CHAPTER

10

GRAPHICS

In this chapter, you will:

- Learn about the paint() and repaint() methods
- Use the drawString() method to draw Strings
- Use the setFont() and setColor() Graphics object methods
- Create Graphics and Graphics 2D objects
- Draw lines, rectangles, ovals, arcs, and polygons
- Copy an area
- Learn more about fonts and their methods
- Draw with Java 2D graphics
- Add sound, images, and simple animations to Swing applets

What are you smiling about?" your mentor, Lynn Greenbrier, asks as she walks by your desk at Event Handlers Incorporated.

"I liked Java programming from the start," you say, "but now that I'm creating applets, I'm really having fun."

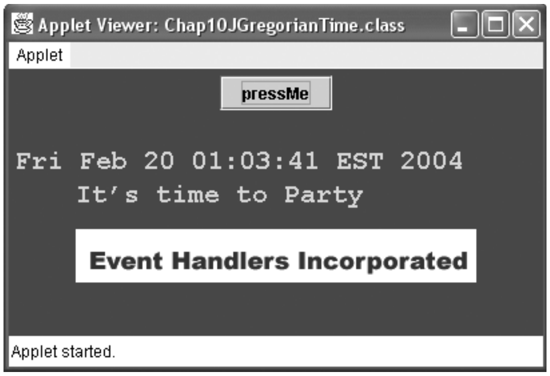
"If you like what you've done with applets so far," Lynn smiles, "just wait until you add colors, shapes, images, and sound. Let me show you how to use graphics and multimedia to add some sizzle."

Previewing the JGregorianTime Swing Applet

The Chap10JGregorianTime Swing applet works as an interactive advertisement for Event Handlers Incorporated and demonstrates several graphics methods. You can now use a completed version of the Swing applet that is saved in the Chapter.10 folder on your Student Disk.

To run the Chap10JGregorianTime Swing applet:

1. Go to the command prompt for the Chapter.10 folder on your Student Disk, type **`appletviewer TestChap10JGregorianTime.html`**, and then press **[Enter]**. It might take a few minutes for the Applet Viewer window shown in Figure 10-1 to open.


Applet Viewer: Chap10JGregorianTime.class
Applet
pressMe

```
Fri Feb 20 01:03:41 EST 2004
    It's time to Party
```

Event Handlers Incorporated
Applet started.

Figure 10-1 Chap10JGregorianTime Swing applet

2. Use the Swing applet by clicking the **PressMe** button to see the time display change. Click the **PressMe** button as many times as you like.
3. Click the **Close** button to close the Applet Viewer window.

Learning About the Paint() and Repaint() Methods

In Chapter 9, you learned that every Swing applet uses four methods: init(), start(), stop(), and destroy(). If you don't write these methods, Java provides you with a "do nothing" copy. You can, however, override any of these automatically supplied methods by writing your own versions.

Actually, a fifth method is used within every Swing applet. The **paint() method** runs when Java displays your Swing applet. You can write your own paint() method to override the automatically supplied one whenever you want to paint graphics, such as shapes, on the screen. As with init(), start(), stop(), and destroy() methods, if you don't write a paint() method, you get an automatic version from Java. The paint() method executes automatically every time you minimize, maximize, or resize the Applet Viewer window.

The paint() method header is `public void paint (Graphics g)`. The header indicates that the method requires a Graphics object argument; here it is named g but you can use any legal identifier. However, you don't usually call the paint() method directly. Instead, you call the **repaint() method**, which you can use when a window needs to be updated, such as when it contains new images. The Java system calls the repaint() method when it needs to update a window, or you can call it yourself—in either case, repaint() creates a Graphics object for you. The repaint() method calls another method named update(), which calls the paint() method. The series of events is best described with an example that you will create in the following steps.

To demonstrate how repaint() and paint() operate:

1. Open a new text file in your text editor.
2. Type the following first few lines of a Swing applet named JDemoPaint:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JDemoPaint extends JApplet
 implements ActionListener
{
```

3. The only component in this Swing applet is a JButton that you can create by typing the following code on the next line: **`JButton pressButton = new JButton("Press");`**.
4. Type the following init() method, which initializes a Container named `con`, sets the `con` layout to FlowLayout, and adds the pressButton to `con`:

```
public void init()
{
 Container con = getContentPane();
 con.setLayout(new FlowLayout() );
 con.add(pressButton);
 pressButton.addActionListener(this);
}
```

5. Override the paint() method by typing the following code, so it prints a message to the screen every time it executes:

```
public void paint(Graphics g)
{
  System.out.println("in paint method");
  pressButton.repaint();
}
```

6. Call the repaint() method when the user clicks the JButton by typing the following:

```
public void actionPerformed(ActionEvent e)
{
 Object source = e.getSource();
 if (source == pressButton)
 {
   repaint();
 }
}
```

7. Add the closing curly brace for the class, and then save the file as **JDemoPaint.java** in the Chapter.10 folder on your Student Disk. Compile the Swing applet using the **`javac JDemoPaint.java`** command.
8. Open a new text file, and then create the following HTML document to host the Swing applet:

```
<HTML>
<APPLET CODE = "JDemoPaint.class"
  WIDTH = 100 HEIGHT = 100>
</APPLET>
</HTML>
```

9. Save the file as **TestJDemoPaint.html** in the Chapter.10 folder on your Student Disk, and then type **`appletviewer TestJDemoPaint.html`** at the command prompt to run the Swing applet using the file. Make sure you can view the command line and the Swing applet on your screen. When the Swing applet starts, the paint() method executes automatically, so the message "in paint method" appears on the command line. Click the **pressButton** in the Swing applet. The actionPerformed() method calls the repaint() method, the repaint() method calls the update() method, which then calls the paint() method, so a second message appears on the command line. Minimize the Applet Viewer window and then restore it. Resize the window by dragging its border. With each action, an "in paint method" message appears on the command line, demonstrating all the conditions under which the paint() method executes.
10. Close the Applet Viewer window.

Tip: The repaint() method only requests that Java repaint the screen. If a second request to repaint() occurs before Java can carry out the first request, then Java executes only the last repaint() method.

Using the drawString() Method to Draw Strings

The **drawString() method** allows you to draw a String in a Swing applet window. The drawString() method requires three arguments: a String, an x-axis coordinate, and a y-axis coordinate.

You are already familiar with x- and y-axis coordinates because you used them with the setLocation() method for components in Chapter 9. However, there is a minor difference in how you place components using the setLocation() method and how you place Strings using the drawString() method. When you use x- and y-coordinates with components, such as JLabels, the upper-left corner of the component is placed at the coordinate position. When you use x- and y-coordinates with drawString(), the lower-left corner of the String appears at the coordinates. Figure 10-2 shows the positions of a JLabel placed at the coordinates 30, 10 and a String placed at the coordinates 10, 30.

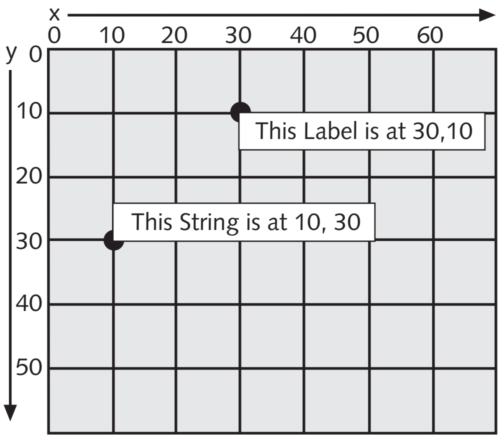


Figure 10-2 JLabel and String coordinates

The drawString() method is a member of the Graphics class, so you need to use a Graphics object to call it. Recall that the paint() method header shows that the method receives a Graphics object from the update() method. If you use drawString() within paint(), then the Graphics object you name in the header is available to you. For example, if you write a paint() method with the header `public void paint(Graphics brush)`, then you can draw a String within your paint() method by using a statement such as `brush.drawString("Hi",50,80);`.

To use drawString() to place a String within a Swing applet:

1. Open a new text file, and begin a class definition for a JDemoGraphics class by typing the following:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class JDemoGraphics extends JApplet
{
```

2. Declare a String to hold the company name for Event Handlers Incorporated by typing **`String companyName = new String("Event Handlers Incorporated");`**.

3. Type the following paint() method that uses a Graphics object to draw the companyName String:

```
 public void paint(Graphics gr)
 {
 gr.drawString(companyName,10,100);
 }
}
```

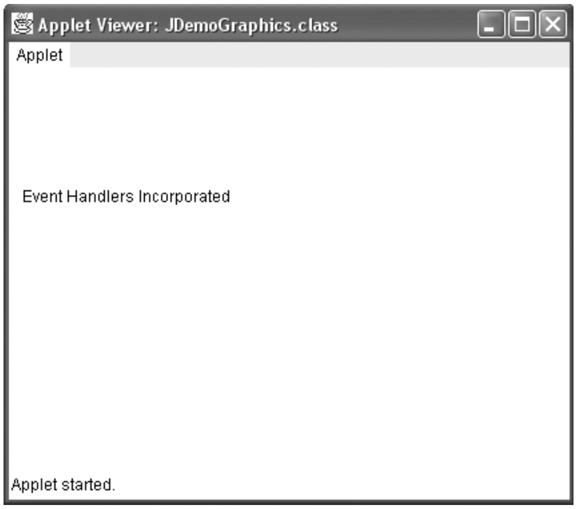
4. Save the file as **JDemoGraphics.java** in the Chapter.10 folder on your Student Disk, and then compile at the command prompt using the **`javac`** command.

5. Open a new text file, and then create the following HTML document for the JDemoGraphics1 class:

```
<HTML>
<APPLET CODE =
 "JDemoGraphics.class" WIDTH = 420 HEIGHT = 300>
</APPLET>
</HTML>
```

6. Save the file as **TestJDemoGraphics.html** in the Chapter.10 folder on your Student Disk, and then use the **`appletviewer TestJDemoGraphics.html`** command to run the program. The program's output appears in Figure 10-3.

7. Close the Applet Viewer window.



```
Event Handlers Incorporated
```

```
Applet started.
```

Figure 10-3 Swing applet using the drawString() method

Using the setFont() and setColor() Graphics Object Methods

You can improve the appearance of Graphics objects by using the setFont() and setColor() Graphics object methods. The setFont() method requires a Font object, which, as you recall, you create with a statement such as `Font someFont = new Font("TimesRoman", Font.BOLD, 16);`. Then you can instruct a Graphics object to use the font by using the font as the argument in a setFont() method. For example, if a Graphics object is named brush, then the font is set to someFont with **`brush.setFont(someFont);`**.

Tip

You learned about the Font object when you changed a JLabel's font in Chapter 9.

You can designate a Graphics color with the setColor() method. The Color class contains 13 constants that appear in Table 10-1. You can use any of these constants as an argument to the setColor() method. For example, you can instruct a Graphics object named brush to apply green paint by using the statement **`brush.setColor(Color.green);`**. Until you change the color, subsequent graphics output will appear as green.

Tip

Java constants are usually written in all uppercase letters, as you learned in Chapter 4. However, even though the color names of the Color class are constants, Java's creators failed to make them uppercase.

Table 10-1 Color class constants

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

Next you will use your knowledge of fonts and colors to set the color and font style of a Swing applet.

To add a Font and color to your JDemoGraphics class:

1. Open the JDemoGraphics.java text file in your text editor and rename the class **JDemoGraphics2**.
2. Just after the companyName declaration, add a Font object by typing:

```
Font bigFont = new Font("Helvetica", Font.ITALIC, 24);
```

3. For the first two statements in the paint() method after the opening curly brace, type the following statements so the gr object uses the bigFont object and the color magenta:

```
gr.setFont(bigFont);
gr.setColor(Color.magenta);
```

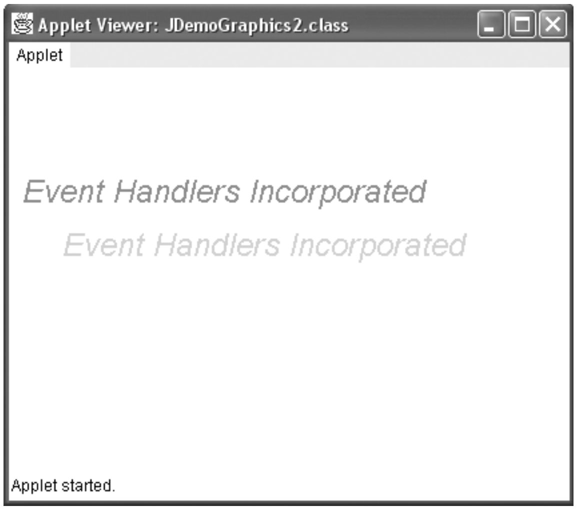
4. Following the existing drawString() method call, type the following lines to change the color and add an additional call to the drawString() method:

```
gr.setColor(Color.orange);
gr.drawString(companyName,40,140);
```

5. Save the file as **JDemoGraphics2.java** in the Chapter.10 folder on your Student Disk, and compile at the command prompt using the **`javac`** command. Modify the **TestJDemoGraphics.html** document for use with the JDemoGraphics2, and then save as **TestJDemoGraphics2.html**. Use the **`appletviewer TestJDemoGraphics2.html`** command to run the program. The program's output appears in Figure 10-4. Although the figure is shown in black and white in this book, notice that the Strings on your screen print as magenta and orange text.

Help The fonts that appear in your Swing applet might be different, depending on your computer's installed fonts. You will learn about installed fonts later in this chapter.

6. Close the Applet Viewer window.


Applet Viewer: JDemoGraphics2.class

Event Handlers Incorporated

Event Handlers Incorporated

Applet started.

Figure 10-4 JDemoGraphics2 Swing applet using font and color

You can also create your own Color object with the statement `Color someColor = new Color(r, g, b);`, where r, g, and b are numbers representing the intensity of red, green, and blue you want in your color. The numbers can range from 0 to 255, with 0 being the darkest shade of the color and 255 being the lightest. For example, `color darkPurple = new Color(100, 0, 100);` produces a dark purple color that has red and blue components, but no green. You can create more than 16 million custom colors using this approach.

