2

In this chapter:
- *Graphics*
- *Point*
- *Dimension*
- *Shape*
- *Rectangle*
- *Polygon*
- *Image*
- *MediaTracker*

Simple Graphics

This chapter digs into the meat of the AWT classes. After completing this chapter, you will be able to draw strings, images, and shapes via the `Graphics` class in your Java programs. We discuss geometry-related classes—`Polygon`, `Rectangle`, `Point`, and `Dimension`, and the `Shape` interface—you will see these throughout the remaining AWT objects. You will also learn several ways to do smooth animation by using double buffering and the `MediaTracker`.

After reading this chapter, you should be able to do simple animation and image manipulation with AWT. For most applications, this should be sufficient. If you want to look at AWT's more advanced graphics capabilities, be sure to take a look at Chapter 12, *Image Processing*.

2.1 Graphics

The `Graphics` class is an abstract class that provides the means to access different graphics devices. It is the class that lets you draw on the screen, display images, and so forth. `Graphics` is an abstract class because working with graphics requires detailed knowledge of the platform on which the program runs. The actual work is done by concrete classes that are closely tied to a particular platform. Your Java Virtual Machine vendor provides the necessary concrete classes for your environment. You never need to worry about the platform-specific classes; once you have a `Graphics` object, you can call all the methods of the `Graphics` class, confident that the platform-specific classes will work correctly wherever your program runs.

You rarely need to create a `Graphics` object yourself; its constructor is protected and is only called by the subclasses that extend `Graphics`. How then do you get a

Graphics object to work with? The sole parameter of the Component.paint() and Component.update() methods is the current graphics context. Therefore, a Graphics object is always available when you override a component's paint() and update() methods. You can ask for the graphics context of a Component by calling Component.getGraphics(). However, many components do not have a drawable graphics context. Canvas and Container objects return a valid Graphics object; whether or not any other component has a drawable graphics context depends on the run-time environment. (The latest versions of Netscape Navigator provide a drawable graphics context for any component, but you shouldn't get used to writing platform-specific code.) This restriction isn't as harsh as it sounds. For most components, a drawable graphics context doesn't make much sense; for example, why would you want to draw on a List? If you want to draw on a component, you probably can't. The notable exception is Button, and that may be fixed in future versions of AWT.

2.1.1 Graphics Methods

Constructors

protected Graphics ()

Because Graphics is an abstract class, it doesn't have a visible constructor. The way to get a Graphics object is to ask for one by calling getGraphics() or to use the one given to you by the Component.paint() or Component.update() method.

The abstract methods of the Graphics class are implemented by some windowing system–specific class. You rarely need to know which subclass of Graphics you are using, but the classes you actually get (if you are using the JDK) are sun.awt.win32.Win32Graphics (JDK1.0), sun.awt.window.WGraphics (JDK1.1), sun.awt.motif.X11Graphics, or sun.awt.macos.MacGraphics.

Pseudo-constructors

In addition to using the graphics contexts given to you by getGraphics() or in Component.paint(), you can get a Graphics object by creating a copy of another Graphics object. Creating new graphics contexts has resource implications. Certain platforms have a limited number of graphics contexts that can be active. For instance, on Windows 95 you cannot have more than four in use at one time. Therefore, it's a good idea to call dispose() as soon as you are done with a Graphics object. Do not rely on the garbage collector to clean up for you.

public abstract Graphics create ()

This method creates a second reference to the graphics context. It is useful for clipping (reducing the drawable area).

public Graphics create (int x, int y, int width, int height)

This method creates a second reference to a subset of the drawing area of the graphics context. The new `Graphics` object covers the rectangle from (`x`, `y`) through (`x+width-1`, `y+height-1`) in the original object. The coordinate space of the new `Graphics` context is translated so that the upper left corner is (0, 0) and the lower right corner is (`width`, `height`). Shifting the coordinate system of the new object makes it easier to work within a portion of the drawing area without using offsets.

Drawing strings

These methods let you draw text strings on the screen. The coordinates refer to the left end of the text's baseline.

public abstract void drawString (String text, int x, int y)

The `drawString()` method draws `text` on the screen in the current font and color, starting at position (`x`, `y`). The starting coordinates specify the left end of the `String`'s baseline.

public void drawChars (char text[], int offset, int length, int x, int y)

The `drawChars()` method creates a `String` from the char array `text` starting at `text[offset]` and continuing for `length` characters. The newly created `String` is then drawn on the screen in the current font and color, starting at position (`x`, `y`). The starting coordinates specify the left end of the `String`'s baseline.

public void drawBytes (byte text[], int offset, int length, int x, int y)

The `drawBytes()` method creates a `String` from the byte array `text` starting at `text[offset]` and continuing for `length` characters. This `String` is then drawn on the screen in the current font and color, starting at position (`x`, `y`). The starting coordinates specify the left end of the `String`'s baseline.

public abstract Font getFont ()

The `getFont()` method returns the current `Font` of the graphics context. See Chapter 3, *Fonts and Colors*, for more on what you can do with fonts. You cannot get meaningful results with `getFont()` until the applet or application is displayed on the screen (generally, not in `init()` of an applet or `main()` of an application).

public abstract void setFont (Font font)

The `setFont()` method changes the current `Font` to `font`. If `font` is not available on the current platform, the system chooses a default. To change the current font to 12 point bold TimesRoman:

```
setFont (new Font ("TimesRoman", Font.BOLD, 12));
```

public FontMetrics getFontMetrics ()

The `getFontMetrics()` method returns the current `FontMetrics` object of the graphics context. You use `FontMetrics` to reveal sizing properties of the current `Font`—for example, how wide the "Hello World" string will be in pixels when displayed on the screen.

public abstract FontMetrics getFontMetrics (Font font)

This version of `getFontMetrics()` returns the `FontMetrics` for the `Font` font instead of the current font. You might use this method to see how much space a new font requires to draw text.

For more information about `Font` and `FontMetrics`, see Chapter 3.

Painting

public abstract Color getColor ()

The `getColor()` method returns the current foreground `Color` of the `Graphics` object. All future drawing operations will use this color. Chapter 3 describes the `Color` class.

public abstract void setColor (Color color)

The `setColor()` method changes the current drawing color to `color`. As you will see in the next chapter, the `Color` class defines some common colors for you. If you can't use one of the predefined colors, you can create a color from its RGB values. To change the current color to red, use any of the following:

```
setColor (Color.red);
setColor (new Color (255, 0, 0));
setColor (new Color (0xff0000));
```

public abstract void clearRect (int x, int y, int width, int height)

The `clearRect()` method sets the rectangular drawing area from (`x`, `y`) to (`x+width-1`, `y+height-1`) to the current background color. Keep in mind that the second pair of parameters is not the opposite corner of the rectangle, but the width and height of the area to clear.

public abstract void clipRect (int x, int y, int width, int height)

The `clipRect()` method reduces the drawing area to the intersection of the current drawing area and the rectangular area from (`x`, `y`) to (`x+width-1`, `y+height-1`). Any future drawing operations outside this clipped area will have no effect. Once you clip a drawing area, you cannot increase its size with `clipRect()`; the drawing area can only get smaller. (However, if the `clipRect()` call is in `paint()`, the size of the drawing area will be reset to its original size on subsequent calls to `paint()`.) If you want the ability to draw to the entire area, you must create a second `Graphics` object that contains a copy of the drawing area before calling `clipRect()` or use `setClip()`. The following code is a simple applet that demonstrates clipping; Figure 2-1 shows the result.

```
import java.awt.*;
public class clipping extends java.applet.Applet {
    public void paint (Graphics g) {
        g.setColor (Color.red);
        Graphics clippedGraphics = g.create();
        clippedGraphics.drawRect (0,0,100,100);
        clippedGraphics.clipRect (25, 25, 50, 50);
        clippedGraphics.drawLine (0,0,100,100);
        clippedGraphics.dispose();
        clippedGraphics=null;
        g.drawLine (0,100,100,0);
    }
}
```

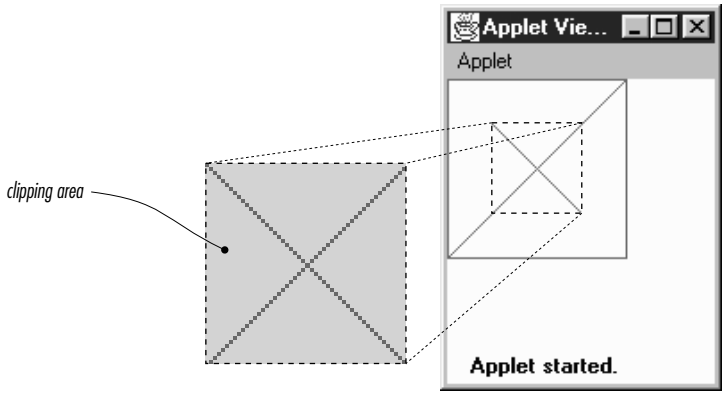

Figure 2–1: Clipping restricts the drawing area

The `paint()` method for this applet starts by setting the foreground color to red. It then creates a copy of the `Graphics` context for clipping, saving the original object so it can draw on the entire screen later. The applet then draws a rectangle, sets the clipping area to a smaller region, and draws a diagonal line across the rectangle from upper left to lower right. Because clipping is in effect, only part of the line is displayed. The applet then discards the clipped `Graphics` object and draws an unclipped line from lower left to upper right using the original object `g`.

public abstract void setClip(int x, int y, int width, int height) ★

This `setClip()` method allows you to change the current clipping area based on the parameters provided. `setClip()` is similar to `clipRect()`, except that it is not limited to shrinking the clipping area. The current drawing area becomes the rectangular area from (`x`, `y`) to (`x+width-1`, `y+height-1`); this area may be larger than the previous drawing area.

public abstract void setClip(Shape clip) ★

This `setClip()` method allows you to change the current clipping area based on the `clip` parameter, which may be any object that implements the `Shape` interface. Unfortunately, practice is not as good as theory, and in practice, `clip` must be a `Rectangle`; if you pass `setClip()` a `Polygon`, it throws an `IllegalArgumentException`.* (The `Shape` interface is discussed later in this chapter.)

public abstract Rectangle getClipBounds () ★
public abstract Rectangle getClipRect () ☆

The `getClipBounds()` methods returns a `Rectangle` that describes the clipping area of a `Graphics` object. The `Rectangle` gives you the (`x`, `y`) coordinates of the top left corner of the clipping area along with its width and height. (`Rectangle` objects are discussed later in this chapter.)

