**Graphics and the Paint Method**

EVERYTHING YOU SEE ON A COMPUTER SCREEN has to be drawn there, even the text. The Java API includes a range of classes and methods that are devoted to drawing on the screen. In this section, I'll look at some of the most important of these. (Note that all the classes mentioned in this section are defined in the package java.awt and must be imported before they can be used.)

To do any drawing at all in Java, you need a graphics context. A graphics context is an object belonging to the class Graphics. Instance methods are provided in this class for drawing shapes, text, and pictures. **The Graphics class is an abstract class**, so you can't create a graphics context directly. There are two ways to get a graphics context: When the paint() method is called, the system provides a graphics context for use in the method. For drawing outside the paint() method, there is a function getGraphics() that returns a graphics context.

There are **two types of graphics contexts**. One is for drawing in Components and one for drawing in off-screen Images. The Component class is one of the superclasses of Applet. Most of the applet methods that you have seen, including paint() and all the event-handling methods, are actually inherited from Component. An object of type Component represents a graphical user interface component -- that is, something that is visible on the screen. An instance method **getGraphics()** is defined in the Component class. It returns a graphics context that can be used for drawing to a particular component. That is, if comp is an object of type Component and if you say

Graphics g = comp.getGraphics();

then g can be used for drawing to the rectangular area of the screen that represents the component, comp. When you call getGraphics() in an applet, it returns a graphics context for drawing in the applet (which, remember is a type of Component, since Applet is a subclass of Component).

The second type of graphics context is for drawing to off-screen Images. Graphics contexts of this type are obtained by calling a getGraphics() function defined in the Image class. Off-screen images are actually just data stored in memory and are not visible on the screen. However, they can be copied to the screen very quickly. Off-screen images are used to do flicker-free animation: Instead of drawing something on the screen as the user watches, you draw it off-screen and then copy the result to the screen all at once. I'll say more about this at the end of the section. For now, just remember that the same drawing commands that can be applied to on-screen Components can also be applied to off-screen Images.

If g is a graphics context that you've obtained with one of the getGraphics() methods, it is a good idea to call the method g.dispose() after you have finished using it. This method frees any system resources that are used by the graphics context. This is a good idea because on many systems, such resources are limited. (You should **never** call dispose() for the graphics context provided in the paint() method.)

**Paint, Repaint, and Update**

Many applets can do all their drawing operations in their paint() methods. The paint() method should be smart enough to correctly redraw the applet at any time, using data stored in instance variables if necessary. If, in the middle of some other method, you realize that the appearance of the applet should change, you can change the instance variables that record the contents of the applet and then call the repaint() method, which tells the system that it should redraw the applet as soon as it gets a chance (by calling the paint() method). This is the only approach to drawing in the applet that I have discussed up till now. It is satisfactory in many cases. However, at some point, you will want to move beyond this simple model to do something fancier. At that point, you need to know more about how the stuff on the screen actually gets drawn.

First of all, it's not just applets that have paint() methods. An applet can contain other components, such as buttons and so-called "canvasses", which are just rectangular areas for drawing. Each component has its own paint() method, which is called by the system when the component needs to be redrawn. So, a component is ordinarily responsible for drawing itself. Thus, for example, if you've added a button to an applet, you don't have to worry about drawing it. It draws itself. (Or, if you prefer, the system draws the button by calling its paint() method.) If you want to use a canvas in an applet, you should define a subclass of the Canvas class, and provide a paint() method in that subclass to draw the canvas. That's how you teach the Canvas to "draw itself." The applet's paint method, then, is only responsible for drawing directly to the applet itself, not to any subcomponents that the applet contains. You'll learn more about components in the [next chapter](http://math.hws.edu/eck/cs124/notes98/c6/index.html).

Another complication arises from the fact the system does not actually call the paint() method of an applet or component directly. There is another method called update() which is the one actually called by the system. The built-in update procedure first erases everything in the applet or component. Then it calls the paint() method to redraw the contents. Usually, erasing the component first is the right thing to do, since the contents of the component might have changed. However, in some cases you will want to avoid this step. (This will certainly be true if you are using an off-screen image, for example.) In that case, you can override update() to read simply:

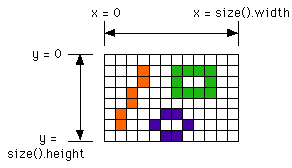
public void update(Graphics g) {

paint(g); // just call paint, without erasing first

}

**Coordinates**

The screen of a computer is a grid of little squares called pixels. The color of each pixel can be set individually, and drawing on the screen just means setting the colors of certain pixels.

A graphics context draws in a rectangle made up of pixels. A position in that rectangle is specified by a pair of integer coordinates, (x,y). The upper left corner has coordinates (0,0). For a component, you can find the size of the rectangle by calling the method size(). The width of the rectangle is size().width pixels, and the height is size().height pixels. The illustration on the right shows a 12-by-8 pixel component (with very large pixels). A small line, rectangle, and oval are shown as they would be drawn by coloring individual pixels. (Note that, properly speaking, the coordinates don't belong to the pixels but to the grid lines between them.)

