Java™ CÓMO PROGRAMAR

Décima edición

Paul Deitel
Deitel & Associates, Inc.

Harvey Deitel
Deitel & Associates, Inc.

Traducción
Alfonso Vidal Romero Elizondo
Ingeniero en Sistemas Electrónicos
Instituto Tecnológico y de Estudios Superiores de Monterrey - Campus Monterrey

Revisión técnica
Sergio Fuenlabrada Velázquez
Edna Martha Miranda Chávez
Judith Sonck Ledezma
Mario Alberto Sesma Martínez
Mario Oviedo Galdeano
José Luis López Goytia
Departamento de Sistemas
Unidad Profesional Interdisciplinaria de Ingeniería y Ciencias Sociales y Administrativas, Instituto Politécnico Nacional, México

Pearson

Gráficos y Java 2D

13

Hay que tratar a la naturaleza en términos de un cilindro, de una esfera, de un cono, todo en perspectiva.

—Paul Cézanne

Los colores, al igual que las características, siguen los cambios de las emociones.

—Pablo Picasso

Nada se vuelve real sino hasta que se experimenta; incluso un proverbio no será proverbio para usted, sino hasta que su vida lo haya ilustrado.

—John Keats

Objetivos

En este capítulo aprenderá:

- Los contextos y los objetos de los gráficos.
- A manipular los colores y los tipos de letra.
- A usar métodos de la clase `Graphics` para dibujar varias figuras.
- A utilizar métodos de la clase `Graphics2D` de la API `Java 2D` para dibujar diversas figuras.
- Las características `Paint` y `Stroke` de las figuras mostradas con `Graphics2D`.

Plan general

13.1 Introducción

En este capítulo veremos varias de las herramientas de Java para dibujar figuras bidimensionales, controlar colores y fuentes. Parte del atractivo inicial de Java era su soporte para gráficos, el cual permitía a los programadores mejorar la apariencia visual de sus aplicaciones. Ahora, Java contiene muchas más herramientas sofisticadas de dibujo como parte de la API Java 2D (que presentamos en este capítulo) y de su tecnología sucesora JavaFX (que presentamos en el capítulo 25 y en otros dos capítulos en línea). Comenzaremos este capítulo con una introducción a muchas de las herramientas de dibujo originales de Java. Después presentaremos varias de las herramientas más poderosas de Java 2D, como el control del *estilo* de líneas utilizadas para dibujar figuras, y el control del *relleno* de las figuras con *colores* y *patrones*. Las clases que eran parte de las herramientas de gráficos originales de Java ahora se consideran parte de la API Java 2D.

En la figura 13.1 se muestra una parte de la jerarquía de clases de Java que incluye varias de las clases de gráficos básicas y las clases e interfaces de la API Java 2D que cubriremos en este capítulo. La clase `Color` contiene métodos y constantes para manipular los colores. La clase `JComponent` contiene el método `paintComponent`, que se utiliza para dibujar gráficos en un componente. La clase `Font` contiene métodos y constantes para manipular los tipos de letras. La clase `FontMetrics` contiene métodos para obtener la información del *tipo de letra*. La clase `Graphics` contiene métodos para dibujar cadenas, líneas, rectángulos y otras figuras. La clase `Graphics2D`, que extiende a la clase `Graphics`, se utiliza para dibujar con la API Java 2D. La clase `Polygon` contiene métodos para crear *polígonos*. La mitad inferior de la figura muestra varias clases e interfaces de la API Java 2D. La clase `BasicStroke` ayuda a especificar las características de dibujo de las *líneas*. Las clases `GradientPaint` y `TexturePaint` ayudan a especificar las características para rellenar *figuras* con *colores* o *patrones*. Las clases `GeneralPath`, `Line2D`, `Arc2D`, `Ellipse2D`, `Rectangle2D` y `RoundRectangle2D` representan varias figuras de Java 2D.

Para empezar a dibujar en Java, primero debemos entender su **sistema de coordenadas** (figura 13.2), el cual representa un esquema para identificar a cada uno de los posibles *puntos* en la pantalla. De manera predeterminada, la *esquina superior izquierda* de un componente de la GUI (como una ventana) tiene las coordenadas (0, 0). Un par de coordenadas está compuesto por una **coordenada *x*** (la **coordenada horizontal**) y una **coordenada *y*** (la **coordenada vertical**). La coordenada *x* es la distancia horizontal que se desplaza *hacia la derecha*, desde la parte izquierda de la pantalla. La coordenada *y* es la distancia vertical que se desplaza *hacia abajo*, desde la parte *superior* de la pantalla. El **eje *x*** describe cada una de las coordenadas horizontales, y el **eje *y*** describe cada una de las coordenadas verticales. Las coordenadas se utilizan para indicar en que parte de la pantalla deben mostrarse los gráficos. Las unidades de las coordenadas se miden en **píxeles** (lo que se conoce como "elementos de imagen"). Un píxel es la *unidad más pequeña de resolución* de un monitor de computadora.

