Chapter 5: Imaging - AWT Facilities, Loading Images

- As with 2D graphics, `awt Graphics` class provides the earliest imaging capabilities, and is restrictive
- Methods for loading an image:

  1. From an Applet (`java.applet.Applet`)

     - Methods:
     
     (a) `Image getImage(URL url)`

         * Returns an `Image` object that can then be painted on the screen
         * The URL must be absolute
         * Call returns immediately
         * Loading begins when the applet attempts to draw the image

     (b) `Image getImage(URL url, String name)`

         * The `String` argument is the name of the image file

     - Examples:

       ```java
       Image image1, image2, image3;

       public void init()
       {
           image1 = getImage(getCodeBase(), "imageFile.gif");
           image2 = getImage(getDocumentBase(), "anImageFile.jpeg");
           image3 = getImage(new URL("http://java.sun.com/graphics/people.gif"));
           ...
       }
       ```

     - `URL getCodeBase()`

         * Gets the base URL
         * This is the URL of the directory which contains this applet

     - `URL getDocumentBase()`

         * Gets the URL of the document in which this applet is embedded
         * If an applet is contained in the document

         ```java
         http://java.sun.com/products/jdk/1.2/index.html
         ```

         the document base is

         ```java
         http://java.sun.com/products/jdk/1.2/index.html
         ```
2. From an application (use Toolkit class)
   - Methods:
     (a) abstract Image getImage(String filename)
     (b) abstract Image getImage(URL url)
   - Examples:

     Image image1, image2;

     public void init()
     {
       Toolkit toolkit = Toolkit.getDefaultToolkit();
       image1 = toolkit.getImage("imageFile.gif");
       image2 = toolkit.getImage(new URL("http://java.sun.com/graphics/people.gif"));
       ...
     }
   - Works for GIF, JPEG, and PNG files
Chapter 5: Imaging - AWT Facilities, Drawing Images

• Methods:

1. `abstract boolean drawImage(Image img, int x, int y, ImageObserver observer)`
   - Draws as much of the specified image as is currently available
   - `(x, y)` locate the upper left corner
   - `bgcolor` is the background color
   - `ImageObserver` is usually implemented by the canvas object; e.g., `JPanel`
     * It could also be an anonymous `Component`:
       ```java
g2.drawImage(image, 0, 0, new Component(){});
```

2. `abstract boolean Graphics.drawImage(Image img, int x, int y, Color bgcolor, ImageObserver observer)`

3. `abstract boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)`
   - Draws as much of the specified image as has already been scaled to fit inside the specified rectangle

4. `abstract boolean drawImage(Image img, int x, int y, int width, int height, Color bgcolor, ImageObserver observer)`

5. `abstract boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, Color bgcolor, ImageObserver observer)`
   - Draws as much of the specified area of the specified image as is currently available, scaling it on the fly to fit inside the specified area of the destination drawable surface
   - `(dx1, dy1)` and `(dx2, dy2)` are the corners of the destination rectangle
   - `(sx1, sy1)` and `(sx2, sy2)` are the corners of the source rectangle

6. `abstract boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, ImageObserver observer)`
Chapter 5: Imaging - AWT Facilities, Monitoring Progress

- These classes monitor the image loading process

  1. The `ImageObserver` interface
     - Monitors the progress of the image construction
     - It’s sole method
       
       ```java
       boolean imageUpdate(Image img, int infoflags, int x, int y, int width, int height)
       ```
     - is called repeatedly as image data becomes available
     - `imageUpdate()` then calls `repaint()` (by default)
     - This process loads the image incrementally, and its initial drawing will be incremental (if downloaded over a network)
     - Example:

