AG2411 – GIS Architecture and Algorithms
Takeshi Shirabe
Geoinformatics, KTH
2021-11-15

Exercise 2
Implement a routine to visualize a raster layer

In the previous exercise, you created the **Layer** class, which enabled you to read and write ASCII files storing raster data. In this exercise, you will add to the class the capability to visualize raster data.

PREPARATION

Stay in the same directory that you designated as your working directory for Exercise 1.

TASKS

Note that there is no self-study guide, step-by-step instruction, or sample code for this exercise. Come to the lab and use all kinds of learning resources including your teaching staff, reference books, and past handouts to get ideas on how to approach to this assignment.

1. Add to the **Layer** class two new methods called **toImage**. They have the same name but take different sets of parameters. The **Layer** class is now structured as below.

Layer
+ **name**: String + **nRows**: int + **nCols**: int + **origin**: double[] + **resolution**: double + **values**: double[] + **nullValule**: double
+ **Layer(String, String)** + **print()**: void + **save(String)**: void + **toImage()**: BufferedImage + **toImage (double[]):** BufferedImage

<Method>

- The first **toImage** method takes no parameter, creates a 24-bit RGB image (which is an instance of **BufferedImage** (see Section 1 of HELPS)) of `this` layer by assigning each cell a 0-255 grayscale (or lightness) value proportional to the difference of the largest value in the layer and that cell's value, and returns the image.
- The second **toImage** method takes an array of double values, creates a 24-bit RGB image of `this` layer by first creating a random RGB color for each of the input values and then assigning each cell the color corresponding to that cell's value, and returns the image.

The image created by either method will be drawn on a **MapPanel**, which you will create in Task 3.

Your code may look like…

```
public class Layer {

	//Attributes
	public String name;        // name of this layer
	public int nRows;          // number of rows
	public int nCols;          // number of columns
	public double[] origin = new double[2];// x,y-coordinates of lower-left corner
	public double resolution;  // cell size
	public double [] values;   // raster data
	public double nullValue;   // value designated as "No data"

	//Constructor
	public Layer(String name, String inputFileName) {
	}
	//Other methods
	public void print(){
		// print this layer on the display
	}
	public void save(String outputFileName) {
		// save this layer as an ASCII file that can be imported to ArcGIS
	}
	public BufferedImage toImage () {
		// create a BufferedImage of the layer in grayscale
	}
	public BufferedImage toImage(double[]) {
		// visualize a BufferedImage of the layer in color
	}
}
```

2. Compile the **Layer** class.

3. Create a class called **MapPanel** in the kth.ag2411.mapalgebra package. It should be saved as "MapPanel.java".
   A MapPanel is like a canvas, on which BufferedImage is drawn at a specified scale (see Section 3 of HELPS).

4. Compile the **MapPanel** class.

5. Create a class called **Ex02** in the kth.ag2411.mapalgebra package. It should be saved as "Ex02.java". This class consists only of a **main** method, which takes four or more arguments indicating 1) the name of an input layer, 2) the name of an existing file that stores the layer, 3) a scale calculated as the ratio of the width (or height) of the image to be drawn to that of the layer, and 4) values of interest. Then it will show two images: one showing the layer in grayscale and the other showing the values of interest in (random) color.

6. Compile and test the **Ex02** class.

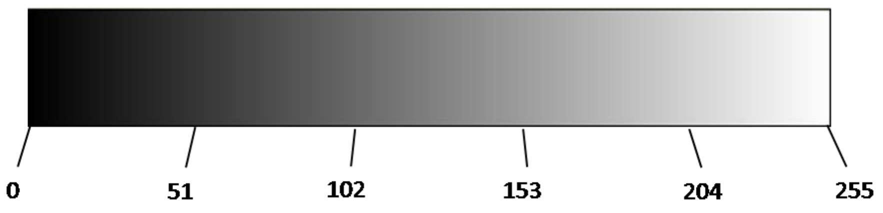
HELPS

1. Creating an image

Java has several image classes, one of which is **BufferedImage** (in **java.awt**). A BufferedImage is comprised of a color model and a raster—more specifically a **WritableRaster** (in **java.awt**). You can edit a BufferedImage through its WritableRaster on a pixel-by-pixel basis. You may find the following lines of code useful when you implement the map methods.

```
// This object represents a 24-bit RBG image with a width of 20 pixels
// (corresponding to the number of columns) and a height of 30 pixels
// (corresponding to the number of rows).
BufferedImage image = new BufferedImage(20, 30, BufferedImage.TYPE_INT_RGB);

// The above image is empty. To color the image, you first need to get access to
// its raster, which is represented by the following object.
WritableRaster raster = image.getRaster();

// These statements make a grayscale value and assign it to the pixel at the
// top-left corner of the raster.
int[] color = new int[3];
color[0] = 128; // Red
color[1] = 128; // Green
color[2] = 128; // Blue
raster.setPixel(19, 0, color); // (19,0) is the pixel at the top-right corner.

// A for-loop statement can easily assign a color value to every cell.
```

