Module IV: Summary - Conditionals and Drawing

Media Concepts

- We learned about some additional image transformation algorithms:
    - Red eye removal is done by selecting a small area of the screen and replacing any pixels close to red with a new color (such as black or brown).
    - Edge detection highlights the "edges" of an image. An edge is a place where a light pixel and a dark pixel are side by side. This algorithm compares the average intensity of a pixel to its neighbor and marks it as an "edge" by turning it black, or a non "edge" by turning it white.
- The distance between two colors can be calculated using the pythagorean theorem, since R, G, and B, can be treated like x, y, and z coordinates.
- Java provides us with a Graphics class that has a rich set of drawing features which allow us to draw shapes and lines easily. To use Graphics, you first get it from the picture object, **Graphics g = pictureObj.getGraphics();**. All drawings assume that (0, 0) is at the top left of the picture. You can then use your **Graphics** object to draw:
    - **g.drawLine(x1,y1,x2,y2)**
    - **g.drawPolygon(xArray, yArray, numPoints)** to draw a polygon with points in xArray and yArray.
    - **g.fillPolygon(xArray, yArray, numPoints)** for a filled-in polygon.
    - **g.drawOval(x,y,width,height)** to draw an oval with a bounding rectangle at (x, y) with width width and height height.
    - **g.fillOval(x,y,width,height)** to draw an oval and fill it in.
    - **g.drawArc(topLeftX,topLeftY,width,height, startAngle, arcAngle)** to draw an arc.
    - **g.fillArc(topLeftX,topLeftY,width,height, startAngle, arcAngle)** to draw a filled-in arc.
    - **g.drawRect(x,y,width,height)** to draw a rectangle.
    - **g.fillRect(x,y,width,height)**
    - **g.drawRoundRect(x,y,width,height, arcWidth, arcHeight)** to draw a rectangle with rounded corners of width arcWidth and height arcHeight.
    - **g.fillRoundRect(x,y,width,height,arcWidth,arcHeight)**

  Use **setColor(color)** to set the color to draw with.
- We can write strings on images with **drawString("some text", offsetX, baselineY)**. We use **setFont(font)** to set the font to write in. **offsetX** is the X position of the leftmost part of the text. **baselineY** is the Y position of the bottom left portion of the text, not including descent elements. You are essentially specifying the bottom left corner of the text when you pass them in. If you need to know the actual size of your text on screen, you can use the **FontMetrics** class. Below is some example code:
    - **FontMetrics currFontMetrics = graphicsObj.getFontMetrics();**
    - **int baselineY = currFontMetrics.getHeight() - currFontMetrics.getDescent();**
    - **int width = currFontMetrics.stringWidth("string");**
- **Graphics** is great for bitmapped images (Raster graphics), but bitmapped images don't scale very well. An alternative is vector graphics, which scale well and are easy to change (since they are essentially represented as programs).
- Using Graphics2D, we can do even more advanced things with graphics in a more object-oriented, encapsulated way. To use Java 2D, you'll need to do the following:

  Get the graphics object from your picture
  Graphics gObj = pictureObj.getGraphics();

  Cast the Graphics class to Graphics2D

Graphics2D g2 = (Graphics2D) gObj;

Set up the stroke if desired (type of pen)
g2.setStroke(new BasicStroke(widthAsFloat));

Set up any Color, GradientPaint, or TexturePaint
g2.setPaint(Color.BLUE);

g2.setPaint(blueToPurpleGradient);

g2.setPaint(texture);

Create a geometric shape
Line2D line2D = new Line2D.Double(0.0,0.0,100.0,100.0);

Draw the outline of a geometric shape
g2.draw(line2D);

Fill a geometric shape
g2.fill(rectangle2D);

Graphics2D and the related classes in java.awt.geom.* add more advanced drawing abilities. Many details are documented on the Java pages for Graphics2D and java.awt.geom.