Tip

Some computers cannot display each of the 16 million possible colors. Each computer will display the closest color it can.

You can discover the red, green, or blue components of any existing color with the methods getRed(), getGreen(), and getBlue(). Each of these methods returns an integer. For example, you can discover the amount of red in a magenta color by printing the value of `Color.magenta.getRed();`.

Next you will use the methods for getting and setting colors to display several hundred colors in a Swing applet.

To create a demonstration program that displays several hundred colors:

1. Open a new text file in your text editor.
2. Type the following import statements, a class header for a JDemoColor program, and the opening curly brace:

```
import javax.swing.*;
import java.awt.*;
public class JDemoColor extends JApplet
{
```

3. Define a small font by typing:

```
Font littleFont = new Font("Helvetica", Font.ITALIC, 6);
```

4. Add the following paint() method with five integer variables—r, g, b, x, and y.

```
public void paint(Graphics gr)
{
 int r, g , b;
 int x = 0, y = 0;
```

5. Set the Graphics object font by typing **`gr.setFont(littleFont);`**.
6. Create a `for` loop in which the red component will vary from 255 down to 0 in decrements of 20. Within the red `for` loop, vary the intensity of green, and within the green `for` loop, vary the intensity of blue. Although you won't get every possible combination of components, you will get a wide variety.

```
for(r = 255; r >= 0; r -= 20)
 for(g = 255; g >= 0; g -= 20)
  for(b = 255; b >= 0; b -= 20)
     {
```

7. Within the body of the innermost `for` loop, create a new color, set the color, and draw an X. After the X is drawn, you increase the x-axis coordinate by 5. When the value of x approaches the horizontal limit of the Swing applet—that is, when it passes 400 or so—increase y and reset x to 0. To accomplish this processing, type the following code:

```
   Color variety = new Color(r, g, b);
   gr.setColor(variety);
   gr.drawString("X",x,y);
   x += 5;
   if (x >= 400)
   {
    x = 0;
    y += 10;
   }//end if
  }//end for
 }//end paint()
}//end JDemoColor class
```

8. Save the file as **JDemoColor.java** in the Chapter.10 folder on your Student Disk, and then compile at the command prompt using the **`javac`** command. Modify the **TestJDemoGraphics2.html** document for use with the JDemoColor class, and then save as **TestJDemoColor.html**. When you run the Swing applet, you should see it filled with hundreds of small Xs in many different colors.
9. Close the Applet Viewer window.

The Swing Applet's Background Color

In addition to changing the color of Strings that you display, you can change the background color of your Swing applet. For example, the statement `setBackground(Color.pink);` changes the Swing applet screen color to pink. You do not need a Graphics object to change the Swing applet's background color; it is the Swing applet itself that changes colors. (You could also write `this.setBackground(Color.pink);` because setBackgound refers to "this" Swing applet.)

Creating Graphics and Graphics 2D Objects

When you call the paint() method, you can use the automatically created Graphics object, but you also can instantiate your own Graphics or Graphics 2D objects. For example, you might want to use a Graphics object when some action occurs, such as a mouse event. Because the ActionPerformed() method does not supply you with a Graphics object automatically, you can create your own.

For example, to display a string when the user clicks a JButton, you can code an ActionPerformed() method such as the following:

```
public void actionPerformed(ActionEvent e)
{
 Object source = e.getSource();
 if (source == button1)
 {
  Graphics draw = getGraphics();
  draw.drawString("You clicked the button!",50,100);
 }
}
```

This method instantiates a Graphics object named draw. (You can use any legal Java identifier.) The getGraphics() method provides the draw object with Graphics capabilities. Then the draw object can employ any of the Graphics methods you have learned—setFont(), setColor(), and drawString().

Tip

Notice that when you create the draw object, you are not calling the Graphics constructor directly. (The name of the graphics constructor is Graphics(), not getGraphics().) You are not allowed to call the Graphics constructor because Graphics() is an abstract class. You will learn about abstract classes in Chapter 11.

Next you will create a Graphics object named pen and use the object to draw a String on the screen. The text of the String will appear to move each time a JButton is clicked.

To write a Swing applet in which you create your own Graphics object:

1. Open a new text file in your text editor, and type the following import statements for the Swing applet:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

2. Start typing the following Swing applet that uses the mouse, and defines a String, a JButton, a Font, and two integers to hold x and y coordinates:

```
public class JDemoCreateGraphicsObject extends JApplet
 implements ActionListener
{
 String companyName = new String
  ("Event Handlers Incorporated");
 JButton moveButton = new JButton("Move It");
 Font hell2Font = new Font("Helvetica", Font.ITALIC, 12);
 int x = 10,y = 50;
```

3. Type the following init() method, which changes the background color, adds the JButton, and prepares the Swing applet to listen for JButton events:

```
public void init()
{
   setBackground(Color.yellow);
   Container con = getContentPane();
   con.setLayout(new FlowLayout() );
   con.add(moveButton);
  moveButton.addActionListener(this);
  }
```

4. Within the actionPerformed() method, you can create a Graphics object and use it to draw the String on the screen. Each time a user clicks the JButton, the x- and y-coordinates both increase so a copy of the company name appears slightly below and to the right of the previous company name. Type the following code to accomplish this processing:

```
     public void actionPerformed(ActionEvent e)
     {
      Object source = e.getSource();
      if (source == moveButton)
      {
       Graphics pen = getGraphics();
       pen.setFont(hell2Font);
       pen.setColor(Color.magenta);
       pen.drawString(companyName, x+=20,y += 20);
      }//end if
    }//end actionPerformed()
  }//end JDemoCreateGraphicsObject class
```

5. Save the file as **JDemoCreateGraphicsObject.java** in the Chapter.10 folder on your Student Disk, and then compile at the command prompt using the `javac` command. Modify the**TestDemoGraphics2.html** document for use with the JDemoCreateGraphicsObject class, and save it as **TestJDemoCreateGraphicsObject.html** in the Chapter.10 folder on your Student Disk. Use the `appletviewer TestJDemoGraphicsObject.html` command to run the program. Click the **moveButton** several times to see the String message appear and move on the screen.
6. When you finish clicking the moveButton, close the Applet Viewer window.

If you run JDemoCreateGraphicsObject and click the JButton enough times, the "Event Handlers Incorporated" String appears to march off the bottom of the Swing applet. Every time you click the JButton, the x- and y-coordinates used by drawString() increase. You can prevent this error by checking the screen coordinates' values to see if they exceed the applet's dimensions.

To avoid the error of exceeding the applet viewing area:

1. Open the **JDemoCreateGraphicsObject** file, and change the class name to **JDemoCreateGraphicsObject2**.
2. Position your insertion point to the right of the statement `pen.setColor (Color.magenta);` in the actionPerformed() method, and then press **[Enter]** to start a new line.

3. Because you add 20 to the x variable each time you draw the String within the applet, you can ensure that the String appears only 12 times by preventing the x-coordinate from exceeding a value of 250. Type the following `if` statement to check the x-coordinate value:

```
if(x < 250)
{
```

4. Position your insertion point to the right of the line `pen.drawString (companyName,x+=20,y+=20);`, press **[Enter]**, type the closing curly brace for the `if` statement, and then press **[Enter]** again.
5. On the new line, type the following `else` statement that disables the JButton after the x-coordinate becomes too large:

```
else
 moveButton.setEnabled(false);
```

6. Save the file as **JDemoCreateGraphicsObject2.java** in the Chapter.10 folder on your Student Disk, and compile at the command prompt using the `javac` command. Modify the**TestJDemoCreateGraphicsObject.html** document for use with the JDemoCreateGraphicsObject2 class, and save as **TestJDemoCreateGraphicsObject2.html** in the Chapter.10 folder on your Student Disk. Now when you click the moveButton until the company name moves to x-coordinate 250, the JButton is disabled, and the company name no longer violates the applet size limits.
7. Close the Applet Viewer window.

DRAWING LINES, RECTANGLES, OVALS, ARCS, AND POLYGONS

Just as you can draw Strings using a Graphics object and the drawString() method, Java provides you with several methods for drawing a variety of lines and geometric shapes.

Tip

Any line or shape will be drawn in the current color you set with the setColor() method.

You can use the **drawLine() method** to draw a straight line between any two points on the screen. The drawLine() method takes four arguments: the x- and y-coordinates of the line's starting point, and the x- and y-coordinates of the line's ending point. For example, if you create a Graphics object named pen, then `pen.drawLine(10,10,100,200);` draws a straight line that slants down and to the right, from position 10, 10 to position 100, 200, as shown in Figure 10-5. Because you can start at either end when you draw a line, an identical line is created with `pen.drawLine(100,200,10,10);`.

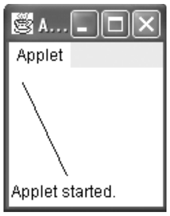


Figure 10-5 Line from position 10, 10 to 100, 200

Tip: It is almost impossible to draw a picture of any complexity without sketching it first on a piece of graph paper to help you determine correct coordinates.

You can use the **drawRect() method** and **fillRect()** method, respectively, to draw the outline of a rectangle or to draw a solid, or filled, rectangle. Each of these methods requires four arguments. The first two arguments represent the x- and y-coordinates of the upper-left corner of the rectangle. The last two arguments represent the width and height of the rectangle. For example, `drawRect(20,100,200,10);` draws a short, wide rectangle that begins at position 20, 100, and is 200 pixels wide by 10 pixels tall.

Tip: For an alternative to the drawRect() method, you can use four calls to drawLine().

The **clearRect() method** also requires four arguments and draws a rectangle. The difference between using the drawRect() and fillRect() methods and the clearRect() method is that the drawRect() and fillRect() methods use the current drawing color, whereas the clearRect() method uses the current background color to draw what appears to be an empty or "clear" rectangle. For example, the JDemoRectangles program shown in Figure 10-6 produces the Swing applet shown in Figure 10-7. The program sets the background color, draws a filled rectangle in a contrasting color, and draws a smaller, clear rectangle (using the background color) within the boundaries of the filled rectangle.

```
import javax.swing.*;
import java.awt.*;
public class JDemoRectangles extends JApplet
{
   public void paint(Graphics gr)
   {
     gr.setColor(Color.red);
     setBackground(Color.blue);
     gr.fillRect(20,20,120,120);
     gr.clearRect(40,40,50,50);
     invalidate();
     validate();
   }
}
```

Figure 10-6 JDemoRectangles program

You can create rectangles with rounded corners when you use the **drawRoundRect method**. The drawRoundRect() method requires six arguments. The first four arguments match the four arguments required to draw a rectangle: the x- and y-coordinates of the upper-left corner, and the width and height. The two additional arguments represent the arc width and height associated with the rounded corners; an **arc** is simply a portion of a circle. If you assign zeros to the arc coordinates, the rectangle will not be rounded; instead, the corners will be square. At the other extreme, if you assign values to the arc coordinates that are at least the width and height of the rectangle, the rectangle is so rounded that it is a circle. The paint() method in Figure 10-8 draws four rectangles with increasingly large corner arcs. Figure 10-9 shows the program's output.

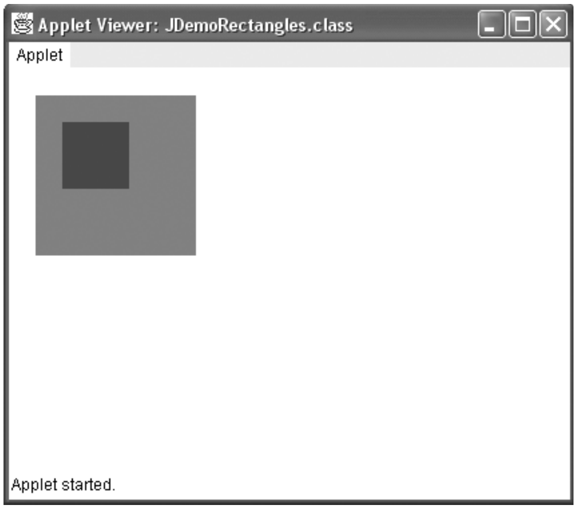


Figure 10-7 Output of the JDemoRectangles program

```
import javax.swing.*;
import java.awt.*;
public class JDemoRectangles2 extends JApplet
{
   public void paint(Graphics gr)
   {
     gr.drawRoundRect(20,20,80,80,0,0);
     gr.drawRoundRect(120,20,80,80,10,10);
     gr.drawRoundRect(220,20,80,80,40,40);
     gr.drawRoundRect(320,20,80,80,80,80);
   }
}
```

Figure 10-8 JDemoRectangles2 program that draws rounded rectangles

Like with the fillRect() method, you can use the fillRoundRect() method to create a filled, rounded rectangle.

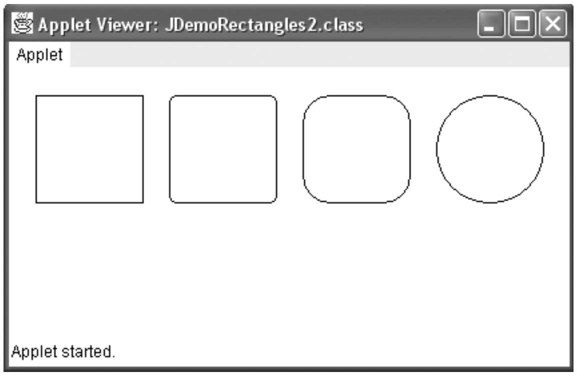


Figure 10-9 Output of the JDemoRectangles2 program that draws rounded rectangles

Drawing Ovals

It is possible to draw an oval using the drawRoundRect() or fillRoundRect() methods, but it is usually easier to use the **drawOval()** and **fillOval() methods**. The drawOval() and fillOval() methods both draw ovals using the same four arguments that rectangles use. When you supply drawOval() or fillOval() with x- and y-coordinates for the upper-left corner and width and height measurements, you can picture an imaginary rectangle that uses the four arguments. The oval is then placed within the rectangle so it touches the rectangle at the center of each of the rectangle's sides. For example, if you create a Graphics object named tool and draw a rectangle with `tool.drawRect(50,50,100,60);`, and an oval with `tool.drawOval(50,50,100,60);`, then the output will appear as shown in Figure 10-10 with the oval edges just skimming the rectangle's sides.

