***Chapter 15***

## *Graphics in Java*

The Abstract Window Toolkit (AWT) class was introduced at the beginning of the semester because it provides support for applets. The AWT contains numerous classes and methods that allow you to create and manage windows. A full description of the AWT could easily consume the entire semester. Therefore, it is not possible to describe in detail every method, instance variable, or class here. The best I can do is to expose you to some of the most fundamental components that enable graphics generation in Java.

The AWT classes are contained in the java.awt package. The following table shows these classes. We will be discussing these classes during the remaining portion of this semester.

|  |  |
| --- | --- |
| **Class** | **Description** |
| BorderLayout | The border layout manager (uses five components: North, South, East, West & Center) |
| Button | Creates a push button control. |
| Canvas | A blank, semantics-free window. |
| CardLayout | The card layout manager which emulates index cards (only the one on top shows). |
| Checkbox | Creates a check box control. |
| CheckboxGroup | Creates a group of check box controls. |
| Choice | Creates a pop-up list. |
| Color | Manages colors in a portable, platform-independent fashion. |
| Component | An abstract superclass for various AWT components |
| Container | An abstract subclass of **Component** that can hold other components |
| Dialog | Creates a dialog window. |
| Dimension | Specifies the dimensions of an object (width is stored in **width**, height store in **height**) |
| Event | Encapsulates events. |
| FileDialog | Creates a window from which a file can be selected. |
| FlowLayout | The flow layout manager. Flow layout positions components left to right, top to bottom. |
| Font | Encapsulates a type font. |
| FontMetrics | Encapsulates various info related to a font (useful when displaying text in a window). |
| Frame | Creates a standard window that has a title bar, resize corners, and a menu bar. |
| Graphics | Encapsulates the graphics context enabling output to be displayed in a window. |
| GridBagConstraints | Defines various constraints relating to the **GridBagLayout** class. |
| GridBagLayout | The grid bag layout manager subject to constraints specified by **GridBagConstraints**. |
| GridLayout | The grid layout manager. Grid layout displays components in a two-dimensional grid. |
| Image | Encapsulates graphical images. |
| Insets | Encapsulates the borders of a container. |
| Label | Creates a label that displays a string. |
| List | Creates a list from which the user can choose (similar to the standard windows list box) |
| MediaTracker | Manages media objects. |
| Menu | Creates a pull-down menu. |
| MenuBar | Creates a menu bar. |
| MenuComponent | An abstract class implemented by various menu classes. |
| MenuItem | Creates a menu item. |
| Panel | The simplest concrete subclass of **Container**. |
| Point | Encapsulates a Cartesian coordinate pair, stored in **x** and **y**. |
| Polygon | Encapsulates a polygon. |
| Rectangle | Encapsulates a rectangle. |
| Scrollbar | Creates a scroll bar control. |
| TextArea | Creates a multiline edit control. |
| TextComponent | A superclass for **TextArea** and **TextField**. |
| TextField | Creates a single-line edit control. |
| Toolkit | Abstract class implemented by the AWT |
| Window | Creates a window with no frame, no menu bar, and no title. |

*Working with Graphics*

The AWT support a rich assortment of graphics methods. All graphics are drawn relative to a window. This can be the main window of an applet, a child window of an applet, or a stand-alone application window. The origin of each window is at the top left corner and is 0,0. Coordinates are specified in pixels. All output to a window takes place through a *graphics context*. A graphics context enables drawing on the screen in Java. A **Graphics** object manages a graphics context by controlling how information is drawn. A graphics context is encapsulated by the **Graphics** class and is obtained in two ways:

* It is passed to an applet when one of its various methods, such as **paint( )** or **update( )** is called.
* It is returned by the **getGraphics( )** method.

Though the following examples demonstrate graphics in a main applet window, the same techniques apply to any other window.

The **Graphics** class defines a number of drawing functions. Each shape can be drawn edge-only or filled. Objects are drawn and filled in the currently selected graphics color, which is black by default. When a graphics object is drawn that exceeds the dimensions of the window, output is automatically clipped. Let's take a look at several of the drawing methods.

*Drawing Lines*

Lines are drawn by the means of the **drawLine( )** method, shown here:

void drawLine(int *startX*, int *startY*, int *endX*, int *endY*)

**drawLine( )** displays a line in the current drawing color that begins at *startX*, *startY* and ends at *endX*, *endY*.

