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

The AWT Package, Graphics- The Utility Methods

Java Programming, Lecture Notes # 164, Revised 02/06/98.

- Preface
- Introduction
- Getting a Graphics Context
  - Sample Program to Get and Use a Graphics Context
- Copying and Clearing a Graphics Context
  - Sample Program to Illustrate Copying and Clearing
- Creating a new Graphics Object
  - Sample Programs to Illustrate Creating a new Graphics Object
- Using the **translate()** Method
  - Sample Program to Illustrate use of the **translate()** Method
- XOR Mode vs Paint Mode
  - Sample Program to Illustrate use of XOR and Paint modes
- Summary

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 an overview of the **Graphics** class, and grouped the methods of that class into several different categories. This lesson will explore some of the methods in the category of graphics utility methods.

To review, the following methods were put into the utility category:

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 other than memory. 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.

Getting a Graphics Context

What does it mean to "get a graphics context?" In layman's terms, this means that your application has gained the ability to draw or place images on a component that has the ability to support drawing and images.

According to Java Software Solutions by Lewis and Loftus:

"Each **Graphics** object represents a particular drawing surface. ... The **Graphics** object defines a *graphics context* through which we manage all graphic activities on that surface."

According to Just Java 1.1 and Beyond by Peter van der Linden:

"A **Graphics** object is what you ultimately draw lines, shapes, and text on. It is also called a "graphics context" in some window systems because it bundles together information about a drawable area, plus font, color, clipping region, and other situational factors."

Now that we know what a *graphics context* is, how do we get one?

To begin with, we don't get one by instantiating an object of type **Graphics**. The **Graphics** class cannot be directly instantiated by the code that you write for your application. Rather, we can get a graphics context in one of several indirect ways.

One way to get a graphics context is to invoke the **getGraphics()** method on another object. However, according to Java AWT Reference by John Zukowski:

"The **getGraphics()** method returns the image's graphics context. The method **getGraphics()** works only for **Image** objects created in memory with **Component.createImage(int, int)**."

The **getGraphics()** method is commonly used when images are being created in memory and then transferred to the screen (double buffering). We will see some examples of double buffering in a subsequent lesson.

This leaves us with two other ways to get a graphics context and they are surprisingly simple.

When you override either the **paint(Graphics g)** method or the **update(Graphics g)** method, the graphics context of the object on which the method is overridden is automatically passed in as a parameter.

We typically override the **paint()** method whenever we want to place graphics material on the screen. While it is possible to also override **update()** this is usually done only in special circumstances (such as animation and double buffering).

The normal approach to displaying graphics material is to place code in the overridden **paint()** method to do the job and then to invoke **repaint()** to ask the system to paint the new material on the screen. Note that the **paint()** method can also be invoked due to external causes (such as the user moving things around on the screen) totally outside the control of your program.

Hopefully the following three quotations from AWT Reference by John Zukowski will help to make this more clear.

public void repaint() - The **repaint()** method requests the scheduler to redraw the component as soon as possible. This will result in **update()** getting called soon thereafter. There is not a one-to-one correlation between **repaint()** and **update()** calls. It is possible that multiple **repaint()** calls can result in a single **update()** - Zukowski

public void update(Graphics g) - The **update()** method is automatically called when you ask to repaint the **Component**. If the component is not lightweight, the default implementation of **update()** clears graphics context **g** by drawing a filled rectangle in the background color, resetting the color to the current foreground color, and calling **paint()**. If you do not override

update() when you do animation, you will see some flickering because **Component** clears the screen. - Zukowski

public void paint(Graphics g) - The **paint()** method is offered so the system can display whatever you want in a **Component**. In the base **Component** class, this method does absolutely nothing. Ordinarily, it would be overridden in an applet to do something other than the default, which is display a box in the current background color. **g** is the graphics context of the component being drawn upon. - Zukowski

Sample Program to Get and Use a Graphics Context

So, let's see some code, and lets start out by making it simple from a graphics viewpoint. The following sample program illustrates getting and using a graphics context.