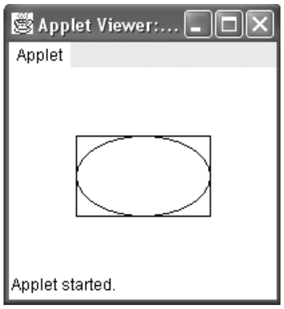


Figure 10-10 Demonstration of the drawOval() method

Tip: If you draw a rectangle with identical height and width, you draw a square. If you draw an oval with identical height and width, you draw a circle.

Next you will add a simple line drawing to the JDemoCreateGraphicsObject2 program. The drawing will appear after the user clicks the JButton enough times to increase the x-coordinate to 250, which disables the JButton.

To add a line drawing to a program:

1. Open the **JDemoCreateGraphicsObject2** program, and rename the class **JDemoCreateGraphicsObject3**.
2. Position your insertion point to the right of the opening curly brace for the actionPerformed() method, and then press **[Enter]** to start a new line.
3. Type the following to define a Graphics object and set its font and color:

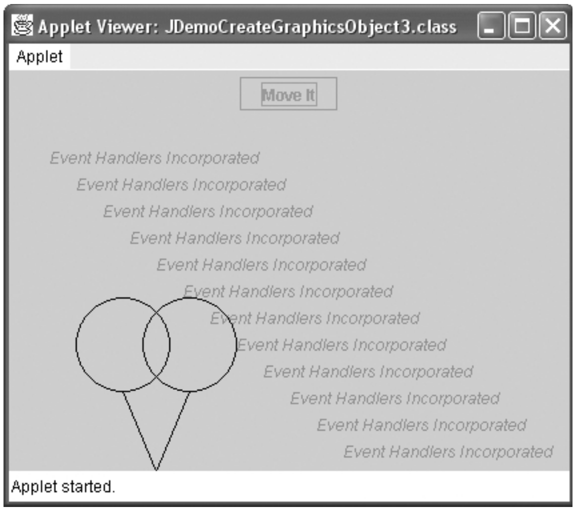
```
Graphics pen = getGraphics();
pen.setFont(hell2Font);
pen.setColor(Color.magenta);
```

4. Replace the current `if...else` structure with the following code that tests the value of x, and either draws the company name or disables the JButton and draws a logo. When you draw the logo, you set the drawing color to black and draw a simple drawing of the Event Handlers Incorporated logo, which is two overlapping balloons with strings attached:

```
if(x < 250)
{
 pen.drawString(companyName,x += 20,y += 20);
}
else
{
 moveButton.setEnabled(false);
 pen.setColor(Color.black);
 pen.drawOval(50,170,70,70);
 pen.drawLine(85,240,110,300);
```

```
    pen.drawOval(100,170,70,70);
    pen.drawLine(135,240,110,300);
   }
```

5. Save the file as **JDemoCreateGraphicsObject3.java** in the Chapter.10 folder on your Student Disk, and compile at the command prompt using the **javac** command. Modify the**TestJDemoCreateGraphicsObject2.html** document for use with the JDemoCreateGraphicsObject3 class, and save as **TestJDemoCreateGraphicsObject3.html** in the Chapter.10 folder on your Student Disk. After the company name moves to x-coordinate 250, the JButton is disabled, and the balloon drawing appears, as shown in Figure 10-11.
6. Close the Applet Viewer window.



```
Event Handlers Incorporated
Event Handlers Incorporated
Event Handlers Incorporated
Event Handlers Incorporated
Event Handlers Incorporated
Event Handlers Incorporated
Event Handlers Incorporated
Event Handlers Incorporated
Event Handlers Incorporated
Event Handlers Incorporated
Event Handlers Incorporated
Event Handlers Incorporated
Applet started.
```

Figure 10-11 Output of the JDemoCreateGraphicObjects3 program with the JButton disabled

Drawing Arcs

In Java, you can draw an arc using the Graphics **drawArc()** method. To use the drawArc() method, you provide six arguments:

- The x-coordinate of the upper-left corner of an imaginary rectangle that represents the bounds of the imaginary circle that contains the arc
- The y-coordinate of the same point
- The width of the imaginary rectangle that represents the bounds of the imaginary circle that contains the arc
- The height of the same imaginary rectangle

- The beginning arc position
- The arc angle

Arc positions and angles are measured in degrees; there are 360 degrees in a circle. The zero-degree position for any arc is at the three o'clock position, as shown in Figure 10-12. The other 359-degree positions increase as you move counterclockwise around an imaginary circle, so that 90 degrees is at the top of the circle in the 12 o'clock position, 180 degrees is opposite the starting position at nine o'clock, and 270 degrees is at the bottom of the circle in the six o'clock position.

The arc angle is the number of degrees over which you want to draw the arc, traveling counterclockwise from the starting position. For example, you could draw a half circle by indicating an arc angle of 180 degrees, or a quarter circle by indicating an arc angle of 90 degrees. If you want to travel clockwise from the starting position, you express the degrees as a negative number. Just as when you draw a line, when drawing any arc you can take one of two approaches: either start at point A and travel to point B, or start at point B and travel to point A. For example, to create an arc object halfarc that looks like the top half of a circle, the statements `halfarc.drawArc(x,y,w,h,0,180);` and `halfarc.drawArc(x,y,w,h,180,-180);` produce identical results. The first statement starts an arc at the three o'clock position and travels 180 degrees counterclockwise to the nine o'clock position. The second statement starts at nine o'clock and travels clockwise to three o'clock.

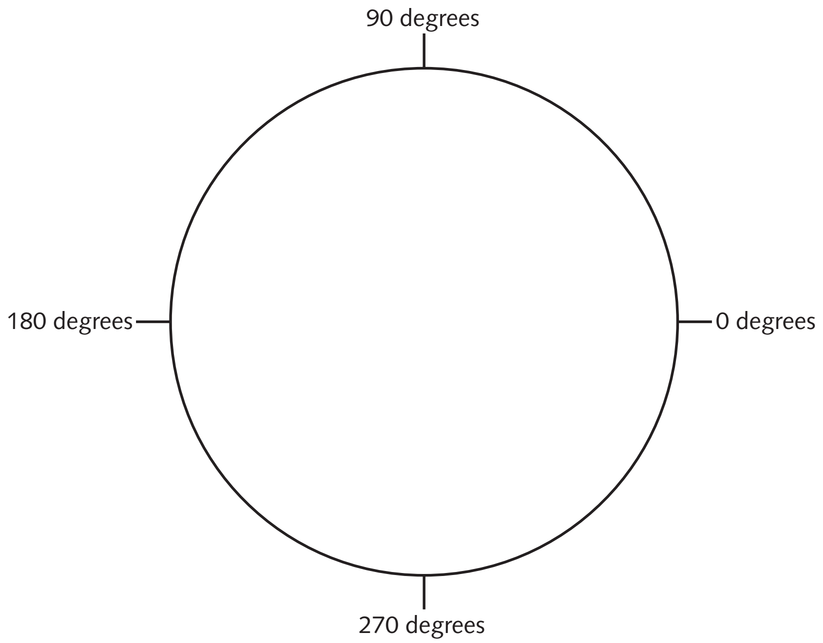


Figure 10-12 Arc positions

The **fillArc() method** creates a solid arc. The arc is drawn and two straight lines are drawn from the arc end points to the center of the imaginary circle whose perimeter the arc occupies. For example, the two statements `solidarc.fillArc(10,50,100,100,20,320);` and `solidarc.fillArc(200,50,100,100,340,40);` together produce the output shown in Figure 10-13. Each of the two arcs is in a circle of size 100 by 100. The first almost completes a full circle, starting at position 20 (near two o'clock) and ending 320 degrees around the circle (at position 340, near four o'clock). The second filled arc more closely resembles a pie slice, starting at position 340 and extending 40 degrees to end at position 40.

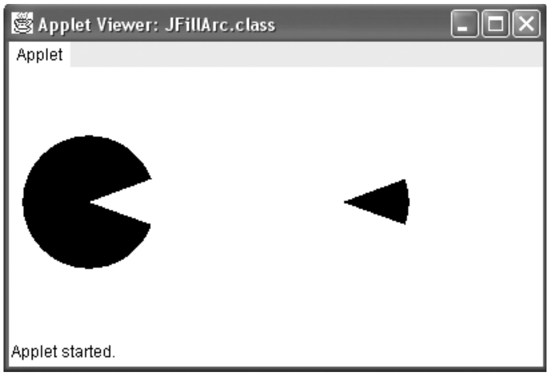


Figure 10-13 Two filled arcs

Creating Three-Dimensional Rectangles

The draw3DRect() method is a minor variation on the drawRect() method. You use the **draw3DRect() method** to draw a rectangle that appears to have "shadowing" on two of its edges—the effect is that of a rectangle that is slightly raised or slightly lowered. The draw3DRect() method requires a fifth argument in addition to the x- and y-coordinates and width and height required by the drawRect() method. The fifth argument is a Boolean value, which is `true` if you want the raised rectangle effect (darker on the right and bottom) and `false` if you want the lowered rectangle effect (darker on the left and top). There is also a **fill3DRect() method** for creating filled three-dimensional rectangles.

Tip

The three-dimensional methods work well only with lighter drawing colors.

Creating Polygons

When you want to create a shape that is more complex than a rectangle, you can use a sequence of calls to the drawLine() method, or you can use the **drawPolygon() method** to draw complex shapes. The drawPolygon() method requires three arguments: two integer arrays and a single integer.

The first integer array holds a series of x-coordinate positions, and the second array holds a series of corresponding y-coordinate positions. The third integer argument is the number of pairs of points you want to connect. If you don't want to connect all the points represented by the array values, you can assign this third argument integer a value that is smaller than the number of elements in each array. However, an error occurs if the third argument is a value higher than the available number of coordinate pairs.

For example, examine the code shown in Figure 10-14, which is a Swing applet that has one task—to draw a red, star-shaped polygon. Two parallel arrays are assigned x- and y-coordinates; the paint() method sets the drawing color to red and draws the polygon. The program's output appears in Figure 10-15.

```
import javax.swing.*;
import java.awt.*;
public class JStar extends JApplet
{
   int xPoints[] = {42, 52, 72, 52,  60, 40,  15, 28,  9, 32, 42};
   int yPoints[] = {38, 62, 68, 80, 105, 85, 102, 75, 58, 60, 38};
   public void paint(Graphics gr)
   {
      gr.setColor(Color.red);
      gr.drawPolygon(xPoints, yPoints, xPoints.length);
   }
}
```

Figure 10-14 JStar Swing applet

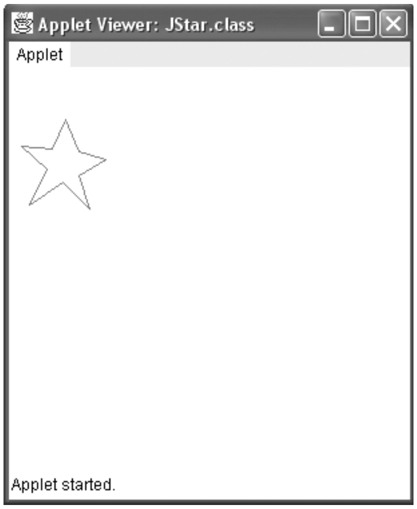


Figure 10-15 Output of the JStar Swing applet

Tip: In Chapter 8, you learned that you can use `length` for the length of an array. Rather than using a constant integer value, such as 11, it is convenient to use the length of one of the coordinate point arrays, as in `xPoints.length`.

You can use the **fillPolygon() method** to draw a solid shape. The major difference between the drawPolygon() and fillPolygon() methods is that if the beginning and ending points used with the fillPolygon() method are not identical, then the two end points will be connected by a straight line before the polygon is filled with color.

The difference is subtle, but rather than providing the fillPolygon() method with three arguments, you can create a Polygon object that defines a polygon, and then pass the constructed object as a single argument to the fillPolygon() method. Note the following statements:

```
Polygon someShape = new Polygon(xPoints, yPoints, size);
fillPolygon(someShape);
```

These statements have the same result as the following:

```
fillPolygon(xPoints, yPoints, size);
```

Additionally, you can instantiate an empty Polygon object (with no points) using the statement `Polygon someFutureShape = new Polygon();`. You use the following statements to add points to the polygon later using a series of calls to the addPoint() method:

```
someFutureShape.addPoint(100,100);
someFutureShape.addPoint(150,200);
someFutureShape.addPoint(50,250);
```

It is practical to use addPoint() instead of coding the point values when you want to write a program in which the user enters polygon point values. Whether the user does so from the keyboard or with a mouse, you can continue to add points to the polygon indefinitely.

Copying an Area

After you create a graphics image, you might want to create copies of the image. For example, you might want a company logo to appear several times in an applet. Of course, you can redraw the picture, but you can also use the **copyArea() method** to copy any rectangular area to a new location. The copyArea() method requires six parameters:

- The x- and y-coordinates of the upper-left corner of the area to be copied
- The width and height of the area to be copied
- The horizontal and vertical displacement of the destination of the copy

Next you will learn how to copy an area containing a logo that you want to appear several times on a Swing applet without re-creating the logo each time.