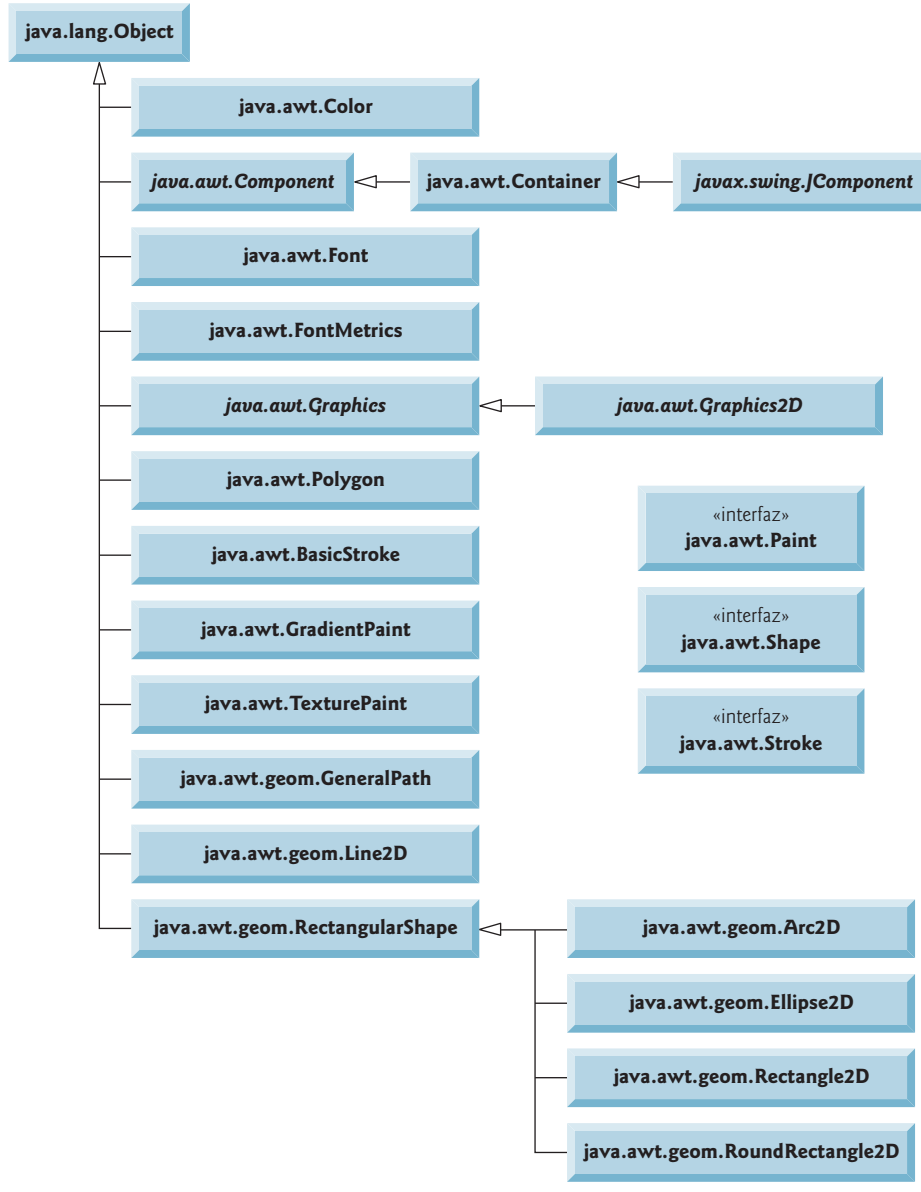

Fig. 13.1 | Clases e interfaces utilizadas en este capítulo, provenientes de las herramientas de gráficos originales de Java y de la API Java2D.

Tip de portabilidad 13.1

Existen distintos tipos de monitores de computadora con distintas resoluciones (es decir, la densidad de los píxeles varía). Esto puede hacer que los gráficos aparezcan de distintos tamaños en distintos monitores, o en el mismo monitor con distintas configuraciones.

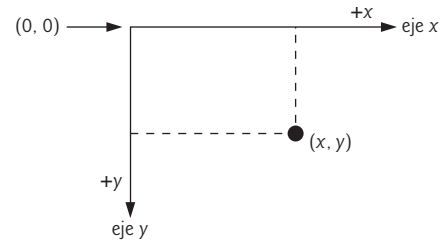

Fig. 13.2 | Sistema de coordenadas de Java. Las unidades se miden en píxeles.

13.2 Contextos y objetos de gráficos

Un **contexto de gráficos** permite dibujar en la pantalla. Un objeto `Graphics` administra un contexto de gráficos y dibuja píxeles en la pantalla que representan *texto* y otros objetos gráficos (como *líneas*, *elipses*, *rectángulos* y otros *polígonos*). Los objetos `Graphics` contienen métodos para *dibujar*, *manipular tipos de letra*, *manipular colores* y varias cosas más.

La clase `Graphics` es una clase `abstract` (es decir, no pueden instanciarse objetos `Graphics`). Esto contribuye a la portabilidad de Java. Como el dibujo se lleva a cabo *de manera distinta* en cada plataforma que soporta a Java, no puede haber sólo una implementación de las herramientas de dibujo en todos los sistemas. Cuando Java se implementa en una plataforma específica, se crea una subclase de `Graphics` que implementa las herramientas de dibujo. Esta implementación está oculta para nosotros por medio de la clase `Graphics`, la cual proporciona la interfaz que nos permite utilizar gráficos de una manera *independiente de la plataforma*.

En el capítulo 12 vimos que la clase `Component` es la *superclase* para muchas de las clases en el paquete `java.awt`. La clase `JComponent` (paquete `javax.swing`), que hereda de manera indirecta de la clase `Component`, contiene un método llamado `paintComponent`, que puede utilizarse para dibujar gráficos. El método `paintComponent` toma un objeto `Graphics` como argumento. El sistema pasa este objeto al método `paintComponent` cuando se requiere volver a pintar un componente ligero de Swing. El encabezado del método `paintComponent` es:

```
public void paintComponent(Graphics g)
```

El parámetro `g` recibe una referencia a una instancia de la subclase específica del sistema de `Graphics`. Tal vez a usted le parezca conocido el encabezado del método anterior; es el mismo que utilizamos en algunas de las aplicaciones del capítulo 12. En realidad, la clase `JComponent` es una *superclase* de `JPanel`. Muchas herramientas de la clase `JPanel` son heredadas de la clase `JComponent`.

El método `paintComponent` raras veces es llamado directamente por el programador, ya que el dibujo de gráficos es un proceso *controlado por eventos*. Como vimos en el capítulo 11, Java usa un modelo *multihilos* de ejecución del programa. Cada hilo es una actividad *paralela*. Cada programa puede tener muchos hilos. Al crear una aplicación de GUI, uno de esos hilos es el **hilo de despachamiento de eventos (EDT)**, que es el que se utiliza para procesar todos los eventos de la GUI. Toda la manipulación de los componentes de GUI debe realizarse en ese hilo. Cuando se ejecuta una aplicación de GUI, el contenedor de la aplicación llama al método `paintComponent` (en el hilo de despachamiento de eventos) para cada componente ligero conforme la GUI se muestra en pantalla. Para que `paintComponent` sea llamado de nuevo, debe ocurrir un evento (como *cubrir* y *descubrir* el componente con otra ventana).

Si el programador necesita hacer que se ejecute `paintComponent` (es decir, si desea actualizar los gráficos dibujados en el componente de Swing), se hace una llamada al método **`repaint`** que devuelve `void` sin recibir argumentos, y todos los objetos `JComponent` lo heredan indirectamente de la clase `Component` (paquete `java.awt`).

13.3 Control de colores

La clase `Color` declara los métodos y las constantes para manipular los colores en un programa de Java. Las constantes de colores previamente declaradas se sintetizan en la figura 13.3, mientras que varios métodos y constructores para los colores se sintetizan en la figura 13.4. Dos de los métodos de la figura 13.4 son métodos de `Graphics` que son específicos para los colores.

Constante de Color	Valor RGB
`public static final Color RED`	255, 0, 0
`public static final Color GREEN`	0, 255, 0
`public static final Color BLUE`	0, 0, 255
`public static final Color ORANGE`	255, 200, 0
`public static final Color PINK`	255, 175, 175
`public static final Color CYAN`	0, 255, 255
`public static final Color MAGENTA`	255, 0, 255
`public static final Color YELLOW`	255, 255, 0
`public static final Color BLACK`	0, 0, 0
`public static final Color WHITE`	255, 255, 255
`public static final Color GRAY`	128, 128, 128
`public static final Color LIGHT_GRAY`	192, 192, 192
`public static final Color DARK_GRAY`	64, 64, 64

Fig. 13.3 | Constantes de `Color` y sus valores RGB.

Método	Descripción
Constructores y métodos de `Color`	
`public Color(int r, int g, int b)`	Crea un color basado en los componentes rojo, verde y azul, expresados como enteros de 0 a 255.
`public Color(float r, float g, float b)`	Crea un color basado en los componentes rojo, verde y azul, expresados como valores de punto flotante de 0.0 a 1.0.
`public int getRed()`	Devuelve un valor entre 0 y 255, el cual representa el contenido rojo.