`getClipRect()` is the Java 1.0 name for this method.

public abstract Shape getClip () ★

The `getClip()` method returns a `Shape` that describes the clipping area of a `Graphics` object. That is, it returns the same thing as `getClipBounds()` but as a `Shape`, instead of as a `Rectangle`. By calling `Shape.getBounds()`, you can get the (`x`, `y`) coordinates of the top left corner of the clipping area along with its width and height. In the near future, it is hard to imagine the actual object that `getClip()` returns being anything other than a `Rectangle`.

public abstract void copyArea (int x, int y, int width, int height, int delta_x, int delta_y)

The `copyArea()` method copies the rectangular area from (`x`, `y`) to (`x+width`, `y+height`) to the area with an upper left corner of (`x+delta_x`, `y+delta_y`). The `delta_x` and `delta_y` parameters are not the coordinates of the second point but an offset from the first coordinate pair (`x`, `y`). The area copied may fall outside of the clipping region. This method is often used to tile an area of the graphics context. `copyArea()` does not save the contents of the area copied.

Painting mode

There are two painting or drawing modes for the `Graphics` class: paint (the default) and XOR mode. In paint mode, anything you draw replaces whatever is already on the screen. If you draw a red square, you get a red square, no matter what was underneath; this is what most programmers have learned to expect.

The behavior of XOR mode is rather strange, at least to people accustomed to modern programming environments. XOR mode is short for eXclusive-OR mode.

---

* It should be simple for Sun to fix this bug; one would expect clipping to a `Polygon` to be the same as clipping to the `Polygon`'s bounding rectangle.

The idea behind XOR mode is that drawing the same object twice returns the screen to its original state. This technique was commonly used for simple animations prior to the development of more sophisticated methods and cheaper hardware.

The side effect of XOR mode is that painting operations don't necessarily get the color you request. Instead of replacing the original pixel with the new value, XOR mode merges the original color, the painting color, and an XOR color (usually the background color) to form a new color. The new color is chosen so that if you repaint the pixel with the same color, you get the original pixel back. For example, if you paint a red square in XOR mode, you get a square of some other color on the screen. Painting the same red square again returns the screen to its original state.

public abstract void setXORMode (Color xorColor)

The `setXORMode()` method changes the drawing mode to XOR mode. In XOR mode, the system uses the `xorColor` color to determine an alternate color for anything drawn such that drawing the same item twice restores the screen to its original condition. The `xorColor` is usually the current background color but can be any color. For each pixel, the new color is determined by an exclusive-or of the old pixel color, the painting color, and the `xorColor`.

For example, if the old pixel is red, the XOR color is blue, and the drawing color is green, the end result would be white. To see why, it is necessary to look at the RGB values of the three colors. Red is (255, 0, 0). Blue is (0, 0, 255). Green is (0, 255, 0). The exclusive-or of these three values is (255, 255, 255), which is white. Drawing another green pixel with a blue XOR color yields red, the pixel's original color, since (255, 255, 255) ^ (0, 0, 255) ^ (0, 255, 0) yields (255, 0, 0).* The following code generates the display shown in Figure 2-2.

```
import java.awt.*;
public class xor extends java.applet.Applet {
    public void init () {
        setBackground (Color.red);
    }
    public void paint (Graphics g) {
        g.setColor (Color.green);
        g.setXORMode (Color.blue);
        g.fillRect (10, 10, 100, 100);
        g.fillRect (10, 60, 100, 100);
    }
}
```

Although it's hard to visualize what color XOR mode will pick, there is one important special case. Let's say that there are only two colors: a background color (the

* ^ is the Java XOR operator.

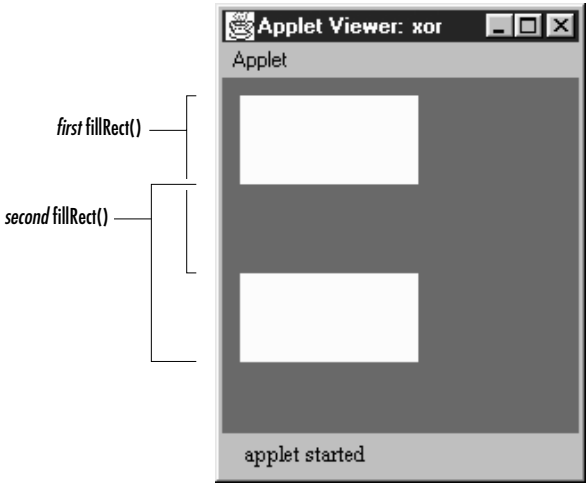

Figure 2–2: Drawing in XOR mode

XOR color) and a foreground color (the painting color). Each pixel must be in one color or the other. Painting "flips" each pixel to the other color. Foreground pixels become background, and vice versa.

public abstract void setPaintMode ()

The `setPaintMode()` method puts the system into paint mode. When in paint mode, any drawing operation replaces whatever is underneath it. Call `setPaintMode()` to return to normal painting when finished with XOR mode.

Drawing shapes

Most of the drawing methods require you to specify a bounding rectangle for the object you want to draw: the location of the object's upper left corner, plus its width and height. The two exceptions are lines and polygons. For lines, you supply two endpoints; for polygons, you provide a set of points.

Versions 1.0.2 and 1.1 of AWT always draw solid lines that are one pixel wide; there is no support for line width or fill patterns. A future version should support lines with variable widths and patterns.

public abstract void drawLine (int x1, int y1, int x2, int y2)

The `drawLine()` method draws a line on the graphics context in the current color from (`x1`, `y1`) to (`x2`, `y2`). If (`x1`, `y1`) and (`x2`, `y2`) are the same point, you will draw a point. There is no method specific to drawing a point. The following code generates the display shown in Figure 2-3.

```
g.drawLine (5, 5, 50, 75);   // line
g.drawLine (5, 75, 5, 75);   // point
g.drawLine (50, 5, 50, 5);   // point
```

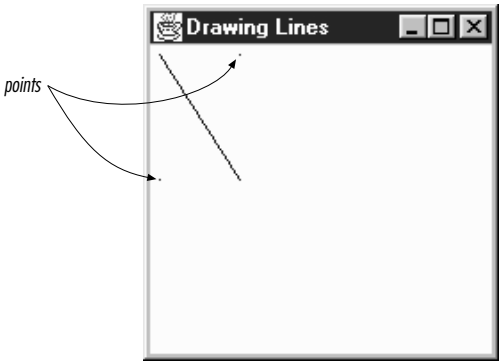

Figure 2–3: Drawing lines and points with drawLine()

public void drawRect (int x, int y, int width, int height)

The `drawRect()` method draws a rectangle on the drawing area in the current color from (`x`, `y`) to (`x+width`, `y+height`). If `width` or `height` is negative, nothing is drawn.

public abstract void fillRect (int x, int y, int width, int height)

The `fillRect()` method draws a filled rectangle on the drawing area in the current color from (`x`, `y`) to (`x+width-1`, `y+height-1`). Notice that the filled rectangle is one pixel smaller to the right and bottom than requested. If `width` or `height` is negative, nothing is drawn.

public abstract void drawRoundRect (int x, int y, int width, int height, int arcWidth, int arcHeight)

The `drawRoundRect()` method draws a rectangle on the drawing area in the current color from (`x`, `y`) to (`x+width`, `y+height`). However, instead of perpendicular corners, the corners are rounded with a horizontal diameter of `arcWidth` and a vertical diameter of `arcHeight`. If `width` or `height` is a negative number, nothing is drawn. If `width`, `height`, `arcWidth`, and `arcHeight` are all equal, you get a circle.

To help you visualize the `arcWidth` and `arcHeight` of a rounded rectangle, Figure 2-4 shows one corner of a rectangle drawn with an `arcWidth` of 20 and a `arcHeight` of 40.

public abstract void fillRoundRect (int x, int y, int width, int height, int arcWidth, int arcHeight)

The `fillRoundRect()` method draws a filled rectangle on the drawing area in the current color from (`x`, `y`) to (`x+width-1`, `y+height-1`). However, instead of having perpendicular corners, the corners are rounded with a horizontal

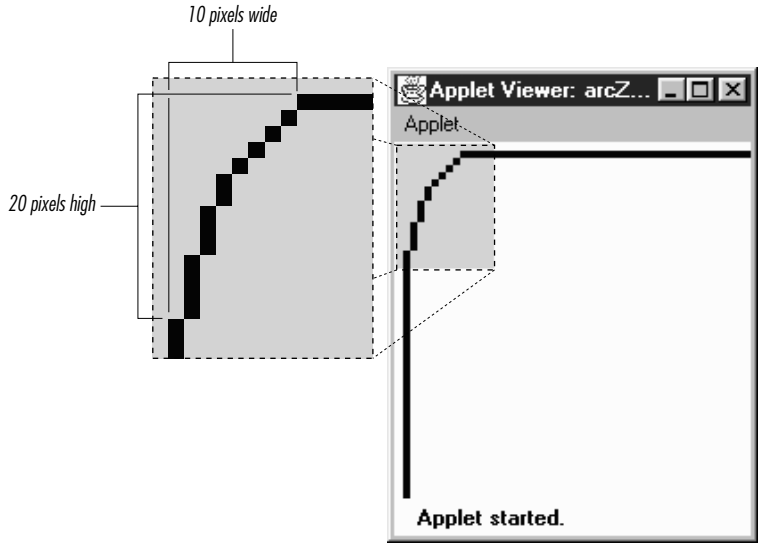

Figure 2–4: Drawing rounded corners

diameter of `arcWidth` and a vertical diameter of `arcHeight` for the four corners. Notice that the filled rectangle is one pixel smaller to the right and bottom than requested. If `width` or `height` is a negative number, nothing is filled. If `width`, `height`, `arcWidth`, and `arcHeight` are all equal, you get a filled circle.