To copy an image:

1. Open a new text file in your text editor, and then enter the beginning statements for a Swing applet that uses the copyArea() method:

```
import javax.swing.*;
import java.awt.*;
public class JThreeStars extends JApplet
{
```

2. Add the following statements, which will create a polygon in the shape of a star:

```
int xPoints[] = {42, 52, 72, 52,
                60, 40,  15, 28,  9, 32, 42};
int yPoints[] = {38, 62, 68, 80,
                105, 85, 102, 75, 58, 60, 38};
Polygon aStar =
 new Polygon(xPoints, yPoints, xPoints.length);
```

3. Add the following paint() method, which sets a color, draws a star, and then draws two additional identical stars:

```
 public void paint(Graphics star)
 {
  star.setColor(Color.red);
  star.drawPolygon(aStar);
  star.copyArea(0,0,75,105,125,130);
  star.copyArea(0,0,75,105,180,70);
 }
}
```

4. Save the program as **JThreeStars.java** in the Chapter.10 folder on your Student Disk, and then compile the program.

5. Open a new file in your text editor, and then enter the following HTML document to test the Swing applet:

```
<HTML>
<APPLET CODE = "JThreeStars.class"
  WIDTH = 420 HEIGHT = 300>
</APPLET>
</HTML>
```

6. Save the HTML document as **TestJThreeStars.html** in the Chapter.10 folder on your Student Disk, and then run it using the `appletviewer TestJThreeStars.html` command. The output should look like Figure 10-16.

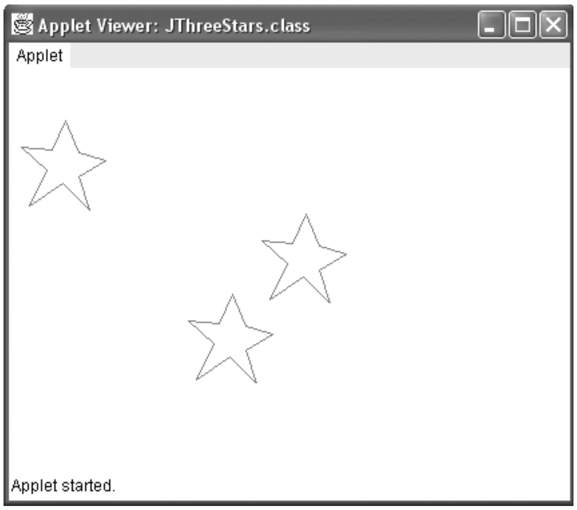


Figure 10-16 Output of the JThreeStars applet

7. Close the Applet Viewer window.
8. Modify the program to add more stars in any location you choose, save and compile the program, and then run the HTML document to confirm that the stars are copied to your desired locations.
9. Close the Applet Viewer window.

LEARNING MORE ABOUT FONTS AND THEIR METHODS

As you add more components in your Swing applet, positioning becomes increasingly important. In particular, when you draw Strings using different fonts, if you do not place the Strings correctly, they overlap and become impossible to read. Additionally, even when you define a font, such as `Font myFont = new Font("TimesRoman",Font.PLAIN,10);`, you have no guarantee that the font will be available on every computer that runs your Swing applet. If your user's computer does not have the font loaded, then Java chooses a default replacement font, so you are never completely sure of how your output will look. Fortunately, Java provides many useful methods for obtaining information about the fonts you use.

You can discover the fonts that are available on your system by using the **getAllFonts() method** that is part of the GraphicsEnvironment class defined in the java.awt package. The GraphicsEnvironment class structure shown in Figure 10-17 describes the collection of GraphicsDevice objects and Font objects available to a Java application on a particular platform. The getAllFonts() method returns an array of String objects that are the names of available fonts, as shown in the following code: `GraphicsEnvironment myFonts = GraphicsEnvironment.getLocalGraphicsEnvironment(); Font myFonts = myFonts.getAllFonts()`.

```
java.lang.Object
  |
  +--java.awt.GraphicsEnvironment
```

Figure 10-17 Structure of the GraphicsEnvironment class

Notice in the example above that you can't call the GraphicsEnvironment object directly. Instead, you must get a reference object to the current computer by calling the static getLocalGraphicsEnvironment() method. In the previous example, myFonts is the reference object. Then in the second statement, the myFonts object is used as the Font reference to the getAllFonts() method. The getAllFonts method returns an array of Font objects existing on the current system.

You can discover the resolution and screen size on your system by using the **getScreenResolution() method** and **getScreenSize() method** that is part of the Toolkit class. The structure of this helpful class is shown in Figure 10-18.

```
java.lang.Object
  |
  +--java.awt.Toolkit
```

Figure 10-18 Structure of the Toolkit class

The **getDefaultToolkit() method** provides information about the system in use. The **getScreenResolution() method** returns the number of pixels as an int type. You can create a Toolkit object and get the screen resolution with the following code:

```
Toolkit tk = Toolkit.getDefaultToolkit();
int resolution = tk.getScreenResolution()
```

The Dimension class structure is shown in Figure 10-19. A Dimension object is useful for representing the width and height of a user interface component. For example, calling the Dimension(int, int) constructor in the following example creates a Dimension object representing the width and height specified as arguments. The Dimension class has three constructors:

- Dimension() method creates an instance of Dimension with a width of zero and a height of zero.
- Dimension(Dimension d) creates an instance of Dimension whose width and height are the same as for the specified dimension.
- Dimension(int width, int height) constructs a Dimension and initializes it to the specified width and specified height.

```
java.lang.Object
  |
  +--java.awt.geom.Dimension2D
        |
        +--java.awt.Dimension
```

Figure 10-19 Structure of the Dimension class

The **getScreenSize() method** that is a member of the Toolkit object returns an object of type Dimension which specifies the width and height of the screen in pixels. Knowing the number of pixels for the width and height of your display is useful to set the coordinates for the position of the window and also set the width and height of the window.

```
Dimension screen = tk.getScreenSize();
String width = screen.width;
String height = sd.height;
```

Next you will write a Swing applet that lists the resolution and screen size along with the fonts available on your system.

To write an applet that lists the resolution, screen size, and fonts on your system:

1. Open a new file in your text editor, and then enter the first few lines of the JFontList Swing applet:

   ```
   import javax.swing.*;
   import java.awt.*;
   public class JFontList extends JApplet
   {
   ```

2. Add the following statement to create two integer variables to hold the x- and y-coordinate positions you will use to draw Strings within the applet: `int x = 10, y = 15;`.

3. Add the following paint() method header and an opening curly brace. Within the method, create a Toolkit object by calling the getDefaultToolkit() method. Call the toString() method to return the value stored in a String named resAndSize. Draw the first String named resAndSize at horizontal position 10 and vertical position 15.

   ```
   public void paint(Graphics gr)
    {
     Toolkit tk = Toolkit.getDefaultToolkit();
     String resAndSize = toString();
     gr.drawString(resAndSize,x, y += 15);
   ```

4. Enter the statement that creates the GraphicsEnvironment object named ge. Enter the following `for` loop to the paint() method. This loop will draw each String in the array that was filled using the getAvailableFontFamilyNames() method. You will draw each subsequent String 15 pixels lower within the applet. Finally, type the closing curly brace for the paint() method.

```
 GraphicsEnvironment ge =
   GraphicsEnvironment.getLocalGraphicsEnvironment();
 String[] fontnames = ge.getAvailableFontFamilyNames();
 for (int i = 0; i < fontnames.length; it=4)
   {
  gr.drawString(fontnames[i], x, y);
  gr.drawString(fontnames[i+1], x+190, y);
  gr.drawString(fontnames[i+2], x+380, y);
  gr.drawString(fontnames[i+3], x+570, y+=15);
   }
```

5. Create a toString() method that constructs and returns information about the screen resolution and size. Within the method, create a Toolkit object named tk and a Dimension object named sd. You can use the methods and fields of these objects to construct a return String containing screen information. Finally add a closing curly brace for the paint() method.

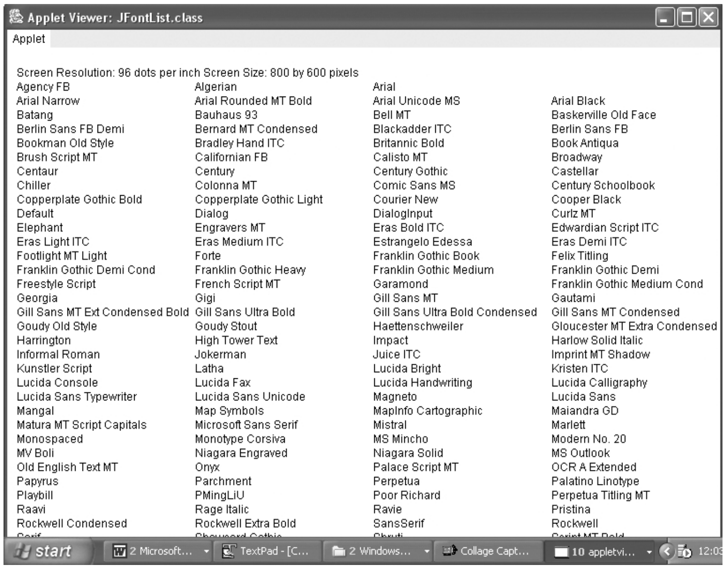
```
 public String toString()
 {
  Toolkit tk = Toolkit.getDefaultToolkit();
  Dimension sd = tk.getScreenSize();
  return "Screen Resolution: " + tk.getScreenResolution()
          + " dots per inch" +
      " Screen Size: " + sd.width + " by " + sd.height +
      " pixels";
 }
}
```

Tip: In Chapter 7, you first used the automatically included toString() method that converts objects to Strings. Now, you override that method for this class by writing your own version. You will learn more about the toString() method in Chapter 12.

6. Save the file as **JFontList.java** in the Chapter.10 folder on your Student Disk, and then compile it using the `javac` command.
7. Open a new text file, and then enter the following text to create an HTML document to host the Swing applet:

```
<HTML>
<APPLET CODE = "JFontList.class" WIDTH = 760
   HEIGHT = 600>
</APPLET>
</HTML>
```

8. Save the HTML document as **TestJFontList.html** in the Chapter.10 folder on your Student Disk, and then run the program using the `appletviewer` command. Your output should look like Figure 10-20. Notice that the Swing applet is not large enough to display all of the fonts that are installed. (Your font list might be different, depending on the fonts installed on your computer. Your list might even be so long that it cannot fully display in the applet.



```
Screen Resolution: 96 dots per inch Screen Size: 800 by 600 pixels
Agency FB                     Algerian                    Arial
Arial Narrow                  Arial Rounded MT Bold       Arial Unicode MS              Arial Black
Batang                        Bauhaus 93                  Bell MT                       Baskerville Old Face
Berlin Sans FB Demi           Bernard MT Condensed        Blackadder ITC                Berlin Sans FB
Bookman Old Style             Bradley Hand ITC            Britannic Bold                Book Antiqua
Brush Script MT               Californian FB              Calisto MT                    Broadway
Centaur                       Century                     Century Gothic                Castellar
Chiller                       Colonna MT                  Comic Sans MS                 Century Schoolbook
Copperplate Gothic Bold       Copperplate Gothic Light    Courier New                   Cooper Black
Default                       Dialog                      DialogInput                   Curlz MT
Elephant                      Engravers MT                Eras Bold ITC                 Edwardian Script ITC
Eras Light ITC                Eras Medium ITC             Estrangelo Edessa             Eras Demi ITC
Footlight MT Light            Forte                       Franklin Gothic Book          Felix Titling
Franklin Gothic Demi Cond     Franklin Gothic Heavy       Franklin Gothic Medium        Franklin Gothic Demi
Freestyle Script              French Script MT            Garamond                      Franklin Gothic Medium Cond
Georgia                       Gigi                        Gill Sans MT                  Gautami
Gill Sans MT Ext Condensed Bold Gill Sans Ultra Bold      Gill Sans Ultra Bold Condensed Gill Sans MT Condensed
Goudy Old Style               Goudy Stout                 Haettenschweiler              Gloucester MT Extra Condensed
Harrington                    High Tower Text             Impact                        Harlow Solid Italic
Informal Roman                Jokerman                    Juice ITC                     Imprint MT Shadow
Kunstler Script               Latha                       Lucida Bright                 Kristen ITC
Lucida Console                Lucida Fax                  Lucida Handwriting            Lucida Calligraphy
Lucida Sans Typewriter        Lucida Sans Unicode         Magneto                       Lucida Sans
Mangal                        Map Symbols                 MapInfo Cartographic          Maiandra GD
Matura MT Script Capitals     Microsoft Sans Serif        Mistral                       Marlett
Monospaced                    Monotype Corsiva            MS Mincho                     Modern No. 20
MV Boli                       Niagara Engraved            Niagara Solid                 MS Outlook
Old English Text MT           Onyx                        Palace Script MT              OCR A Extended
Papyrus                       Parchment                   Perpetua                      Palatino Linotype
Playbill                      PMingLiU                    Poor Richard                  Perpetua Titling MT
Raavi                         Rage Italic                 Ravie                         Pristina
Rockwell Condensed            Rockwell Extra Bold         SansSerif                     Rockwell
```

Figure 10-20 Output of the JFontList program

9. Close the Applet Viewer window.

Typesetters and desktop publishers measure the height of every font in three parts: leading, ascent, and descent. **Leading** is the amount of space between baselines. **Ascent** is the height of an uppercase character from a baseline to the top of the character. **Descent** measures the size of characters that "hang below" the baseline, such as the tails on the lowercase letters g and j. The **height** of a font is the sum of the leading, ascent, and descent. Figure 10-21 shows each of these measurements.

Tip Leading is pronounced "ledding."

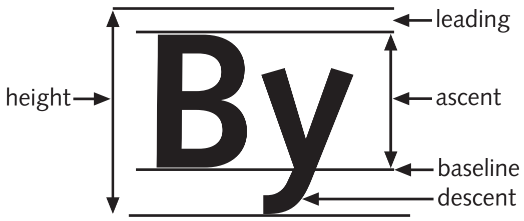


Figure 10-21 Parts of a font's height

You can discover a font's height by using the **getFontMetrics() method**. The getFontMetrics() method is part of the Graphics class and returns a FontMetrics object. The FontMetrics class contains the following methods that return a font's statistics:

- `public int getLeading()`
- `public int getAscent()`
- `public int getDescent()`
- `public int getHeight()`

Each of these methods returns an integer value representing the font size in points (one point measures 1/72 of an inch) of the requested portion of the Font object. For example, if you define a Font object named myFont, and a Graphics object named paintBrush, then you can set the current font for the Graphics object with the statement `paintBrush.setFont(myFont);`. When you code `int heightOfFont = paintBrush.getFontMetrics().getHeight();`, then the heightOfFont variable holds the total height of myFont characters.