Fig. 13.4 | Los métodos de `Color` y los métodos de `Graphics` relacionados con los colores (parte 1 de 2).

Método	Descripción
`public int getGreen()`	
	Devuelve un valor entre 0 y 255, el cual representa el contenido verde.
`public int getBlue()`	
	Devuelve un valor entre 0 y 255, el cual representa el contenido azul.
Métodos de `Graphics` para manipular objetos `Colors`	
`public Color getColor()`	
	Devuelve un objeto Color que representa el color actual para el contexto de gráficos.
`public void setColor(Color c)`	
	Establece el color actual para dibujar con el contexto de gráficos.

Fig. 13.4 | Los métodos de `Color` y los métodos de `Graphics` relacionados con los colores (parte 2 de 2).

Todo color se crea a partir de un valor rojo, verde y azul. En conjunto, a estos componentes se les llama **valores RGB**. Los tres componentes RGB pueden ser enteros en el rango de 0 a 255, o pueden ser valores de punto flotante en el rango de 0.0 a 1.0. El primer componente RGB especifica la cantidad de rojo, el segundo la cantidad de verde y el tercero la cantidad de azul. Entre mayor sea el valor RGB mayor será la cantidad de ese color en particular. Java permite seleccionar de entre 256 × 256 × 256 (o aproximadamente 16.7 millones de) colores. No todas las computadoras son capaces de mostrar todos estos colores. La pantalla mostrará el color más cercano que pueda.

En la figura 13.4 se muestran dos de los constructores de la clase `Color` (uno que toma tres argumentos `int` y otro que toma tres argumentos `float`, en donde cada argumento especifica la cantidad de rojo, verde y azul). Los valores `int` deben estar en el rango de 0 a 255 y los valores `float` deben estar en el rango de 0.0 a 1.0. El nuevo objeto `Color` tendrá las cantidades de rojo, verde y azul que se especifiquen. Los métodos **`getRed`**, **`getGreen`** y **`getBlue`** de `Color` devuelven valores enteros de 0 a 255, los cuales representan la cantidad de rojo, verde y azul, respectivamente. El método **`getColor`** de `Graphics` devuelve un objeto `Color` que representa el color actual de dibujo. El método **`setColor`** de `Graphics` establece el color actual de dibujo.

Dibujar en distintos colores

Las figuras 13.5 y 13.6 demuestran varios métodos de la figura 13.4 al dibujar *rectángulos rellenos* y objetos `String` en varios colores distintos. Cuando la aplicación empieza a ejecutarse, se hace una llamada al método `paintComponent` de la clase `JPanelColor` (líneas 10 a 37 de la figura 13.5) para pintar la ventana. En la línea 17 se utiliza el método `setColor` de `Graphics` para establecer el color actual de dibujo. El método `setColor` recibe un objeto `Color`. La expresión `new Color(255, 0, 0)` crea un nuevo objeto `Color` que representa rojo (valor `255` para rojo y `0` para los valores verde y azul). En la línea 18 se utiliza el método **`fillRect`** de `Graphics` para dibujar un *rectángulo relleno* con el color actual. El método `fillRect` dibuja un rectángulo con base en sus cuatro argumentos. Los primeros dos valores enteros representan la coordenada *x* superior izquierda y la coordenada *y* superior izquierda en donde el objeto `Graphics` empieza a dibujar el rectángulo. El tercer y cuarto argumentos son enteros no negativos que representan la anchura y la altura del rectángulo en píxeles, respectivamente. Un rectángulo que se dibuja usando el método `fillRect` se rellena con el color actual del objeto `Graphics`.

```
1  // Fig. 13.5: PanelColor.java
2  // Cambiar los colores del dibujo.
3  import java.awt.Graphics;
4  import java.awt.Color;
5  import javax.swing.JPanel;
6
7  public class JPanelColor extends JPanel
8  {
9     // dibuja rectángulos y objetos String en distintos colores
10    @Override
11    public void paintComponent(Graphics g)
12    {
13       super.paintComponent(g);
14       this.setBackground(Color.WHITE);
15
16       // establece nuevo color de dibujo, usando valores enteros
17       g.setColor(new Color(255, 0, 0));
18       g.fillRect(15, 25, 100, 20);
19       g.drawString("RGB actual: " + g.getColor(), 130, 40);
20
21       // establece nuevo color de dibujo, usando valores de punto flotante
22       g.setColor(new Color(0.50f, 0.75f, 0.0f));
23       g.fillRect(15, 50, 100, 20);
24       g.drawString("RGB actual: " + g.getColor(), 130, 65);
25
26       // establece nuevo color de dibujo, usando objetos Color static
27       g.setColor(Color.BLUE);
28       g.fillRect(15, 75, 100, 20);
29       g.drawString("RGB actual: " + g.getColor(), 130, 90);
30
31       // muestra los valores RGB individuales
32       Color color = Color.MAGENTA;
33       g.setColor(color);
34       g.fillRect(15, 100, 100, 20);
35       g.drawString("Valores RGB: " + color.getRed() + ", " +
36          color.getGreen() + ", " + color.getBlue(), 130, 115);
37    }
38 } // fin de la clase JPanelColor
```

Fig. 13.5 | Cómo cambiar colores de dibujo.

```
1  // Fig. 13.6: MostrarColores.java
2  // Demostración de objetos Color.
3  import javax.swing.JFrame;
4
5  public class MostrarColores
6  {
7     // ejecuta la aplicación
8     public static void main(String[] args)
9     {
```

Fig. 13.6 | Demostración de los objetos `Color` (parte 1 de 2).

```
10         // crea marco para objeto JPanelColor
11         JFrame frame = new JFrame("Uso de colores");
12         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14         JPanelColor jPanelColor = new JPanelColor();
15         frame.add(jPanelColor);
16         frame.setSize(400, 180);
17         frame.setVisible(true);
18      }
19   } // fin de la clase MostrarColores
```

Fig. 13.6 | Demostración de los objetos `Color` (parte 2 de 2).

En la línea 19 se utiliza el método **`drawString`** de `Graphics` para dibujar un objeto `String` en el color actual. La expresión `g.getColor()` recupera el color actual del objeto `Graphics`. Después concatenamos el objeto `Color` devuelto con la cadena "`RGB actual:`", lo que produce una llamada *implícita* al método `toString` de la clase `Color`. La representación `String` de un objeto `Color` contiene el nombre de la clase y el paquete (`java.awt.Color`), además de los valores rojo, verde y azul.

Observación de apariencia visual 13.1

Todos percibimos los colores de una forma distinta. Elija sus colores con cuidado, para asegurarse que su aplicación sea legible, tanto para las personas que pueden percibir el color, como para aquellas que no pueden ver ciertos colores. Trate de evitar usar muchos colores distintos muy cerca unos de otros.

En las líneas 22 a 24 y 27 a 29 se llevan a cabo de nuevo las mismas tareas. En la línea 22 se utiliza el constructor de `Color` con tres argumentos `float` para crear un color verde oscuro (`0.50f` para rojo, `0.75f` para verde y `0.0f` para azul). Observe la sintaxis de los valores. La letra `f` anexada a una literal de punto flotante indica que la literal debe tratarse como de tipo `float`. Recuerde que de manera predeterminada las literales de punto flotante se tratan como de tipo `double`.

