Richard G Baldwin (512) 223-4758, baldwin@austin.cc.tx.us,
http://www2.austin.cc.tx.us/baldwin/

The AWT Package, Graphics and Colors - An Overview

Java Programming, Lecture Notes # 162, Revised 02/06/98.

Preface

Students in Prof. Baldwin's **Advanced Java Programming** classes at ACC are responsible for knowing and understanding all of the material in this lesson.

Introduction

A previous lesson provided some sample programs which make use of the features of the **Graphics** class and the **Color** class without much in the way of explanation. The purpose of this lesson is to present an overview of the **Graphics** class and the **Color** class. Subsequent lessons will explore many aspects of these classes in depth.

The Graphics Class

The **Graphics** class is a large and complex class. It is the abstract base class that provides all, or at least most, of the functionality for an application to draw onto components and onto off-screen images as well.

A **Graphics** object encapsulates the following state information that is needed for basic rendering operations.

- The object of type **Component** on which to draw.
- A translation origin for rendering and clipping coordinates.
- The current clipped region.
- The current color.
- The current font.
- The current logical pixel operation function (XOR or Paint).
- The current XOR alternation color.

So, what happens when you use one of the **Graphics** methods to draw a line or a figure on a **Component**? Here are some of the facts according to the JavaSoft documentation from JDK 1.1.3:

"Coordinates are infinitely thin and lie between the pixels of the output device.

Operations which draw the outline of a figure operate by traversing an infinitely thin path between pixels with a pixel-sized pen that hangs down and to the right of the anchor point on the path.

Operations which fill a figure operate by filling the interior of that infinitely thin path.

Operations which render horizontal text render the ascending portion of character glyphs entirely above the baseline coordinate."

The fact that the graphics pen hangs down and to the right has some important implications:

If you draw a figure that covers a given rectangle, that figure occupies one extra row of pixels on the right and bottom edges as compared to filling a figure that is bounded by that same rectangle. We saw an example of this in a previous lesson where we drew a rectangle with the same dimensions as the **Canvas** object on which it was drawn, and discovered that the right and bottom borders of the rectangle hung off the edge of the object and were not visible.

Another implication is that if you draw a horizontal line along the same Y-coordinate as the baseline of a line of text, that line will be drawn entirely below the text, except for any descenders.

When you pass coordinates to the methods of a **Graphics** object, they are considered relative to the translation origin of the **Graphics** object prior to the invocation of the method.

A **Graphics** object describes a graphics context. A graphics context has a **current clip**. Any rendering operation that you perform will modify only those pixels which lie within the area bounded by the **current clip** of the graphics context and the component that was used to create the **Graphics** object.

When you draw or write, that drawing or writing is done in the **current color** using the **current paint mode** in the **current font**.

Numerous other classes, such as the **Rectangle** class and the **Polygon** class are used in support of operations involving the **Graphics** class.

Constructors

The **Graphics** class has a single constructor that takes no arguments. However, **Graphics** is an **abstract** class so your applications cannot call the constructor directly. You can obtain a **Graphics** context from another **Graphics** context by calling **getGraphics()** on a component.

You also receive a **Graphics** context as a parameter whenever you override either the **paint()** or **update()** methods.

Methods of the Graphics Class

The **Graphics** class has many methods. The following sections contain a representative sampling of those methods. I have attempted to group them into meaningful categories. A complete listing of the methods along with a description of each is contained in the JavaSoft documentation for the AWT.

Utility Methods for Graphics

clearRect(int, int, int, int) - Clears the specified rectangle by filling it with the background color of the current drawing surface.

copyArea(int, int, int, int, int, int) - Copies an area of the component specified by the first four parameters to another location on the graphics context at a distance specified by the last two parameters.

create() - Creates a new **Graphics** object that is a copy of the **Graphics** object on which it is invoked.

dispose() - Disposes of the graphics context on which it is invoked and releases any system resources that it is using. This includes system resources <u>other than memory</u>. A Graphics object cannot be used after dispose has been called. It is important that you manually dispose of your **Graphics** objects (created directly from a component or other **Graphics** object) when you no longer need them rather than to wait for finalization.

finalize() - Disposes of this graphics context once it is no longer referenced.

getColor() - Gets this graphics context's current color.

setColor(Color) - Sets this graphics context's current color to the specified color. Subsequent graphics operations using this graphics context use this specified color.

setPaintMode() - Sets the paint mode of this graphics context to overwrite the destination with this graphics context's current color (as opposed to XORMODE). Subsequent rendering operations will overwrite the destination with the current color.

setXORMode(Color) - Sets the paint mode of this graphics context to alternate between this graphics context's current color and the new specified color.

toString() - Returns a **String** object representing this **Graphics** object's value.

translate(int, int) - Translates the origin of the graphics context to the point (*x*, *y*) in the current coordinate system.