// Draw lines

import java.awt.\*;

import java.applet.\*;

/\*

<applet code="Lines" width=300 height=200>

</applet>

\*/

public class Lines extends Applet {

public void paint (Graphics g) {

g.drawLine(0, 0, 100, 100);

g.drawLine(0, 100, 100, 0);

g.drawLine(40, 25, 250, 180);

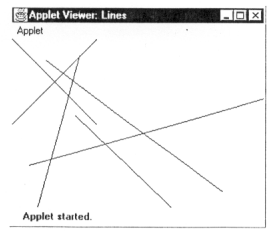
g.drawLine(75, 90, 400, 400);

g.drawLine(20, 150, 400, 40);

g.drawLine(5, 290, 80, 19);

}

} // Sample output from this program is shown here:



*Drawing Rectangles*

The drawRect( ) and filRect( ) methods display an outlines and filled rectangle, respectively. They are shown here:

void drawRect(int *top*, int *left*, int *width*, int *height*)

void fillRect(int *top*, int *left*, int *width*, int *height*)

The upper-left corner of the rectangle is at *top*, *left*. The dimensions of the rectangle are specified by *width* and *height*.

To draw a rounded rectangle, use drawRoundRect( ) or fillRoundRect( ), both are shown here:

void drawRoundRect (int *top*, int *left*, int *width*, int *height*,

int *xDiam*, int *yDiam*)

void fillRoundRect (int *top*, int *left*, int *width*, int *height*,

int *xDiam*, int *yDiam*)

A rounded rectangle has rounded corners. The upper-left corner of the rectangle is at *top*, *left*. The dimensions of the rectangle are specified by width, and height. The diameter of the rounding arc along the X axis is specified by *xDiam*. The diameter of the rounding arc along the Y axis is specified by *yDiam*. The following applet draws several rectangles:

// Draw rectangles

import java.awt.\*;

import java.applet.\*;

/\*

<applet code="Rectangles" width=300 height=200>

</applet>

\*/

public class Rectangles extends Applet {

public void paint(Graphics g) {

g.drawRect(10, 10, 60, 50);

g.fillRect(100, 10, 60, 50);

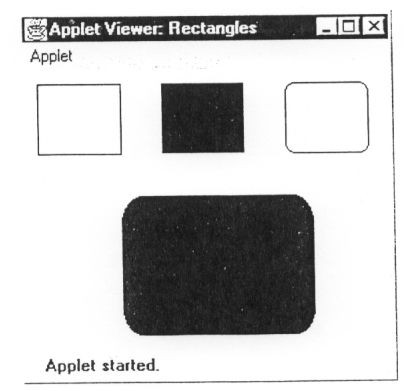
g.drawRoundRect(190, 10, 60, 50, 15, 15);

g.fillRoundRect(70, 90, 140, 100, 30, 40);

}

}

Sample output from this program is shown here:

*Drawing Ellipses and Circles*

To draw an ellipse, use **drawOval( )**. To fill an ellipse, use **fillOval( )**. These methods are shown here:

void drawOval(int *top*, int *left*, int *width*, int *height*)

void fillOval(int *top*, int *left*, int *width*, int *height*)

The ellipse is drawn within a bounding rectangle whose upper-left corner is specified by *top*, *left* and whose width and height are specified by *width* and *height*. To draw a circle, specify a square as the bounding rectangle. The following program draws several ellipses:

// Draw Ellipses

import java.awt.\*;

import java.applet.\*;

/\*

<applet code="Ellipses" width=300 height=200>

</applet>

\*/

public class Ellipses extends Applet {

public void paint(Graphics g) {

g.drawOval (10, 10, 50, 50);

g.fillOval (100, 10, 75, 50);

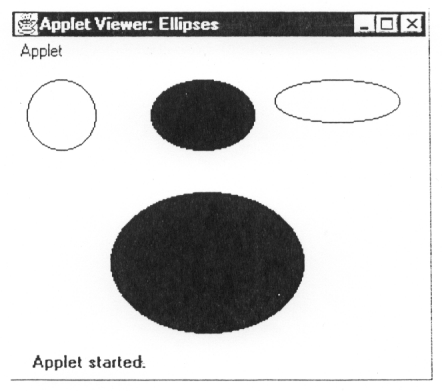
g.drawOval (190, 10, 90, 30);

g.fillOval (70, 90, 140, 100);

}

}

Sample output from this program is shown here:

*Drawing Arcs*

Arcs can be drawn with **drawArc( )** and **fillArc( )**, shown here:

void drawArc(int *top*, int *left*, int *width*, int *height*, int *startAngle*,

int *sweepAngle*)

void fillArc(int *top*, int *left*, int *width*, int *height*, int *startAngle*,

int *sweepAngle*

The arc is bounded by the rectangle whose upper-left corner is specified by *top*, *left* and whose width and height are specified by *width* and *height*. The arc is drawn from *startAngle* through the angular distance specified by *sweepAngle*. Angles are specified in degrees. Zero degrees is on the horizontal, at the 3 o'clock position. The arc is drawn counterclockwise if *sweepAngle* is positive, and clockwise if *sweepAngle* is negative. Therefore, to draw an arc from 12:00 to 6:00, the start angle would be 90, and the sweep angle, 180.