Tip

Notice the object-dot-method-dot-method construction of the getHeight() statement. Alternately, if it is clearer to you, you can write two statements. The first statement declares a FontMetrics object: `FontMetrics fmObject = paintBrush.getFontMetrics();`. The second statement assigns a value to heightOfFont: `int heightOfFont = fmObject.getHeight();`.

Tip

When you define a Font object, you use point size. However, when you use the getFontMetrics() methods, the sizes are returned in pixels.

Next you will write a Swing applet to demonstrate the FontMetrics() methods. You will create three Font objects and display their metrics.

To demonstrate the FontMetrics methods:

1. Open a new text file in your text editor, and then enter the first few lines of the JDemoFontMetrics program:

```
import javax.swing.*;
import java.awt.*;
public class JDemoFontMetrics extends JApplet
{
```

2. Type the following code to create a String and a few fonts to use for demonstration purposes:

```
String companyName =
 new String("Event Handlers Incorporated");
Font
 courierItalic = new Font("Courier", Font.ITALIC, 16),
 timesPlain = new Font("TimesRoman", Font.PLAIN, 16),
 helvetBold = new Font("Helvetica", Font.BOLD, 16);
```

Tip

If your JFontList program showed that you do not have one of these fonts, then substitute another font that you do have.

3. Add the following code to define four integer variables to hold the four font measurements, and two integer variables to hold the current horizontal and vertical output positions within the Swing applet:

```
int ascent, descent, height, leading;
int x = 10, y = 15;
```

4. Within the Swing applet, you will draw Strings for output that you will position 15 pixels apart vertically on the screen. Type the following statement to create a constant to hold this vertical increase value:

```
static final int INCREASE = 15;
```

5. Add the following statements to start writing a paint() method. Within the method, you set the Font to courierItalic, draw the companyName String to show a working example of the font, and then call a displayMetrics() method that you will write in Step 6. You will pass the Graphics object to the displayMetrics() method, so the displayMetrics() method can discover the sizes associated with the current font. Perform the same three steps using the timesPlain and helvetBold fonts.

```
public void paint(Graphics pen)
{
 pen.setFont(courierItalic);
 pen.drawString(companyName, x, y);
 displayMetrics(pen);
 pen.setFont(timesPlain);
 pen.drawString(companyName, x, y += 40);
 displayMetrics(pen);
 pen.setFont(helvetBold);
 pen.drawString(companyName, x, y += 40);
 displayMetrics(pen);
}
```

6. Next add the header and opening curly brace for the displayMetrics() method. The method will receive a Graphics object from the paint() method. Add the following statements to call the four getFontMetrics() methods to obtain values for the leading, ascent, descent, and height variables:

```
public void displayMetrics(Graphics metrics)
{
 leading = metrics.getFontMetrics().getLeading();
 ascent = metrics.getFontMetrics().getAscent();
 descent = metrics.getFontMetrics().getDescent();
 height = metrics.getFontMetrics().getHeight();
```

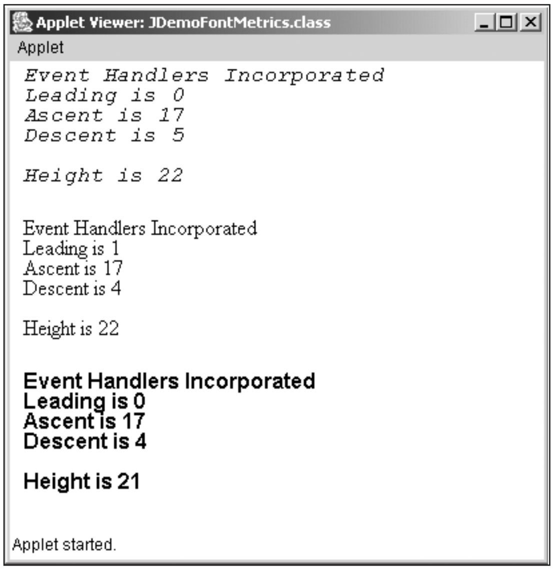
7. Add the following five drawString() statements to display the values. Use the expression `y += INCREASE` to change the vertical position of each String by the INCREASE constant.

```
 metrics.drawString("Leading is " + leading,
    x, y +=  INCREASE);
 metrics.drawString("Ascent is " + ascent,
    x, y +=  INCREASE);
 metrics.drawString("Descent is " + descent,
    x, y +=  INCREASE);
 metrics.drawString("     ", x, y += INCREASE);
 metrics.drawString("Height is " + height,
    x, y +=  INCREASE);
 }
}
```

8. Save the program as **JDemoFontMetrics.java** in the Chapter.10 folder on your Student Disk, and then compile it using the **`javac`** command.
9. Open a new text file in your text editor, and then enter the following HTML document to host the JDemoFontMetrics Swing applet:

```
<HTML>
<APPLET CODE =
 "JDemoFontMetrics.class" WIDTH = 400 HEIGHT = 350>
</APPLET>
</HTML>
```

10. Save the HTML document as **TestJFontMetrics.html** in the Chapter.10 folder on your Student Disk. At the command prompt, type **`appletviewer TestJFontMetrics.html`**. Your output should look like Figure 10-22. Notice that even though each Font object was constructed with a size of 16, the individual statistics vary for each Font object.



```
Event Handlers Incorporated
Leading is 0
Ascent is 17
Descent is 5

Height is 22


Event Handlers Incorporated
Leading is 1
Ascent is 17
Descent is 4

Height is 22


Event Handlers Incorporated
Leading is 0
Ascent is 17
Descent is 4

Height is 21
```

Figure 10-22 Output of the JDemoFontMetrics Swing applet

11. Close the Applet Viewer window.

A practical use for discovering the height of your font is to space Strings correctly as you print them. For example, instead of placing every String in a series vertically equidistant from the previous String with a statement, such as `pen.drawString("Some string", x, y += INCREASE);` (where INCREASE is always the same), you can make the actual increase in the vertical position dependent on the font. If you code `pen.drawString("Some string", x, y += pen.getFontMetrics().getHeight());`, then you are assured that each String has enough room, and will appear regardless of the font currently in use by the Graphics pen object.

When you create a String, you know how many characters are in the String. However, you cannot be sure which font Java will use or substitute, and because fonts have different measurements, it is difficult to know the exact width of the String in a Swing applet. Fortunately, the FontMetrics class contains a **stringWidth() method** that returns the integer width of a String. As an argument, the stringWidth() method requires the name of a String. For example, if you create a String named myString, then you can retrieve the width of myString with `int width = gr.getFontMetrics().stringWidth(myString);`.

Next you will use the FontMetrics methods to draw a rectangle around a String. Instead of guessing at appropriate pixel positions, you can use the height and width of the String to create a box with borders placed symmetrically around the String.

To draw a rectangle around a String:

1. Open a new file in your text editor, and then enter the first few lines of a JBoxAround Swing applet:

```
import javax.swing.*;
import java.awt.*;
public class JBoxAround extends JApplet
{
```

2. Enter the following statements to add a String, a Font, and variables to hold the font metrics and x- and y-coordinates:

```
String companyName =
 new String("Event Handlers Incorporated");
Font serifItalic = new Font("Serif", Font.ITALIC, 20);
int leading, ascent, height, width;
int x = 40, y = 60;
```

3. Create the following constant variable that holds a number of pixels indicating the dimensions of the rectangle that you will draw around the String:

```
static final int BORDER = 5;
```

4. Add the following paint() method, which sets the font, draws the String, and obtains the font metrics:

```
public void paint(Graphics gr)
{
    gr.setFont(serifItalic);
    gr.drawString(companyName,x,y);
    leading = gr.getFontMetrics().getLeading();
    ascent = gr.getFontMetrics().getAscent();
    height = gr.getFontMetrics().getHeight();
    width = gr.getFontMetrics().stringWidth(companyName);
```

5. Draw a rectangle around the String using the following drawRect() method. In Figure 10-23, the x- and y-coordinates of the upper-left edge are set at 40-border, 60-(ascent + leading + border). The proper width and height are then determined to draw a uniform rectangle around the string.

The values of the x- and y-coordinates used in the drawString() method indicate the left side of the baseline of the first character in the String. You want to position the upper-left corner of the rectangle five pixels to the left of the String, so the first argument to drawRect() is five less than x, or `x - BORDER`. The second argument to drawRect() is the y-coordinate of the String minus the ascent of the String, minus the leading of the String, minus five, or `y - (ascent + leading + BORDER)`. The last two arguments to drawRectangle() are the width and the height of the rectangle. The width is the String's width plus five pixels on the left and five pixels on the right. The height of the rectangle is the String's height, plus five pixels above the String and five pixels below the String.

```
    gr.drawRect(x - BORDER, y - (ascent + leading + BORDER),
    width + 2 * BORDER, height + 2 * BORDER);
    repaint ();
    }
  }
```

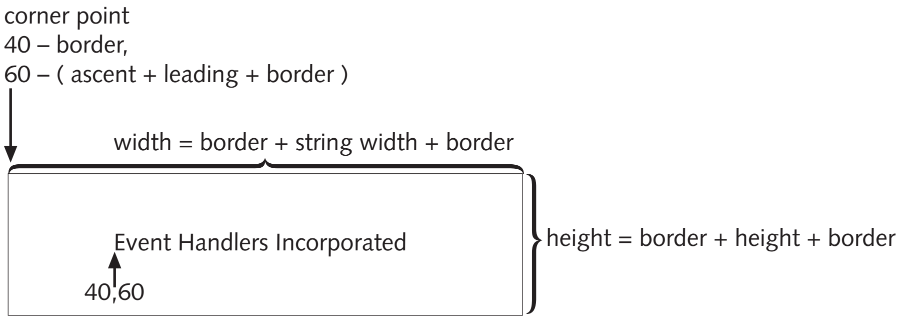

corner point
40 – border,
60 – (ascent + leading + border)
width = border + string width + border
Event Handlers Incorporated
40,60
height = border + height + border

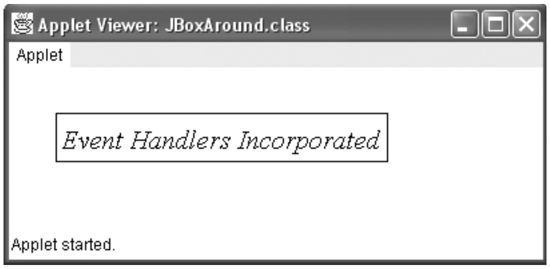
Figure 10-23 Rectangle surrounding a String

6. Save the file as **JBoxAround** in the Chapter.10 folder on your Student Disk, and then compile it using the **javac** command.

7. Open a new text file in your text editor, and then enter the following HTML document to host the applet:

   ```
   <HTML>
   <APPLET CODE = "JBoxAround.class" WIDTH = 400 HEIGHT =
   120>
   </APPLET>
   </HTML>
   ```

8. Save the HTML document as **TestJBoxAround.html** in the Chapter.10 folder on your Student Disk, and then run the program using the **`appletviewer TestJBoxAround.html`** command. Your output should look like Figure 10-24.
9. Close the Applet Viewer window, and then experiment with changing the contents of the String and the size of the BORDER constant. Confirm that the rectangle is drawn symmetrically around any String object. When you finish, close the Applet Viewer window.



Event Handlers Incorporated

Applet started.

Figure 10-24 Output of the JBoxAround program

Drawing with Java 2D Graphics

Drawing operations earlier in this chapter are called using an object. In addition, you can call drawing operations on a Graphics2D object. The structure of the Graphics2D class is shown in Figure 10-25.

```
java.lang.Object
  |
  +--java.awt.Graphics
         |
         +--java.awt.Graphics2D
```

Figure 10-25 Structure of the Graphics2D class

The advantage of using Java 2D is the enhanced classes offered to create higher-quality two-dimensional (2D) graphics, images, and text for use in your programs. They don't replace the existing java.awt classes, though—you can still use the other classes and programs that use them.

One of the advantages of Java 2D is a set of high-quality classes to offer enhanced 2D graphics, images, and text. Some of these classes include features such as:

- Fill patterns such as gradients
- Strokes that define the width and style of a drawing stroke
- Anti-aliasing, a graphics technique for producing smoother on-screen graphics

Graphics2D is found in the java.awt package. A Graphics2D object is produced by casting a Graphics object and is commonly referred to as a graphics context. For example, the void paint()method of the previous JBoxAround applet could be cast to create a Graphics2D object as follows:

```
public void paint(Graphics pen)
{
 Graphics2D newpen = (Graphics2D)pen;
```

The JBoxAround2 program created by casting is shown in Figure 10-26. It produces identical output to the JBoxAround program, as shown earlier in Figure 10-24.

```
import javax.swing.*;
import java.awt.*;
public class JBoxAround2 extends JApplet
{
   String companyName =
     new String("Event Handlers Incorporated");
   Font serifItalic = new Font("Serif". Font.ITALIC. 20);
   int leading, ascent, height, width;
   int x = 40, y = 60;

   static final int BORDER = 5

   public void paint(Graphics pen)
   {
     Graphics2D two = (Graphics2D)pen;
     two.setFont(serifItalic);
     two.drawString(companyName,x,y);
     leading = two.getFontMetrics().getLeading();
     ascent = two.getFontMetrics().getAscent();
     height = two.getFontMetrics().getHeight();
     width = two.getFontMetrics().stringWidth(companyName);
     two.drawRect(x - BORDER, y - (ascent + leading + BORDER),
       width + 2 = BORDER, height +2 = BORDER);
     repaint();
   }
}
```

Figure 10-26 JBoxAround2 program

One concept introduced with Java 2D distinguishes between an output device's coordinate space and the coordinate space you refer to when drawing an object. **Coordinate space** is any 2D area that can be described using x- and y-coordinates. For all drawing operations so far in this chapter, the only coordinate space used is for the **device coordinate space**. Recall that you have specified the x- and y-coordinates for an output area such as a container on a Swing applet, and those coordinates have been used to draw lines, text, and other objects. Java 2D adds a **user coordinate space** that you refer to when creating and drawing a 2D drawing object. The upper-left corner 0,0 of the drawing area represents both the device space and the user space. Whereas the device space coordinate is constant, the user space coordinate (0,0) can move as a result of a 2D drawing such as a 2D rotation operation. You will learn more about the two coordinate systems as you work with the 2D examples in this chapter.