```java
Image image1;

public void init()
{
    Toolkit toolkit = Toolkit.getDefaultToolkit();
    image1 = toolkit.getImage(new URL("http://java.sun.com/graphics/people.gif"));
    ...
}

public void paintComponent(Graphics g)
{
    g.drawImage(image1, 0, 0, this);
    ...
}
```
2. *MediaTracker* class

- Used to track the status of media objects
- Theoretically could be used for both sound and image objects, but currently only supports images
- Does not return until object has been completely loaded

- Constructors:
  - (a) *MediaTracker(Component comp)*

- Methods:
  - (a) *void addImage(Image image, int id)*
    - *Adds an image to the list of images being tracked by this media tracker*
    - *id* is an identifier used to track the image
  - (b) *void addImage(Image image, int id, int w, int h)*
    - *Adds a scaled image to the list of images being tracked by this media tracker*
    - *(w, h)* represent the dimensions at which the image is to be displayed
  - (c) *boolean checkAll()*
    - *Checks to see if all images being tracked by this media tracker have finished loading*
    - *It does not start loading any images*
  - (d) *boolean checkAll(boolean load)*
    - *As above, but if load is true, it will start loading images that have not started loading yet*
  - (e) *boolean checkID(int id)*
    - *Checks to see if all images tracked by this media tracker that are tagged with the specified identifier have finished loading*
  - (f) *boolean checkID(int id, boolean load)*
  - (g) *void waitForAll()*
    - *Starts loading all images tracked by this media tracker*
    - *It waits until all the images have finished loading*
  - (h) *boolean waitForAll(long ms)*
    - *As above, but will also stop waiting when ms milliseconds have elapsed*
(i) `void waitForID(int id)`

* Starts loading all images tracked by this media tracker with the specified identifier

(j) `boolean waitForID(int id, long ms)`

- When using `MediaTracker`, `null` can be used instead of an `ImageObserver` object in the call to `drawImage()` since drawing will now occur only after loading is complete

- Example:
  ```java
  Image image1;
  public void init()
  {
    Toolkit toolkit = Toolkit.getDefaultToolkit();
    image1 = toolkit.getImage(new URL("http://java.sun.com/graphics/people.gif"));

    MediaTracker tracker = new MediaTracker(this);
    tracker.addImage(image1, 0);
    try {
      tracker.waitForID(0);
    } catch (InterruptedException e) {
      System.out.println("Download error");
    }
    ...
  }

  public void paintComponent(Graphics g)
  {
    g.drawImage(image1, 0, 0, null);
    ...
  }
  ```
3. *ImageIcon* class

- Achieves loading of image via an *ImageIcon*
- Constructors:
  (a) *ImageIcon()*
    * Creates an uninitialized image icon
  (b) *ImageIcon(Image image, String description)*
    * Creates an *ImageIcon* from an image object
    * *description* is simply that
  (c) *ImageIcon(Image image)*
    * As above, with description set to *comment* property of image
  (d) *ImageIcon(String filename, String description)*
    * Creates an *ImageIcon* from the specified file
    * *filename* can be a file name or a file path
  (e) *ImageIcon(String filename)*
    * The description is set to the *filename* string
  (f) *ImageIcon(URL location, String description)*
    * Creates an *ImageIcon* from the specified URL
    * Image is preloaded using MediaTracker
  (g) *ImageIcon(URL location)*
- Methods:
  (a) *void loadImage(Image image)*
    * Loads the image, returning only when the image is loaded
  (b) *Image getImage()*
  (c) *void paintIcon(Component c, Graphics g, int x, int y)*
    * Paints the icon
    * The top-left corner of the icon is drawn at \((x, y)\)
    * If this icon has no image observer, this method uses the \(c\) component as the observer
Chapter 5: Imaging - AWT Facilities, Monitoring Progress

Examples for loading an image:

(a)
```
Image image1;

public void init()
{
    ImageIcon icon = new ImageIcon(new URL("http://java.sun.com/graphics/people.gif"));
    Image image1 = icon.getImage();

    image1 = icon.getImage();
    ...

    public void paintComponent(Graphics g)
    {
        g.drawImage(image1, 0, 0, null);
        ...
    }
```

(b)
```
Image image1;

public void init()
{
    ImageIcon icon = new ImageIcon(new URL("http://java.sun.com/graphics/people.gif"));
    Image image1 = new Image();

    image1 = icon.getImage();
    ...