The **drawString()** method in the following sample program is invoked on the graphics context of a **Frame** object to display the string "**Hello World**".

When you compile and run this program, a **Frame** object will appear on the screen. The client area of the **Frame** object will display the words "Hello World".

When you press the close button on the **Frame** object, the program will terminate and control will be returned to the operating system.

The key item in this program, from a graphics viewpoint, is the overridden **paint()** method that draws the string data on the graphics context passed in as a parameter to **paint()**. That material is highlighted in **boldface** so that you can locate it easily.

```
/*File Graphics01.java
Copyright 1997, R.G.Baldwin

This program was tested using JDK 1.1.3 under Win95.

**********************************************************/
import java.awt.*;
import java.awt.event.*;

class Graphics01 extends Frame{ //controlling class

  //Override the paint method to display the string "Hello
  // World" on the graphics context of the Frame object.
  public void paint(Graphics g){
    g.drawString("Hello World",100,40);
  }//end paint()

  public Graphics01(){//constructor
    this.setTitle("Copyright 1997, R.G.Baldwin");
    this.setSize(350,50);
    this.setVisible(true);

    //Anonymous inner-class listener to terminate program
    this.addWindowListener(
      new WindowAdapter(){//anonymous class definition
        public void windowClosing(WindowEvent e){
          System.exit(0);//terminate the program
        }//end windowClosing()
      }//end WindowAdapter
    );//end addWindowListener

  }//end constructor

  public static void main(String[] args){
    new Graphics01();//instantiate this object
  }//end main
}//end Graphics01 class
//=========================================================//
```

What about that anonymous inner-class listener statement. Well, I told you that the program was going to be simple *from a graphics viewpoint*. At least the **paint()** method is simple. In case you don't know about inner-classes, you can learn about them in one of my earlier lessons.

Copying and Clearing in a Graphics Context

The two methods that are of primary interest in this section are **copyArea()** and **clearRect()**.

We'll begin with **clearRect()** because it is the simpler of the two. This method clears a specified rectangle by filling it with the background color of the current drawing surface. The method has four parameters as follows:

Parameters:
- **x** - the *x* coordinate of the rectangle to clear.
- **y** - the *y* coordinate of the rectangle to clear.
- **width** - the width of the rectangle to clear.

height - the height of the rectangle to clear.

As in virtually all graphics methods in Java, the **x** and **y** coordinates refer to the upper left-hand corner of the rectangular area to be cleared.

Now consider the method named **copyArea()**. This method copies a rectangular area of the current drawing surface to another area which is separated from the first by a distance specified by **dx** and **dy**. The method copies downwards and to the right. To copy an area of the drawing surface to the left or upwards, specify a negative value for **dx** or **dy**. Any portion of the source rectangle that is outside the current drawing surface won't be copied.

This method has six parameters as shown below.

Parameters:
- **x** - the *x* coordinate of the source rectangle.
- **y** - the *y* coordinate of the source rectangle.
- **width** - the width of the source rectangle.
- **height** - the height of the source rectangle.
- **dx** - the horizontal distance to copy the pixels.
- **dy** - the vertical distance to copy the pixels.

Sample Program to Illustrate Copying and Clearing

The following program illustrates the use of the **copyArea()** and **clearRect()** methods of the **Graphics** class.

This program draws the string "**Hello World**" in the upper left corner of a **Frame** object. Then it uses the **copyArea()** method to make two additional copies of the drawing by copying a rectangular area from the upper left corner to two other areas.

Then it uses the **clearRect()** method to erase most of the letter "H" from the second copy by clearing a rectangular portion of the screen that contains part of the drawing of the letter "H".

When you compile and run this program, a **Frame** object will appear on the screen. The client area of the **Frame** object will display the words **Hello World** in two different locations, and the same words with part of the "**H**" missing in another location.

When you press the close button on the **Frame** object, the program will terminate and control will be returned to the operating system.