You can think of the process of drawing with Java 2D objects as involving:

- Specifying the rendering attributes
- Setting a drawing stroke
- Creating objects to draw

Specifying the Rendering Attributes

The first step in drawing a 2D object is to specify how a drawn object will be rendered. Whereas drawings that are not 2D can only specify the attribute Color, 2D can designate other attributes such as line width and fill patterns. You specify 2D colors by using the setColor() method, which works like the Graphics method of the same name. Using a Graphics2D object, the color can be set to black using the code:

```
gr2D.setColor(Color.black);
```

Fill patterns control how a drawing object will be filled in. In addition to using a solid color, 2D fill patterns can be a gradient fill, texture, or even a pattern that you devise. A fill pattern is created by using the setPaint() method of Graphics2D with a bPaint object as the only argument. Classes that can be a fill pattern include GradientPaint, TexturePaint, and Color.

A **gradient fill** is a gradual shift from one color at one coordinate point to a different color at a second coordinate point. If the color shift occurs once between the points, it is called an **acyclic gradient**. If the shift occurs repeatedly, it is called a **cyclic gradient**. Figure 10-28 shows the top rectangle with an acyclic shift, and the bottom rectangle with a cyclic shift between white and darkGray colors.

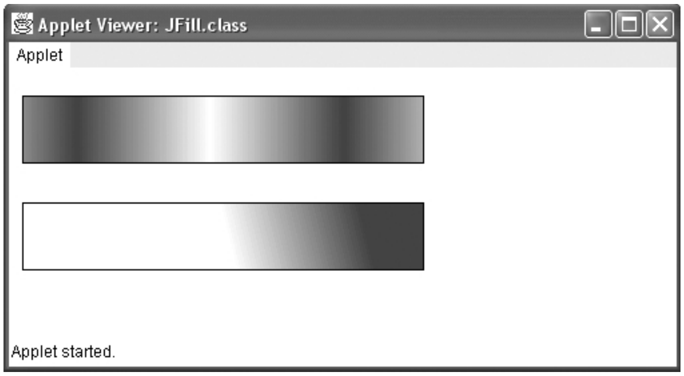


Figure 10-27 One rectangle has an acyclic gradient; the other has a cyclic gradient

Setting a Drawing Stroke

All lines in non-2D graphics operations are drawn solid, with square ends and a line width of 1 pixel. With the new 2D methods, the drawing line width can be changed using the setStroke() method. The Stroke is actually an interface; the class that defines line types and implements the Stroke interface is named **BasicStroke**. A BasicStroke constructor takes three arguments:

- A `float` value representing the line width, with 1.0 as the norm
- An `int` value determining the type of cap decoration at the end of a line
- An `int` value determining the style of juncture between two line segments

BasicStroke class variables determine the endcap and juncture style arguments. **Endcap** styles apply to the end of lines that do not join with other lines, and include CAP_BUT, CAP_ROUND, and CAP_SQUARE. Juncture styles, for lines that join, include JOIN_MITER, JOIN_ROUND, and JOIN_BEVEL.

Tip

Technically, the term stroke has been defined as a single movement using or as if using, a tool or implement such as a pen or pencil.

The following statements create a BasicStroke object and make it the current stroke:

```
BasicStroke aLine = new BasicStroke(1.0f,
 BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND);
gr2D.setStroke(aLine);
```

The `f` indicates that the 1.0 argument is a floating-point type.

Creating Objects to Draw

After you have created a Graphics2D object and specified the rendering attributes, the final steps are to create the different draw objects and then draw them. Objects that are drawn in Java 2D are first created by defining them as geometric shapes using the java.awt.geom package classes. You can define the shape of lines, rectangles, ovals, and arcs; after you define the shape, you use it as an argument to the draw() or fill() methods. The Graphics2D class does not have different methods for each shape you can draw.

Lines

Lines are created using the Line2D.Float class that takes four arguments. The arguments are the x- and y-coordinates of the two endpoints of the line. For example, to create a line from the endpoint (60,5) to the endpoint (13,28), the arguments are:

```
Line2D.Float line = new Line2D.Float(60F, 5F, 13F, 28F);
```

Tip

Note that F or f is used with the literal arguments so that the Java compiler will not mistake them for integers.

It is possible to create lines based on x and y points defined in the Point2D class. The **Point.2D.Float class** defines a point from a pair of x- and y-coordinates of type `float`. In the following example, you replace the x- and y-coordinates with Point2D objects:

```
Point2D.Float pos1 = new Point2D.Float(60,5);
Point2D.Float pos2 = new Point2D.Float(13,28);
```

The code to create a line then becomes `Line2D.Float line = new Line2D.Float (pos1, pos2);`.

Next you will create a line with a drawing stroke to illustrate how a drawing stroke can be created with different end types and juncture types where lines intersect.

To create a line with a drawing stroke:

1. Open a new file in your text editor, and then enter the first few lines of a J2DLine Swing applet. (Note that you are importing the java.awt.geom package.)

```
import javax.swing.*;
import java.awt.*;
import java.awt.geom.*;
public class J2DLine extends JApplet
{
```

2. Enter the following statements to create a paint() method, create a Graphics environment gr, and cast the Graphics environment to a Graphics2D environment gr2D. Create x and y points with the Point2D.Float class.

```
public void paint(Graphics gr)
{
 Graphics2D gr2D = (Graphics2D)gr;
 Point2D.Float pos1 = new Point.2D.Float(50,10);
 Point2D.Float pos2 = new Point2D.Float(13,28);
```

3. Create a BasicStroke object, and then create a drawing stroke named aLine. Note that the line width is set to 5 pixels and the endcap style and juncture style are set to CAP_ROUND.

```
BasicStroke aLine = new BasicStroke(5.0f,
BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND);
```

4. Add the following code to create a line between the points posx and posy, and draw the line:

```
  gr2D.setStroke(aline);
  Line2D.Float line = new Line2D.Float(pos1, pos2);
  gr2D.draw(line);
  repaint();
  }
}
```

5. Save the file as **J2DLine.java** in the Chapter.10 folder on your Student Disk, and then compile it using the **`javac`** command.

6. Open a new file in your text editor, and then enter the following HTML document to host the applet:

```
<HTML>
<APPLET CODE = "J2DLine.class" WIDTH = 50 HEIGHT = 50>
</APPLET>
</HTML>
```

7. Save the HTML document as **TestJ2DLine.html** in the Chapter.10 folder on your Student Disk, and then run the program using the **`appletviewer TestJ2DLine.html`** command. Your output should look like Figure 10-28.

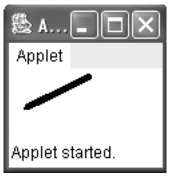


Figure 10-28 Output of the J2DLine program

Rectangles

Rectangles can be created by using a Rectangle2D.Float or a Rectangle2D.Double class. The two classes are distinguished by the type of arguments used in the constructor—float or double. Rectangle2D.Float and Rectangle2D.Double both require four arguments representing the x-coordinate, y-coordinate, width, and height. The code to create a Rectangle2D.Float object named rect at (10,10) with a width of 50 and height of 40 is `Rectangle2D. Float rect = new Rectangle2D.Float(10F, 10F, 50F, 40F);`.

Ovals

Oval objects can be created with the Ellipse2D.Float class. The Ellipse2D.Float constructor requires four arguments representing the x-coordinate, y-coordinate, width, and height. The code to create an Ellipse2D.Float object named ell at (10,73) with a width of 40 and height of 20 is `Ellipse2D.Float ell = new Ellipse2D.Float(10F,73F,40F,20F);`.

Arcs

Arcs can be created with the Arc2D.Float class. The Arc2D.Float constructor takes seven arguments. The first four are arguments representing the x-coordinate, y-coordinate, width, and height that apply to the ellipse of which the arc is a part. The remaining three arguments are:

- The starting degree of the arc
- The number of degrees it travels
- An integer indicating how it is closed

The number of degrees traveled by the arc is specified in a counterclockwise direction using positive numbers. The last argument uses one of the three class variables:

- Arc2D.PIE connects the arc to the center of an ellipse and looks like a pie slice.
- Arc2D.CHORD connects the arc's endpoints with a straight line.
- Arc2D.OPEN is an unclosed arc.

To create an Arc2D.Float object named ac at (10,133) with a width of 30 and height of 33, a starting degree of 30, 120 degrees traveled, and using the class variable Arc.2D.PIE, you use the following statment: `Arc2D.Float ac = new Arc2D.Float(10,133,30,33, 30,120,Arc2D.PIE);`.

Polygons

A Polygon is created by defining the movements from one point to another. The movement that creates a polygon is defined as a GeneralPath object found in the java.awt.geom package.

- The statement `GeneralPath pol = new GeneralPath();` creates a GeneralPath object named pol.
- The moveTo() method of GeneralPath is used to create the beginning point on the polygon. Thus, the statement `pol.moveTo(10F,193F);` starts the polygon named pol at the coordinates (10,193).

- The lineTo() method is used to create a line that ends at a new point. The statement `pol.lineTo(25F,183F);` creates a second point using the arguments of 25 and 183 as the x- and y-coordinates of the new point.
- The statement `pol.lineTo(100F,223F);` creates a third point. The lineTo() method can be used to connect the current point to the original point or, alternately, you can use the closePath() method without any arguments. Here we use the statement `pol.closePath()` to close the polygon.

Next you will you use the Java 2D drawing object types to create a Swing applet that illustrates sample rectangles, ovals, arcs, and polygons.

To create the JShapes2D Swing applet:

1. Open a new file in your text editor, and then enter the first few lines of a JShapes2D Swing applet:

```
import javax.swing.*;
import java.awt.*;
import java.awt.geom.*;
public class JShapes2D extends JApplet
{
```

2. Enter the following statements to create a paint() method, create a Graphics environment gr, and cast the Graphics environment to a Graphics2D environment gr2D:

```
public void paint(Graphics gr)
{
 Graphics2D gr2D = (Graphics2D)gr;
```

3. Create two Rectangle2D.Float objects named rect and rect2. Draw the rect object, and then fill the rect2 object.

```
Rectangle2D.Float rect = new Rectangle2D.Float(10F, 10F,
  40F, 20F);
Rectangle2D.Float rect2 = new Rectangle2D.Float(10F, 40F,
  40F, 20F);
gr2D.draw(rect);
gr2D.fill(rect2);
```

4. Create two Ellipse2D.Float objects named ell and ell2. Draw the ell object and fill the ell2 object.

```
Ellipse2D.Float ell = new Ellipse2D.Float(10F,73F,40F,
  20F);
Ellipse2D.Float ell2 = new Ellipse2D.Float(10F,103F,40F,
  20F);
gr2D.draw(ell);
gr2D.fill(ell2);
```

5. Create two Arc2D.Float objects named ac and ac2. Draw the ac object and fill the ac2 object.

```
Arc2D.Float ac = new Arc2D.Float(10,133,30,33,30,120,
  Arc2D.PIE);
Arc2D.Float ac2 = new Arc2D.Float(10,163,30,33,30,120,
  Arc2D.PIE);
gr2D.draw(ac);
gr2D.fill(ac2);
```

6. Create a new GeneralPath object named pol. Set the starting point of the polygon and create two additional points. Use the closePath() method to close the polygon by connecting the current point to the starting point. Draw the pol object.

```
  GeneralPath pol = new GeneralPath();
  pol.moveTo(10F,193F);
  pol.lineTo(25F,183F);
  pol.lineTo(100F,223F);
  pol.closePath();
  gr2D.draw(pol);
 }
}
```

7. Save the file as**JShapes2D.java** in the Chapter.10 folder on your Student Disk, and then compile it using the `javac` command.

8. Open a new file in your text editor, and then enter the following HTML document to host the applet:

```
<HTML>
<APPLET CODE = "JShapes2D.class" WIDTH = 250
   HEIGHT = 250>
</APPLET>
</HTML>
```

9. Save the HTML document as **TestJShapes2D.html** in the Chapter.10 folder on your Student Disk, and then run the program using the `appletviewer TestJShapes2D.html` command. Your output should look like Figure 10-29.

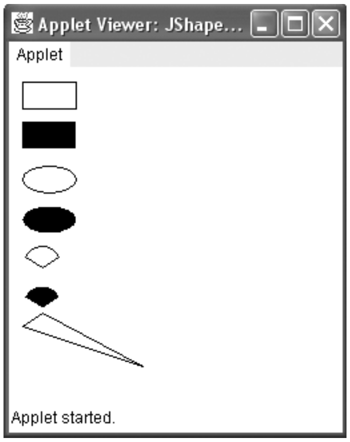


Figure 10-29 Output of the JShapes2D program

Adding Sound, Images, and Simple Animation to Swing Applets

Java 2 still supports sound using applet methods that have been available since the introduction of Java. This allows you to make Java programs audible using the methods of the Applet class to retrieve and play sound files in programs using various sound formats. Sound formats include the Windows Wave file format(.wav), Sun Audio file format(.au), and Music and Instrument Digital Interface file format(.midi).

The simplest way to retrieve and play a sound is to use the play() method of the Applet class. The **play() method** retrieves and plays the sound as soon as possible after it is called. The play() method takes one of two forms:

- play() with one argument—the argument is a Uniform Resource Locater (URL) object that loads and plays an audio clip when both the URL object and the audio clip are stored at the same URL.
- play() with two arguments—loads and plays the audio file (the first argument is a URL object and the second argument is a folder path name). The first argument will often be a call to a getCodeBase() method or getDocumentBase() method to retrieve the URL object; the second argument is the name of the audio clip within the folder path that is stored at that URL.