    public void paintComponent(Graphics g)
    {
        icon.paintIcon(this, g, 0, 0);
        ...
    }
```
• This approach is used by AWT

• An ImageProducer generates a stream of pixel data

• This data is ”pushed” to an ImageConsumer

• Pixel data can be manipulated by an ImageFilter, which intercepts the data from the producer before it gets to the consumer

• This model was intended for downloading images over the network

• The problem with this is that it makes it difficult to process groups of pixels together

• This can be remedied using additional classes

  1. PixelGrabber collects all the pixel data into an array

  2. MemoryImageSource outputs the array’s data as a stream
Chapter 5: Imaging - AWT Facilities, Push Model (2)
Chapter 5: Imaging - Java2D BufferedImage Class

- Offers many advantages over the Image class:
  1. Loading BufferedImage is much simpler than regular Images
     - Java2D uses immediate mode as opposed to the push model
  2. Provides methods for image manipulation
  3. May be stored as managed images
     - Hardware acceleration can be used for rendering

- Structure:

1. Raster stores pixel data
   - DataBuffer holds the actual pixel data
   - SampleModel defines how the samples are clustered (e.g., color triads, stripes of colors, etc.)
Chapter 5: Imaging - Java2D  *BufferedImage* Class (2)

2. *ColorModel* has methods for converting pixel data to colors
   - It provides mappings from pixel data to color components
   - *ColorSpace* defines how to combine the components

- The general algorithm for processing *BufferedImages*
  1. Create a *BufferedImage* object for storing the pixel data
  2. Read the file into the object
  3. Process the image using the image-processing facilities of Java2D (see later)
  4. Display and/or save the image to a file

- Details:
  1. Creating *BufferedImage* objects
     (a) Using *BufferedImage* constructors:
        i. *BufferedImage*(int *width*, int *height*, int *imageType*)
           - *(width, height)* specify the image dimensions
           - *imageType* specifies the pixel format
           * Typical formats are *BufferedImage.TYPE_INT_RGB* and *BufferedImage.TYPE_INT_ARGB*
           - The color space is *sRGB*
        ii. *BufferedImage*(ColorModel *cm*, WritableRaster *raster*, boolean *isRasterPremultiplied*, Hashtable<?, ?> *properties*)
           - See online documentation
        iii. *BufferedImage*(int *width*, int *height*, int *imageType*, IndexColorModel *cm*)
           - See online documentation
     (b) Using *Image Component.createImage*(int *width*, int *height*)
        - The resulting *Image* can be cast to *BufferedImage*
Chapter 5: Imaging - Java2D BufferedImage Class (3)

(c) Using the abstract GraphicsConfiguration class
- This class describes the characteristics of a graphics device
- Methods:
  i. abstract BufferedImage createCompatibleImage(int width, int height)
     * Returns a BufferedImage with a data layout and color model compatible with this GraphicsConfiguration
  ii. abstract BufferedImage createCompatibleImage(int width, int height, int transparency)
     * Returns a BufferedImage with a data layout and color model compatible with this GraphicsConfiguration that supports the specified transparency
- The advantage to creating a ”compatible” image is that the new image will share the storage properties of the original
  * If the device is the screen, then colors will be stored in the same format for both the image and the screen
  * This makes display more efficient as no conversions will be necessary
- Example:
  BufferedImage im, copy1, copy2;
  ...
  int imWidth = im.getWidth();
  int imHeight = im.getHeight();
  int transparency = im.getColorModel().getTransparency();
  BufferedImage copy = gc.createCompatibleImage(imWidth, imHeight, transparency);
  Graphics2D imgc = im.createGraphics();
  BufferedImage copy2 = imgc.createCompatibleImage(imWidth, imHeight);

2. Loading BufferedImage images
- ImageIO class
  * Methods:
    (a) static BufferedImage read(File input) throws IOException
        * Returns a BufferedImage as the result of decoding a supplied File with an ImageReader chosen automatically from among those currently registered
        * Sample code:
          BufferedImage img;
          try {
            img = ImageIO.read(new File("strawberry.jpg"));
          } catch (IOException e) { 
          }
Chapter 5: Imaging - Java2D *BufferedImage* Class (4)

(b) *static BufferedImage read(URL input)*

* Returns a *BufferedImage* as the result of decoding a supplied URL with an *ImageReader* chosen automatically from among those currently registered