En la línea 27 se establece el color actual de dibujo a una de las constantes de `Color` declaradas con anterioridad (`Color.BLUE`). Las constantes de `Color` son `static` por lo que se crean cuando la clase `Color` se carga en memoria en tiempo de ejecución.

La instrucción de las líneas 35 y 36 hace llamadas a los métodos `getRed`, `getGreen` y `getBlue` de `Color` en la constante `Color.MAGENTA` antes declarada. El método `main` de la clase `MostrarColores` (líneas 8 a 18 de la figura 13.6) crea el objeto `JFrame` que contendrá un objeto `ColoresJPanel` en el que se mostrarán los colores.

Observación de ingeniería de software 13.1

Para cambiar el color es necesario crear un nuevo objeto `Color` (o utilizar una de las constantes de `Color` previamente declaradas). Al igual que los objetos `String`, los objetos `Color` son inmutables (no pueden modificarse).

El componente **JcolorChooser** (paquete `javax.swing`) permite a los usuarios de aplicaciones seleccionar colores. En las figuras 13.7 y 13.8 se muestra un cuadro de diálogo `JColorChooser`. Al hacer clic en el botón **Cambiar color**, aparece un cuadro de diálogo `JColorChooser`. Al seleccionar un color y oprimir el botón **Aceptar** del diálogo, el color de fondo de la ventana de la aplicación cambia.

```
1  // Fig. 13.7: MostrarColores2JFrame.java
2  // Selección de colores con JColorChooser.
3  import java.awt.BorderLayout;
4  import java.awt.Color;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7  import javax.swing.JButton;
8  import javax.swing.JFrame;
9  import javax.swing.JColorChooser;
10 import javax.swing.JPanel;
11
12 public class MostrarColores2JFrame extends JFrame
13 {
14    private final JButton cambiarColorJButton;
15    private Color color = Color.LIGHT_GRAY;
16    private final JPanel coloresJPanel;
17
18    // establece la GUI
19    public MostrarColores2JFrame()
20    {
21       super("Uso de JColorChooser");
22
23       // crea objeto JPanel para mostrar color
24       coloresJPanel = new JPanel();
25       coloresJPanel.setBackground(color);
26
27       // establece cambiarColorJButton y registra su manejador de eventos
28       cambiarColorJButton = new JButton("Cambiar color");
29       cambiarColorJButton.addActionListener(
30          new ActionListener() // clase interna anónima
31          {
32             // muestra JColorChooser cuando el usuario hace clic con el botón
33             @Override
34             public void actionPerformed(ActionEvent evento)
35             {
36                color = JColorChooser.showDialog(
37                   MostrarColores2JFrame.this, "Seleccione un color", color);
38
39                // establece el color predeterminado, si no se devuelve un color
40                if (color == null)
41                   color = Color.LIGHT_GRAY;
42
43                // cambia el color de fondo del panel de contenido
44                coloresJPanel.setBackground(color);
45             } // fin del método actionPerformed
46          } // fin de la clase interna anónima
47       ); // fin de la llamada a addActionListener
```

Fig. 13.7 | Selección de colores con `JColorChooser` (parte 1 de 2).

```
48
49          add(coloresJPanel, BorderLayout.CENTER);
50          add(cambiarColorJButton, BorderLayout.SOUTH);
51
52          setSize(400, 130);
53          setVisible(true);
54       } // fin del constructor de MostrarColores2JFrame
55    } // fin de la clase MostrarColores2JFrame
```

Fig. 13.7 | Selección de colores con `JColorChooser` (parte 2 de 2).

```
1  // Fig. 13.8: MostrarColores2.java
2  // Selección de colores con JColorChooser.
3  import javax.swing.JFrame;
4
5  public class MostrarColores2
6  {
7     // ejecuta la aplicación
8     public static void main(String[] args)
9     {
10       MostrarColores2JFrame aplicacion = new MostrarColores2JFrame();
11       aplicacion.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12    }
13 } // fin de la clase MostrarColores2
```

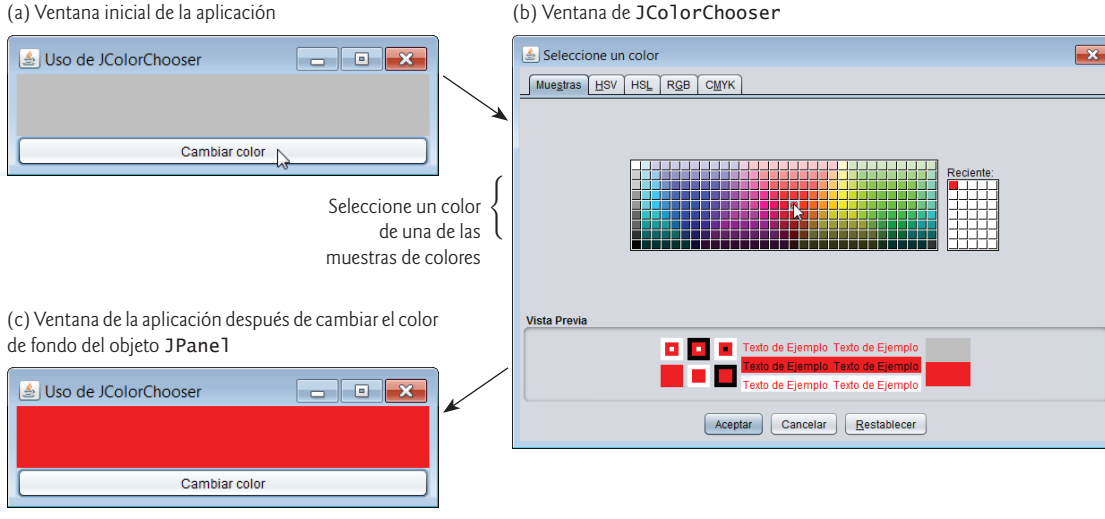

Fig. 13.8 | Selección de colores con `JColorChooser`.

La clase `JColorChooser` proporciona el método estático **`showDialog`** que crea un objeto `JColorChooser`, lo adjunta a un cuadro de diálogo y lo muestra en pantalla. Las líneas 36 y 37 de la figura 13.7

invocan a este método para mostrar el cuadro de diálogo del selector de colores. El método `showDialog` devuelve el objeto `Color` seleccionado, o `null` si el usuario oprime **Cancelar** o si cierra el cuadro de diálogo sin oprimir **Aceptar**. Este método recibe tres argumentos: una referencia a su objeto `Component` padre, un objeto `String` que muestra en la barra de título del cuadro de diálogo y el `Color` inicial seleccionado para el cuadro de diálogo. El componente padre es una referencia a la ventana desde la que se muestra el diálogo (en este caso el objeto `JFrame`, con el nombre de referencia `marco`). Este diálogo estará centrado en el componente padre. Si el padre es `null`, entonces el cuadro de diálogo se centra en la pantalla. Mientras el diálogo para seleccionar colores se encuentre en la pantalla, el usuario no podrá interactuar con el componente padre sino hasta cerrar el diálogo. A este tipo de cuadro de diálogo se le conoce como cuadro de diálogo modal.