Tip

The `<APPLET>` tag was introduced in Chapter 9 to run an Applet from within an HTML document using the attributes CODE, HEIGHT, and WIDTH. The getCodeBase() and getDocumentBase()methods are Applet methods. By using these methods when loading sound or images, you make it possible for the applet to work even if you move it to another Web server.

Used with the CODEBASE attribute, which indicates the filename of the applet's main class file, the above methods direct the browser to look in a different folder for the applet and other files it uses. This is necessary when the desired files are in a different location than the Web page containing the applet. By calling getCodeBase() in an applet, you get a URL object that represents the folder where the applet's class file is stored. For example, the following statement retrieves and plays the event.au sound file which is stored at the same place as the applet: `play(getCodeBase(),"event.au");`.

Tip

The getDocumentBase() method returns an absolute URL naming the directory of the document in which the applet is stored. It is sometimes used instead of getCodeBase() as a matter of preference.

To play a sound more than once, or start or stop the sound, you must load the sound into an AudioClip object using the applet's newAudioClip() method. AudioClip is part of the java.awt.Applet class and must be imported into your program. The getAudioClip() method can take one or two arguments similar to the play() method. The first argument (or only argument, if there is only one) is a URL argument that identifies the sound file; the second argument is a folder path reference needed for locating the file.

The following statement loads the sound file from the previous example into the clip object: `AudioClip aClip = newAudioClip(getCodeBase(), "audio/event.au");`. Here the sound file reference indicates that the event.au sound file is located in the audio folder. After you have created an AudioClip object, you can use the play() method to call and play the sound, the stop() method to halt the playback, and the loop() method to play the sound repeatedly.

Next you will use the play() method and AudioClip to play a sound in a Swing applet. You will also create and add a Graphics2D object.

To play a sound and add a Graphics2D object in a Swing applet for Event Handlers, Inc.:

1. Open a new file in your text editor, and then enter the first few lines of the JEventSound Swing applet:

   ```
   import java.awt.*;
   import java.applet.*;
   import javax.swing.*;
   public class JEventSound extends JApplet
   {
   ```

2. Enter the following statement to declare an AudioClip object named sound:

   ```
   AudioClip sound;
   ```

3. Create the init() method and an AudioClip object to play the event.au sound file with the code:

```
public void init()
 {
  sound = getAudioClip(getCodeBase(),"event.au");
 }
```

4. Create the following start() method. The start method uses the loop() method to play the event.au sound file continually:

```
public void start()
 {
  sound.loop();
 }
```

5. Create the following stop() method to halt the event.au sound file.

```
public void stop()
{
 sound.stop();
}
```

6. Create a Graphics object using paint (Graphics g), and then use a cast to change the graphics context to a Graphics2D object. Use the drawstring() method to create a message which appears on the screen while the Swing applet play. Add a closing curly brace for the class.

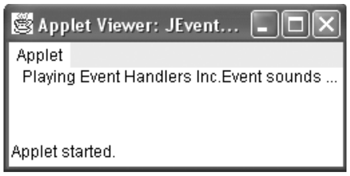
```
public void paint(Graphics g)
  {
  Graphics2D g2D = (Graphics2D)g;
  g2D.drawString
     ("Playing Event Handlers Inc. Event
  sounds ...", 10, 10);
 }
}
```

7. Save the file as **JEventSound.java** in the Chapter.10 folder on your Student Disk, and then compile it using the **`javac`** command.

8. Open a new file in your text editor, and then enter the following HTML document to test the Swing applet:

```
<HTML>
<APPLET CODE = "JEventSound.class" WIDTH = 400
   HEIGHT = 250 >
</APPLET>
</HTML>
```

9. Save the HTML document as **TestJEventSound.html** in the Chapter.10 folder on your Student Disk, and then run it using the **`appletviewer TestJEventSound.html`** command. The output should look like Figure 10-30. You should also be able to hear sound playing continually if speakers are installed for your system.



Playing Event Handlers Inc.Event sounds ...

Applet started.

Figure 10-30 Output of the JEventSound Swing applet

Adding Images

An **image** is a likeness of a person or thing. Images abound on the Internet in all shapes, colors and sizes. Images formats supported by Java include:

- Graphics Interchange Format (GIF), which can contain a maximum of 256 different colors
- Join Photographic Experts Group (JPEG) which is commonly used to store photographs, and is a more sophisticated way to represent a color image
- Portable Network Graphics (PNG), which is more flexible than the GIF, and stores its images in a lossless form (It was originally designed to be a portable image storage form for computer-originated images.)

Java's image capabilities include using the Image class and ImageIcon class to load images in one of the formats discussed earlier. The Image class is an abstract class. An **abstract** class is one from which you cannot create any objects, but from which you can inherit, so you must create Image objects indirectly. The Image class is found in the java.awt package, as shown in Figure 10-31. The ImageIcon class is particularly useful because it can be used to easily load an image into either an applet or an application. The ImageIcon class is part of the Swing package, as shown in Figure 10-32.

```
java.lang.Object
  |
  +--java.awt.Image
```

Figure 10-31 Structure of Image class

```
java.lang.Object
  |
  +--javax.swing.ImageIcon
```

Figure 10-32 Structure of ImageIcon class

To declare an Image with the name eventLogo1, you use the declaration `Image eventLogo1;`. The getImage() method is used to load an Image into the applet. Like the AudioClip method used for loading sound, one version of the getImage() method can take up to two arguments—a location where the image is stored and the filename of the image. You create and load the Image named eventImage with the statement:

```
eventLogo1 = getImage(getCodeBase(),"event.gif");
```

Notice that the Image object is not created directly. Instead, you request that an Image be loaded and returned to you.

Because the ImageIcon class is not an abstract class, you can create the ImageIcon eventLogo2 with the following statement:

```
eventLogo2 = new ImageIcon("event.gif");
```

This creates an ImageIcon object that holds the same event.gif as the eventLogo1 Image.

Tip

The ImageIcon class provides several constructors that allow an ImageIcon object to be initialized with an image from a local computer or an image stored on a Web server.

The applet's paint() method is used to display both Image and ImageIcon object images. The drawImage() method is a Graphics method which uses the following four arguments:

- The first argument is a reference to the Image object in which the image is stored.
- The second argument is the x-coordinate where the image will appear on the applet.
- The third argument is the y-coordinate where the image will appear on the applet.
- The fourth argument is a reference to an ImageObserver object.

An ImageObserver object can be any object that implements the ImageObserver interface. The Component class implements the ImageObserver interface so all components will inherit this implementation. Usually, the ImageObserver object is the object on which the image appears. For example, when the JApplet class implements this interface, it can track the progress of an image. This useful feature allows you to display a message such as "Please wait. Loading images." while graphics files are being loaded. Recall from Chapter 4 what you learned about the `this` reference. Here you use the `this` reference to refer to the applet in the following example. The code to display the eventLogo1 image is:

```
g.drawImage(eventLogo1,0,0,this);
```

A second version of the Graphics method, the drawImage() method, is unavailable to Image objects, but can be used with ImageIcon images. In this version, the drawImage()

method is used to output a scaled image. This method takes six arguments. Note that the first three arguments are the same as those for the other drawImage() method.

- The first argument is a reference to the Image object in which the image is stored.
- The second argument is the x-coordinate where the image will appear on the applet.
- The third argument is the y-coordinate where the image should appear on the applet.
- The fourth argument is a call to the getWidth() method to specify the image width for display purposes.
- The fifth argument is a call to the getHeight() method to specify the image height for display purposes (in the following example, the height should be 100 pixels fewer than the height of the applet).
- The sixth argument uses the `this` reference to implement the ImageObserver object.

The code to display the eventLogo2 image is:

```
g.drawImage(eventLogo1,0,120, getWidth(), getHeight() -
100, this);
```

You can also use the paintIcon() method to display ImageIcon images. This method requires four arguments:

- The first argument is a reference to the Component on which the image will appear—`this` in the following example.
- The second argument is a reference to the Graphics object that will be used to render the image—g in the following example.
- The third argument is the x-coordinate for the upper-left corner of the image.
- The fourth argument is the y-coordinate for the upper-left corner of the image.

The code to display the eventLogo2 image using the paintIcon() method is:

```
eventLogo2.paintIcon(this, g, 180, 0);
```

The completed Swing applet is shown in Figure 10-33 with the class name JEventImage. Output of the JEventImage is shown in Figure 10-34. You should review the preceding paragraphs while viewing the program and program output.

```
import javax.applet.*;
import java.awt.*;
import javax.swing.*;

public class JEventImage extends JApplet
{
   Image eventLogo1;
   ImageIcon eventLogo2;

   public void init()
   {
     eventLogo1 = getImage(getCodeBase(),"event.gif");
     eventLogo2 = new ImageIcon("event.gif");
   }
   public void paint (Graphics g)
     {
     g.drawImage(eventLogo1,0,0,this);
     g.drawImage(eventLogo1,0,120, getWidth(), getHeight()-100, this);
     eventLogo2.paintIcon(this, g, 180, 0);
   }
}
```

Figure 10-33 The JEventImage program

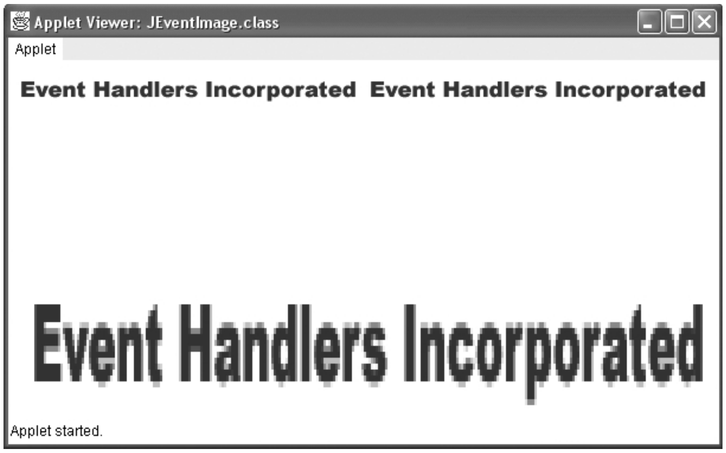

Applet Viewer: JEventImage.class
Applet
Event Handlers Incorporated
Event Handlers Incorporated
Event Handlers Incorporated
Applet started.

Figure 10-34 Output of the JEventImage program

Adding Simple Animation

At some time in your life, you probably created simple animation by drawing a series of figures on the pages of a book, and slightly changing each version of the figure from the previous one. When you flipped through the pages of the book, the figure appeared to move. Movies, whether animated or not, are created in a similar manner—you see a succession of film frames, and each one contains a slightly modified image. Computer animation is achieved in the same fashion—a series of images appear on your screen in rapid succession. You will be able to create fairly sophisticated animation after you have covered Chapter 17; for now, you can use an ActionListener to control drawing different images using the paint() method. Although the results will not be truly animated, you can achieve dynamic results in which the time appears to change. This change is accomplished by displaying time as a String and updating the String time contents with successive clicks of a JButton.

Next you will create a Swing applet for Event Handlers Incorporated that contains a graphical representation of the current day, date, and time, which changes when the user clicks a JButton. In addition, a sound plays while the message " It's time to Party…" appears under the graphical time representation. An image of a banner of the Event Handlers company name appears under the message. A snapshot of the Swing applet at a random time appears in Figure 10-35.

To create the JGregorianTime applet:

1. Open a new text file, and enter the first few lines of the JGregorianTime Swing applet:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
public class JGregorianTime extends JApplet implements
ActionListener
{
```

2. Add the following statements to create an AudioClip named sound, a new Color named tan, an empty String named lastTime, and an ImageIcon named eventLogo:

```
private AudioClip sound;
private Color tan = new Color(255, 204, 102);
private String lastTime = "";
private ImageIcon eventLogo ;
```

3. Add the following statements to create a JButton for the user to click:

```
JButton pressMe = new JButton("pressMe");
```

4. Begin the init() method, and enter the following statements to add a sound object and ImageIcon object:

```
public void init()
{
  sound = getAudioClip(getCodeBase(),"event.au");
  eventLogo = new ImageIcon("event.gif");
```

5. Add the following statements to set the background of the applet, create a container name `con`, and change the default layout from BorderLayout to FlowLayout:

```
setBackground(Color.blue);
Container con = getContentPane();
con.setLayout(new FlowLayout());
```

6. Add the following statements, which add the JButton named pressMe to the container and add an ActionListener() for pressMe. Then add the closing curly brace to the init() method.

```
  con.add(pressMe);
  pressMe.addActionListener(this);
}
```

7. Begin the paint() method and cast the Graphics context to Graphics 2D. Create a new font named monoFont and set the font of the Graphics2D object to monoFont. (Note the syntax for setting monoFont in a Graphics 2D environment.)

```
public void paint(Graphics g)
{
  Graphics2D g2D = (Graphics2D)g;
  Font monoFont = new Font("Monospaced", Font.BOLD, 20);
  g2D.setFont(monoFont);
```

8. Create a new GregorianCalendar object named day, create a String named time using the getTime() method, and then convert the result using the toString() method.

```
GregorianCalendar day = new GregorianCalendar();
String time = day.getTime().toString();
```

9. Use the setColor()method and drawString() method to set the color and draw the strings lastTime and time. (Note that when the animation starts, the String lastTime is empty.) Set the g2D object's color to tan, and then draw the time string. Finally, reference the String lastTime to the String time.

```
g2D.setColor(Color.blue);
g2D.drawString(lastTime, 5, 75);
g2D.setColor(tan);
g2D.drawString(time, 5, 75);
lastTime = time;
```

10. Add the ImageIcon eventLogo to the paint() method below the graphic representation of the day, date, and time. Add the drawString() method under the eventLogo Image to display the string "It's time to Party...". Add a repaint() method so that the JButton will be redrawn each time the paint() method is called, and then add the closing curly brace to the paint() method.