- Here are some image transformations we covered:
  - Sepia toned images are created by darkening shadows, turning mid-range colors more brown, and making the highlights more yellow
  - Posterizing involves reducing the number of different colors in an image. Typically, all color values in a given range are assigned to be a single value (middle of the range). This is done separately for the red, green, and blue values of a color. A typical example would be to convert each as follows:

Range	Color Value to Use
0 to 63	31
64 to 127	95
128 to 191	159
192 to 255	223

  - Background replacement requires three images -- the original background picture, the same picture with a foreground element (such as a person), and the new background (which can be complex and non-uniform). For each pixel, if the original background and the picture with the foreground element have a color difference of less than 15, the new background is copied into that spot on the image, overwriting the old background.
  - Chroma key is similar, but assumes that the old background is a uniform, blue color. This means that only two pictures are required -- an original picture with a blue background and a foreground element, and the new background to copy the foreground element onto. If a pixel in the original picture is blue, defined as (red + green) < blue, then the new background is copied into that spot on the image.
  - Scaling is done by defining an AffineTransform and then calling **g2D.drawImage(originalImage, affineTransform, null)**
  - Gradients are created as follows:
    - Specifying a point and the color at that point
    - Then specify a second point and the color at that point
    - Create a GradientPaint using those points
    - call **g2D.setPaint(gradientPaint)**

    After that, anything you draw will have a gradient along the vector between the two points

Programming Concepts

- A conditional is used to decide whether to execute a piece of code. The syntax for a conditional is:

```
if (expression) then {
        statement1;
        statement2;
        ...
// optionally...
} else {
        statement1;
        statement2;
        ...
}
```

- Example uses of a conditional include counting how often one thing occurs (such as how many pixels in an image are white); taking action only when something is true (such as only changing pixels to black that are close in color to red, or marking something as an edge only if it is dissimilar from nearby pixels)
- conditional expressions may use any combination of AND (**expression1** && **expression2**), OR (**expression1** || **expression2**), EXCLUSIVE OR (**expression1** ^ **expression2**), EQUALS OR (**expression1** == **expression2**), and NOT (!**expression**). Note the common usage of repeated characters. This is important, as something like **expression1**= **expression2** would mean something very different from **expression1** == **expression2**.
- The OR operator is very useful for checking if at least one of several things is true. For example, if want to check that x is greater than 4 or y is less than 6 or z is equal to 7, The code in java would be written as: x > **4 || y < 6 || z == 7**
- The AND operator is very useful for checking that a value is within a given range. For example, if we want to check if a value lies between 0 and 255 (inclusive), we would usually write it out like so: *0 <= x < 255*. In Java however, this would be written as
    - **0 <= x && x <= 255**
    - **x >= 0 && x <= 255**
- If statements often use ranges to define when something should be done. A range may be *inclusive* (meaning the endpoints are included, in which case something like (**lowend** <= x && x <= **highend**) is used. A range may be *exclusive* (meaning the endpoints are not included) in which case something like (**lowend** < x && x < **highend**) is used. In other words, inclusive ranges are defined using an equal sign and exclusive ranges are defined without the equal.
- We can use if-then-else statements to execute code based on certain conditions. Else statements execute code only if the first condition is false, which we can combine in a lot of useful ways. A common use of if-then-else statements is to act differently for different ranges of values, for example: if (x < 5) { /* x < 5 */ } else if (x < 10) { /* 5 <= x < 10 */ } else { /* x >= 10 */ }. The syntax for this is something like:

```
if (expression1)
   statement1
else if (expression2)
   statement2
else if (expression3)
   statement3
   :
   :
else
   code that does not fit any other range
```

We use conditional statements to test if colors are in a certain range. This can allow us to implement image transformation algorithms such as posterization, background replacement and sepia toning. These algorithms involve looping through the pixels and using if statements to test if the pixels lie in certain ranges, then making different replacements depending on what range the pixel lies in.