A listing of the program follows with interesting code fragments highlighted in **boldface**.

```
/*File Graphics02.java
Copyright 1997, R.G.Baldwin

This program was tested using JDK 1.1.3 under Win95.

**********************************************************/
import java.awt.*;
import java.awt.event.*;

class Graphics02 extends Frame{ //controlling class

  //Override the paint method to display the string "Hello
  // World" on the graphics context of the Frame object.
  public void paint(Graphics g){
    g.drawString("Hello World",10,40);//draw the string
    g.copyArea(0,0,100,100,100,0); //copy to another spot
    g.copyArea(0,0,100,100,100,50); //copy to another spot
    g.clearRect(100,50,15,50); //erase part of second copy
  }//end paint()

  public Graphics02(){//constructor
    this.setTitle("Copyright 1997, R.G.Baldwin");
    this.setSize(350,150);
    this.setVisible(true);

    //Anonymous inner-class listener to terminate program
    this.addWindowListener(
      new WindowAdapter(){//anonymous class definition
        public void windowClosing(WindowEvent e){
          System.exit(0);//terminate the program
        }//end windowClosing()
      }//end WindowAdapter
    );//end addWindowListener
  }//end constructor

  public static void main(String[] args){
    new Graphics02();//instantiate this object
  }//end main
}//end Graphics02 class
//=========================================================//
```

Creating a new Graphics Object

According to some authors, the **create**() method creates a new **Graphics** object that is a copy of the **Graphics** object on which it is invoked. It might be more appropriate to say that it creates a second reference to the **Graphics** object on which it is invoked because figures drawn using the new reference appear on the original graphics context when it is rendered.

Whether it is a new object, or simply a second reference to the original object, we will see that in many ways, it behaves as though it is a second reference to the original object.

The primary purpose of this section is to illustrate the use of the **create()** method. Along the way, we will also illustrate a number of other concepts:

- The requirement to contend with *insets* when drawing on objects that have borders, such as **Frame** objects.
- The use of **clipping**.
- The use of **setColor()** to change the current drawing color.
- The use of the **dispose()** method to return *graphics context* resources to the operating system.

We will see two sample programs in the next section. The first program will deal with the *insets* problem by adding compensating offset values to coordinates as parameters to method calls. The second program will avoid the *insets* problem by overlaying the client area of a **Frame** object with a **Canvas** object for which there are no *insets*. (A sample program in a later section will eliminate the *insets* problem through use of the **translate()** method.

Also at this point we should say a few words about the use of the **dispose()** method. Since the different books seem to have different explanations for the need to dispose of the graphics contexts that you create, probably the best thing to do is simply to provide the following material extracted directly from the JavaSoft documentation for JDK 1.1.3

dispose()

Disposes of this graphics context and releases any system resources that it is using. A Graphics object cannot be used after dispose has been called.

When a Java program runs, a large number of Graphics objects can be created within a short time frame. Although the finalization process of the garbage collector also disposes of the same system resources, it is preferable to manually free the associated resources by calling this method rather than to rely on a finalization process which may not run to completion for a long period of time.

Graphics objects which are provided as arguments to the paint and update methods of components are automatically released by the system when those methods return. For efficiency, programmers should call dispose when finished using a Graphics object only if it was created directly from a component or another Graphics object.

Sample Programs to Illustrate Creating a new Graphics Object

The idea for the example programs in this section is based on an applet from the book entitled AWT Reference by John Zukowski. Mr. Zukowski designed his applet to explain the use of clipping in Java. It was modified here to include insets, color, etc.

The first program illustrates the following graphics concepts:

- Creation of a second reference to a graphics context.
- The requirement to contend with insets when drawing on objects that have borders, such as **Frame** objects.
- Drawing rectangles and lines.
- The use of **clipping**.
- The use of **setColor()** to change the current drawing color.
- The use of the **dispose()** method to return graphics context resources to the operating system.

The **paint()** method in an extended **Frame** class is overridden to perform the following tasks in order:

- Get the left and top insets of a **Frame** object so that they can be used as arithmetic offsets later when drawing in the **Frame** object.
- Set the drawing color to red.
- Create a second reference to the graphics context passed to the **paint()** method.
- Use the second reference to draw the outline of a red square 100 pixels on each side.