Figure 2-4 shows how AWT generates rounded corners. Figure 2-5 shows the collection of rectangles created by the following code. The rectangles in Figure 2-5 are filled and unfilled, with rounded and square corners.

```
g.drawRect (25, 10, 50, 75);
g.fillRect (25, 110, 50, 75);
g.drawRoundRect (100, 10, 50, 75, 60, 50);
g.fillRoundRect (100, 110, 50, 75, 60, 50);
```

public void draw3DRect (int x, int y, int width, int height, boolean raised)

The `draw3DRect()` method draws a rectangle in the current color from (`x`, `y`) to (`x+width`, `y+height`); a shadow effect makes the rectangle appear to float slightly above or below the screen. The `raised` parameter has an effect only if the current color is not black. If `raised` is `true`, the rectangle looks like a button waiting to be pushed. If `raised` is `false`, the rectangle looks like a depressed button. If `width` or `height` is negative, the shadow appears from another direction.

public void fill3DRect (int x, int y, int width, int height, boolean raised)

The `fill3DRect()` method draws a filled rectangle in the current color from (`x`, `y`) to (`x+width`, `y+height`); a shadow effect makes the rectangle appear to float slightly above or below the screen. The `raised` parameter has an effect

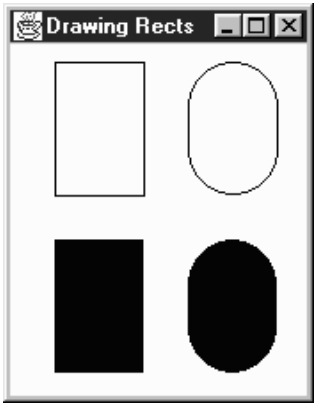

Figure 2–5: Varieties of rectangles

only if the current color is not black. If `raised` is `true`, the rectangle looks like a button waiting to be pushed. If `raised` is `false`, the rectangle looks like a depressed button. To enhance the shadow effect, the depressed area is given a slightly deeper shade of the drawing color. If `width` or `height` is negative, the shadow appears from another direction, and the rectangle isn't filled. (Different platforms could deal with this differently. Try to ensure the parameters have positive values.)

Figure 2-6 shows the collection of three-dimensional rectangles created by the following code. The rectangles in the figure are raised and depressed, filled and unfilled.

```
g.setColor (Color.gray);
g.draw3DRect (25, 10, 50, 75, true);
g.draw3DRect (25, 110, 50, 75, false);
g.fill3DRect (100, 10, 50, 75, true);
g.fill3DRect (100, 110, 50, 75, false);
```

public abstract void drawOval (int x, int y, int width, int height)

The `drawOval()` method draws an oval in the current color within an invisible bounding rectangle from (`x`, `y`) to (`x+width`, `y+height`). You cannot specify the oval's center point and radii. If `width` and `height` are equal, you get a circle. If `width` or `height` is negative, nothing is drawn.

public abstract void fillOval (int x, int y, int width, int height)

The `fillOval()` method draws a filled oval in the current color within an invisible bounding rectangle from (`x`, `y`) to (`x+width-1`, `y+height-1`). You cannot specify the oval's center point and radii. Notice that the filled oval is one pixel smaller to the right and bottom than requested. If `width` or `height` is negative, nothing is drawn.

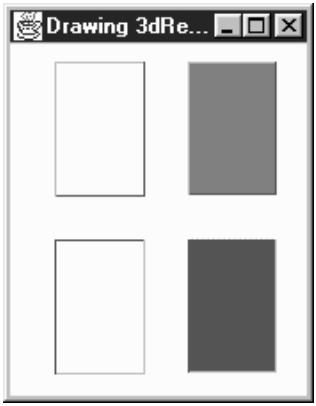

Figure 2–6: Filled and unfilled 3D rectangles

Figure 2-7 shows the collection of ovals, filled and unfilled, that were generated by the following code:

```
g.drawOval (25, 10, 50, 75);
g.fillOval (25, 110, 50, 75);
g.drawOval (100, 10, 50, 50);
g.fillOval (100, 110, 50, 50);
```

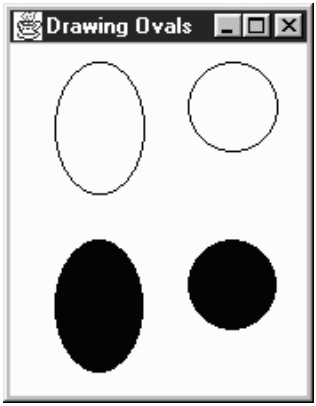

Figure 2–7: Filled and unfilled ovals

public abstract void drawArc (int x, int y, int width, int height, int startAngle, int arcAngle)
The `drawArc()` method draws an arc in the current color within an invisible bounding rectangle from (`x`, `y`) to (`x+width`, `y+height`). The arc starts at `startAngle` degrees and goes to `startAngle` + `arcAngle` degrees. An angle of 0 degrees is at the 3 o'clock position; angles increase counter-clockwise. If

`arcAngle` is negative, drawing is in a clockwise direction. If `width` and `height` are equal and `arcAngle` is 360 degrees, `drawArc()` draws a circle. If `width` or `height` is negative, nothing is drawn.

public abstract void fillArc (int x, int y, int width, int height, int startAngle, int arcAngle)

The `fillArc()` method draws a filled arc in the current color within an invisible bounding rectangle from (`x`, `y`) to (`x+width-1`, `y+height-1`). The arc starts at `startAngle` degrees and goes to `startAngle` + `arcAngle` degrees. An angle of 0 degrees is at the 3 o'clock position; angles increase counter-clockwise. If `arcAngle` is negative, drawing is in a clockwise direction. The arc fills like a pie (to the origin), not from arc endpoint to arc endpoint. This makes creating pie charts easier. If `width` and `height` are equal and `arcAngle` is 360 degrees, `fillArc()` draws a filled circle. If `width` or `height` is negative, nothing is drawn.

Figure 2-8 shows a collection of filled and unfilled arcs that were generated by the following code:

```
g.drawArc (25, 10, 50, 75, 0, 360);
g.fillArc (25, 110, 50, 75, 0, 360);
g.drawArc (100, 10, 50, 75, 45, 215);
g.fillArc (100, 110, 50, 75, 45, 215);
```

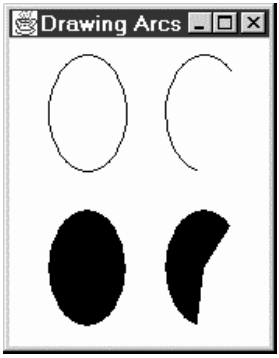

Figure 2–8: Filled and unfilled arcs

public void drawPolygon (Polygon p)

The `drawPolygon()` method draws a path for the points in polygon `p` in the current color. Section 2.6 discusses the `Polygon` class in detail.

The behavior of `drawPolygon()` changes slightly between Java 1.0.2 and 1.1. With version 1.0.2, if the first and last points of a `Polygon` are not the same, a call to `drawPolygon()` results in an open polygon, since the endpoints are not connected for you. Starting with version 1.1, if the first and last points are not the same, the endpoints are connected for you.

public abstract void drawPolygon (int xPoints[], int yPoints[], int numPoints)

The `drawPolygon()` method draws a path of `numPoints` nodes by plucking one element at a time out of `xPoints` and `yPoints` to make each point. The path is drawn in the current color. If either `xPoints` or `yPoints` does not have `numPoints` elements, `drawPolygon()` throws a run-time exception. In 1.0.2, this exception is an `IllegalArgumentException`; in 1.1, it is an `ArrayIndexOutOfBoundsException`. This change shouldn't break older programs, since you are not required to catch run-time exceptions.

public abstract void drawPolyline (int xPoints[], int yPoints[], int numPoints) ★

The `drawPolyline()` method functions like the 1.0 version of `drawPolygon()`. It plays connect the dots with the points in the `xPoints` and `yPoints` arrays and does not connect the endpoints. If either `xPoints` or `yPoints` does not have `numPoints` elements, `drawPolygon()` throws the run-time exception, `ArrayIndexOutOfBoundsException`.

Filling polygons is a complex topic. It is not as easy as filling rectangles or ovals because a polygon may not be closed and its edges may cross. AWT uses an even-odd rule to fill polygons. This algorithm works by counting the number of times each scan line crosses an edge of the polygon. If the total number of crossings to the left of the current point is odd, the point is colored. If it is even, the point is left alone. Figure 2-9 demonstrates this algorithm for a single scan line that intersects the polygon at x values of 25, 75, 125, 175, 225, and 275.

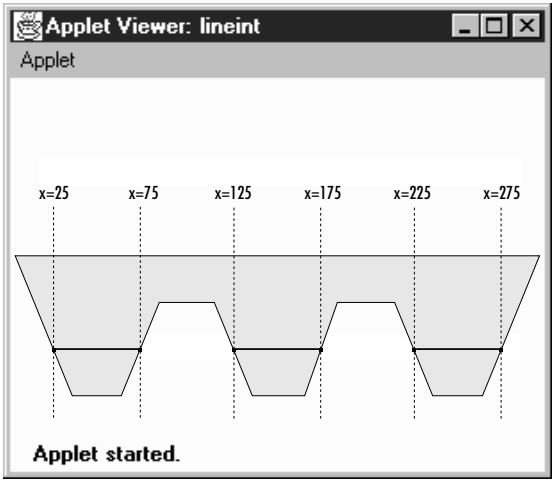

Figure 2–9: Polygon fill algorithm

The scan line starts at the left edge of the screen; at this point there haven't been

any crossings, so the pixels are left untouched. The scan line reaches the first crossing when x equals 25. Here, the total number of crossings to the left is one, so the scan line is inside the polygon, and the pixels are colored. At 75, the scan line crosses again; the total number of crossings is two, so coloring stops.

public void fillPolygon (Polygon p)

The `fillPolygon()` method draws a filled polygon for the points in `Polygon p` in the current color. If the polygon is not closed, `fillPolygon()` adds a segment connecting the endpoints. Section 2.6 discusses the `Polygon` class in detail.

public abstract void fillPolygon (int xPoints[], int yPoints[], int nPoints)

The `fillPolygon()` method draws a polygon of `numPoints` nodes by plucking one element at a time out of `xPoints` and `yPoints` to make each point. The polygon is drawn in the current color. If either `xPoints` or `yPoints` does not have `numPoints` elements, `fillPolygon()` throws the run-time exception `IllegalArgumentException`. If the polygon is not closed, `fillPolygon()` adds a segment connecting the endpoints.*

Figure 2-10 shows several polygons created by the following code, containing different versions of `drawPolygon()` and `fillPolygon()`:

```
int[] xPoints[] = {{50, 25, 25, 75, 75},
                   {50, 25, 25, 75, 75},
                   {100, 100, 150, 100, 150, 150, 125, 100, 150},
                   {100, 100, 150, 100, 150, 150, 125, 100, 150}};
int[] yPoints[] = {{10, 35, 85, 85, 35, 10},
                   {110, 135, 185, 185, 135},
                   {85, 35, 35, 85, 85, 35, 10, 35, 85},
                   {185, 135, 135, 185, 185, 135, 110, 135, 185}};
int   nPoints[] = {5, 5, 9, 9};
g.drawPolygon (xPoints[0], yPoints[0], nPoints[0]);
g.fillPolygon (xPoints[1], yPoints[1], nPoints[1]);
g.drawPolygon (new Polygon(xPoints[2], yPoints[2], nPoints[2]));
g.fillPolygon (new Polygon(xPoints[3], yPoints[3], nPoints[3]));
```

Drawing images

An `Image` is a displayable object maintained in memory. To get an image on the screen, you must draw it onto a graphics context, using the `drawImage()` method of the `Graphics` class. For example, within a `paint()` method, you would call `g.drawImage(image, ... , this)` to display some image on the screen. In other situations, you might use the `createImage()` method to generate an offscreen `Graphics` object, then use `drawImage()` to draw an image onto this object, for display later.