- Many of the algorithms we have discussed in this module and the last one are transformations of pictures. These transformation algorithms are a general class of algorithm that tend to have the following structure:

```
public void transformationName(params) {
        Pixel pixelObj;
        <other necessary variable declarations>
        for (int sourceX=val; sourceX < <end condition>; sourceX++) {
                for (int sourceY=val; sourceY < <end condition>; sourceY++) {
                        <apply transformation (change the color of a pixel, copy it into a specified location, etc)>
                }
        }
}
```

- Inheritance defines a relationship between classes. When one class inherits from another, it gets all the methods and properties of the parent class. Inheritance allows us to extend an existing class with additional functionality by adding additional methods and fields. The child class gets all the methods and properties of the parent class. For example, Graphics2D inherits from Graphics. Now we can use a Graphics2D everywhere we use a Graphics, since it is defined to have the same capabilities.
- Interfaces provide a specification for a set of methods that a class implements. A class can only inherit from one parent, but it may implement many additional interfaces.
- We also expanded on our understanding of methods.
    - We learned about setting up return values for methods. The syntax for this is public **ReturnType methodName**(**type1 parameter1**,...) An example use for this is picture transformations that create a new image. A prototypical version of this method would be:

```
public Picture someTransformation() {

        // if we want to modify the current image, do the following
        // create a new picture to draw into
        Picture result = new Picture(width height);
        Graphics2D g2D = (Graphics2D) result.getGraphics()
        // possibly set up some sort of transform or fancy paint stuff
        g2D.drawImage(this.getImage(), ...)
        return result
}
```

    - Methods have parameters which have types. Types can be primitive, a class, or an interface.

      Passing a primitive
      A parameter can be a primitive type such as int (red box)

      Passing an object whose type is a Class
      A parameter can have the type of a class. For example, a method could be defined that takes type Object as a parameter (public void **myMethod**(Object myObj). In that case, any of the classes shown in the image (blue and green rectangles) would be valid parameters. This is because they all inherit from Object or from something that inherits from Object. Similarly, a method that takes an object of type MultipleGradientPaint could also be passed an object of type LinearGradientPaint

      Passing an object whose type is an Interface
      All of the blue boxes are classes that implement Paint (an interface) either explicitly (Color, GradientPaint, MultipleGradientPaint), or through inheritance (LinearGradientPaint). An object created from any of the four classes shown in blue could be passed into a method requiring a parameter of type Paint.

- The Java API documentation will be a very important resource for you moving forward. No one can remember every detail of all of the classes provided in the Java API. Instead, good programmers know to make use of documentation (this is one reason it's so very important to carefully document your code -- you won't even remember everything your own code does once you write enough of it!).

Inheritance	Interface	Primitive Types
Lots of classes can inherit from the same object	Lots of classes can implement the same interface	Primitive types are not classes
The syntax to inherit from a class is extends classname	The syntax to implement an interface is implements interface [extends classname]	Primitive types cannot be extended
A class includes fields and methods	An interface includes only methods	A primitive contains only a value
A class can only inherit from a SINGLE object	A class can implement any number of different interfaces	
A variable var can be defined using a class Classname as its type	A variable var can be defined using an interface Interfacename as its type	A variable var can be defined using a primitive as its type
Any instance of Classname can be assigned to var	Any instance of *any* class that implements Interfacename can be assigned to var	Anything value of that primitive type can be assigned to var
var doesn't know what interfaces Classname implements, only what it's class is.	var doesn't know about methods or fields associated with the class implementing Interfacename, only the methods defined in Interfacename	varcontains a value, not a class
If var's type (Classname) implements an interface Interfacename, and var2 is of type Interfacename, then var2 = var is a valid statement. This is because Java knows that Classname implements Interfacename	If var's type is Interfacename; var contains an instance of Classname; and var2 is of type Classname, then var2 = var is *not* a valid statement. Instead, you must use var2 = (Classname) var. This is because Java has no way of knowing which of the classes that implement Interfacename are contained in var	If var and var2 are of the same primitive type, then var2 = var is a valid statement. Otherwise you must use casting to assign var to var2
Inheritance is single, not multiple	Interfaces support "multiple inheritance"	primitives do not support inheritance