When you are writing an applet, you don't know its size. The size is specified in an <APPLET> tag, and it's not good form to depend on that size being set to some particular value. It's also true that applets can be resized while they are running. (For the moment, this is not true for applets on pages in Web browsers, but that will probably change in the future.) If you do all your drawing in the applet's paint() method, you can check its size there and base your drawing on the actual height and width of the applet:

public void paint(Graphics g) {

int width = size().width; // get actual width of applet

int height = size().height; // get actual height

. . . // draw the contents of the applet

}

But if you do any calculations elsewhere based on the size of the applet, you might want to do something like this:

int width = -1; // width of applet (initially unknown)

int height = -1; // height of applet (initially unknown)

// values of -1 here will force checkSize() to

// set the correct sizes the first time it is called

void checkSize() { // check if applet size has changed

int w = size().width; // actual current width

int h = size().height; // actual current height

if ( w != width || h != height ) { // size has changed!

width = w; // record new width

height = h; // record new height

. . . // recalculate size-dependent stuff

}

}

public void paint(Graphics g) {

checkSize(); // always check size before redrawing

. . . // draw the contents of the applet

}

The checkSize() method should also be called from other methods that depend on the size of the applet. It all gets a bit tricky, but this is probably the easiest way to account for changes in an applet's size.

**Shapes**

The Graphics class provides a large number of methods for drawing various shapes, such as lines, rectangles, and ovals. The shapes are specified using the coordinate system described above. They are drawn in the current drawing color, as set by the setColor() method. Here is a list of some of the most important drawing methods. Note that all these methods are in the Graphics class, so they all must be called through an object of type Graphics. (For example: g.drawLine(0,0,w,h);.)

**drawLine(int x1, int y1, int x2, int y2)**

Draws a line from the point (x1,y1) to the point (x2,y2).

**drawRect(int x, int y, int width, int height)**

Draws the outline of a rectangle. The upper left corner is at (x,y), and the width and height of the rectangle are as specified. The width and height must be positive, or nothing will be drawn.

**drawOval(int x, int y, int width, int height)**

Draws the outline of an oval. The oval is one that just fits inside the rectangle specified by x, y, width, and height. If width equals height, the oval is a circle.

**drawRoundRect(int x, int y, int width, int height, int xdiam, int ydiam)**

Draws a rectangle with rounded corners. The degree of rounding is given by xdiam and ydiam. The corners are arcs of an ellipse with horizontal diameter xdiam and vertical diameter ydiam. (A reasonable value for xdiam and ydiam is 16.)

**draw3DRect(int x, int y, int width, int height, boolean raised)**

Draws a rectangle that is supposed to have a three-dimensional effect, as if it raised from the screen or pushed into the screen.

**fillRect(int x, int y, int width, int height)**

Draws a filled-in rectangle.

**fillOval(int x, int y, int width, int height)**

Draws a filled-in oval.

**fillRoundRect(int x, int y, int width, int height, int xdiam, int ydiam)**

Draws a filled-in rounded rectangle.

**fill3DRect(int x, int y, int width, int height, boolean raised)**

Draws a filled-in three-dimensional rectangle.

**drawString(String str, int x, int y)**

Draws the string str, starting at the point (x,y). x gives the position of the left end of the string. y gives the height of the baseline, which is like the line you write on in a ruled tablet. (There is more about drawing text later in this section.)

**drawImage(Image img, int x, int y, ImageObserver observer)**

Draws a copy of an image. The upper left corner of the image is placed at the point (x,y). The mysterious fourth parameter can usually be set to the special variable this. (There is more about images later in this section.)

**Colors**

Java is designed to work with so-called RGB colors. An RGB color is specified by three numbers that give the level of red, green, and blue, respectively, in the color. A color in Java is an object of the class Color. You can construct a new color by specifying its red, blue, and green components. For example,

Color myColor = new Color(r,g,b);

There are actually two constructors that you can call in this way. In one, r, g, and b are integers in the range 0 to 255. In the other, they are numbers of type float in the range 0.0 to 1.0. Often, you can avoid constructing new colors altogether, since the Color class defines several constants representing common colors: Color.white, Color.black, Color.red, Color.green, Color.blue, Color.cyan, Color.magenta, Color.yellow, Color.pink, Color.orange, Color.lightGray, Color.gray, and Color.darkGray.

One of the instance variables in a graphics context is the current drawing color, which is used for all the drawing commands listed above (except drawImage.) If g is a graphics context, you can change the current drawing color for g using the method g.setColor(c), where c is a Color. For example, if you want to draw in green, you would just say g.setColor(Color.green). The graphics context continues to use the color until you explicitly change it. If you want to know what the current drawing color is, you can call the function g.getColor(), which returns a value of type color.