3. Displaying *BufferedImage* images

- Methods:
  
  (a) *abstract void drawImage(BufferedImage img, BufferedImageOp op, int x, int y)*

  * (x, y) specify location of the image
  * op specifies the type of pixel manipulation required (see later)

  (b) *abstract boolean drawImage(Image img, AffineTransform xform, ImageObserver obs)*

  * xform is a transform from image space to user space to be applied before drawing
  * It is applied before any transforms in the current context are applied

  * The return value indicates whether the image is fully loaded and rendered

4. Drawing to *BufferedImage* images

- To be able to draw to a *BufferedImage* object, you need a *Graphics2D* context

  * *Graphics2D createGraphics()* returns a graphics context for the image

    ```java
    BufferedImage bi = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);
    Graphics2D g2 = (Graphics2D)bi.createGraphics();
    ```

  - Use the normal *Graphics2D* methods for drawing discussed earlier
5. Writing BufferedImage to a file

- Use `static boolean ImageIO.write(RenderedImage im, String formatName, File output) throws IOException`
- This will allow you to save an image that you created or manipulated to a file
- `formatName` is a `String` that represents the graphics format of the file
  * Standard formats that should be supported include PNG, GIF, JPEG, and BMP
  * This is extensible, meaning that other plugins can be used to support other formats
- Sample code:
  ```java
  try {
      BufferedImage bi = getMyImage(); // retrieve image
      File outputfile = new File("saved.png");
      ImageIO.write(bi, "png", outputfile);
  } catch (IOException e) {
      ...
  }
  ```
Chapter 5: Imaging - Managed Images

- A *managed* image is automatically cached in VRAM by the JVM
- `drawImage()` uses the VRAM version of an image instead of the original
  - The advantage is that hardware blitting is used with the VRAM version
  - The non-VRAM version can only use software blitting
- Managed versions are created at the whim of the JVM
  - `Image`, `ImageIcon`, and `BufferedImage` objects can be managed if created with
    1. `createImage()`
    2. `createCompatibleImage()`
    3. A `BufferedImage` constructor
  - or loaded using
    1. `getImage()`
    2. `ImageIO.read()`
  - Transparency also affects the ability to be managed
    * `OPAQUE` and `BITMASK` transparency can be managed
    * Translucent images can be managed if system-dependent flags are set
- An image will be copied to VRAM if it has not been modified for some interval
  - If edited with a non-accelerated operation, the next `draw` operation will use the RAM version
  - This is all system-dependent
  - The upshot is that you should minimize modification of images during runtime
Chapter 5: Imaging - *VolatileImage* Class

- Volatile images are created and managed by the programmer, not by the JVM

- To create:
  1. `VolatileImage Component.createComponentVolatileImage(int width, int height, ImageCapabilities caps)` throws `AWTException`
  2. `VolatileImage GraphicsConfiguration.createComponentCompatibleVolatileImage(int width, int height, ImageCapabilities caps)` throws `AWTException`

- They exist solely in VRAM
  - No copy exists in RAM

- Their operations are accelerated (dependent on OS)

- Disadvantages:
  1. Image may be lost (dependent on OS)
     - Program must monitor and recreate when necessary
  2. All processing done in VRAM
     - If operation native to graphics card, will be accelerated
     - If not, software operations in VRAM slower than in RAM
Chapter 5: Imaging - Image Processing Introduction

- **Image processing** refers to manipulation of an image, usually at the pixel level
  - It’s sometimes referred to as image *filtering*, analogous to real-world use of filters with cameras

- Java2D allows processing of *BufferedImage* objects via the *java.awt.image.BufferedImageOp* interface
  - ”Regular” images can be manipulated using the older *java.awt.image.ImageFilter* class
    * This not addressed in these notes

- Image processing starts with a *source image*, applies an operation to this image, and generates a *destination image*

The primary method is *filter()*:

1. *BufferedImage* filter(*BufferedImage src, BufferedImage dest*)
   - If *dest* is not *null*, the filtered image is returned in it
   - Regardless of whether *dest* is *null*, a new *BufferedImage* object is created to hold the filtered image
   - Regardless of whether *dest* is *null*, the method returns the filtered image
   - *src* is unaffected

For example:

```java
BufferedImage b1, b2, b3;
BufferedImageOp op = new $<specific>$Op(...);
...

b2 = op.filter(b1, null);
op.filter(b1, b3);
...
```
Chapter 5: Imaging - Image Processing Introduction (2)

- The predefined operators:

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Supporting Classes</th>
<th>Effects</th>
<th>In place?</th>
</tr>
</thead>
<tbody>
<tr>
<td>AffineTransformOp</td>
<td>java.awt.geom.AffineTransform</td>
<td>transformation</td>
<td>no</td>
</tr>
<tr>
<td>LookupOp</td>
<td>LookupTable</td>
<td>posterizing</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>ByteLookupTable</td>
<td>inversion</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ShortLookupTable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RescaleOp</td>
<td></td>
<td>brightening</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>darkening</td>
<td></td>
</tr>
<tr>
<td>ColorConvertOp</td>
<td>java.awt.color.ColorSpace</td>
<td>color space conversion</td>
<td>yes</td>
</tr>
<tr>
<td>ConvolveOp</td>
<td>Kernel</td>
<td>blurring</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sharpening</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>edge detection</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 5: Imaging - Image Processing Rendering Hints

• Hints are suggestions provided to the JVM as to how the programmer would like something rendered
  – They deal with quality/time tradeoffs
  – They may or may not be heeded by the JVM

• They are implemented by the `java.awt.RenderingHints` class

• Hints are specified in terms of a `key/value` pair
  – The `key` specifies the attribute of interest
  – The `value` specifies how you’d like that attribute handled

• Keys and values (partial list):

<table>
<thead>
<tr>
<th>Key</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY_ANTIALIASING</td>
<td>VALUE_ANTIALIAS_ON</td>
</tr>
<tr>
<td></td>
<td>VALUE_ANTIALIAS_OFF</td>
</tr>
<tr>
<td></td>
<td>VALUE_ANTIALIAS_DEFAULT</td>
</tr>
<tr>
<td>KEY_RENDERING</td>
<td>VALUE_RENDER_QUALITY</td>
</tr>
<tr>
<td></td>
<td>VALUE_RENDER_SPEED</td>
</tr>
<tr>
<td></td>
<td>VALUE_RENDER_DEFAULT</td>
</tr>
<tr>
<td>KEY_DITHERING</td>
<td>VALUE_DITHER_DISABLE</td>
</tr>
<tr>
<td></td>
<td>VALUE_DITHER_ENABLE</td>
</tr>
<tr>
<td></td>
<td>VALUE_DITHER_DEFAULT</td>
</tr>
<tr>
<td>KEY_COLOR_RENDERING</td>
<td>VALUE_COLOR_RENDER_QUALITY</td>
</tr>
<tr>
<td></td>
<td>VALUE_COLOR_RENDER_SPEED</td>
</tr>
<tr>
<td></td>
<td>VALUE_COLOR_RENDER_DEFAULT</td>
</tr>
<tr>
<td>KEY_FRACTIONALMETRICS</td>
<td>VALUE_FRACTIONALMETRICS_ON</td>
</tr>
<tr>
<td></td>
<td>VALUE_FRACTIONALMETRICS_OFF</td>
</tr>
<tr>
<td></td>
<td>VALUE_FRACTIONALMETRICS_DEFAULT</td>
</tr>
<tr>
<td>KEY_TEXT_ANTIALIASING</td>
<td>VALUE_TEXT_ANTIALIAS_ON</td>
</tr>
<tr>
<td></td>
<td>VALUE_TEXT_ANTIALIAS_OFF</td>
</tr>
<tr>
<td></td>
<td>VALUE_TEXT_ANTIALIAS_DEFAULT</td>
</tr>
<tr>
<td>KEY_INTERPOLATION</td>
<td>VALUE_INTERPOLATION_BICUBIC</td>
</tr>
<tr>
<td></td>
<td>VALUE_INTERPOLATION_BILINEAR</td>
</tr>
<tr>
<td></td>
<td>VALUE_INTERPOLATION_NEAREST_NEIGHBOR</td>
</tr>
<tr>
<td>KEY_ALPHA_INTERPOLATION</td>
<td>VALUE_ALPHA_INTERPOLATION_QUALITY</td>
</tr>
<tr>
<td></td>
<td>VALUE_ALPHA_INTERPOLATION_SPEED</td>
</tr>
<tr>
<td></td>
<td>VALUE_ALPHA_INTERPOLATION_DEFAULT</td>
</tr>
</tbody>
</table>
Chapter 5: Imaging - Image Processing Rendering Hints (2)

- What they affect:

  1. **KEY_ANTIALIASING**: Smoothness of lines
  2. **KEY_RENDERING**: Generic quality v speed
  3. **KEY_DITHERING**: For increasing the apparent color palette on devices with a limited gamut
  4. **KEY_COLOR_RENDERING**: Controls color correction for devices with color profiles
  5. **KEY_FRACTIONALMETRICS**: Controls use of floats or integers for fonts
  6. **KEY_TEXT_ANTIALIASING**: 
  7. **KEY_INTERPOLATION**: Controls how pixels are combined with neighbors for some pixel operations
  8. **KEY_ALPHA_INTERPOLATION**: Controls how alpha values are computed

- These used to control *Graphics2D* rendering in general, and filtering of *BufferedImages*

- Constructors:

  1. `RenderingHints(RenderingHints.Key key, Object value)`
     - Creates a single hint
  2. `RenderingHints(Map<RenderingHints.Key, ?> init)`
     - Allows multiple hints to be specified at one time by passing them in the *Map* argument

- Methods:

  1. `Object put(Object key, Object value)`
     - Adds the *key/value* pair to the given object
     - It returns the previous value of *key* in the object, or *null* otherwise
  2. `void add(RenderingHints hints)`
     - Adds the *key/value* pairs in *hints* to the given object
  3. `Object remove(Object key)`
     - Deletes the *key* and its values from this object
     - It returns the previous value of *key* in the object, or *null* otherwise
Classes `Graphics2D`, and those that implement `BufferedImageOp` and `RasterOp` provide methods to get and possibly set hints

Example:

```java
Graphics2D g2 = (Graphics2D) g;
RenderingHints hints = new RenderingHints(RenderingHints.KEY_ALIASING,
                                          RenderingHints.VALUE_ANTIALIAS_ON);
hints.put(RenderingHints.KEY_RENDERING, RenderingHints.VALUE_RENDER_ON);
g2.setRenderingHints(hints);
... ```
Chapter 5: Imaging - Image Processing  

* AffineTransformOp Class

- Creates a new image that is a transformed version of the original
- Not the same as the Graphic2D transforms
- Creating the destination image
  - Keep in mind that both the source and destination are rectangular arrays of pixel information (color)
  - The transformed source image is overlaid on top of the destination image
  - There is usually not a one-to-one correspondence between a destination pixel and source pixels that overlap it

- You need to specify how such situations should be handled
  1. Nearest neighbor hint
     * Use color of the source pixel that is closest to the destination pixel
  2. Bilinear interpolation hint
     * Colors of overlapping source pixels combined

* Constructors;
  1. AffineTransformOp(AffineTransform xform, int interpolationType)
     - Note that xform is a standard Graphics2D transform as discussed previously
     - interpolationType is one of TYPE_NEAREST_NEIGHBOR or TYPE_BILINEAR
     - Default is nearest neighbor (if type is null)
  2. AffineTransformOp(AffineTransform xform, RenderingHints hints)
     - In this version, hints should contain a key/value pair that affects interpolation
  3. Example:

```java
BufferedImage b1, b2;
//...
AffineTransform t = AffineTransform.getRotateInstance(Math.PI/6, 0, 80);
RenderingHints hints = new RenderingHints(RenderingHints.KEY_INTERPOLATION,
    RenderingHints.VALUE_INTERPOLATION_BILINEAR);
AffineTransformOp myOp = new AffineTransformOp(t, hints);
myOp.filter(b1, b2);
```
Chapter 5: Imaging - Image Processing \textit{AffineTransformOp} Class (2)

- **Methods**
  - Note: These also defined for other \textit{BufferedImageOp} classes

1. \textit{BufferedImage createCompatibleDestImage(BufferedImage src, ColorModel destCM)}
   - Creates a zeroed destination image with the correct size and number of bands
   - \textit{src} is the source image for the filter operation
   - \textit{destCM} is the destination’s ColorModel, which can be \textit{null}
     * If \textit{null}, an appropriate ColorModel will be used

2. \textit{Rectangle2D getBounds2D(BufferedImage src)}
   - Returns the bounding box of the filtered destination image

3. \textit{Point2D getPoint2D(Point2D srcPt, Point2D dstPt)}
   - Returns the location of the destination point given a point in the source
   - If \textit{dstPt} is not \textit{null}, it will be used to hold the return value