Una vez que el usuario selecciona un color en las líneas 40 y 41 se determina si color es `null`, en cuyo caso `color` se establece en el valor predeterminado `Color.LIGHT_GRAY`. En la línea 44 se invoca el método `setBackground` para cambiar el color de fondo del objeto `JPanel`. El método `setBackground` es uno de los muchos métodos de la clase `Component` que pueden utilizarse en la mayoría de los componentes de la GUI. El usuario puede seguir utilizando el botón **Cambiar color** para cambiar el color de fondo de la aplicación. La figura 13.8 contiene el método `main`, que ejecuta el programa.

La figura 13.8(b) muestra el cuadro de diálogo `JColorChooser` predeterminado, que permite al usuario seleccionar un color de una variedad de **muestras de colores**. Hay cinco fichas en la parte superior del cuadro de diálogo: **Muestras**, **HSV**, **HSL**, **RGB** y **CMYK**. Estas fichas representan cinco distintas formas de seleccionar un color. La ficha **HSV** le permite seleccionar un color con base en el **matiz** (hue), la **saturación** (saturation) y el **brillo** (brightness), valores que se utilizan para definir la cantidad de luz en un color. Para obtener más información sobre HSV (o HSB), visite `http://es.wikipedia.org/wiki/Modelo_de_color_HSV`. La ficha **RGB** le permite seleccionar un color mediante el uso de controles deslizables para seleccionar los componentes rojo, verde y azul del color. Las fichas **HSV** y **RGB** se muestran en la figura 13.9.

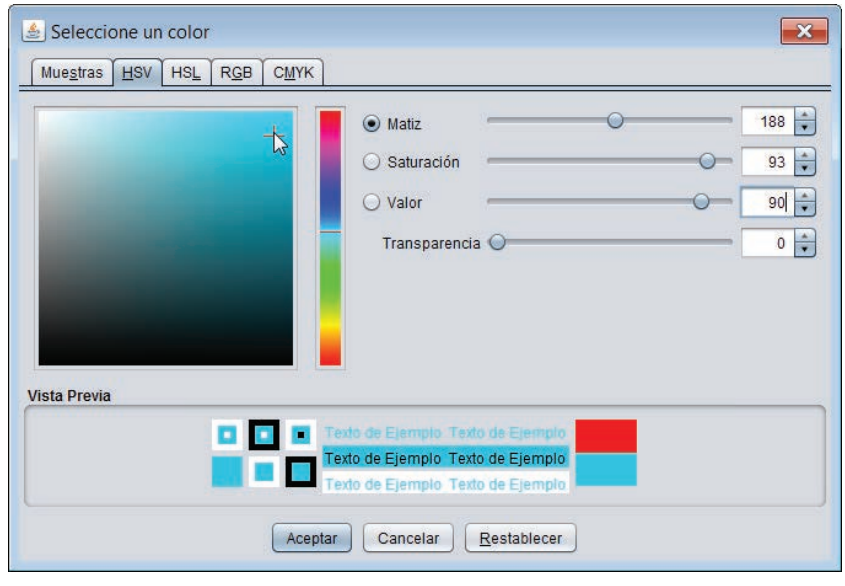

Fig. 13.9 | Las fichas **HSV** y **RGB** del cuadro de diálogo `JColorChooser` (parte 1 de 2).

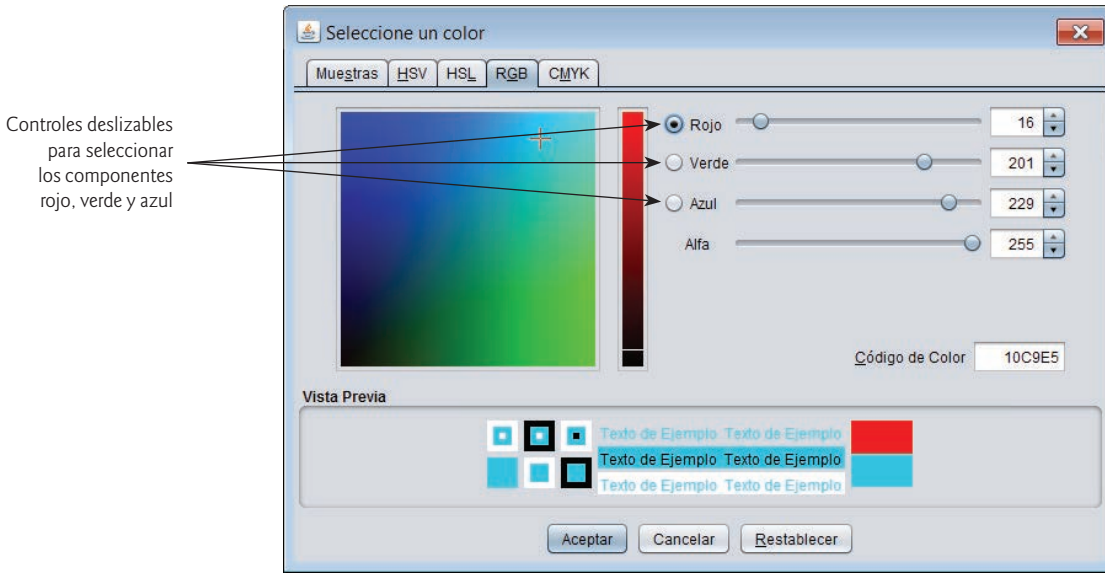

Fig. 13.9 | Las fichas **HSV** y **RGB** del cuadro de diálogo `JColorChooser` (parte 2 de 2).

13.4 Manipulación de tipos de letra

En esta sección presentaremos los métodos y constantes para controlar los tipos de letras. La mayoría de los métodos y constantes de tipos de letra son parte de la clase `Font`. Algunos constructores, métodos y constantes de la clase `Font` y la clase `Graphics` se sintetizan en la figura 13.10.

Método o constante	Descripción
Constantes, constructores y métodos de `Font`	
`public static final int PLAIN`	Constante que representa un estilo de tipo de letra simple.
`public static final int BOLD`	Constante que representa un estilo de tipo de letra en negritas.
`public static final int ITALIC`	Constante que representa un estilo de tipo de letra en cursivas.
`public Font(String nombre, int estilo, int tamanio)`	Crea un objeto `Font` con el nombre de tipo de letra, estilo y tamaño especificados.
`public int getStyle()`	Devuelve un `int` que indica el estilo actual de tipo de letra.
`public int getSize()`	Devuelve un `int` que indica el tamaño actual del tipo de letra.
`public String getName()`	Devuelve el nombre actual del tipo de letra, como una cadena.
`public String getFamily()`	Devuelve el nombre de la familia del tipo de letra, como una cadena.
`public boolean isPlain()`	Devuelve `true` si el tipo de letra es simple; `false` en caso contrario.
`public boolean isBold()`	Devuelve `true` si el tipo de letra está en negritas; `false` en caso contrario.
`public boolean isItalic()`	Devuelve `true` si el tipo de letra está en cursivas; `false` en caso contrario.

Fig. 13.10 | Métodos y constantes relacionados con `Font` (parte 1 de 2).