Every component has an associated foreground color and background color. When the component is erased by the update() method, it is filled with the background color. When a new graphics context is created for a component, the current drawing color is equal to the foreground color. You can set the foreground and background colors for a component by calling the methods

void setForeground(Color c)

and

void setBackground(Color c)

Note that these are instance methods in the Component class, **not** in the Graphics class. So in an applet, which is one type of component, you would say simply setBackground(Color.white), and not g.setBackground(Color.white), to set the background color for the applet to be white.

**Fonts and FontMetrics**

A font represents a particular type of text. The same character will appear different in different fonts. In Java, a font is characterized by a font name, a style, and a size. The available font names are system dependent, but you can always use the following three names: TimesRoman, Helvetica, and Courier. The style of a font is one of the values:

* Font.PLAIN,
* Font.ITALIC,
* Font.BOLD, or
* Font.BOLD + Font.ITALIC.

The size of a font is an integer. Size typically ranges from about 10 to 36, although larger sizes can also be used. The size of a font is usually about equal to the height of the largest characters in the font, in pixels, but this is not a definite rule.

Java has a class named Font for representing fonts. You can construct a new font by specifying its font name, style, and size:

Font plainFont = new Font("TimesRoman",Font.PLAIN,12);

Font boldFont = new Font("Helvetica",Font.BOLD,12);

A graphics context has a current font, which is used in the drawString() method. You can set the current font with the setFont() method. For example, if g is a graphics context and boldFont is a font, then the command g.setFont(boldFont) will set the current font of g to boldFont. For example, you could print out a big, bold "Hello World" by saying:

Font boldFont = new Font("Helvetica", Font.BOLD, 24);

g.setFont(boldFont);

g.drawString("Hello World", 20, 30);

You can find out the current font of g by calling the method g.getFont(), which returns an object of type Font. A newly created graphics context is set up to use some default font, which is system dependent. Sometimes, you might want to use a modified version of the same font. There are methods in the Font class that make this possible. Here's an example:

Font F = g.getFont();

Font boldFont = new Font(F.getName(), Font.BOLD, F.getSize());

Font bigFont = new Font(F.getName(), F.getStyle(), 2 \* F.getSize());

When you draw a string, sometimes you need to know how much space it is going to occupy. To do this in Java, you need an object of type FontMetrics, which contains information about the size of text drawn in some particular font. If F is a font, you can get a FontMetrics object for that font by calling g.getFontMetrics(F). Once you have a FontMetrics object, FM, you can call FM.stringWidth(str) to find the width of a string str when drawn in that font. FM.getHeight() gives the standard distance between the baselines of consecutive lines of text.

**Images**

In Java, an object of type Image is a picture that can be copied into a graphics context using the drawImage() method. Java is meant to work with images that can be downloaded over the net when they are needed. To accommodate this possibility, the drawImage() method is a bit strange: It doesn't necessarily draw the image immediately. If the image is not yet available -- for example, if it has not yet been downloaded -- then the drawImage() method returns without drawing it, and the system will draw it later when it is available! Usually, though, you can just call drawImage() and let the system handle the rest.

In any case, I am not interested in downloaded images here. I am interested in off-screen images, which are images that can be drawn to using a graphics context. You can create such an image with the createImage() method, which is defined in class Component (and is therefore available, in particular, in applets). All you have to specify is the width and the height of the image you want:

Image OSC = createImage(width,height);

Once you've created an off-screen image, OSC, you can call its getGraphics() method to get a graphics context for drawing to the image:

Graphics g = OSC.getGraphics();

And once you have g, you can draw anything you want. You can then use drawImage to copy the image you've created onto the screen, or into any other graphics context.

In a technique called double buffering, a copy of an entire component is kept in an off-screen image. All drawing operations are performed on this copy. The paint method then just has to copy the image onto the screen. Here's an outline of how things might work:

Image OSC = null; // the off-screen image for double buffering

Graphics OSC\_g; // a graphics context for OSC

public void update(Graphics g) {

// redefine this so it doesn't erase before drawing

paint(g);

}

public void paint(Graphics g) {

if (OSC != null) { // to be safe, check that OSC is not null

g.drawImage(OSC,0,0,this); // copy image to screen

}

}

void drawStuff() {

if (OSC == null) { // create off-screen canvas, if it doesn't yet exist

OSC = createImage(size().width, size().height());

OSC\_g = OSC.getGraphics();

OSC\_g.setColor(Color.white); // fill OSC with white

OSC\_g.fillRect(0,0,size().width,size.height());

}

. . . // draw to OSC\_g

repaint(); // ask system to update the screen

}

For an applet, you can create the off-screen image in the init() method. Other components, however, don't have init()methods, and you have to be careful that you don't create the off-screen image until you know the size of the component. (This is probably **not** known when the component object is first constructed.) If you want to deal with changes in the size of the component, things become even harder.

Double buffering can be used to do flicker-free animation. It's a must for professional-looking graphics. However, you should be aware that it does eat up a lot of memory, and you pay some performance penalty because of the time it takes to copy an image to the screen. It takes experience and taste to know how to use double buffering and other graphical techniques well. The best advice is to practice. Write a lot of sample applets, and see what happens!