- Set the drawing color for the second reference to blue.
- Clip the drawing area for the second reference to a square 50 pixels on each side centered in the original square.
- Use the second reference to attempt to draw a blue line from the upper left-hand corner to the lower right-hand corner of the original rectangle. Only that portion of the line that is inside the new clipped drawing area will actually be drawn.
- Call the **dispose()** method to return the resources occupied by the second reference to the operating system.
- Set the second reference to **null** so that it will become eligible for *garbage collection*.
- Use the original graphics context to set the drawing color to green.
- Use the original graphics context to draw a green line from the lower left-hand corner to the upper right-hand corner of the original red rectangle. This entire line will be visible because the clipping area does not apply to the original graphics context.

When you compile and run this program, a **Frame** object will appear on the screen. A red square will appear in the upper left portion of the client area of the **Frame**.

A green line will run from the bottom left corner to the upper right corner within the square.

A segment of a blue line will appear as the center portion of an imaginary line that runs from the top left corner to the lower right corner within the square. This blue line segment is the result of clipping the line at the boundaries of the clipping area that was applied to the second reference to the graphics context.

When you press the close button on the **Frame** object, the program will terminate and control will be returned to the operating system.

This program was tested using JDK 1.1.3 under Win95.

A complete listing follows with interesting code fragments highlighted in **boldface**.

```
/*File Graphics03.java
Copyright 1997, R.G.Baldwin

This program was tested using JDK 1.1.3 under Win95.

**********************************************************/
import java.awt.*;
import java.awt.event.*;

class Graphics03 extends Frame{ //controlling class
  //Override the paint method
  public void paint(Graphics g){
    int top = this.getInsets().top;//get top inset value
    int left = this.getInsets().left;//get left inset value
    g.setColor(Color.red);

    //Create another reference to the Graphics context g
    Graphics clpArea = g.create();
    //Use original clpArea reference to draw a
    // red rectangle
    clpArea.drawRect(0+left,0+top,100,100);

    clpArea.setColor(Color.blue);

    //Reduce clpArea reference to rectangle shown
    clpArea.clipRect(25+left,25+top,50,50);

    //Use clpArea ref to try to draw a blue diagonal line
    // across the entire original rectangle.  Only middle
    // portion actually gets drawn.
    clpArea.drawLine(0+left,0+top,100+left,100+top);

    clpArea.dispose();//free system resources
    clpArea = null;//make eligible for garbage collection
    g.setColor(Color.green);

    //Use the original graphics context to draw a green
    // diagonal line across the entire rectangle.
    g.drawLine(0+left,100+top,100+left,0+top);
  }//end paint()

  public Graphics03(){//constructor
    this.setTitle("Copyright 1997, R.G.Baldwin");
    this.setSize(300,150);
    this.setVisible(true);

    //Anonymous inner-class listener to terminate program
    this.addWindowListener(
      new WindowAdapter(){//anonymous class definition
        public void windowClosing(WindowEvent e){
          System.exit(0);//terminate the program
        }//end windowClosing()
      }//end WindowAdapter
    );//end addWindowListener
  }//end constructor

  public static void main(String[] args){
    new Graphics03();//instantiate this object
  }//end main
}//end Graphics03 class
//=======================================================//
```

The second program illustrates the use of a **Canvas** object as the *drawing surface* in order to eliminate the *insets* problem. Otherwise, it is very similar to the first program.

Use of the **Canvas** object as a drawing surface eliminates the *insets* problem simply because a **Canvas** object doesn't have any borders.

The *insets* result from the fact that some containers, such as a **Frame** object, have borders, and the area covered by the borders is considered to be a part of the drawing surface, insofar as coordinate values are concerned.

In other words, the 0,0 coordinate position is the upper left-hand corner of the container, outside the borders if it has borders.