The following applet draws several arcs:

// Draw Arcs

import java.awt.\*;

import java.applet.\*;

/\*

<applet code="Arcs" width=300 height=200>

</applet>

\*/

public class Arcs extends Applet {

public void paint(Graphics g) {

g.drawArc (10, 40, 70, 70, 0, 75);

g.fillArc (100, 40, 70, 70, 0, 75);

g.drawArc (10, 100, 70, 80, 0, 175);

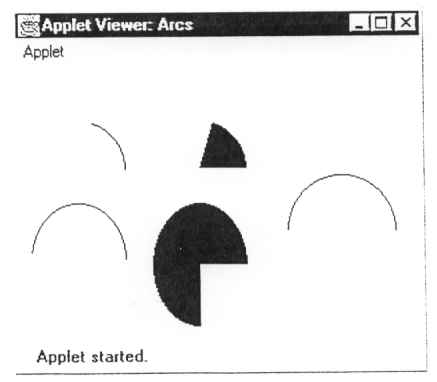
g.fillArc (100, 100, 70, 90, 0, 270);

g.drawArc (200, 80, 80, 80, 0, 180);

}

}

Sample output from this program is shown here:

*Drawing Polygons*

It is possible to draw arbitrarily shaped figures using drawPolygon( ) and fillPolygon( ) shown here:

void drawPolygon(int *x[ ]*, int *y[ ]*, int *numPoints*)

void fillPolygon(int *x[ ]*, int *y[ ]*, int *numPoints*)

The polygon's endpoints are specified by the coordinate pairs contained within the *x* and *y* arrays. The number of points defined by *x* and *y* is specified by *numPoints*. There are alternative forms of these methods in which the polygon is specified by a **Polygon** object.

The following applet draws an hourglass shape:

// Draw Polygon

import java.awt.\*;

import java.applet.\*;

/\*

<applet code="HourGlass" width=230 height=210>

</applet>

\*/

public class HourGlass extends Applet {

public void paint(Graphics g) {

int xpoints[ ] = {30, 200, 30, 200, 30};

int ypoints[ ] = {30, 30, 200, 200, 30};

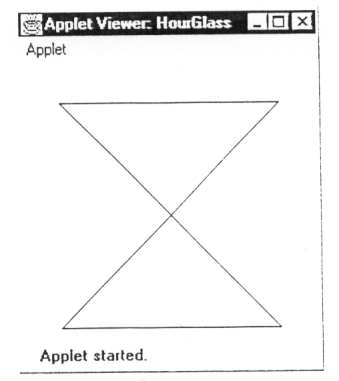
int num = 5;

g.drawPolygon(xpoints, ypoints, num);

}

}

Sample output from this program is shown here:

*Sizing Graphics*

Often, you will want to size a graphics object to fit the current size of the window in which it is drawn. To do so, first call **size( )** on the window object. It is shown here: **Dimension size( )**

It returns the dimensions of the window encapsulated within a Dimension object. Once you have the current size of the window, you can scale your graphical output accordingly.

To demonstrate this technique, here is an applet that will start as a 200x200 pixel square, and grow by 25 pixels in width and height with each mouse click until the applet gets larger than 500x500. At that point, the next click will return it to 200x200 and the process starts over. Within the window, a rectangle is drawn so that it fills the window. This applet uses the **mouseUp( )** event method (to be discussed immediately after the midterm).

// Resizing output to fit the current size of a window.

import java.applet.\*;

import java.awt.\*;

/\*

<applet code="ResizeMe" width=200 height=200>

</applet>

\*/

public class ResizeMe extends Applet {

final int inc = 25;

int max = 500;

int min = 200;

Dimension d;

public void paint (Graphics g) {

d = size( );

g.drawLine(0, 0, d.width-1, d.height-1);

g.drawLine(0, d.height-1, d.width-1, 0);

g.drawRect(0, 0, d.width-1, d.height-1);

}

public boolean mouseUp(java.awt.Event evt, int x, int y) {

int w = (d.width + inc) > max?min : (d.width + inc);

int h= (d.height + inc) > max?min : (d.height + inc);

resize(w, h);

return true;

}

}

*Working with Color*

Java supports color in a portable, device-independent fashion. The AWT color system allows you to specify any color you want. It then finds the best match for that color, given the limits of the display hardware currently executing your program or applet. Thus, your code does not need to be concerned with the differences in the way color is supported by various hardware devices. Color is encapsulated by the **Color** class.