---

* In Java 1.1, this method throws `ArrayIndexOutOfBoundsException`, not `IllegalArgumentException`.

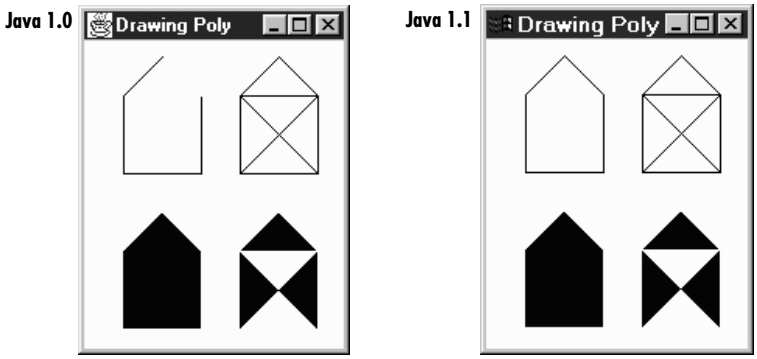

Figure 2–10: Filled and unfilled polygons

This begs the question: where do images come from? We will have more to say about the `Image` class later in this chapter. For now, it's enough to say that you can call `getImage()` to load an image from disk or across the Net. There are versions of `getImage()` in the `Applet` and `Toolkit` classes; the latter is for use in applications. You can also call `createImage()`, a method of the `Component` class, to generate an image in memory.

What about the last argument to `drawImage()`? What is `this` for? The last argument of `drawImage()` is always an image observer—that is, an object that implements the `ImageObserver` interface. This interface is discussed in detail in Chapter 12. For the time being, it's enough to say that the call to `drawImage()` starts a new thread that loads the requested image. An image observer monitors the process of loading an image; the thread that is loading the image notifies the image observer whenever new data has arrived. The `Component` class implements the `ImageObserver` interface; when you're writing a `paint()` method, you're almost certainly overriding some component's `paint()` method; therefore, it's safe to use `this` as the image observer in a call to `drawImage()`. More simply, we could say that any component can serve as an image observer for images that are drawn on it.

public abstract boolean drawImage (Image image, int x, int y, ImageObserver observer)

The `drawImage()` method draws `image` onto the screen with its upper left corner at `(x, y)`, using `observer` as its `ImageObserver`. Returns `true` if the object is fully drawn, `false` otherwise.

public abstract boolean drawImage (Image image, int x, int y, int width, int height, ImageObserver observer)

The `drawImage()` method draws `image` onto the screen with its upper left corner at `(x, y)`, using `observer` as its `ImageObserver`. The system scales `image` to fit into a `width height` area. The scaling may take time. This method returns `true` if the object is fully drawn, `false` otherwise.

With Java 1.1, you don't need to use `drawImage()` for scaling; you can prescale the image with the `Image.getScaledInstance()` method, then use the previous version of `drawImage()`.

public abstract boolean drawImage (Image image, int x, int y, Color backgroundColor, ImageObserver observer)

The `drawImage()` method draws `image` onto the screen with its upper left corner at (`x`, `y`), using `observer` as its `ImageObserver`. `backgroundColor` is the color of the background seen through the transparent parts of the image. If no part of the image is transparent, you will not see `backgroundColor`. Returns `true` if the object is fully drawn, `false` otherwise.

public abstract boolean drawImage (Image image, int x, int y, int width, int height, Color backgroundColor, ImageObserver observer)

The `drawImage()` method draws `image` onto the screen with its upper left corner at (`x`, `y`), using `observer` as its `ImageObserver`. `backgroundColor` is the color of the background seen through the transparent parts of the image. The system scales `image` to fit into a `width` x `height` area. The scaling may take time. This method returns `true` if the image is fully drawn, `false` otherwise.

With Java 1.1, you can prescale the image with the `AreaAveragingScaleFilter` or `ReplicateScaleFilter` described in Chapter 12, then use the previous version of `drawImage()` to display it.

The following code generated the images in Figure 2-11. The images on the left come from a standard JPEG file. The images on the right come from a file in GIF89a format, in which the white pixel is "transparent." Therefore, the gray background shows through this pair of images.

```
import java.awt.*;
import java.applet.*;
public class drawingImages extends Applet {
    Image i, j;
    public void init () {
        i = getImage (getDocumentBase(), "rosey.jpg");
        j = getImage (getDocumentBase(), "rosey.gif");
    }
    public void paint (Graphics g) {
        g.drawImage (i, 10, 10, this);
        g.drawImage (i, 10, 85, 150, 200, this);
        g.drawImage (j, 270, 10, Color.lightGray, this);
        g.drawImage (j, 270, 85, 150, 200, Color.lightGray, this);
    }
}
```

public abstract boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, ImageObserver observer) ★

The `drawImage()` method draws a portion of `image` onto the screen. It takes the part of the image with corners at (`sx1`, `sy1`) and (`sx2`, `sy2`); it places this

Figure 2–11: Scaled and unscaled images

rectangular
snippet on the screen with one corner at (`dx1`, `dy1`) and another at (`dx2`, `dy2`), using `observer` as its `ImageObserver`. (Think of `d` for destination location and `s` for source image.) This method returns `true` if the object is fully drawn, `false` otherwise.

`drawImage()` flips the image if source and destination endpoints are not the same corners, crops the image if the destination is smaller than the original size, and scales the image if the destination is larger than the original size. It does not do rotations, only flips (i.e., it can produce a mirror image or an image rotated 180 degrees but not an image rotated 90 or 270 degrees).

public abstract boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2, int sx1, int sy1, int sx2, int sy2, Color backgroundColor, ImageObserver observer) ★

The `drawImage()` method draws a portion of `image` onto the screen. It takes the part of the image with corners at (`sx1`, `sy1`) and (`sx2`, `sy2`); it places this rectangular snippet on the screen with one corner at (`dx1`, `dy1`) and another at (`dx2`, `dy2`), using `observer` as its `ImageObserver`. (Think of `d` for destination location and `s` for source image.) `backgroundColor` is the color of the background seen through the transparent parts of the image. If no part of the image is transparent, you will not see `backgroundColor`. This method returns `true` if the object is fully drawn, `false` otherwise.

Like the previous version of `drawImage()`, this method flips the image if source and destination endpoints are not the same corners, crops the image if the

destination is smaller than the original size, and scales the image if the destination is larger than the original size. It does not do rotations, only flips (i.e., it can produce a mirror image or an image rotated 180 degrees but not an image rotated 90 or 270 degrees).

The following code demonstrates the new `drawImage()` methods in Java 1.1. They allow you to scale, flip, and crop images without the use of image filters. The results are shown in Figure 2-12.

```
// Java 1.1 only
import java.awt.*;
import java.applet.*;
public class drawingImages11 extends Applet {
    Image i, j;
    public void init () {
        i = getImage (getDocumentBase(), "rosey.gif");
    }
    public void paint (Graphics g) {
        g.drawImage (i, 10, 10, this);
        g.drawImage (i, 10, 85,
                     i.getWidth(this)+10, i.getHeight(this)+85,
                     i.getWidth(this), i.getHeight(this), 0, 0, this);
        g.drawImage (i, 270, 10,
                     i.getWidth(this)+270, i.getHeight(this)*2+10, 0, 0,
                     i.getWidth(this), i.getHeight(this), Color.gray, this);
        g.drawImage (i, 10, 170,
                     i.getWidth(this)*2+10, i.getHeight(this)+170, 0,
                     i.getHeight(this)/2, i.getWidth(this)/2, 0, this);
    }
}
```

Miscellaneous methods

public abstract void translate (int x, int y)

The `translate()` method sets how the system translates the coordinate space for you. The point at the (`x`, `y`) coordinates becomes the origin of this graphics context. Any future drawing will be relative to this location. Multiple translations are cumulative. The following code leaves the coordinate system translated by (100, 50).

```
translate (100, 0);
translate (0, 50);
```

Note that each call to `paint()` provides an entirely new `Graphics` context with its origin in the upper left corner. Therefore, don't expect translations to persist from one call to `paint()` to the next.

Figure 2–12: Flipped, mirrored, and cropped images

public abstract void dispose ()

The `dispose()` method frees any system resources used by the `Graphics` context. It's a good idea to call `dispose()` whenever you are finished with a `Graphics` object, rather than waiting for the garbage collector to call it automatically (through `finalize()`). Disposing of the `Graphics` object yourself will help your programs on systems with limited resources. However, you should not dispose the `Graphics` parameter to `Component.paint()` or `Component.update()`.

public void finalize ()

The garbage collector calls `finalize()` when it determines that the `Graphics` object is no longer needed. `finalize()` calls `dispose()`, which frees any resources that the `Graphics` object has used.

public String toString ()

The `toString()` method of `Graphics` returns a string showing the current font and color. However, `Graphics` is an abstract class, and classes that extend `Graphics` usually override `toString()`. For example, on a Windows 95 machine, `sun.awt.win32.Win32Graphics` is the concrete class that extends `Graphics`. The class's `toString()` method displays the current origin of the `Graphics` object, relative to the original coordinate system:

```
sun.awt.win32.Win32Graphics[0,0]
```

2.2 Point

The `Point` class encapsulates x and y coordinates within a single object. It is probably one of the most underused classes within Java. Although there are numerous places within AWT where you would expect to see a `Point`, its appearances are surprisingly rare. Java 1.1 is starting to use `Point` more heavily. The `Point` class is most often used when a method needs to return a pair of coordinates; it lets the method return both x and y as a single object. Unfortunately, `Point` usually is not used when a method requires x and y coordinates as arguments; for example, you would expect the `Graphics` class to have a version of `translate()` that takes a point as an argument, but there isn't one.

The `Point` class does *not* represent a point on the screen. It is not a visual object; there is no `drawPoint()` method.

2.2.1 Point Methods

Variables

The two public variables of `Point` represent a pair of coordinates. They are accessible directly or use the `getLocation()` method. There is no predefined origin for the coordinate space.

public int x
: The coordinate that represents the horizontal position.

public int y
: The coordinate that represents the vertical position.

Constructors

public Point ()
: The first constructor creates an instance of `Point` with an initial x value of 0 and an initial y value of 0.

public Point (int x, int y)
: The next constructor creates an instance of `Point` with an initial x value of `x` and an initial y value of `y`.

public Point (Point p)
: The last constructor creates an instance of `Point` from another point, the x value of `p.x` and an initial y value of `p.y`.