- Pixels of destination that are not covered by source pixels are filled with 0’s
  - If not using an alpha channel, these will be black
  - If there is an alpha channel, these will be transparent

- If pixels of source fall outside of destination rectangle, these pixels will be lost
  - Destination rectangle effectively acts like a clipping rectangle
• A **LookupOp** is specified in terms of a **LookupTable** object
  - Lookup tables implement color indexing
    * Lookup table(s) hold(s) color values
    * A pixel value is not interpreted as a color, but as an index into a lookup table(s)
    * The color retrieved from the table(s) is displayed

  ![Diagram of a raster and lookup table](image)

  - **LookupTable** abstract class used to support color indexing
    * Color data can be stored in tables in several ways:
      1. You can use a single table whose value represents all channels of a color (e.g., RGB or RGBA)
      2. You can use one table for each channel of a color
    * **LookupTable** is extended by classes **ByteLookupTable** and **ShortLookupTable**
      - These differ by the data types used for color data
Chapter 5: Imaging - Image Processing

* LookupOp Class (2)

* Constructors (**Byte** shown; **Short** is correspondingly similar):

1. **ByteLookupTable**(*int offset, byte[] data*)
   - *data* holds the color data as a single value (e.g., RGB)
   - *offset* is used to modify the index values
   - For example, if pixel values are known to be in the range
     \(50 \leq c \leq 150\)
     you could use an offset of 50 and a table dimensioned to hold 101
     values
   - If you used a value of 0 for *offset*, your table would have to have 151
     slots

2. **ByteLookupTable**(*int offset, byte[][] data*)
   - In this case, you need one table for each color channel
     (a) If using RGB, you need 3
     (b) If using RGBA, you need 4
   - *data* holds all of the channel tables
   - The tables are used in tandem

* LookupOp class

- Constructor:

  * **LookupOp**(*LookupTable lookup, RenderingHints hints*)

- Example:

  ```java
  short[] v1 = new short[256];
  // Load v1 with values
  LookupTable t1 = new ShortLookupTable(0, v1);
  LookupOp luOp1 = new LookupOp(t1, null);
  ...
  
  short[] v2 = new short[256];
  short[] v3 = new short[256];
  short[] v4 = new short[256];
  // Load v2 with values
  // Load v3 with values
  // Load v4 with values
  short[][] mv = {v1, v2, v3};
  LookupTable t2 = new ShortLookupTable(0, mv);
  LookupOp luOp2 = new LookupOp(t2, null);
  ...
  ```
Chapter 5: Imaging - Image Processing *RescaleOP* Class

- Scales (multiplies) every color component of a pixel’s color by a scale factor

\[ c_d = s \times c_s + o \]

where \( o \) is an offset

- Components are bounded:

\[
\begin{align*}
0.0 & \leq c_d \leq 1.0 \\
0 & \leq c_d \leq 255 
\end{align*}
\]

- Constructors:

1. *RescaleOp(float scaleFactor, float offset, RenderingHints hints)*
   - Equation applied to all channels except *alpha*

2. *RescaleOp(float[] scaleFactors, float[] offsets, RenderingHints hints)*
   - This version allows non-uniform scaling of components
   - The arrays must have one value per channel of the pixel data

- Example:

```java
RescaleOp brightenOp = new RescaleOp(1.5f, 0, null);
```
Chapter 5: Imaging - Image Processing  

ConvolveOp Class

- In convolution, a color is generated from a source by combining the color of the source with those of its neighbors
- The process is specified in terms of a *kernel*
  - Kernel represented by a 3 X 3 matrix
  - The central slot represents the source
  - The others represent the neighbors

<table>
<thead>
<tr>
<th>s1</th>
<th>s2</th>
<th>s3</th>
</tr>
</thead>
<tbody>
<tr>
<td>s4</td>
<td>s5</td>
<td>s6</td>
</tr>
<tr>
<td>s7</td>
<td>s8</td>
<td>s9</td>
</tr>
</tbody>
</table>

- The color of the destination is computed as
  \[ c_d = n1 \times s1 + n2 \times s2 + n3 \times s3 + n4 \times s4 + \text{source} \times s5 + n6 \times s6 + n7 \times s7 + n8 \times s8 + n9 \times s9 \]