To visualize a layer in grayscale, you will need to design a formula to convert a cell value to a grayscale value. Let it be a linear function of a cell value such that it ranges from 0 to 255, 0 corresponding to the largest cell value and 255 corresponding to the smallest cell value (see the picture below). We want cells of greater values to be shaded more darkly.



2. Creating Graphical User Interface (GUI) Components

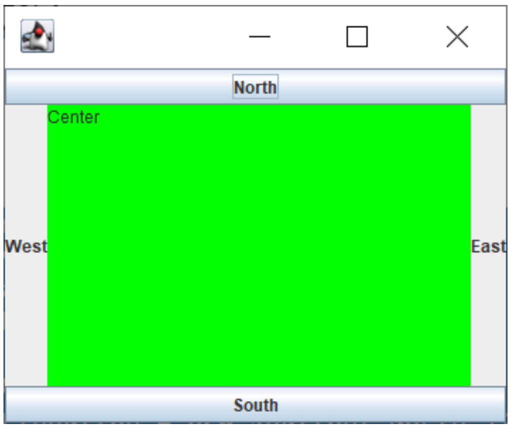
Java has a package called Swing (**javax.swing**), which provides a variety of classes of graphical user interface (GUI) components such as windows (**JFrame** and **JPanel**), buttons (**JButton**), labels (**JLabel**), text areas (**JTextArea**), checkboxes (**JCheckBox**), menu bars (**JMenuBar**), and slider bars (**JSlider**).

A **JFrame** is a window with a title bar and a close control. You will likely to use a JFrame as the main window of an application. As demonstrated below, it is easy to create GUI components and add them to a JFrame.

```
public class GUIDemo {
	public static void main(String[] args) {
		// Create a JFrame, which will be the main window of this demo application
		JFrame appFrame = new JFrame();
```

```
            // Create GUI components
            JButton northButton = new JButton("North");
            JButton southButton = new JButton("South");
            JLabel eastLabel = new JLabel("East");
            JLabel westLabel = new JLabel("West");
            JTextArea centerTextArea = new JTextArea("Center");

            // Set centerTextArea's background color to green
            centerTextArea.setBackground(Color.GREEN);
            // Set centerTextArea's preferredSize to 300 x 200
            Dimension dimension = new Dimension(300 , 200);
            centerTextArea.setPreferredSize(dimension);
            // Make centerTextArea wrap text lines
            centerTextArea.setLineWrap(true);

            // Add the GUI components to appFrame
            appFrame.add(northButton, BorderLayout.PAGE_START);
            appFrame.add(southButton, BorderLayout.PAGE_END);
            appFrame.add(westLabel, BorderLayout.LINE_START);
            appFrame.add(eastLabel, BorderLayout.LINE_END);
            appFrame.add(centerTextArea, BorderLayout.CENTER);

            // Optimize the arrangement the components contained by appFrame
            appFrame.pack();

            // make appFrame visible
            appFrame.setVisible(true);
      }
}
```

Do you see something like this?



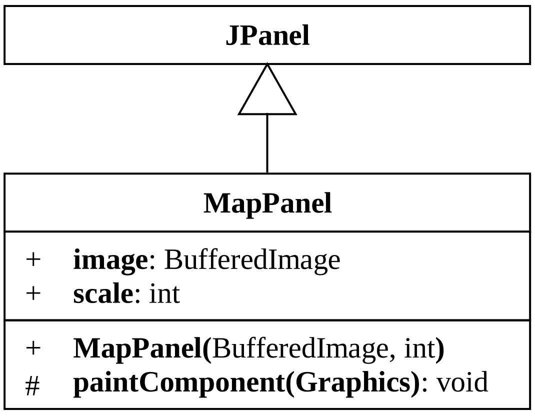
3. Drawing Image on JPanel

Images cannot be seen unless they are drawn on GUI components. As illustrated in the previous section, a JFrame stands alone but other GUI components need to be nested in another GUI component. So, just drawing an image involves more than one GUI component—in particular, **JPanel**.