Locations

public Point getLocation () ★

The `getLocation()` method retrieves the current location of this point as a new `Point`.

public void setLocation (int x, int y) ★
public void move (int x, int y) ☆

The `setLocation()` method changes the point's location to `(x, y)`.

`move()` is the Java 1.0 name for this method.

public void setLocation (Point p) ★

This `setLocation()` method changes the point's location to `(p.x, p.y)`.

public void translate (int x, int y)

The `translate()` method moves the point's location by adding the parameters `(x, y)` to the corresponding fields of the `Point`. If the original `Point p` is (3, 4) and you call `p.translate(4, -5)`, the new value of `p` is (7, -1).

Miscellaneous methods

public int hashCode ()

The `hashCode()` method returns a hash code for the point. The system calls this method when a `Point` is used as the key for a hash table.

public boolean equals (Object object)

The `equals()` method overrides the `Object.equals()` method to define equality for points. Two `Point` objects are equal if their x and y values are equal.

public String toString ()

The `toString()` method of `Point` displays the current values of the x and y variables. For example:

```
java.awt.Point[x=100,y=200]
```

2.3 Dimension

The `Dimension` class is similar to the `Point` class, except it encapsulates a width and height in a single object. Like `Point`, `Dimension` is somewhat underused; it is used primarily by methods that need to return a width and a height as a single object; for example, `getSize()` returns a `Dimension` object.

2.3.1 Dimension Methods

Variables

A `Dimension` instance has two variables, one for width and one for height. They are accessible directly or through use of the `getSize()` method.

public int width

The width variable represents the size of an object along the x axis (left to right). Width should not be negative; however, there is nothing within the class to prevent this from happening.

public int height

The height variable represents the size of an object along the y axis (top to bottom). Height should not be negative; however, there is nothing within the class to prevent this from happening.

Constructors

public Dimension ()

This constructor creates a `Dimension` instance with a width and height of 0.

public Dimension (Dimension dim)

This constructor creates a copy of `dim`. The initial width is `dim.width`. The initial height is `dim.height`.

public Dimension (int width, int height)

This constructor creates a `Dimension` with an initial width of `width` and an initial height of `height`.

Sizing

public Dimension getSize () ★

The `getSize()` method retrieves the current size as a new `Dimension`, even though the instance variables are public.

public void setSize (int width, int height) ★

The `setSize()` method changes the dimension's size to `width height`.

public void setSize (Dimension d) ★

The `setSize()` method changes the dimension's size to `d.width d.height`.

Miscellaneous methods

public boolean equals (Object object)

The `equals()` method overrides the `Object.equals()` method to define equality for dimensions. Two `Dimension` objects are equal if their width and height values are equal.

public String toString ()

The `toString()` method of `Dimension` returns a string showing the current width and height settings. For example:

```
java.awt.Dimension[width=0,height=0]
```

2.4 Shape

The new `Shape` interface defines a single method; it requires a geometric object to be able to report its bounding box. Currently, the `Rectangle` and `Polygon` classes implement `Shape`; one would expect other geometric classes to implement `Shape` in the future. Although `Component` has the single method defined by the `Shape` interface, it does not implement the interface.

2.4.1 Shape Method

public abstract Rectangle getBounds() ★

The `getBounds()` method returns the shape's bounding `Rectangle`. Once you have the bounding area, you can use methods like `Graphics.copyArea()` to copy the shape.

2.5 Rectangle

The `Rectangle` class encapsulates x and y coordinates and width and height (`Point` and `Dimension` information) within a single object. It is often used by methods that return a rectangular boundary as a single object: for example, `Polygon.getBounds()`, `Component.getBounds()`, and `Graphics.getClipBounds()`. Like `Point`, the `Rectangle` class is not a visual object and does not represent a rectangle on the screen; ironically, `drawRect()` and `fillRect()` don't take `Rectangle` as an argument.

2.5.1 Rectangle Methods

Variables

The four public variables available for `Rectangle` have the same names as the public instance variables of `Point` and `Dimension`. They are all accessible directly or through use of the `getBounds()` method.

public int x

The x coordinate of the upper left corner.

public int y

The y coordinate of the upper left corner.

public int width

The width variable represents the size of the `Rectangle` along the horizontal axis (left to right). Width should not be negative; however, there is nothing within the class to prevent this from happening.

public int height

The height variable represents the size of the `Rectangle` along the vertical axis (top to bottom). Height should not be negative; however, there is nothing within the class to prevent this from happening.

Constructors

The following seven constructors create `Rectangle` objects. When you create a `Rectangle`, you provide the location of the top left corner, along with the `Rectangle`'s width and height. A `Rectangle` located at (0,0) with a width and height of 100 has its bottom right corner at (99, 99). The `Point` (100, 100) lies outside the `Rectangle`, since that would require a width and height of 101.

public Rectangle ()

This `Rectangle` constructor creates a `Rectangle` object in which x, y, width, and height are all 0.

public Rectangle (int width, int height)

This `Rectangle` constructor creates a `Rectangle` with (`x`, `y`) coordinates of (0,0) and the specified `width` and `height`. Notice that there is no `Rectangle(int x, int y)` constructor because that would have the same method signature as this one, and the compiler would have no means to differentiate them.

public Rectangle (int x, int y, int width, int height)

The `Rectangle` constructor creates a `Rectangle` object with an initial x coordinate of `x`, y coordinate of `y`, width of `width`, and height of `height`. Height and width should be positive, but the constructor does not check for this.

public Rectangle (Rectangle r)

This `Rectangle` constructor creates a `Rectangle` matching the original. The (`x`, `y`) coordinates are (`r.x`, `r.y`), with a width of `r.width` and a height of `r.height`.

public Rectangle (Point p, Dimension d)

This `Rectangle` constructor creates a `Rectangle` with (`x`, `y`) coordinates of (`p.x`, `p.y`), a width of `d.width`, and a height of `d.height`.

public Rectangle (Point p)

This `Rectangle` constructor creates a `Rectangle` with (`x`, `y`) coordinates of (`p.x`, `p.y`). The width and height are both zero.

public Rectangle (Dimension d)

The last `Rectangle` constructor creates a `Rectangle` with (`x`, `y`) coordinates of (0, 0). The initial `Rectangle` width is `d.width` and height is `d.height`.

Shaping and sizing

public Rectangle getBounds() ★

The `getBounds()` method returns a copy of the original `Rectangle`.

public void setBounds (int x, int y, int width, int height) ★
public void reshape (int x, int y, int width, int height) ☆

The `setBounds()` method changes the origin of the `Rectangle` to (`x`, `y`) and changes the dimensions to `width` by `height`.

`reshape()` is the Java 1.0 name for this method.

public void setBounds (Rectangle r) ★

The `setBounds()` method changes the origin of the `Rectangle` to (`r.x`, `r.y`) and changes the dimensions to `r.width` by `r.height`.

public Point getLocation() ★

The `getLocation()` retrieves the current origin of this rectangle as a `Point`.

public void setLocation (int x, int y) ★
public void move (int x, int y) ☆

The `setLocation()` method changes the origin of the `Rectangle` to (`x`, `y`).

`move()` is the Java 1.0 name for this method.

public void setLocation (Point p) ★

The `setLocation()` method changes the `Rectangle`'s origin to (`p.x`, `p.y`).

public void translate (int x, int y)

The `translate()` method moves the `Rectangle`'s origin by the amount (`x`, `y`). If the original `Rectangle`'s location (r) is (3, 4) and you call `r.translate (4, 5)`, then r's location becomes (7, 9). x and y may be negative. `translate()` has no effect on the `Rectangle`'s width and height.

public Dimension getSize () ★

The `getSize()` method retrieves the current size of the rectangle as a `Dimension`.

public void setSize() (int width, int height) ★
public void resize (int width, int height) ☆

The `setSize()` method changes the `Rectangle`'s dimensions to `width` x `height`.

`resize()` is the Java 1.0 name for this method.

public void setSize() (Dimension d) ★

The `setSize()` method changes the `Rectangle`'s dimensions to `d.width` x `d.height`.

public void grow (int horizontal, int vertical)

The `grow()` method increases the `Rectangle`'s dimensions by adding the amount `horizontal` on the left and the right and adding the amount `vertical` on the top and bottom. Therefore, all four of the rectangle's variables change. If the original location is (`x`, `y`), the new location will be (`x-horizontal`, `y-vertical`) (moving left and up if both values are positive); if the original size is (`width`, `height`), the new size will be (`width+2*horizontal`, `height+2*vertical`). Either horizontal or vertical can be negative to decrease the size of the `Rectangle`. The following code demonstrates the changes:

```
import java.awt.Rectangle;
public class rect {
    public static void main (String[] args) {
        Rectangle r = new Rectangle (100, 100, 200, 200);
        System.out.println (r);
        r.grow (50, 75);
        System.out.println (r);
        r.grow (-25, -50);
        System.out.println (r);
    }
}
```

This program produces the following output:

```
java.awt.Rectangle[x=100,y=100,width=200,height=200]
java.awt.Rectangle[x=50,y=25,width=300,height=350]
java.awt.Rectangle[x=75,y=75,width=250,height=250]
```

public void add (int newX, int newY)

The `add()` method incorporates the point (`newX`, `newY`) into the `Rectangle`. If this point is already in the `Rectangle`, there is no change. Otherwise, the size of the `Rectangle` increases to include (`newX`, `newY`) within itself.

public void add (Point p)

This `add()` method incorporates the point (`p.x`, `p.y`) into the `Rectangle`. If this point is already in the `Rectangle`, there is no change. Otherwise, the size of the `Rectangle` increases to include (`p.x`, `p.y`) within itself.

public void add (Rectangle r)

This `add()` method incorporates another `Rectangle r` into this `Rectangle`. This transforms the current rectangle into the union of the two `Rectangles`. This method might be useful in a drawing program that lets you select multiple objects on the screen and create a rectangular area from them.

We will soon encounter a method called `union()` that is almost identical. `add()` and `union()` differ in that `add()` modifies the current `Rectangle`, while `union()` returns a new `Rectangle`. The resulting rectangles are identical.

Intersections

public boolean contains (int x, int y) ★
public boolean inside (int x, int y) ☆

The `contains()` method determines if the point (`x`, `y`) is within this `Rectangle`. If so, `true` is returned. If not, `false` is returned.

