `readBitmap` should open the indicated (BMP) file for reading, and then extract information from the file needed to "fill" the instance variables. Both `readBitmap` and the `Bitmap` constructor should throw any `IOException` that occurs (as should `writeBitmap`). Handle such an `IOException` gracefully inside the calling method in `BitmapGUI`, for example, by informing and prompting the user.

The `writeBitmap` method is essentially the reverse of `readBitmap`. Use the information stored in the instance variables to write out the bitmap to the specified file, again throwing any `IOException` that occurs.

The methods `flip`, `blur`, and `enhanceColor` should only modify `pixels` (nothing else). The parameter(s) for `enhanceColor` may depend on your approach in Lab 2.

The method `getImage` returns the pixel data stored in `pixels` as a `BufferedImage`, which is useful for displaying the image in a GUI. When creating a `BufferedImage` to return, use the constructor that takes three arguments: width, height, and image type – for image type, use the `BufferedImage` class constant `TYPE_INT_RGB`. A `BufferedImage` is essentially a 2-D array of pixel data, but each pixel is stored in a "packed" form, i.e., all the color components are packed into the bytes of a single `int`. To fill the `BufferedImage` before returning it, take each `Color` object in `pixels`, convert it to packed form using the `getRGB` method in the `Color` class, and assign it to the corresponding entry in the `BufferedImage` using the `setRGB` method (the one with three parameters – the third parameter should be passed the packed `int`). Note that you are essentially writing the `static` method `read(File)` that is in the `ImageIO` class (but don't use this method, or any method like it).

## The `BitmapGUI` class

The `BitmapGUI` class implements the GUI by constructing and managing all GUI components, and handling all user events. As usual, make `BitmapGUI` a subclass of `JFrame`. Your GUI should contain, *at minimum*, the following:

- A *File* menu containing menu items *Open*, *Save*, *Save As*, *Close*, and *Exit.*
- A button for each image manipulation operation.
- Something to indicate when the current image has been modified (but not yet saved).

You have a lot of freedom in the layout of your GUI (don't feel confined by the example GUI that I demonstrated in class). Also, a certain amount of redundancy is fine – for example, you can provide both buttons and menu items that accomplish the same task, as long as you include the components listed above.

***Instance variables***

You should have an instance variable for each GUI component. The type of one of the components should be a class that extends `JPanel`; this is the "canvas" on which you will draw the `BufferedImage` that gets passed back by the `getImage` method in the `Bitmap` class (more on this below). In addition, include the following three instance variables:

```
private File mostRecentInputFile;
private Bitmap bmp;
private boolean modified;
```

Each time the user selects an input file, construct a `File` object and reference it with `mostRecentInputFile`. Figure out how to make a `JFileChooser` take advantage of this information so that the user will automatically be placed back in the most recent folder used to select an input file when selecting another file (except the first time, of course, when there is no "most recent file").

Also, each time the user selects an input file, construct a new `Bitmap` object referenced by `bmp`, and set `modified` to `false`.

Provide one constructor with no parameters. This constructor will set up the initial GUI. To keep your code tidy, it would be a good idea to have the constructor call other methods that do most of the "heavy lifting." For example, you might have methods such as:

```
addMenus()
addButtons()
addOtherComponents()
addMenuItemEventListeners()
addButtonEventListeners()
enableDisable()
```

These are also mostly self-explanatory. Inside `enableDisable` (or whatever you name it), call the method `setEnabled` on each relevant GUI component, passing it `true` or `false` as appropriate so that when there is no current image, certain menu items, buttons, etc., are inactive ("grayed out").

Finally, include a `main` method in which you simply construct a `BitmapGUI` object.

***More on event listeners***

In this section I will briefly discuss a couple of the event listeners you need to write; this will give you a sense of the details you should think about. In general, use anonymous inner classes for event listeners.

1. In the `actionPerformed` method associated with the *Open* file menu item, pop up a `JFileChooser` that defaults to the most recent folder used to select an input file. If the user clicks **Cancel** before completing, do nothing. If the user selects a file and clicks **Open**, *and* if the current image (if any) has *not* been modified, then update all relevant instance variables (in particular, `mostRecentInputFile`, `bmp`, and `modified`), update all relevant GUI components (e.g., you might have a text label that shows the full path name of the current input file), call your version of `enableDisable`, then call `pack` and `repaint`. If the current image *has* been modified, pop up a dialog box telling the user that the current image is unsaved, asking if s/he wants to save it before proceeding. (You can use `JOptionPane.showConfirmDialog`.)

2. In the `actionPerformed` method associated with the button that allows the user to flip the image, call the `flip` method on `bmp`, set `modified` to `true`, update any GUI components (e.g., anything that indicates that the image has been modified), and call `repaint`. If you set up the canvas as described below, the flipped image will automatically be redrawn.

3. When the user closes an image, set `bmp` to `null` and set `modified` to `false`. The `null` or non-`null` status of `bmp` can be useful at various places in your code.

***Custom*** `JPanel`
As stated above, it is a good idea for the GUI component that acts as your canvas to be a subclass of `JPanel`. Make this a (non-anonymous) inner class. The constructor can simply call the superclass constructor with no arguments. You need to include two methods:

```
public Dimension getPreferredSize()
public void paintComponent(Graphics g)
```

When `repaint` is called at various places in your code, the windowing system will redraw each of the components. It turns out that if you want a component to redraw in a certain way, it is better to override the `paintComponent` method for that component, rather than the `paint` method (as we have often done). In addition, the windowing system will "ask" each component what its size is. For a canvas who size will not change (think of the graphical labs in COMP 2611), it suffices to set the size once using the `setPreferredSize` method, but if the size of a component is dynamic (here the size of the canvas will depend on the size of the BMP image), it is better to override the `getPreferredSize` method so that the windowing system can query the component about its size at any time. Inside `getPreferredSize`, construct and return an object of class `Dimension`. Pass to the `Dimension` constructor the width and height of the canvas – if there is no open image, use default values stored in class constants (the idea is to have a blank canvas with a fixed size when there is no image); if there *is* an open image, use the width and height of that image (this is where the accessor methods in `Bitmap` come in handy).

Inside `paintComponent`, the first thing you should do is call `super.paintComponent(g)`. Then either draw a blank rectangle with the default empty canvas size (e.g., in white or light gray), or draw the actual image stored in `bmp`. To draw the image, call the `drawImage` method on `g`. You can use the version of `drawImage` that takes four arguments: the first is the image itself (the `BufferedImage` returned by `getImage` in `Bitmap` will be accepted here), the next two are the coordinates of the top-left corner of the canvas (use $(0, 0)$), and `null` suffices as the fourth argument.

## ***Additional details***

- You can work alone or in pairs.
- In your `BitmapGUI` constructor, make the window initially appear somewhere other than the very top-left of the screen.
- Use class constants where appropriate.
- You can add helper methods if they help.
- It would be a nice touch to make the `JFileChooser` filter for `.bmp`/`.BMP` files.

**BONUS #1 (5%)** Add an *Undo* feature. This will require some modifications to your code. By repeatedly selecting *Undo*, the user should be able to work back to the original image, which should then be marked as unmodified.

**BONUS #2 (5%)** Incorporate the *Combine Two Images* feature from Lab 2. Try to make this as easy as possible for the user. (What about displaying two/three images side by side? This might involve scaling down the images so that they fit on the screen, which is not hard to do in Java.)