The **getInsets()** method provides the width in pixels of the four borders which makes it possible to compensate arithmetically for the borders when working with coordinate values.

In order to be able to use a **Canvas** object as a drawing surface, it is necessary to extend the **Canvas** class so that the **paint()** method can be overridden. In the following program, the **Canvas** class was extended into a new class named **MyClass** where the **paint()** method was overridden to perform the graphics operations.

An object of the **MyCanvas** class was instantiated, made yellow, and added to the **Frame** object in such a way as to cover the entire client area of the **Frame** object and act as a drawing surface on top of the **Frame** object. Since the **MyCanvas** object has no borders, the *insets* problem was eliminated.

As with the first program, the idea for this program was based on an applet from the book entitled AWT Reference by John Zukowski.

A complete listing of the program follows with interesting code fragments highlighted in **boldface**.

```
/*File Graphics04.java
Copyright 1997, R.G.Baldwin

This sample program replicates the functionality of the
program named Graphics03.  However, it was modified to
eliminate the nuisance of having to contend with insets
when drawing on a Frame object.

The requirement to contend with insets was eliminated by
adding a Canvas object to the Frame object and drawing on
the Canvas object instead of on the Frame object.  A
Canvas object doesn't have borders, so there are no insets
to contend with.

This program was tested using JDK 1.1.3 under Win95.

**********************************************************/
import java.awt.*;
import java.awt.event.*;

//Extend Canvas in order to make it possible to override
// the paint() method.
class MyCanvas extends Canvas{
  //Override the paint method
  public void paint(Graphics g){
    g.setColor(Color.red);

    //Create another reference to the Graphics context g
    Graphics clpArea = g.create();
    //Use original clpArea reference to draw a
    // red rectangle
    clpArea.drawRect(0,0,100,100);
    clpArea.setColor(Color.blue);

    //Reduce clpArea reference to rectangle shown
    clpArea.clipRect(25,25,50,50);

    //Use clpArea ref to try to draw a blue diagonal line
    // across the entire original rectangle.  Only middle
    // portion actually gets drawn.
    clpArea.drawLine(0,0,100,100);

    clpArea.dispose();//free system resources
    clpArea = null;//make eligible for garbage collection
    g.setColor(Color.green);

    //Use the original graphics context to draw a green
    // diagonal line across the entire rectangle.
    g.drawLine(0,100,100,0);
  }//end paint()

}//end class MyCanvas

//=========================================================//
class Graphics04 extends Frame{ //controlling class

  public Graphics04(){//constructor
    this.setTitle("Copyright 1997, R.G.Baldwin");
    this.setSize(300,150);

    //Create a yellow drawing surface and use it to cover
    // the client area of the Frame object.
    MyCanvas myDrawingSurface = new MyCanvas();
    myDrawingSurface.setBackground(Color.yellow);
    this.add(myDrawingSurface);
    this.setVisible(true);

    //Anonymous inner-class listener to terminate program
    this.addWindowListener(
      new WindowAdapter(){//anonymous class definition
        public void windowClosing(WindowEvent e){
          System.exit(0);//terminate the program
        }//end windowClosing()
      }//end WindowAdapter
    );//end addWindowListener
  }//end constructor

  public static void main(String[] args){
    new Graphics04();//instantiate this object
  }//end main
}//end Graphics04 class

//=========================================================//
```

Using the translate() Method

The **translate(int x, int y)** method translates the origin of the graphics context to the point (**x**, **y**) in the current coordinate system. The method modifies the graphics context so that its new origin corresponds to the point (**x**, **y**) in this graphics context's original coordinate system.

All coordinates used in subsequent rendering operations on the graphics context (or a copy or second reference to the graphics context made after the translation takes place) will be relative to the new origin.

Sample Program to Illustrate Use of the translate() Method

The **translate()** method provides another useful way to eliminate the *insets* problem. In particular, the method can be used to translate the origin of the graphics context to the upper left-hand corner of the client area of the **Frame** object (inside the borders).