`inside()` is the Java 1.0 name for this method.

public boolean contains (Point p) ★

The `contains()` method determines if the point (`p.x`, `p.y`) is within this `Rectangle`. If so, `true` is returned. If not, `false` is returned.

public boolean intersects (Rectangle r)

The `intersects()` method checks whether `Rectangle r` crosses this `Rectangle` at any point. If it does, `true` is returned. If not, `false` is returned.

public Rectangle intersection (Rectangle r)

The `intersection()` method returns a new `Rectangle` consisting of all points that are in both the current `Rectangle` and `Rectangle r`. For example, if `r = new Rectangle (50, 50, 100, 100)` and `r1 = new Rectangle (100, 100, 75, 75)`, then `r.intersection (r1)` is the `Rectangle (100, 100, 50, 50)`, as shown in Figure 2-13.

public Rectangle union (Rectangle r)

The `union()` method combines the current `Rectangle` and `Rectangle r` to form a new `Rectangle`. For example, if `r = new Rectangle (50, 50, 100, 100)` and `r1 = new Rectangle (100, 100, 75, 75)`, then `r.union (r1)` is the `Rectangle (50, 50, 125, 125)`. The original rectangle is unchanged. Figure 2-14 demonstrates the effect of `union()`. Because `fillRect()` fills to `width-1`

and `height-1`, the rectangle drawn appears slightly smaller than you would expect. However, that's an artifact of how rectangles are drawn; the returned rectangle contains all the points within both.

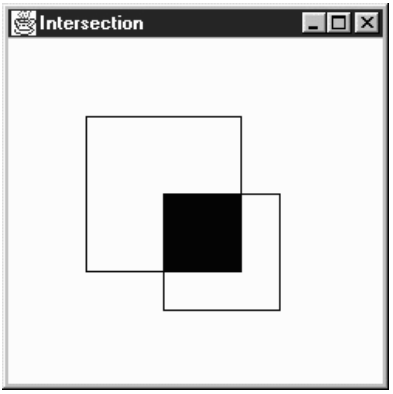

Figure 2–13: Rectangle intersection

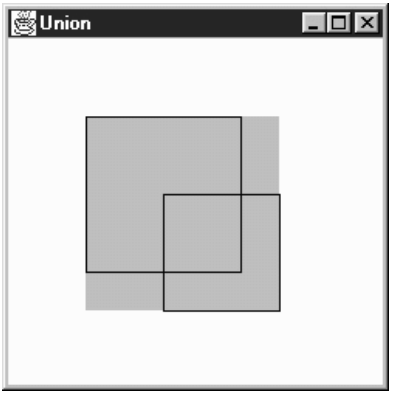

Figure 2–14: Rectangle union

Miscellaneous methods

public boolean isEmpty ()

The `isEmpty()` method checks whether there are any points within the `Rectangle`. If the width and height of the `Rectangle` are both 0 (or less), the `Rectangle` is empty, and this method returns `true`. If either width or height is greater than zero, `isEmpty()` returns `false`. This method could be used to check the results of a call to any method that returns a `Rectangle` object.

public int hashCode ()

The `hashCode()` method returns a hash code for the rectangle. The system calls this method when a `Rectangle` is used as the key for a hash table.

public boolean equals (Object object)

The `equals()` method overrides the `Object`'s `equals()` method to define what equality means for `Rectangle` objects. Two `Rectangle` objects are equal if their `x`, `y`, width, and height values are equal.

public String toString ()

The `toString()` method of `Rectangle` displays the current values of the `x`, `y`, width, and height variables. For example:

```
java.awt.Rectangle[x=100,y=200,width=300,height=400]
```

2.6 Polygon

A `Polygon` is a collection of points used to create a series of line segments. Its primary purpose is to draw arbitrary shapes like triangles or pentagons. If the points are sufficiently close, you can create a curve. To display the `Polygon`, call `drawPolygon()` or `fillPolygon()`.

2.6.1 Polygon Methods

Variables

The collection of points maintained by `Polygon` are stored in three variables:

public int npoints

The `npoints` variable stores the number of points.

public int xpoints[]

The `xpoints` array holds the x component of each point.

public int ypoints[]

The `ypoints` array holds the y component of each point.

You might expect the `Polygon` class to use an array of points, rather than separate arrays of integers. More important, you might expect the instance variables to be private or protected, which would prevent them from being modified directly. Since the three instance variables are public, there is no guarantee that the array sizes are in sync with each other or with `npoints`. To avoid trouble, always use `addPoints()` to modify your polygons, and avoid modifying the instance variables directly.

Constructors

public Polygon ()

This constructor creates an empty `Polygon`.

public Polygon (int xPoints[], int yPoints[], int numPoints)

This constructor creates a `Polygon` that consists of `numPoints` points. Those points are formed from the first `numPoints` elements of the `xPoints` and `yPoints` arrays. If the `xPoints` or `yPoints` arrays are larger than `numPoints`, the additional entries are ignored. If the `xPoints` or `yPoints` arrays do not contain at least `numPoints` elements, the constructor throws the run-time exception `ArrayIndexOutOfBoundsException`.

Miscellaneous methods

public void addPoint (int x, int y)

The `addPoint()` method adds the point (`x`, `y`) to the `Polygon` as its last point. If you alter the `xpoints`, `ypoints`, and `npoints` instance variables directly, `addPoint()` could add the new point at a place other than the end, or it could throw the run-time exception `ArrayIndexOutOfBoundsException` with a message showing the position at which it tried to add the point. Again, for safety, don't modify a `Polygon`'s instance variables yourself; always use `addPoint()`.

public Rectangle getBounds () ★
public Rectangle getBoundingBox () ☆

The `getBounds()` method returns the `Polygon`'s bounding `Rectangle` (i.e., the smallest rectangle that contains all the points within the polygon). Once you have the bounding box, it's easy to use methods like `copyArea()` to copy the `Polygon`.

`getBoundingBox()` is the Java 1.0 name for this method.

public boolean contains (int x, int y) ★
public boolean inside (int x, int y) ☆

The `contains()` method checks to see if the (`x`, `y`) point is within an area that would be filled if the `Polygon` was drawn with `Graphics.fillPolygon()`. A point may be within the bounding rectangle of the polygon, but `contains()` can still return `false` if not within a closed part of the polygon.

`inside()` is the Java 1.0 name for this method.

public boolean contains (Point p) ★

The `contains()` method checks to see if the point `p` is within an area that would be filled if the `Polygon` were drawn with `Graphics.fillPolygon()`.

public void translate (int x, int y) ★

The `translate()` method moves all the `Polygon`'s points by the amount `(x, y)`. This allows you to alter the location of the `Polygon` by shifting the points.

2.7 Image

An `Image` is a displayable object maintained in memory. AWT has built-in support for reading files in GIF and JPEG format, including GIF89a animation. Netscape Navigator, Internet Explorer, HotJava, and Sun's JDK also understand the XBM image format. Images are loaded from the filesystem or network by the `getImage()` method of either `Component` or `Toolkit`, drawn onto the screen with `drawImage()` from `Graphics`, and manipulated by several objects within the `java.awt.image` package. Figure 2-15 shows an `Image`.

Figure 2–15: An Image

`Image` is an abstract class implemented by many different platform-specific classes. The system that runs your program will provide an appropriate implementation; you do not need to know anything about the platform-specific classes, because the `Image` class completely defines the API for working with images. If you're curious, the platform-specific packages used by the JDK are:

- `sun.awt.win32.Win32Image` on Java 1.0 Windows NT/95 platforms
- `sun.awt.windows.WImage` on Java 1.1 Windows NT/95 platforms
- `sun.awt.motif.X11Image` on UNIX/Motif platforms
- `sun.awt.macos.MacImage` on the Macintosh

This section covers only the `Image` object itself. AWT also includes a package named `java.awt.image` that includes more advanced image processing utilities. The classes in `java.awt.image` are covered in Chapter 12.

2.7.1 Image Methods

Constants

public static final Object UndefinedProperty

In Java 1.0, the sole constant of `Image` is `UndefinedProperty`. It is used as a return value from the `getProperty()` method to indicate that the requested property is unavailable.

Java 1.1 introduces the `getScaledInstance()` method. The final parameter to the method is a set of hints to tell the method how best to scale the image. The following constants provide possible values for this parameter.

public static final int SCALE_DEFAULT ★

The `SCALE_DEFAULT` hint should be used alone to tell `getScaledInstance()` to use the default scaling algorithm.

public static final int SCALE_FAST ★

The `SCALE_FAST` hint tells `getScaledInstance()` that speed takes priority over smoothness.

public static final int SCALE_SMOOTH ★

The `SCALE_SMOOTH` hint tells `getScaledInstance()` that smoothness takes priority over speed.

public static final int SCALE_REPLICATE ★

The `SCALE_REPLICATE` hint tells `getScaledInstance()` to use `ReplicateScaleFilter` or a reasonable alternative provided by the toolkit. `ReplicateScaleFilter` is discussed in Chapter 12.

public static final int SCALE_AREA_AVERAGING ★

The `SCALE_AREA_AVERAGING` hint tells `getScaledInstance()` to use `AreaAveragingScaleFilter` or a reasonable alternative provided by the toolkit. `AreaAveragingScaleFilter` is discussed in Chapter 12.

Constructors

There are no constructors for `Image`. You get an `Image` object to work with by using the `getImage()` method of `Applet` (in an applet), `Toolkit` (in an application), or the `createImage()` method of `Component` or `Toolkit`. `getImage()` uses a separate thread to fetch the image. The thread starts when you call `drawImage()`, `prepareImage()`, or any other method that requires image information. `getImage()` returns immediately. You can also use the `MediaTracker` class to force an image to load before it is needed. `MediaTracker` is discussed in the next section.

Characteristics

public abstract int getWidth (ImageObserver observer)

The `getWidth()` method returns the width of the image object. The width may not be available if the image has not started loading; in this case, `getWidth()` returns –1. An image's size is available long before loading is complete, so it is often useful to call `getWidth()` while the image is loading.

public abstract int getHeight (ImageObserver observer)

The `getHeight()` method returns the height of the image object. The height may not be available if the image has not started loading; in this case, the `getHeight()` method returns –1. An image's size is available long before loading is complete, so it is often useful to call `getHeight()` while the image is loading.

Miscellaneous methods

public Image getScaledInstance (int width, int height, int hints) ★

The `getScaledInstance()` method enables you to generate scaled versions of images before they are needed. Prior to Java 1.1, it was necessary to tell the `drawImage()` method to do the scaling. However, this meant that scaling didn't take place until you actually tried to draw the image. Since scaling takes time, drawing the image required more time; the net result was degraded appearance. With Java 1.1, you can generate scaled copies of images before drawing them; then you can use a version of `drawImage()` that does not do scaling, and therefore is much quicker.