- Kernel represented by java.awt.image.Kernel
  - Constructors:
    1. `Kernel(int width, int height, float[] data)`
       * `width` and `height` specify how many `data` values are copied
         - The first `width * height` are copied
       * Values in `data` are stored in row major order
  - How should situation where kernel falls outside of source area be handled?
    1. `public static final int EDGE_ZERO_FILL`
       * Destination edge pixels filled with 0’s
    2. `public static final int EDGE_NO_OP`
       * Destination edge pixels left unchanged

- ConvolveOp Constructors:
  1. `ConvolveOp(Kernel kernel)`
     - By default, `EDGE_ZERO_FILL` is used
  2. `ConvolveOp(Kernel kernel, int edgeCondition, RenderingHints hints)`
Chapter 5: Imaging - Image Processing *ConvolveOp* Class

- Example:

```java
float identity = {0.0f, 0.0f, 0.0f
                0.0f, 1.0f, 0.0f,
                0.0f, 0.0f, 0.0f};
Kernel identityKernel = new Kernel(3, 3, identity);
ConvolveOp nullOp = new ConvolveOp(identity, ConvolveOp.EDGE_NO_OP, null);
```
You can create custom operators by implementing the `BufferedImageOp` interface

You must define the following methods:

1. `BufferedImage createCompatibleDestImage(BufferedImage src, ColorModel destCM)`
   - Creates a zeroed destination image with the correct size and number of channels
   - `destCM` is the `ColorModel` of the destination
     * If null, the `ColorModel` of the source is used
2. `BufferedImage filter(BufferedImage src, BufferedImage dest)`
   - Performs a single-input/single-output operation on a `BufferedImage` - this does the work
   - If the color models for the two images do not match, a color conversion into the destination color model is performed
   - If the destination image is null, a `BufferedImage` with an appropriate ColorModel is created
3. `Rectangle2D getBounds2D(BufferedImage src)`
   - Returns the bounding box of the filtered destination image
4. `Point2D getPoint2D(Point2D srcPt, Point2D dstPt)`
   - Returns the location of the corresponding destination point given a point in the source image
   - These will always be the same except for operators that transform the source image
   - If `dstPt` is specified, it is used to hold the return value
5. `RenderingHints getRenderingHints()`
   - Returns the rendering hints for this operation
Chapter 5: Imaging - Image Processing User-Defined Operators (2)

- Example:
  - This example demonstrates the creation of a thresholding operator
    
    * Given a threshold, max, and min value, this operator sets pixel values below the threshold to the min value, and all others to the max value
  
    - Taken from *Java 2D Graphics* (O'Reilly)

```java
public class ThresholdOp implements BufferedImageOp {
    protected int threshold, min, max;

    public class ThresholdOp (int threshold, int min, int max) {
        this .threshold = threshold;
        this.min = min;
        this.max = max;
    }  

    public final BufferedImage filter (BufferedImage src, BufferedImage dst) {
        if (dst == null)
            dst = createCompatibleDestImage(src, null);
        for (int y = 0; y < src.getHeight(); y++)
            for (int x = 0; x < src.getWidth(); x++) {
                int srcPixel = src.getRGB(x, y);
                Color c = new Color(srcPixel);
                int red = threshold(c.getRed());
                int blue = threshold(c.getBlue());
                int green = threshold(c.getGreen());
                dst.setRGB(x, y, new Color(red, green, blue).getRGB());
            }
        return dst;
    }

    public int threshold (int input) {
        if (input < threshold)
            return min;
        else
            return max;
    }

    public BufferedImage createCompatibleDestImage (BufferedImage src, ColorModel dstCM) {
        BufferedImage bi;
        if (dstCM == null)
            dstCM = src.getColorModel();
        int width = src.getWidth();
        int height = src.getHeight();
        bi = new BufferedImage(dstCM,  
            dstCM.createCompatibleWriteableRaster(width, height),
            dstCM.isAlphaPremultiplied(), null);  
        return bi;
    }
}
```
public final Rectangle2D getBounds2D(BufferedImage src) {
    return src.getRaster().getBounds();
}

public final Point2D (Point2D srcPt, Point2D dstPt) {
    if (dstPt == null)
        dstPt = new Point2d.Float();
    dstPt.setLocation(srcPt.getX(), srcPt.getY());
    return dstPt;
}

public final RenderingHints () {
    return null;
}