The following program replicates the functionality of the program named **Graphics03**. However, it eliminates the problem of *insets* by invoking the **translate()** method on the original graphics context to shift the origin to the upper left- hand corner of the the client area of the **Frame** object (inside the borders).

The program illustrates the same concepts as previous programs, and in addition illustrates the use of the **translate()** method to shift the 0,0 coordinate position to a different spot on the graphics context.

Minimal changes were required to the original program named **Graphics03** to implement this solution to the *insets* problem. Those changes are highlighted in **boldface** in the program listing that follows.

```
/*File Graphics05.java
Copyright 1997, R.G.Baldwin

This program was tested using JDK 1.1.3 under Win95.

**********************************************************/
import java.awt.*;
import java.awt.event.*;

class Graphics05 extends Frame{ //controlling class
  //Override the paint method
  public void paint(Graphics g){
    g.setColor(Color.red);

    //Translate the 0,0 coordinate of the graphics context
    // to the upper left-hand corner of the client area of
    // the Frame object.
    g.translate(
               this.getInsets().left,this.getInsets().top);
    //Create another reference to the Graphics context g
    Graphics clpArea = g.create();

    //Use original clpArea reference to draw a
    // red rectangle
    clpArea.drawRect(0,0,100,100);
    clpArea.setColor(Color.blue);

    //Reduce clpArea reference to rectangle shown
    clpArea.clipRect(25,25,50,50);

    //Use clpArea ref to try to draw a blue diagonal line
    // across the entire original rectangle.  Only middle
    // portion actually gets drawn.
    clpArea.drawLine(0,0,100,100);

    clpArea.dispose();//free system resources
    clpArea = null;//make eligible for garbage collection
    g.setColor(Color.green);

    //Use the original graphics context to draw a green
    // diagonal line across the entire rectangle.
    g.drawLine(0,100,100,0);
  }//end paint()

  public Graphics05(){//constructor
    this.setTitle("Copyright 1997, R.G.Baldwin");
    this.setSize(300,150);
    this.setVisible(true);

    //Anonymous inner-class listener to terminate program
    this.addWindowListener(
      new WindowAdapter(){//anonymous class definition
        public void windowClosing(WindowEvent e){
          System.exit(0);//terminate the program
        }//end windowClosing()
      }//end WindowAdapter
    );//end addWindowListener
  }//end constructor

  public static void main(String[] args){
    new Graphics05();//instantiate this object
  }//end main
}//end Graphics05 class
//=======================================================//
```

.

XOR Mode vs Paint Mode

This section of this lesson explains "**how** to do it" and not "**why** to do it". If you don't already know that you need to render your drawing in **XOR** *(exclusive or)* mode, you probably don't need to render it in **XOR** mode.

The **Paint** mode that results from invoking the **setPaintMode()** method is easy to explain. Each new pixel that you render simply replaces the existing pixel with the color of the new pixel.

However, the **XOR** mode that results from invoking the **setXORMode()** method is much more complex and requires an explanation.

In this section, we will look at some specific examples of performing an *exclusive or* on selected bit patterns. The results will be important in understanding the programming example in the next section.

The rules for the determining the *exclusive or* of two bits are shown in the following box:

```
0 or 0 = 0
1 or 0 = 1
0 or 1 = 1
1 or 1 = 0
```

In other words, if either but not both of the bits is a 1, the output is a 1. Otherwise, the output is a 0.

The following box shows the bit patterns for the 24 bits that comprise the color portion of four different Java RGB color values (the remaining eight bits are not shown).

```
111111110000000000000000 = red
000000001111111100000000 = green
000000000000000011111111 = blue
111111111111111111111111 = white
```

When you invoke the **setXORMode(Color altColor)** method, the color for each pixel that is rendered thereafter is determined by the **XOR** of three color values:

- the current drawing color,
- the **altColor** value passed as a parameter to the **setXORMode()** mode, and
- the current color of the pixel.