The `width` parameter of `getScaledInstance()` is the new width of the image. The `height` parameter is the new height of the image. If either is –1, the scaling retains the aspect ratio of the original image. For instance, if the original image size was 241 by 72 pixels, and `width` and `height` were 100 and –1, the new image size would be 100 by 29 pixels. If both width and height are –1, the `getScaledInstance()` method retains the image's original size. The `hints` parameter is one of the `Image` class constants.

```
Image i = getImage (getDocumentBase(), "rosey.jpg");
Image j = i.getScaledInstance (100, -1, Image.SCALE_FAST);
```

public abstract ImageProducer getSource ()

The `getSource()` method returns the image's producer, which is an object of type `ImageProducer`. This object represents the image's source. Once you have the `ImageProducer`, you can use it to do additional image processing; for example, you could create a modified version of the original image by using a `FilteredImageSource`. Image producers and image filters are covered in Chapter 12.

public abstract Graphics getGraphics ()

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)`. If the image came from a URL or a file (i.e., from `getImage()`), `getGraphics()` throws the run-time exception `ClassCastException`.

public abstract Object getProperty (String name, ImageObserver observer)

The `getProperty()` method interacts with the image's property list. An object representing the requested property `name` will be returned for `observer`. `observer` represents the `Component` on which the image is rendered. If the property `name` exists but is not available yet, `getProperty()` returns `null`. If the property name does not exist, the `getProperty()` method returns the `Image.UndefinedProperty` object.

Each image type has its own property list. A property named `comment` stores a comment `String` from the image's creator. The `CropImageFilter` adds a property named `croprect`. If you ask `getProperty()` for an image's `croprect` property, you get a `Rectangle` that shows how the original image was cropped.

public abstract void flush()

The `flush()` method resets an image to its initial state. Assume you acquire an image over the network with `getImage()`. The first time you display the image, it will be loaded over the network. If you redisplay the image, AWT normally reuses the original image. However, if you call `flush()` before redisplaying the image, AWT fetches the image again from its source. (Images created with `createImage()` aren't affected.) The `flush()` method is useful if you expect images to change while your program is running. The following program demonstrates `flush()`. It reloads and displays the file *flush.gif* every time you click the mouse. If you change the file *flush.gif* and click on the mouse, you will see the new file.

```
import java.awt.*;
public class flushMe extends Frame {
    Image im;
    flushMe () {
        super ("Flushing");
        im = Toolkit.getDefaultToolkit().getImage ("flush.gif");
        resize (175, 225);
    }
    public void paint (Graphics g) {
        g.drawImage (im, 0, 0, 175, 225, this);
    }
    public boolean mouseDown (Event e, int x, int y) {
        im.flush();
        repaint();
        return true;
    }
```

```
    public static void main (String [] args) {
        Frame f = new flushMe ();
        f.show();
    }
}
```

2.7.2 Simple Animation

Creating simple animation sequences in Java is easy. Load a series of images, then display the images one at a time. Example 2-1 is an application that displays a simple animation sequence. Example 2-2 is an applet that uses a thread to run the application. These programs are far from ideal. If you try them, you'll probably notice some flickering or missing images. We discuss how to fix these problems shortly.

Example 2–1: Animation Application

```
import java.awt.*;
public class Animate extends Frame {
    static Image im[];
    static int numImages = 12;
    static int counter=0;
    Animate () {
        super ("Animate");
    }
    public static void main (String[] args) {
        Frame f = new Animate();
        f.resize (225, 225);
        f.show();
        im = new Image[numImages];
        for (int i=0;i<numImages;i++) {
            im[i] = Toolkit.getDefaultToolkit().getImage ("clock"+i+".jpg");
        }
    }
    public synchronized void paint (Graphics g) {
        g.translate (insets().left, insets().top);
        g.drawImage (im[counter], 0, 0, this);
        counter++;
        if (counter == numImages)
            counter = 0;
        repaint (200);
    }
}
```

This application displays images with the name *clock*n*.jpg*, where *n* is a number between 0 and 11. It fetches the images using the `getImage()` method of the `Toolkit` class—hence, the call to `Toolkit.getDefaultToolkit()`, which gets a `Toolkit` object to work with. The `paint()` method displays the images in sequence, using `drawImage()`. `paint()` ends with a call to `repaint(200)`, which schedules another call to `paint()` in 200 milliseconds.

The `AnimateApplet`, whose code is shown in Example 2-2, does more or less the same thing. It is able to use the `Applet.getImage()` method. A more significant difference is that the applet creates a new thread to control the animation. This thread calls `sleep(200)`, followed by `repaint()`, to display a new image every 200 milliseconds.

Example 2–2: Multithreaded Animation Applet

```
import java.awt.*;
import java.applet.*;
public class AnimateApplet extends Applet implements Runnable {
    static Image im[];
    static int numImages = 12;
    static int counter=0;
    Thread animator;
    public void init () {
        im = new Image[numImages];
        for (int i=0;i<numImages;i++)
            im[i] = getImage (getDocumentBase(), "clock"+i+".jpg");
    }
    public void start() {
        if (animator == null) {
            animator = new Thread (this);
            animator.start ();
        }
    }
    public void stop() {
        if ((animator != null) && (animator.isAlive())) {
            animator.stop();
            animator = null;
        }
    }
    public void run () {
        while (animator != null) {
            try {
                animator.sleep(200);
                repaint ();
                counter++;
                if (counter==numImages)
                    counter=0;
            } catch (Exception e) {
                e.printStackTrace ();
            }
        }
    }
    public void paint (Graphics g) {
        g.drawImage (im[counter], 0, 0, this);
    }
}
```

One quick fix will help the flicker problem in both of these examples. The

update() method (which is inherited from the Component class) normally clears the drawing area and calls paint(). In our examples, clearing the drawing area is unnecessary and, worse, results in endless flickering; on slow machines, you'll see update() restore the background color between each image. It's a simple matter to override update() so that it doesn't clear the drawing area first. Add the following method to both of the previous examples:

```
public void update (Graphics g) {
    paint (g);
}
```

Overriding update() helps, but the real solution to our problem is double buffering, which we'll turn to next.

2.7.3 Double Buffering

Double buffering means drawing to an offscreen graphics context and then displaying this graphics context to the screen in a single operation. So far, we have done all our drawing directly on the screen—that is, to the graphics context provided by the paint() method. As your programs grow more complex, paint() gets bigger and bigger, and it takes more time and resources to update the entire drawing area. On a slow machine, the user will see the individual drawing operations take place, which will make your program look slow and clunky. By using the double buffering technique, you can take your time drawing to another graphics context that isn't displayed. When you are ready, you tell the system to display the completely new image at once. Doing so eliminates the possibility of seeing partial screen updates and flickering.

The first thing you need to do is create an image as your drawing canvas. To get an image object, call the createImage() method. createImage() is a method of the Component class, which we will discuss in Chapter 5, *Components*. Since Applet extends Component, you can call createImage() within an applet. When creating an application and extending Frame, createImage() returns null until the Frame's peer exists. To make sure that the peer exists, call addNotify() in the constructor, or make sure you call show() before calling createImage(). Here's the call to the createImage() method that we'll use to get an Image object:

```
Image im = createImage (300, 300); // width and height
```

Once you have an Image object, you have an area you can draw on. But how do you draw on it? There are no drawing methods associated with Image; they're all in the Graphics class. So we need to get a Graphics context from the Image. To do so, call the getGraphics() method of the Image class, and use that Graphics context for your drawing:

```
Graphics buf = im.getGraphics();
```

Now you can do all your drawings with `buf`. To display the drawing, the `paint()` method only needs to call `drawImage(im, . . .)`. Note the hidden connection between the `Graphics` object, `buf`, and the `Image` you are creating, `im`. You draw onto `buf`; then you use `drawImage()` to render the image on the on-screen `Graphics` context within `paint()`.

Another feature of buffering is that you do not have redraw the entire image with each call to `paint()`. The buffered image you're working on remains in memory, and you can add to it at will. If you are drawing directly to the screen, you would have to recreate the entire drawing each time `paint()` is called; remember, `paint()` always hands you a completely new `Graphics` object. Figure 2-16 shows how double buffering works.

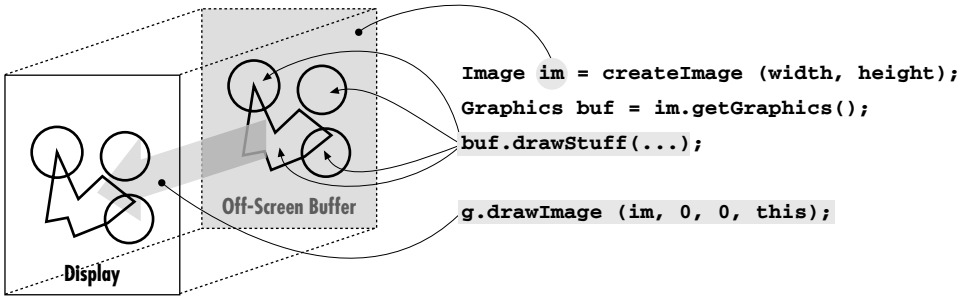

Figure 2–16: Double buffering

Example 2-3 puts it all together for you. It plays a game, with one move drawn to the screen each cycle. We still do the drawing within `paint()`, but we draw into an offscreen buffer; that buffer is copied onto the screen by `g.drawImage(im, 0, 0, this)`. If we were doing a lot of drawing, it would be a good idea to move the drawing operations into a different thread, but that would be overkill for this simple applet.

Example 2–3: Double Buffering—Who Won?

```
import java.awt.*;
import java.applet.*;
public class buffering extends Applet {
    Image im;
    Graphics buf;
    int pass=0;
    public void init () {
        // Create buffer
        im = createImage (size().width, size().height);
        // Get its graphics context
        buf = im.getGraphics();
        // Draw Board Once
```

Example 2–3: Double Buffering—Who Won? (continued)

```
            buf.setColor (Color.red);
            buf.drawLine (  0,  50, 150,  50);
            buf.drawLine (  0, 100, 150, 100);
            buf.drawLine ( 50,   0,  50, 150);
            buf.drawLine (100,   0, 100, 150);
            buf.setColor (Color.black);
        }
        public void paint (Graphics g) {
            // Draw image - changes are written onto buf
            g.drawImage (im, 0, 0, this);
            // Make a move
            switch (pass) {
                case 0:
                    buf.drawLine (50, 50, 100, 100);
                    buf.drawLine (50, 100, 100, 50);
                    break;
                case 1:
                    buf.drawOval (0, 0, 50, 50);
                    break;
                case 2:
                    buf.drawLine (100, 0, 150, 50);
                    buf.drawLine (150, 0, 100, 50);
                    break;
                case 3:
                    buf.drawOval (0, 100, 50, 50);
                    break;
                case 4:
                    buf.drawLine (0, 50, 50, 100);
                    buf.drawLine (0, 100, 50, 50);
                    break;
                case 5:
                    buf.drawOval (100, 50, 50, 50);
                    break;
                case 6:
                    buf.drawLine (50,  0, 100, 50);
                    buf.drawLine (50, 50, 100,  0);
                    break;
                case 7:
                    buf.drawOval (50, 100, 50, 50);
                    break;
                case 8:
                    buf.drawLine (100, 100, 150, 150);
                    buf.drawLine (150, 100, 100, 150);
                    break;
            }
            pass++;
            if (pass <= 9)
                repaint (500);
        }
    }
```

2.8 MediaTracker

The `MediaTracker` class assists in the loading of multimedia objects across the network. Tracking is necessary because Java loads images in separate threads. Calls to `getImage()` return immediately; image loading starts only when you call the method `drawImage()`. `MediaTracker` lets you force images to start loading before you try to display them; it also gives you information about the loading process, so you can wait until an image is fully loaded before displaying it.