Método o constante	Descripción
Métodos de `Graphics` para manipular objetos `Font`	
`public Font getFont()`	Devuelve la referencia a un objeto `Font` que representa el tipo de letra actual.
`public void setFont(Font f)`	Establece el tipo de letra, el estilo y el tamaño actuales especificados por la referencia `f` al objeto `Font`.

Fig. 13.10 | Métodos y constantes relacionados con `Font` (parte 2 de 2).

El constructor de la clase `Font` recibe tres argumentos: el **nombre del tipo de letra**, su **estilo** y su **tamaño**. El nombre del tipo de letra es cualquier tipo soportado por el sistema en el que se esté ejecutando el programa, como los tipos de letra estándar de Java `Monospaced`, `SansSerif` y `Serif`. El estilo de tipo de letra es `Font.PLAIN` (simple), `Font.ITALIC` (cursivas) o `Font.BOLD` (negritas); cada uno es un campo `static` de la clase `Font`. Los estilos de letra pueden usarse combinados (por ejemplo, `Font.ITALIC + Font.BOLD`). El tamaño del tipo de letra se mide en puntos. Un **punto** es 1/72 de una pulgada. El método **`setFont`** de `Graphics` establece el tipo de letra a dibujar en ese momento (el tipo de letra en el cual se mostrará el texto) con base en su argumento `Font`.

Tip de portabilidad 13.2

El número de tipos de letra varía enormemente entre sistemas. Java proporciona cinco tipos de letras (`Serif`, `Monospaced`, `SansSerif`, `Dialog` y `DialogInput`) que pueden usarse en todas las plataformas de Java. El entorno en tiempo de ejecución de Java (JRE) en cada plataforma asigna estos nombres de tipos de letras lógicos a los tipos de letras que están realmente instalados en la plataforma. Los tipos de letras reales que se utilicen pueden variar de una plataforma a otra.

La aplicación de las figuras 13.11 y 13.12 muestra texto en cuatro tipos de letra distintos y con diferente tamaño. La figura 13.11 utiliza el constructor de `Font` para inicializar objetos `Font` (en las líneas 17, 21, 25 y 30) que se pasan al método `setFont` de `Graphics` para cambiar el tipo de letra para dibujar. Cada llamada al constructor de `Font` pasa un nombre de tipo de letra (`Serif`, `Monospaced`, o `SansSerif`) como una cadena, un estilo de tipo de letra (`Font.PLAIN`, `Font.ITALIC` o `Font.BOLD`) y un tamaño de tipo de letra. Una vez que se invoca el método `setFont` de `Graphics`, todo el texto que se muestre después de la llamada aparecerá en el nuevo tipo de letra hasta que éste se modifique. La información de cada tipo de letra se muestra en las líneas 18, 22, 26, 31 y 32, usando el método `drawString`. Las coordenadas que se pasan a `drawString` corresponden a la esquina inferior izquierda de la línea base del tipo de letra. En la línea 29 se cambia el color de la letra a rojo, por lo que la siguiente cadena que se muestra aparece en color rojo. En las líneas 31 y 32 se muestra información acerca del objeto `Font` final. El método **`getFont`** de la clase `Graphics` devuelve un objeto `Font` que representa el tipo de letra actual. El método **`getName`** devuelve el nombre del tipo de letra actual como una cadena. El método **`getSize`** devuelve el tamaño del tipo de letra, en puntos.

Observación de ingeniería de software 13.2

Para cambiar el tipo de letra, debe crear un nuevo objeto `Font`. Los objetos `Font` son inmutables; la clase `Font` no tiene métodos establecer para modificar las características del tipo de letra actual.

La figura 13.12 contiene el método `main`, que crea un objeto `JFrame` para mostrar un objeto `FontJPanel`. Agregamos un objeto `FontJPanel` a este objeto `JFrame` (línea 15), el cual muestra los gráficos creados en la figura 13.11.

```
1  // Fig. 13.11: FontJPanel.java
2  // Muestra cadenas en distintos tipos de letra y colores.
3  import java.awt.Font;
4  import java.awt.Color;
5  import java.awt.Graphics;
6  import javax.swing.JPanel;
7
8  public class FontJPanel extends JPanel
9  {
10    // muestra objetos String en distintos tipos de letra y colores
11    @Override
12    public void paintComponent(Graphics g)
13    {
14       super.paintComponent(g);
15
16       // establece el tipo de letra a Serif (Times), negrita, 12 puntos y dibuja
         una cadena
17       g.setFont(new Font("Serif", Font.BOLD, 12));
18       g.drawString("Serif 12 puntos, negrita.", 20, 50);
19
20       // establece el tipo de letra a Monospaced (Courier), cursiva, 24 puntos y
         dibuja una cadena
21       g.setFont(new Font("Monospaced", Font.ITALIC, 24));
22       g.drawString("Monospaced 24 puntos, cursiva.", 20, 70);
23
24       // establece el tipo de letra a SansSerif (Helvetica), simple, 14 puntos y
         dibuja una cadena
25       g.setFont(new Font("SansSerif", Font.PLAIN, 14));
26       g.drawString("SansSerif 14 puntos, simple.", 20, 90);
27
28       // establece el tipo de letra a Serif (Times), negrita/cursiva, 18 puntos
         y dibuja una cadena
29       g.setColor(Color.RED);
30       g.setFont(new Font("Serif", Font.BOLD + Font.ITALIC, 18));
31       g.drawString(g.getFont().getName() + " " + g.getFont().getSize() +
32          " puntos, negrita cursiva.", 20, 110);
33    }
34 } // fin de la clase FontJPanel
```

Fig. 13.11 | Mostrar cadenas en distintos tipos de letra y colores.

```
1  // Fig. 13.12: TiposDeLetra.java
2  // Uso de tipos de letra.
3  import javax.swing.JFrame;
4
5  public class TiposDeLetra
6  {
7     // ejecuta la aplicación
8     public static void main(String[] args)
9     {
10       // crea marco para FontJPanel
11       JFrame marco = new JFrame("Uso de tipos de letra");
12       marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14       FontJPanel fontJPanel = new FontJPanel();
15       marco.add(fontJPanel);
```

Fig. 13.12 | Uso de tipos de letra (parte 1 de 2).

```
16          marco.setSize(475, 170);
17          marco.setVisible(true);
18       }
19    } // fin de la clase TiposDeLetra
```

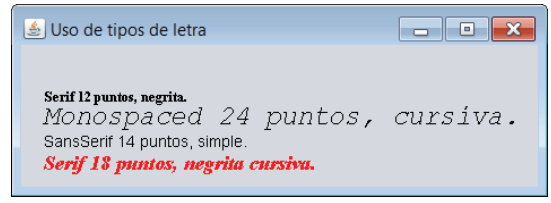

Fig. 13.12 | Uso de tipos de letra (parte 2 de 2).