Consider the following **XOR** example of rendering a new *red* pixel on an existing *red* pixel where the **XOR altColor** value has been set to *green*. The three color values are shown below along with the intermediate and final **XOR** of the three.

```
111111110000000000000000 = red
111111110000000000000000 = red
000000000000000000000000 = intermediate result
000000001111111100000000 = green
000000001111111100000000 = final result which is green
```

As you can see, the **XOR Mode** result of drawing *red* on *red* produces the value of the third color which in this case is the **altColor** value passed to the **setXORMode()** method.

Thus, if we are in **XOR Mode** and draw *red* on *red*, the actual color rendered will be the color that was passed to the **setXORMode()** method when it was invoked. This will be true when drawing any color on top of the same color.

Now consider the following **XOR** example of rendering a *red* pixel on a *white* pixel where the **XOR altColor** value has been set to *green*. The three color values are shown below along with the intermediate and final **XOR** of the three.

```
111111110000000000000000 = red
111111111111111111111111 = white
000000001111111111111111 = intermediate result
000000001111111100000000 = green
000000000000000011111111 = final result which is blue
```

As you can see, the **XOR** of *red*, *white*, and *green* produces the color value for *blue*. Using this scheme, you should be able to determine the numeric value of the color value that will be produced for the **XOR** of any three colors.

One of the reasons for using **XOR** rendering is the fact that when a figure is redrawn on itself in **XOR** mode, the result is to erase the figure and restore the background to its state prior to the drawing of the figure in the first place. This is true even for multi-colored backgrounds. This is sometimes used in animation processes as a way to draw and erase a figure very quickly. With this concept in mind, consider the process of performing the same **XOR** two times in succession. Use the *red-red-green* case as an example.

```
111111110000000000000000 = red
111111110000000000000000 = red
000000000000000000000000 = intermediate result
000000001111111100000000 = green
000000001111111100000000 = final result which is green

Now starting with green, XOR the same red pixel

000000001111111100000000 = green
111111110000000000000000 = red
111111111111111100000000 = intermediate result
000000001111111100000000 = green
111111110000000000000000 = red which is the original color of the pixel
```

As you can see, the final color of the pixel is the same that it was before the new pixel color was rendered twice in succession. This is as we would expect based on the previous discussion of the figure erasing itself on the second rendering in **XOR** mode.

Sample Program to Illustrate use of XOR and Paint Modes

This section contains a sample program that matches the **XOR** bit-pattern examples given above.

This program illustrates the use of **setXORMode()** and **setPaintMode()**. It shows the result of overlapping drawings in both modes. It also shows the result of redrawing a figure that was originally drawn in **XOR** mode.

It is strongly recommended that you compile and run this program because you will probably need to see the display to understand the following description.

This program draws <u>two sets</u> of four overlapping filled squares at different locations on the screen with a drawing color of red. The current drawing color is not changed during the entire sequence of drawing squares.

For both sets, the first two overlapping squares are drawn in the default **Paint** mode. This produces two red squares which overlap and merge in the overlapping area. Except for the fact that you know they are squares (because I told you so), it is not possible to discern the shape of the overlapping area. If they were overlapping polygons, for example, you would not be able to discern the shape of the overlapping area.

Then the mode is changed to **XOR** with an **XOR altColor** value of green and a third overlapping square is drawn with the current drawing color being red.

In this case, <u>the overlap between the two squares is green</u>. (You should have expected this based on the discussion in the previous section.) Thus, it is now possible to discern the shape of of the overlapping area.

In addition, that portion of the square that doesn't overlap the red square but is drawn on the white background is rendered in blue. (Again, you should have expected this on the basis of the discussion in the previous section.)

Then the mode is reset to **Paint** and a fourth overlapping square is drawn. It simply overdraws the blue square with red in the intersecting area as would be expected for the **Paint** mode.

To demonstrate the manner in which drawing the same figure twice in the **XOR** mode causes it to be erased, the same sequence is repeated again in a location further to the right on the screen.

However, in this case, after all four squares have been drawn, the mode is set to **XOR** and another square is drawn in the exact location of the third square in the sequence (the one previously drawn in **XOR** mode). In other words, the third square is redrawn in **XOR** mode.