Color defines several constants (for example **Color.black**, **Color.blue** etc.) to specify a number of common colors. You can also create your own colors, using one of the color constructors, shown here:

Color(int *red*, int *green*, int *blue*)

or

Color(float *red*, float *green*, float *blue*)

The 1st constructor takes three integers that specify the color as a mix of red, green, and blue. These values must be between 0 and 255, as in this example:

new Color(255, 100, 100) // light red.

The 2nd color constructor takes three float values (between 0.0 and 1.0) that specify the relative mix of red, green, and blue.

Once you have created a color, you can use it to set the foreground and/or background color by using the **setForeground( )** and **setBackground( )** methods.

The Color class defines several methods that help manipulate colors. You can obtain the red, green, and blue components of a color independently using **getRed( )**, **getGreen( )**, and **getBlue( )**, shown here:

int getRed( )

int getGreen( )

int getBlue( )

Each of these methods returns the RGB color component found in the invoking **Color** object into the lower 8 bits of an integer.

By default, graphics objects are drawn in the current foreground color. You can change this color by calling the Graphics method **setColor( )**:

void setColor(Color *newColor*)

Here, *newColor* specifies the new drawing color. You can obtain the current color by calling **getColor( )**, shown here:

Color getColor( )

// Demonstrate color.

import java.awt.\*;

import java.applet.\*;

/\*

<applet code="ColorDemo" width=300 height=200>

</applet>

\*/

public class ColorDemo extends Applet {

// draw lines

public void paint(Graphics g) {

Color c1 = new Color(255, 100, 100);

Color c2 = new Color(100, 255, 100);

Color c3 = new Color(100, 100, 255);

g.setColor(c1);

g.drawLine(0, 0, 100, 100);

g.drawLine(0, 100, 100, 0);

g.setColor(c2);

g.drawLine(40, 25, 250, 180);

g.drawLine(75, 90, 400, 400);

g.setColor(c3);

g.drawLine(20, 150, 400, 40);

g.drawLine(5, 290, 80, 19);

g.setColor(Color.red);

g.drawOval(10, 10, 50, 50);

g.fillOval(70, 90, 140, 100);

g.setColor(Color.blue);

g.drawOval(190, 10, 90, 30);

g.drawRect(10, 10, 60, 50);

g.setColor(Color.blue);

g.drawOval(190, 10, 90, 30);

g.drawRect(10, 10, 60, 50);

g.setColor(Color.cyan);

g.fillRect(100, 10, 60, 50);

g.drawRoundRect(190, 10, 60, 50, 15, 15);

}

}

*Foreground & Background Colors*

To set the background color of an applet's window, use **setBackground( )**. To set the foreground color (the color in which text is shown, for example), use **setForeground( )**. These methods are defined by **Component**, and they have the following general forms:

void setBackground(Color *newColor*)

void setForeground(Color *newColor*)

The following sets the background to green and foreground to red:

setBackground(Color.green);

setForeground(Color.red);

A good place to set the foreground and background colors is in the **init( )** method. Of course, you can change these colors as often as necessary during the execution of your applet. The default foreground color is black. The default background color is light gray.

**Assignment for Chapter 15**

Write an applet that simulates a screen saver. The applet should randomly draw lines using method **drawLine** of class **Graphics**. After drawing 100 lines, the applet should clear itself and start drawing lines again. To allow the program to draw continuously, place a call to **repaint** as the last line in the method **paint**.

Some issues for you to consider:

1. The **update** method automatically clears the screen before calling **paint**. You will need to override this method so that it simply calls **paint**.
2. When you want to erase the screen (i.e., the line counter gets to 100), consider calling **super.update**.
3. You must not have a loop in **paint** that prints the 100 lines. You must draw a single line in **paint**, and call **repaint**, then fall through the bottom of the **paint** routine.
4. You’ll need to pause between each line that’s drawn. There are several ways of doing this. You might use the index to read about a version of **repaint** that lets you pass a pause time, or you could also look up **sleep**.

Extra Credit: Draw the individual lines with random colors, changing the background color to a new color for each set of 100 lines.

Update: The repaint method with the millisecond pause is currently not working properly with JBuilder. Currently, the best way to pause uses the following code:

try { Thread.sleep( *the amount of milliseconds put here* ); }

catch (InterruptedException e) { }