Métrica de los tipos de letra

En ocasiones es necesario obtener información sobre el tipo de letra actual (como el nombre, el estilo y el tamaño) para dibujar. En la figura 13.10 se sintetizan varios métodos de `Font` que se utilizan para obtener información sobre el tipo de letra. El método **`getStyle`** devuelve un valor entero que representa el estilo actual. El valor entero devuelto puede ser `Font.PLAIN`, `Font.ITALIC`, `Font.BOLD` o la combinación de `Font.ITALIC` y `Font.BOLD`. El método **`getFamily`** devuelve el nombre de la familia a la que pertenece el tipo de letra actual. El nombre de la familia del tipo de letra es específico de la plataforma. También hay métodos de `Font` disponibles para probar el estilo del tipo de letra actual, los cuales se sintetizan también en la figura 13.10. Los métodos **`isPlain`**, **`isBold`** e **`isItalic`** devuelven `true` si el estilo del tipo de letra actual es simple, negrita o cursiva, respectivamente.

En la figura 13.13 se muestran algunos elementos comunes de la **métrica de los tipos de letras** que proporcionan información precisa acerca de un tipo de letra, como la **altura**, el **descendente** (la distancia entre la base de la línea y el punto inferior del tipo de letra), el **ascendente** (la cantidad que se eleva un carácter por encima de la base de la línea) y el **interlineado** (la diferencia entre el descendente de una línea de texto y el ascendente de la línea de texto que está debajo; es decir, el espaciamiento entre líneas).

Fig. 13.13 | Métrica de los tipos de letra.

La clase **`FontMetrics`** declara varios métodos para obtener información métrica de los tipos de letra. En la figura 13.14 se sintetizan estos métodos, junto con el método **`getFontMetrics`** de la clase `Graphics`. La aplicación de las figuras 13.15 y 13.16 utiliza los métodos de la figura 13.14 para obtener la información métrica de dos tipos de letra.

Método	Descripción
Métodos de `FontMetrics`	
`public int getAscent()`	Devuelve un valor que representa el ascendente de un tipo de letra, en puntos.
`public int getDescent()`	Devuelve un valor que representa el descendente de un tipo de letra, en puntos.
`public int getLeading()`	Devuelve un valor que representa el interlineado de un tipo de letra, en puntos.
`public int getHeight()`	Devuelve un valor que representa la altura de un tipo de letra, en puntos.
Métodos de `Graphics` para obtener la métrica de un tipo de letra	
`public FontMetrics getFontMetrics()`	
	Devuelve el objeto `FontMetrics` para el objeto `Font` actual.
`public FontMetrics getFontMetrics(Font f)`	
	Devuelve el objeto `FontMetrics` para el argumento `Font` especificado.

Fig. 13.14 | Métodos de `FontMetrics` y `Graphics` para obtener la métrica de los tipos de letra.

```
1  // Fig. 13.15: MetricaJPanel.java
2  // Métodos de FontMetrics y Graphics útiles para obtener la métrica de los tipos
   de letra.
3  import java.awt.Font;
4  import java.awt.FontMetrics;
5  import java.awt.Graphics;
6  import javax.swing.JPanel;
7
8  public class MetricaJPanel extends JPanel
9  {
10    // muestra la métrica de los tipos de letra
11    @Override
12    public void paintComponent(Graphics g)
13    {
14       super.paintComponent(g);
15
16       g.setFont(new Font("SansSerif", Font.BOLD, 12));
17       FontMetrics metrica = g.getFontMetrics();
18       g.drawString("Tipo de letra actual: " + g.getFont(), 10, 40);
19       g.drawString("Ascendente: " + metrica.getAscent(), 10, 55);
20       g.drawString("Descendente: " + metrica.getDescent(), 10, 70);
21       g.drawString("Altura: " + metrica.getHeight(), 10, 85);
22       g.drawString("Interlineado: " + metrica.getLeading(), 10, 100);
23
24       Font tipoLetra = new Font("Serif", Font.ITALIC, 14);
25       metrica = g.getFontMetrics (tipoLetra);
26       g.setFont(tipoLetra);
27       g.drawString("Tipo de letra actual: " + tipoLetra, 10, 130);
28       g.drawString("Ascendente: " + metrica.getAscent(), 10, 145);
29       g.drawString("Descendente: " + metrica.getDescent(), 10, 160);
30       g.drawString("Altura: " + metrica.getHeight(), 10, 175);
31       g.drawString("Interlineado: " + metrica.getLeading(), 10, 190);
32    }
33 } // fin de la clase MetricaJPanel
```

Fig. 13.15 | Métodos de `FontMetrics` y `Graphics` útiles para obtener la métrica de los tipos de letra.

```
1  // Fig. 13.16: Metrica.java
2  // Muestra de la métrica de los tipos de letra.
3  import javax.swing.JFrame;
4
5  public class Metrica
6  {
7     // ejecuta la aplicación
8     public static void main(String[] args)
9     {
10       // crea marco para objeto MetricaJPanel
11       JFrame marco = new JFrame("Demostración de FontMetrics");
12       marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14       MetricaJPanel metricaJPanel = new MetricaJPanel();
15       marco.add(metricaJPanel);
16       marco.setSize(530, 250);
17       marco.setVisible(true);
18    }
19 } // fin de la clase Metrica
```

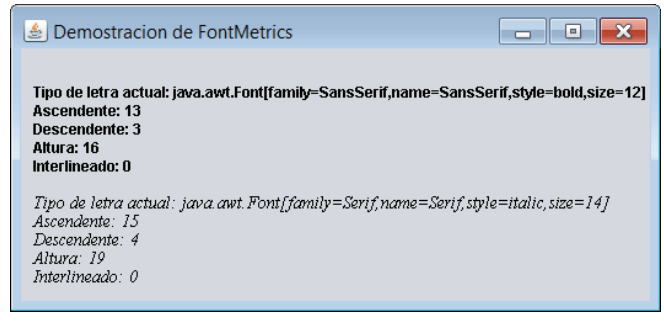

Fig. 13.16 | Mostrar la métrica de los tipos de letra.

En la línea 16 de la figura 13.15 se crea y se establece el tipo de letra actual para dibujar en `SansSerif`, negrita, de 12 puntos. En la línea 17 se utiliza el método `getFontMetrics` de `Graphics` para obtener el objeto `FontMetrics` del tipo de letra actual. En la línea 18 se imprime la representación `String` del objeto `Font` devuelto por `g.getFont()`. En las líneas 19 a 22 se utilizan los métodos de `FontMetrics` para obtener el ascendente, el descendente, la altura y el interlineado del tipo de letra.

En la línea 24 se crea un nuevo tipo de letra `Serif`, cursiva, de 14 puntos. En la línea 25 se utiliza una segunda versión del método `getFontMetrics` de `Graphics`, la cual recibe un argumento `Font` y devuelve su correspondiente objeto `FontMetrics`. En las líneas 28 a 31 se obtiene el ascendente, el descendente, la altura y el interlineado de ese tipo de letra. Cabe mencionar que la métrica es ligeramente distinta para cada uno de los tipos de letra.

13.5 Dibujo de líneas, rectángulos y óvalos

En esta sección presentaremos varios métodos de `Graphics` para dibujar líneas, rectángulos y óvalos. Los métodos y sus parámetros se sintetizan en la figura 13.17. Para cada método de dibujo que requiere un parámetro `anchura` y otro parámetro `altura`, sus valores deben ser números no negativos. De lo contrario, no se mostrará la figura.