Drawing and Filling Shapes

drawLine(int, int, int, int) - Draws a line, using the current color, between two points in this graphics context's coordinate system.

drawPolyline(int[], int[], int) - Draws a sequence of connected lines defined by arrays of *x* and *y* coordinates. The figure will not be closed if the first point differs from the last point.

drawRect(int, int, int, int) - Draws the outline of the specified rectangle using the current color of the graphics context..

fillRect(int, int, int, int) - Fills the specified rectangle with the context's current color. Be sure to check the documentation regarding the coordinates of the right edge and bottom edge of the rectangle before using. This comment applies to all the fill methods.

drawRoundRect(int, int, int, int, int, int) - Draws an outlined round-cornered rectangle using this graphics context's current color. You might need to look at a book containing a diagram to learn how to specify how the corners are rounded.

fillRoundRect(int, int, int, int, int, int) - Fills the specified rounded corner rectangle with the current color.

draw3DRect(int, int, int, int, boolean) - Draws a 3-D highlighted outline of the specified rectangle. The edges of the rectangle are highlighted so that they appear to be beveled and lit from the upper left corner. The boolean parameter determines whether the rectangle appears to be raised above the surface or sunk into the surface. It is raised when the parameter is true.

fill3DRect(int, int, int, int, boolean) - Paints a 3-D highlighted rectangle filled with the current color.

drawOval(int, int, int, int) - Draws the outline of an oval in the current color. When the last two parameters are equal, this method draws a circle.

fillOval(int, int, int, int) - Fills an oval bounded by the specified rectangle with the current color. As with drawOval(), when the last two parameters are equal, the method fills a circle.

drawArc(int, int, int, int, int, int) - Draws the outline of a circular or elliptical arc covering the specified rectangle. You will probably need to examine the documentation to figure out how to specify the parameters for this method as well as the fillArc() method.

fillArc(int, int, int, int, int, int) - Fills a circular or elliptical arc covering the specified rectangle.

drawPolygon(Polygon) - Draws the outline of a polygon defined by the specified **Polygon** object. Another overloaded version is available that accepts a list of coordinate values to specify the polygon. The following description of a **Polygon** object was taken from the JavaSoft documentation for JDK 1.1.3.

"The Polygon class encapsulates a description of a closed, two-dimensional region within a coordinate space. This region is bounded by an arbitrary number of line segments, each of which is one side of the polygon. Internally, a polygon comprises of a list of (x, y) coordinate pairs, where each pair defines a vertex of the polygon, and two successive pairs are the endpoints of a line that is a side of the polygon. The first and final pairs of (x, y) points are joined by a line segment that closes the polygon."

fillPolygon(Polygon) - Fills the polygon defined by the specified Polygon object with the graphics context's current color. Another overloaded version is available that accepts a list of coordinate values to specify the polygon.

Drawing Text

drawString(String, int, int) - Draws the text given by the specified string, using this graphics context's current font and color.

drawChars(char[], int, int, int, int) - Draws the text given by the specified character array, using this graphics context's current font and color. Another version lets you pass an array of bytes to represent the characters to be drawn.

getFont() - Gets the current font and returns an object of type **Font** which describes the context's current font.

getFontMetrics() - Gets the font metrics of the current font. Returns an object of type **FontMetrics**. Methods of the **FontMetrics** class can be used to obtain metrics information (size, etc.) about the font to which the **getFontMetrics()** method is applied.

getFontMetrics(Font) - Gets the font metrics for the specified font.

setFont(Font) - Sets this graphics context's font to the specified font.

Clipping

Clipping is the process of defining a region of the **Graphics** context inside of which the pixels may be modified in some subsequent rendering process. Pixels outside the clip area are immune from modification.

clipRect(int, int, int, int) - Intersects the current clip with the specified rectangle. This results in a clipping area that is the intersection of the current clipping area and the specified rectangle. Future rendering operations have no effect outside of the clipping area. This method can only be used to reduce the size of the clipping area. It cannot be used to increase the size of the clipping area.

getClip() - Gets the current clipping area and returns it as an object of type **Shape**. Note that **Shape** is an interface. The following information and caution regarding the **Shape** interface was taken from the JavaSoft documentation for JDK 1.1.3:

Shape: The interface for objects which represent some form of geometric shape. This interface will be revised in the upcoming Java2D project. It is meant to provide a common interface for various existing geometric AWT classes and methods which operate on them. Since it may be superseded or expanded in the future, developers should avoid implementing this interface in their own classes until it is completed in a later release.

getClipBounds() - Returns the bounding rectangle of the current clipping area.

setClip(int, int, int, int) - Sets the current clip to the rectangle specified by the given coordinates.

Drawing Images

There are about six different **drawImage()** methods, differing primarily by the features of the method and the parameters passed in to support the features of that particular method. The following method is one of the more complex ones and is representative of the group. We will deal with the features of the methods and the associated parameters in the lessons on "Working with Images".

drawImage(Image, int, int, int, int, int, int, int, int, Color, ImageObserver) - 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.