Currently, `MediaTracker` can monitor the loading of images, but not audio, movies, or anything else. Future versions are rumored to be able to monitor other media types.

2.8.1 MediaTracker Methods

Constants

The `MediaTracker` class defines four constants that are used as return values from the class's methods. These values serve as status indicators.

public static final int LOADING
: The `LOADING` variable indicates that the particular image being checked is still loading.

public static final int ABORTED
: The `ABORTED` variable indicates that the loading process for the image being checked aborted. For example, a timeout could have happened during the download. If something `ABORTED` during loading, it is possible to `flush()` the image to force a retry.

public static final int ERRORED
: The `ERRORED` variable indicates that an error occurred during the loading process for the image being checked. For instance, the image file might not be available from the server (invalid URL) or the file format could be invalid. If an image has `ERRORED`, retrying it will fail.

public static final int COMPLETE
: The `COMPLETE` flag means that the image being checked successfully loaded.

If `COMPLETE`, `ABORTED`, or `ERRORED` is set, the image has stopped loading. If you are checking multiple images, you can OR several of these values together to form a composite. For example, if you are loading several images and want to find out about any malfunctions, call `statusAll()` and check for a return value of `ABORTED | ERRORED`.

Constructors

public MediaTracker (Component component)

The `MediaTracker` constructor creates a new `MediaTracker` object to track images to be rendered onto `component`.

Adding images

The `addImage()` methods add objects for the `MediaTracker` to track. When placing an object under a `MediaTracker`'s control, you must provide an identifier for grouping purposes. When multiple images are grouped together, you can perform operations on the entire group with a single request. For example, you might want to wait until all the images in an animation sequence are loaded before starting the animation; in this case, assigning the same ID to all the images makes good sense. However, when multiple images are grouped together, you cannot check on the status of a single image. The moral is: if you care about the status of individual images, put each into a group by itself.

Folklore has it that the identifier also serves as a loading priority, with a lower ID meaning a higher priority. This is not completely true. Current implementations start loading lower IDs first, but at most, this is implementation-specific functionality for the JDK. Furthermore, although an object with a lower identifier might be told to start loading first, the `MediaTracker` does nothing to ensure that it finishes first.

public synchronized void addImage (Image image, int id, int width, int height)

The `addImage()` method tells the `MediaTracker` instance that it needs to track the loading of `image`. The `id` is used as a grouping. Someone will eventually render the `image` at a scaled size of `width height`. If `width` and `height` are both –1, the image will be rendered unscaled. If you forget to notify the `MediaTracker` that the `image` will be scaled and ask the `MediaTracker` to `waitForID (id)`, it is possible that the image may not be fully ready when you try to draw it.

public void addImage (Image image, int id)

The `addImage()` method tells the `MediaTracker` instance that it needs to track the loading of `image`. The `id` is used as a grouping. The `image` will be rendered at its actual size, without scaling.

Removing images

Images that have finished loading are still watched by the `MediaTracker`. The `removeImage()` methods, added in Java 1.1, allow you to remove objects from the `MediaTracker`. Once you no longer care about an image (usually after waiting for

it to load), you can remove it from the tracker. Getting rid of loaded images results in better performance because the tracker has fewer objects to check. In Java 1.0, you can't remove an image from `MediaTracker`.

public void removeImage (Image image) ★

The `removeImage()` method tells the `MediaTracker` to remove all instances of `image` from its tracking list.

public void removeImage (Image image, int id) ★

The `removeImage()` method tells the `MediaTracker` to remove all instances of `image` from group `id` of its tracking list.

public void removeImage (Image image, int id, int width, int height) ★

This `removeImage()` method tells the `MediaTracker` to remove all instances of `image` from group `id` and scale `width height` of its tracking list.

Waiting

A handful of methods let you wait for a particular image, image group, all images, or a particular time period. They enable you to be sure that an image has finished trying to load prior to continuing. The fact that an image has finished loading does not imply it has successfully loaded. It is possible that an error condition arose, which caused loading to stop. You should check the status of the image (or group) for actual success.

public void waitForID (int id) throws InterruptedException

The `waitForID()` method blocks the current thread from running until the images added with `id` finish loading. If the wait is interrupted, `waitForID()` throws an `InterruptedException`.

public synchronized boolean waitForID (int id, long ms) throws InterruptedException

The `waitForID()` method blocks the current thread from running until the images added with `id` finish loading or until `ms` milliseconds have passed. If all the images have loaded, `waitForID()` returns `true`; if the timer has expired, it returns `false`, and one or more images in the `id` set have not finished loading. If `ms` is 0, it waits until all images added with `id` have loaded, with no timeout. If the wait is interrupted, `waitForID()` throws an `InterruptedException`.

public void waitForAll () throws InterruptedException

The `waitForAll()` method blocks the current thread from running until all images controlled by this `MediaTracker` finish loading. If the wait is interrupted, `waitForAll()` throws an `InterruptedException`.

public synchronized boolean waitForAll (long ms) throws InterruptedException

The `waitForAll()` method blocks the current thread from running until all images controlled by this `MediaTracker` finish loading or until `ms` milliseconds have passed. If all the images have loaded, `waitForAll()` returns `true`; if the timer has expired, it returns `false`, and one or more images have not finished loading. If `ms` is 0, it waits until all images have loaded, with no timeout. When you interrupt the waiting, `waitForAll()` throws an `InterruptedException`.

Checking status

Several methods are available to check on the status of images loading. None of these methods block, so you can continue working while images are loading.

public boolean checkID (int id)

The `checkID()` method determines if all the images added with the `id` tag have finished loading. The method returns `true` if all images have completed loading (successfully or unsuccessfully). Since this can return `true` on error, you should also use `isErrorID()` to check for errors. If loading has not completed, `checkID()` returns `false`. This method does not force images to start loading.

public synchronized boolean checkID (int id, boolean load)

The `checkID()` method determines if all the images added with the `id` tag have finished loading. If the `load` flag is `true`, any images in the `id` group that have not started loading yet will start. The method returns `true` if all images have completed loading (successfully or unsuccessfully). Since this can return `true` on error, you should also use `isErrorID()` to check for errors. If loading has not completed, `checkID()` returns `false`.

public boolean checkAll ()

The `checkAll()` method determines if all images associated with the `MediaTracker` have finished loading. The method returns `true` if all images have completed loading (successfully or unsuccessfully). Since this can return `true` on error, you should also use `isErrorAny()` to check for errors. If loading has not completed, `checkAll()` returns `false`. This method does not force images to start loading.

public synchronized boolean checkAll (boolean load)

The `checkAll()` method determines if all images associated with the `MediaTracker` have finished loading. If the `load` flag is `true`, any image that has not started loading yet will start. The method returns `true` if all images have completed loading (successfully or unsuccessfully). Since this can return `true` on error, you should also use `isErrorAny()` to check for errors. If loading has not completed, `checkAll()` returns `false`.

public int statusID (int id, boolean load)

The `statusID()` method checks on the load status of the images in the `id` group. If there are multiple images in the group, the results are ORed together. If the `load` flag is `true`, any image in the `id` group that has not started loading yet will start. The return value is some combination of the class constants `LOADING`, `ABORTED`, `ERRORED`, and `COMPLETE`.

public int statusAll (boolean load)

The `statusAll()` method determines the load status of all the images associated with the `MediaTracker`. If this `MediaTracker` is watching multiple images, the results are ORed together. If the `load` flag is `true`, any image that has not started loading yet will start. The return value is some combination of the class constants `LOADING`, `ABORTED`, `ERRORED`, and `COMPLETE`.

public synchronized boolean isErrorID (int id)

The `isErrorId()` method checks whether any media in the `id` group encountered an error while loading. If any image resulted in an error, `isErrorId()` returns `true`; if there were no errors, it returns `false`.

public synchronized boolean isErrorAny ()

The `isErrorAny()` method checks to see if any image associated with the `MediaTracker` encountered an error. If there was an error, the method returns `true`; if none, `false`.

public synchronized Object[] getErrorsID (int id)

The `getErrorsID()` method returns an array of the objects that encountered errors in the group ID during loading. If loading caused no errors, the method returns `null`. The return type is an `Object` array instead of an `Image` array because `MediaTracker` will eventually support additional media types.

public synchronized Object[] getErrorsAny ()

The `getErrorsAny()` method returns an array of all the objects that encountered an error during loading. If there were no errors, the method returns `null`. The return type is an `Object` array instead of an `Image` array because `MediaTracker` will eventually support additional media types.

2.8.2 Using a MediaTracker

The `init()` method improves the `AnimateApplet` from Example 2-2 to ensure that images load before the animation sequence starts. Waiting for images to load is particularly important if there is a slow link between the computer on which the applet is running and the server for the image files. Note that in a few cases, like interlaced GIF files, you might be willing to display an image before it has completely loaded. However, judicious use of `MediaTracker` will give you much more control over your program's behavior.

The new `init()` method creates a `MediaTracker`, puts all the images in the animation sequence under the tracker's control, and then calls `waitForAll()` to wait until the images are loaded. Once the images are loaded, it calls `isErrorsAny()` to make sure that the images loaded successfully.

```
public void init () {
    MediaTracker mt = new MediaTracker (this);
    im = new Image[numImages];
    for (int i=0;i<numImages;i++) {
        im[i] = getImage (getDocumentBase(), "clock"+i+".jpg");
        mt.addImage (im[i], i);
    }
    try {
        mt.waitForAll();
        if (mt.isErrorAny())
            System.out.println ("Error loading images");
    } catch (Exception e) {
        e.printStackTrace ();
    }
}
```