This causes the green and blue portions of that square to be replaced by red and white, effectively erasing the square and returning the display to its original form (as you should expect).

However, that portion of the third square that was previously overlapped by the fourth square (causing it to be red instead of blue), was rendered as green because the fourth square wasn't there when the third square was originally drawn. Therefore, redrawing it causes the redrawn square to overlap the fourth square and produce the green overlap area.

A program listing follows. Interesting code fragments are highlighted in **boldface**.

```
/*File Graphics06.java
Copyright 1997, R.G.Baldwin

This program was tested using JDK 1.1.3 under Win95.

**********************************************************/
import java.awt.*;
import java.awt.event.*;

class Graphics06 extends Frame{ //controlling class
  //Override the paint method
  public void paint(Graphics g){
    g.setColor(Color.red);

    //Translate the 0,0 coordinate of the graphics context
    // to the upper left-hand corner of the client area of
    // the Frame object.
    g.translate(
               this.getInsets().left,this.getInsets().top);

    //Draw first set of four overlapping filled red
    // squares.  Start drawing in the default Paint mode.
    g.fillRect(0,0,50,50);
    g.fillRect(25,25,50,50);
    //Set to XOR mode and draw another overlapping square
    // with the drawing color set to red and the XOR color
    // set to green.  This will produce a square that is
    // green where it overlaps a red square and is blue
    // where it doesn't overlap the red square but is
    // being drawn on the white background.
    g.setXORMode(Color.green);
    g.fillRect(50,50,50,50);
    //Reset to default Paint mode and draw another
    // overlapping square. This will simply draw a red
    // square covering part of the blue square and covering
    // the white background
    g.setPaintMode();
    g.fillRect(75,75,50,50);

    //Now demonstrate the cancelling effect of redrawing
    // a square in XOR mode.
    //Draw second set of four overlapping filled red
    // squares exactly as before but at a different
    // location on the screen.
    g.fillRect(200,0,50,50);
    g.fillRect(225,25,50,50);
    g.setXORMode(Color.green);
    g.fillRect(250,50,50,50);
    g.setPaintMode();
    g.fillRect(275,75,50,50);
    //***Important concept demonstrated here ***
    //Now redraw the third square in the second set
    // in XOR mode.  This will erase the one originally
    // drawn except where it overlaps the fourth square.
    // That overlap will be green.
    g.setXORMode(Color.green);
    g.fillRect(250,50,50,50);

  }//end paint()

  public Graphics06(){//constructor
    this.setTitle("Copyright 1997, R.G.Baldwin");
    this.setSize(350,200);
    this.setVisible(true);

    //Anonymous inner-class listener to terminate program
    this.addWindowListener(
      new WindowAdapter(){//anonymous class definition
        public void windowClosing(WindowEvent e){
          System.exit(0);//terminate the program
        }//end windowClosing()
      }//end WindowAdapter
    );//end addWindowListener
  }//end constructor

  public static void main(String[] args){
    new Graphics06();//instantiate this object
  }//end main
}//end Graphics06 class
//=========================================================//
```

Summary

This lesson has explained and illustrated the use of the following methods from the original list of **Graphics** utility methods given at the beginning of the lesson:

- **clearRect**(int, int, int, int)
- **copyArea**(int, int, int, int, int, int)
- **create**()
- **dispose**()
- **setColor**(Color)
- **translate**(int, int)
- **setPaintMode**()
- **setXORMode**(Color)

We did not illustrate or explain the use of the following methods:

- **toString**()
- **getColor**()
- **finalize**()

The **toString()** method is simply overridden to return a **String** object that describes a **Graphics** object. We have seen many examples of the overridden **toString()** method in earlier lessons.

The **getColor()** method is the flip side of the **setColor()** method and should not be difficult for you to understand on your own.

The **finalize()** method is overridden to dispose of the **Graphics** object prior to garbage collection. However, as explained above, you should manually dispose of the **Graphics** objects that you create and not wait for finalization to take place.

-end-