```
  eventLogo.paintIcon(this, g, 50, 120);
  g2D.drawString("It's time to Party...", 50, 100);
  pressMe.repaint();
}
```

11. Add the Swing applet's start() and stop() methods. Add the loop() method to the applet's start() method to play the sound continually. Then add the stop method to the applet's stop() method.

```
public void start()
{
  sound.loop();
}
public void stop()
{
  sound.stop();
}
```

12. At this point, the Swing applet is almost completed. You still must add the actionPerformed() method that executes when the user clicks the JButton. The only task performed by the method is to call the repaint() method. Add the following method to your applet. Then add the closing curly braces for the acitonPerformed() method and the class.

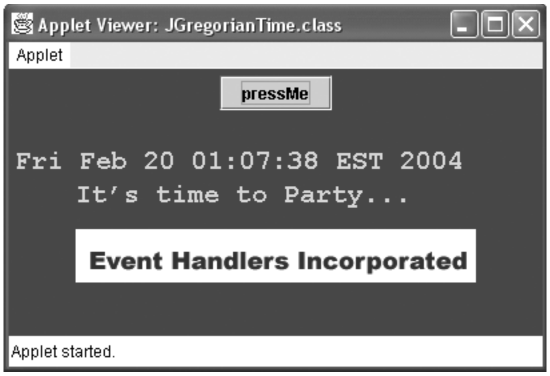
```
public void actionPerformed(ActionEvent e)
 {
  Object source = e.getSource();
  if(source == pressMe)
   repaint();
  }
}
```

13. Save the file as **JGregorianTime** in the Chapter.10 folder on your Student Disk, and then compile it using the **`javac`** command.

14. Open a new file in your text editor, and then enter the code for an HTML document to host the applet:

```
<HTML>
<APPLET CODE = "JGregorianTime.class"
   WIDTH = 400 HEIGHT = 200>
</APPLET>
</HTML>
```

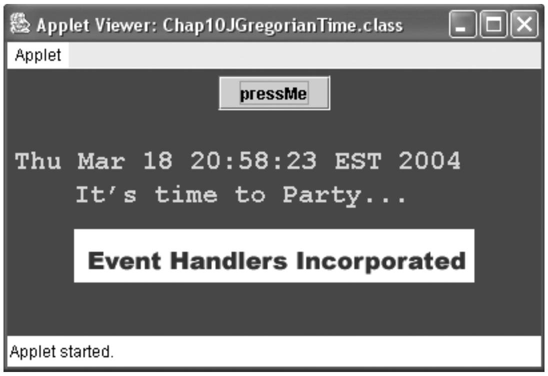
15. Save the HTML document as **TestJGregorianTime.html** in the Chapter.10 folder on your Student Disk, and then run it using the **`appletviewer TestJGregorianTime.html`** command. The output appears in Figure 10-35. Wait at least a couple of minutes, click the **pressMe** button, wait a period of time, and observe the time changes. Figure 10-36 shows the output of the program after the JButton has been clicked.



```
Fri Feb 20 01:07:38 EST 2004
    It's time to Party...
```

Event Handlers Incorporated

Figure 10-35 A snapshot of the JGregorianTime applet



```
Thu Mar 18 20:58:23 EST 2004
    It's time to Party...
```

Event Handlers Incorporated

Figure 10-36 Output of the JGregorianTime program after clicking the JButton

16. Close the Applet Viewer window.

Chapter Summary

- The paint() method header, `public void paint (Graphics g)`, requires a Graphics object argument. The paint() method runs when you display, maximize, minimize, or restore an applet. You can use the paint() method automatically supplied by Java, or you can write your own. You don't usually call the paint() method directly. Instead, you call the repaint() method, which calls the update() method, which then calls the paint() method.
- You use the drawString() method to draw a String in an applet. The drawString() method requires three arguments: a String, an x-coordinate, and a y-coordinate. The x- and y-coordinates represent the lower-left position of the String. The drawString() method is a member of the Graphics class, so you need to use a Graphics object to call it.

- You can designate a Graphics color with the setColor() method. The Color class contains 13 constants: black, blue, cyan, darkGray, gray, green, lightGray, magenta, orange, pink, red, white, and yellow. You can create your own Color object with the statement `Color someColor = new Color(r, g, b);`, where r, g, and b are numbers representing the intensity of red, green, and blue you want in your color.
- You can use the drawLine() method to draw a straight line between any two points on the screen. The drawLine() method takes four arguments: the x- and y-coordinates of the line's starting point, and the x- and y-coordinates of the line's ending point.
- You can use the drawRect() and fillRect() methods, respectively, to draw the outline of a rectangle or to draw a solid, filled rectangle. Each of these methods requires four arguments. The first two arguments represent the x- and y-coordinates of the upper-left corner of the rectangle. The last two arguments represent the width and height of the rectangle.
- The drawOval() and fillOval() methods draw ovals using four arguments—the x- and y-coordinates for the upper-left corner, and the width and height measurements of an imaginary rectangle that surrounds the oval.
- An arc is a portion of a circle that you can draw using the Graphics method drawArc(). The drawArc() method requires six arguments: the x-coordinate of the upper-left corner of an imaginary rectangle that represents the bounds of the imaginary circle that contains the arc, the y-coordinate of the same point, the width of the imaginary rectangle that represents the bounds of the imaginary circle that contains the arc, the height of the same point, the beginning arc position, and the arc angle.
- You can discover the fonts that are available on your system by calling the getLocalGraphicsEnvironment() method and then the resulting object to reference its getAllFonts() method. You can also discover the resolution and screen size on your computer system by using the getScreenResolution()and getScreenSize() methods that are part of the Toolkit class.
- A Graphics2D environment can be created in a paint() method by casting a Graphics environment.
- Drawn objects in Java 2D are created by defining them as geometric shapes using the java.awt.geom package classes. You can draw lines, rectangles, ovals, arcs, and polygons. The Graphics2D class does not have different methods for each of the shapes you can draw. Instead, you define the shape and use it as an argument to the draw() or fill() methods.
- Images can be added to an applet using the Image and ImageIcon classes.
- Sound can be added to an applet by first retrieving a sound file with WAV, AU, or MIDI formats. The sound can then be played using an AudioClip object.

REVIEW QUESTIONS

1. The method that calls the paint() method for you is ________________.
   a. callPaint()
   b. repaint()
   c. requestPaint()
   d. draw()
2. The paint() method header requires a(n) ________________ argument.
   a. void
   b. integer
   c. String
   d. Graphics
3. The statement `g.drawString(someString, 50, 100);` places someString's ________________ corner at position 50, 100.
   a. upper-left
   b. lower-left
   c. upper-right
   d. lower-right
4. If you use the setColor() method to change a Graphics object's color to yellow, ________________ will appear in yellow.
   a. only the next graphics output
   b. all graphics output for the remainder of the method
   c. all graphics output for the remainder of the applet
   d. all graphics output until you change the color
5. The correct statement to instantiate a Graphics object named picasso is ________________.
   a. `Graphics picasso;`
   b. `Graphics picasso = new Graphics();`
   c. `Graphics picasso = getGraphics();`
   d. `Graphics picasso = getGraphics(new);`
6. The statement `g.drawRoundRect(100,100,100,100,0,0);` draws a shape that looks most like a ________________.
   a. square
   b. round-edged rectangle
   c. circle
   d. straight line

7. If you draw an oval with the same value for width and height, then you draw a(n) ________________.
   a. circle
   b. square
   c. rounded square
   d. ellipsis
8. The zero-degree position for any arc is at the ________________ o'clock position.
   a. three
   b. six
   c. nine
   d. 12
9. The method you use to create a solid arc is ________________.
   a. solidArc()
   b. fillArc()
   c. arcSolid()
   d. arcFill()
10. You use the ________________ method to copy any rectangular area to a new location.
    a. copyRect()
    b. copyArea()
    c. repeatRect()
    d. repeatArea()
11. The measurement of an uppercase character from the baseline to the top of the character is its ________________.
    a. ascent
    b. descent
    c. leading
    d. height
12. To be sure that a vertical series of Strings has enough room to appear in an applet, you would use which of the following statements?
    a.
    ```
    g.drawString("Some string", x,
       y += g.getFontMetrics(). getHeight());
    ```
    b.
    ```
    g.drawString("Some string", x,
       y += g.getFontMetrics(). getLeading());
    ```

c. `g.drawString("Some string", x,`
   `y += g.getFontMetrics(). getAscent());`

d. `g.drawString("Some string", x,`
   `y += g.getFontMetrics(). getDescent());`

13. You can discover the fonts that are available on your system by using the ________________.
    a. getAllFonts() method of the GraphicsEnvironment class
    b. getAllFonts() method of the Graphics class
    c. setAllFonts() method of the GraphicsEnvironment class
    d. getAllFonts() method of the ImageEnvironment class
14. The getScreenResolution() method and getScreenSize() method ________________.
    a. both return the number of pixels as an int type
    b. return the number of pixels as an int type and an object of type Dimension
    c. both return an object of type Dimension
    d. return the number of pixels as a double type and an object of type Dimension
15. A Graphics2D object is produced by ________________.
    a. the setGraphics2D() method
    b. the `Graphics2D newpen = Graphics2D()` statement
    c. the `Graphics2D = Graphics(g)` statement
    d. casting a Graphics object
16. Java 2D uses ________________ when creating and drawing a 2D drawing object.
    a. only coordinate space
    b. only user coordinate space
    c. both coordinate and user coordinate space
    d. only 2D coordinate space
17. A gradient fill is a gradual change in ________________.
    a. color
    b. font size
    c. drawing style
    d. line thickness
18. After the getAudioClip() method retrieves a sound object named mysound, the ________________ plays a sound continually in a Swing applet.
    a. sound.loop() method
    b. loop() method
    c. mysound.loop() method
    d. mysound.continuous() method

19. The ______________ is particularly useful for loading an image into either an applet or application.
    a. Image class
    b. ImageLogo class
    c. ImageIcon class
    d. GetImage class
20. Showing successive images on the screen is called ______________.
    a. action-oriented
    b. object-oriented
    c. animation
    d. volatility

EXERCISES

1. Write a Swing applet that demonstrates displaying your first name in every even-numbered font size from 4 through 24. Save the program as **JFontSizeDemo.java** in the Chapter.10 folder on your Student Disk.
2. Write a Swing applet that displays your name in blue the first time the user clicks a JButton, and then displays your name larger and in gray the second time the user clicks the JButton. Save the program as **JBlueGray.java** in the Chapter.10 folder on your Student Disk.
3. Write a Swing applet that displays a form for creating an e-mail directory. The form should contain three JTextFields and three JLabels for first name, last name, and e-mail address. After the user enters an e-mail address and presses [Enter], the program should display the information that was entered. Use the drawString() method to display the information. Use the paint() method to display a heading line for the information display, such as "The e-mail information you entered is: ". Save the program as **JEmailForm.java** in the Chapter.10 folder on your Student Disk.
4. a. Write a Swing applet that displays a yellow smiling face on the screen. Save the program as **JSmileFace.java** in the Chapter.10 folder on your Student Disk.
   b. Add a JButton to the JSmileFace Swing applet so the smile changes to a frown when the user clicks the JButton. Save the program as **JSmileFace2.java** in the Chapter.10 folder on your Student Disk.
5. a. Use polygons and lines to create a graphics image that looks like a fireworks display. Write a Swing applet that displays the fireworks. Save the program as **JFireworks.java** in the Chapter.10 folder on your Student Disk.
   b. Add a JButton to the JFireworks Swing applet. Do not show the fireworks until the user clicks the JButton. Save the program as **JFireworks2.java** in the Chapter.10 folder on your Student Disk.

6. a. Write a Swing applet to display your name. Place boxes around your name at intervals of 10, 20, 30, and 40 pixels. Save the program as **JBorders.java** in the Chapter.10 folder on your Student Disk.
   b. Make each of the four borders in the JBorders.java applet display a different color. Save the program as **JBorders2.java** in the Chapter.10 folder on your Student Disk.
7. Create a Swing applet and use dialog boxes to prompt the user to enter a name and weight in pounds. Once the name and weight are entered, use Graphics2D methods to display the user's name and weight, with weight displayed in pounds, ounces, kilograms, and metric tons on separate lines. Use the following conversion factors:

   1 pound = 16 ounces

   1 kilogram = 1 pound / 2.204623

   1 metric ton = 1pound / 2204.623

   Save the program as **JCalculateWeight.java** in the Chapter.10 folder on your Student Disk.
8. Write a Swing applet that uses the Graphics2D environment to create a GeneralPath object. Use the General path object to create the outline of your favorite state. Display the state name at the approximate center of the state boundaries. Save the program as **JFavoriteState.java** in the Chapter.10 folder on your Student Disk.

9. Write a Swing applet that draws a realistic-looking Stop sign. Save the program as **JStopSign.java** in the Chapter.10 folder on your Student Disk.
10. Write a Swing applet that uses the ImageIcon class to place image icon objects on four JButtons. Download your favorite GIF files from the Internet. If necessary, reduce the size of the GIF images to approximately 30 by 30 pixels, or you can use the four GIF files in your Student Disk if you wish. Each time a JButton is clicked, display a different message below the JButtons. Save the program as **JButtonIcons.java** in the Chapter.10 folder on your Student Disk.
11. Each of the following files in the Chapter.10 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with Fix. For example, DebugTen1.java will become FixDebugTen1.java. You can test each applet with the TestDebugTen.html files on your Student Disk. Remember to change the Java class file referenced in the HTML document so it matches the DebugTen applet on which you are working.
    a. DebugTen1.java
    b. DebugTen2.java
    c. DebugTen3.java
    d. DebugTen4.java

Case Project

The Party Planners organization in your local town is sponsoring a contest to see who can program the best Java Swing applet to be used as an advertisement for their party events. You can download sound clips and graphics images from the Internet to use in your program. Create a Swing applet named JPartyPlanner and an HTML test file to run the Swing applet. Good luck! I hope you win!