A JPanel is a container that can contain other GUI components. More importantly here, you can draw on it graphic objects such as vector drawings and raster images, and scale (i.e., expand or shrink) them easily. The graphic objects drawn on the JPanel will be *automatically* redrawn whenever necessary, for example, when the JFrame containing it first appears, when it is obstructed by another window, or when the obstructing window is removed.

You might expect that JPanel has a method like "setImage" that draws an image on it. Unfortunately not. So, we will modify JPanel into a new class—let's call it **MapPanel**—so that an instance of it can store an image and a scale and use a method of JPanel to draw that image at that scale.

The relationship between java's JPanel and our MapPanel is illustrated in the following "class diagram." MapPanel extends JPanel, which means that MapPanel is a subclass of JPanel, which means that MapPanel inherits all the attributes and methods (not shown in the diagram) of JPanel. MapPanel has two additional attributes and one additional constructor, and modifies one of the methods inherited from JPanel, **paintComponent**.


JPanel
MapPanel
+ image: BufferedImage
+ scale: int
+ MapPanel(BufferedImage, int)
# paintComponent(Graphics): void

Here is a sample code for MapPanel.

```
public class MapPanel extends JPanel { // MapPanel is a subclass of JPanel and thus
// inherits all its attributes and methods.

        // ATTRIBUTES
        // All the attributes of JPanel (which are automatically inherited), plus:
        public BufferedImage image;
        public int scale;

        // CONSRUCTORS
        // All the constructors of JPanel (which are automatically inherited), plus:
        public MapPanel(BufferedImage image, int scale) {
                super(); // first, instantiate a MapPanel in the same way JPanel does.
                this.image = image; // then, initialize additional attributes
                this.scale = scale;
        }

        // All the other methods of JPanel (which are automatically inherited), plus:
        @Override
        protected void paintComponent(Graphics g) {
                super.paintComponent(g); // first, do what JPanel would normally do. Then do:
                g.drawImage(image, 0, 0, image.getWidth()*scale, image.getHeight()*scale, this);
        }
        // The @Override tag will be ignored by the complier. It just signifies that
        // MapPanel modifies JPanel's paintComponent() method.
}
```

To show an image at a specific scale, instantiate a MapPanel with that image and that scale, and add it to a JFrame in the same way as demonstrated in the previous section.

4. Private methods

It often happens that a conceptually simple task requires a computationally complex procedure. For example, a procedure to find the maximum value in a layer may be coded as follows.

```
double max = Double.NEGATIVE_INFINITY;
for (int i = 0; i < nRows; i++) {
      for (int j = 0; j < nCols; j++) {
            if (values[i*nCols+j] > max) {
                  max = values[i*nCols+j];
            }
      }
}
```

The above lines of code may not be terribly long or complicated. If they were, however, it would be a good idea to separate them as an independent method, for ease of reading. Such a method is usually designed to be accessible (and useable) only within the class in which that method resides, and thus called a **private method** and preceded by a keyword, **private** when it is defined. For example

```
private double getMax() {
      double max = Double.NEGATIVE_INFINITY;
      for (int i = 0; i < nRows; i++) {
            for (int j = 0; j < nCols; j++) {
                  if (values[i*nCols+j] > max) {
                        max = values[i*nCols+j];
                  }
            }
      }
      return max;
}
```

As shown below, if you add the above method to the Layer class, then you can call it from anywhere within the Layer class.

```
public class Layer {

      .
      .
      .

      public double useGetMaxIndirectly() {
            double max = this.getMax(); // calling getMax()
            return max;
      }

      private double getMax() {
            // see above
      }
}
```

Notice, however, that the Ex02 class has no access to the getMax method. To see it, try this.

```
public class Ex02 {
      public static void main(String[] args) {
      Layer layer = new Layer("elevation", "./data/elevation.txt");
      System.out.println(layer.useGetMaxIndirectly()); // This works.
      System.out.println(layer.getMax()); // This doesn't.
}
```

SUBMISSION

Submit the following.

- Source code of **Layer** class and of **Ex02** class.

DUE DATE

To avoid penalty points, your submission must be made through **CANVAS** no later than **November 18, 2021, 11:59 PM.**