The Color Class

The **Color** class is reasonably simple to understand and use. It encapsulates colors using the **RGB** format. This is a format in which a particular color is created by adding contributions from the primary colors: red, green, and blue.

In **RGB** format, the red, green, and blue components of a color are each represented by an integer in the range 0-255.

The value 0 indicates no contribution from the associated primary color. A value of 255 indicates the maximum intensity of the primary color component.

There is another color model called the **HSB** model (hue, saturation, and brightness). The **Color** class provides a set of convenience methods for converting between **RGB** and **HSB** colors.

Variables

The **Color** class provides a set of *static final* variables which make it convenient to specify any one of thirteen predefined colors.

All that is required to use these variables for the specification of a color is to reference the color by variable name as illustrated in the following code fragment:

```
object.setBackground(Color.red);
```

The list of predefined color values in the class is:

- black
- blue
- cyan
- darkGray
- gray
- green
- lightGray
- magenta
- orange
- pink
- red
- white
- yellow

Constructors

There are three constructors that you can use to instantiate a **Color** object. Two allow you to specify the contributions of red, green, and blue respectively as parameters. These two constructors instantiate a new **Color** object of the proper color.

One of these two constructors allows you to specify the contributions of red, green, and blue with integer values ranging between 0 and 255 where 0 represents no contribution of a particular primary color and 255 represents a maximum contribution of the primary color. The description of this constructor is:

Color(int, int, int) - Creates a color with the specified RGB components.

The other constructor in this category allows you to specify the contributions of the three primary colors using **float** values ranging between 0.0 and 1.0. The description of this constructor is:

Color(float, float, float) - Creates a color with the specified red, green, and blue values, where each of the values is in the range 0.0-1.0.

Finally, you can also construct your own **int** color value and pass it as a parameter to one of the three overloaded constructors. In this case, the format of the **int** must be as defined below:

Color(int) - Creates a color with the specified **RGB** value, where the red component is in bits 16-23 of the argument, the green component is in bits 8-15 of the argument, and the blue component is in bits 0-7.

Methods of the Color Class

As of JDK 1.1.3, the following methods are available for use with colors. I have attempted to group these methods into meaningful categories. A complete listing of the methods along with a description of each is contained in the JavaSoft documentation for the AWT.

Utility Methods for Colors

equals(Object) - Determines whether another object is equal to this color.

hashCode() - Computes the hash code for this color.

toString() - Creates a string that represents this color and indicates the values of its RGB components.

Investigate a Color Object

These methods return an integer that represents the composite RGB value of a particular **Color** object.

getRed() - Gets the red component of this color as an integer in the range 0 to 255.

getGreen() - Gets the green component of this color as an integer in the range 0 to 255.

getBlue() - Gets the blue component of this color as an integer in the range of 0 to 255.

getRGB() - Gets the RGB value representing the color. The red, green, and blue components of the color are each scaled to be a value between 0 and 255. Bits 24-31 of the returned integer are 0xff, bits 16-23 are the red value, bit 8-15 are the green value, and bits 0-7 are the blue value.

Create a Color Object

These methods return a **Color** object.

brighter() - Creates a brighter version of this color. This method was used in an earlier lesson where I created a fake button and used this method to provide highlighting on the edges.

darker() - Creates a darker version of this color. This method was used in an earlier lesson where I created a fake button and used this method to provide shadows on the edges.

decode(String) - Converts a string to an integer and returns the specified color.

For use with System Properties

These methods also return a **Color** object, but they are specialized to work with the System Properties.

getColor(String) - Finds a color in the system properties. The **String** object is used as the *key* value in the *key/value* scheme used to describe properties in Java. The *value* is then used to return a **Color** object.

getColor(String, Color) - Finds a color in the system properties. Same as the previous method except that the second parameter is returned if the first parameter doesn't result in a valid **Color** object.

getColor(String, int) - Finds a color in the system properties. Similar to the previous method except that the second parameter is used to instantiate and return a **Color** object.

HSB Methods

These methods are used to convert between the **RGB** color model and the **HSB** color model.

getHSBColor(float, float, float) - Creates a **Color** object based on values supplied for the HSB color model.

HSBtoRGB(float, float, float) - Converts the components of a color, as specified by the HSB model, to an equivalent set of values for the RGB model.

RGBtoHSB(int, int, int, float[]) - Converts the components of a color, as specified by the RGB model, to an equivalent set of values for hue, saturation, and brightness, the three components of the HSB model.

Summary

This lesson has presented an overview of the **Graphics** class and the **Color** class. Several subsequent lessons will be dedicated to expanding on the material presented here.

One lesson will be dedicated to working with some of the utility methods in the **Graphics** class, and specific lessons will be dedicated to working with shapes, text, clipping, and images.

In addition, the utility methods of the **Graphics** class and the methods and variables of the **Color** class will be used throughout those lessons.

-end-