Método	Descripción
`public void drawLine(int x1, int y1, int x2, int y2)`	Dibuja una línea entre el punto (`x1`, `y1`) y el punto (`x2`, `y2`).
`public void drawRect(int x, int y, int anchura, int altura)`	Dibuja un rectángulo con la `anchura` y `altura` especificadas. La esquina superior izquierda del rectángulo tiene las coordenadas (`x`, `y`). Sólo el contorno del rectángulo se dibuja usando el color del objeto `Graphics`; el cuerpo del rectángulo no se rellena con este color.
`public void fillRect(int x, int y, int anchura, int altura)`	Dibuja un rectángulo relleno con la `anchura` y `altura` especificadas. La esquina superior izquierda del rectángulo tiene las coordenadas (`x`, `y`).
`public void clearRect(int x, int y, int anchura, int altura)`	Dibuja un rectángulo relleno con la `anchura` y `altura` especificadas, en el color de fondo actual. La *esquina superior* izquierda del rectángulo tiene las coordenadas (`x`, `y`). Este método es útil si el programador desea eliminar una porción de una imagen.
`public void drawRoundRect(int x, int y, int anchura, int altura, int anchuraArco, int alturaArco)`	Dibuja un rectángulo con esquinas redondeadas, en el color actual y con la `anchura` y `altura` especificadas. Los valores de `anchuraArco` y `alturaArco` determinan el grado de redondez de las esquinas (vea la figura 13.20). Sólo se dibuja el contorno de la figura.
`public void fillRoundRect(int x, int y, int anchura, int altura, int anchuraArco, int alturaArco)`	Dibuja un rectángulo relleno con esquinas redondeadas, en el color actual y con la `anchura` y `altura` especificadas. Los valores de `anchuraArco` y `alturaArco` determinan el grado de redondez de las esquinas (vea la figura 13.20).
`public void draw3DRect(int x, int y, int anchura, int altura, boolean b)`	Dibuja un rectángulo tridimensional en el color actual, con la `anchura` y `altura` especificadas. La esquina superior izquierda del rectángulo tiene las coordenadas (`x`, `y`). El rectángulo aparece con relieve cuando `b` es `true` y sin relieve cuando `b` es `false`. Sólo se dibuja el contorno de la figura.
`public void fill3DRect(int x, int y, int anchura, int altura, boolean b)`	Dibuja un rectángulo tridimensional relleno en el color actual, con la `anchura` y `altura` especificadas. La esquina superior izquierda del rectángulo tiene las coordenadas (`x`, `y`). El rectángulo aparece con relieve cuando `b` es `true` y sin relieve cuando `b` es `false`.
`public void drawOval(int x, int y, int anchura, int altura)`	Dibuja un óvalo en el color actual, con la `anchura` y `altura` especificadas. La esquina *superior izquierda* del rectángulo imaginario que lo rodea tiene las coordenadas (`x`, `y`). El óvalo toca el centro de cada uno de los cuatro lados del rectángulo imaginario (vea la figura 13.21). Sólo se dibuja el contorno de la figura.
`public void fillOval(int x, int y, int anchura, int altura)`	Dibuja un óvalo relleno en el color actual, con la `anchura` y `altura` especificadas. La esquina *superior izquierda* del rectángulo imaginario que lo rodea tiene las coordenadas (`x`, `y`). El óvalo toca el centro de cada uno de los cuatro lados del rectángulo imaginario (vea la figura 13.21).

Fig. 13.17 | Métodos de `Graphics` para dibujar líneas, rectángulos y óvalos.

La aplicación de las figuras 13.18 y 13.19 demuestra cómo dibujar una variedad de líneas, rectángulos, rectángulos tridimensionales, rectángulos con esquinas redondeadas y óvalos. En la figura 13.18, en la línea 17 se dibuja una línea roja, en la línea 20 se dibuja un rectángulo vacío de color azul y en la línea 21 se dibuja un rectángulo relleno de color azul. Los métodos `fillRoundRect` (línea 24) y `drawRoundRect` (línea 25) dibujan rectángulos con esquinas redondeadas. Sus primeros dos argumentos especifican las coordenadas de la esquina superior izquierda del **rectángulo delimitador** (el área en la que se dibujará el rectángulo redondeado). Las coordenadas de la esquina superior izquierda *no* son el borde del rectángulo redondeado, sino las coordenadas en donde se encontraría el borde si el rectángulo tuviera esquinas cuadradas. El tercer y cuarto argumentos especifican la anchura y altura del rectángulo. Sus últimos dos argumentos determinan los diámetros horizontal y vertical del arco (es decir, la anchura y la altura del arco) que se utiliza para representar las esquinas.

En la figura 13.20 se muestran la anchura y altura del arco, junto con la anchura y la altura de un rectángulo redondeado. Si se utiliza el mismo valor para la anchura y la altura del arco, se produce

```
1  // Fig. 13.18: LineasRectsOvalosJPanel.java
2  // Dibujo de líneas, rectángulos y óvalos.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import javax.swing.JPanel;
6
7  public class LineasRectsOvalosJPanel extends JPanel
8  {
9     // muestra varias líneas, rectángulos y óvalos
10    @Override
11    public void paintComponent(Graphics g)
12    {
13       super.paintComponent(g);
14       this.setBackground(Color.WHITE);
15
16       g.setColor(Color.RED);
17       g.drawLine(5, 30, 380, 30);
18
19       g.setColor(Color.BLUE);
20       g.drawRect(5, 40, 90, 55);
21       g.fillRect(100, 40, 90, 55);
22
23       g.setColor(Color.CYAN);
24       g.fillRoundRect(195, 40, 90, 55, 50, 50);
25       g.drawRoundRect(290, 40, 90, 55, 20, 20);
26
27       g.setColor(Color.GREEN);
28       g.draw3DRect(5, 100, 90, 55, true);
29       g.fill3DRect(100, 100, 90, 55, false);
30
31       g.setColor(Color.MAGENTA);
32       g.drawOval(195, 100, 90, 55);
33       g.fillOval(290, 100, 90, 55);
34    }
35 } // fin de la clase LineasRectsOvalosJPanel
```

Fig. 13.18 | Dibujo de líneas, rectángulos y óvalos.

```
1  // Fig. 13.19: LineasRectsOvalos.java
2  // Prueba de LineasRectsOvalosJPanel.
3  import java.awt.Color;
4  import javax.swing.JFrame;
5
6  public class LineasRectsOvalos
7  {
8     // ejecuta la aplicación
9     public static void main(String[] args)
10    {
11       // crea marco para LineasRectsOvalosJPanel
12       JFrame marco =
13          new JFrame("Dibujo de lineas, rectangulos y ovalos");
14       marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15
16       LineasRectsOvalosJPanel lineasRectsOvalosJPanel =
17          new LineasRectsOvalosJPanel();
18       lineasRectsOvalosJPanel.setBackground(Color.WHITE);
19       marco.add(lineasRectsOvalosJPanel);
20       marco.setSize(400, 210);
21       marco.setVisible(true);
22    }
23 } // fin de la clase LineasRectsOvalos
```

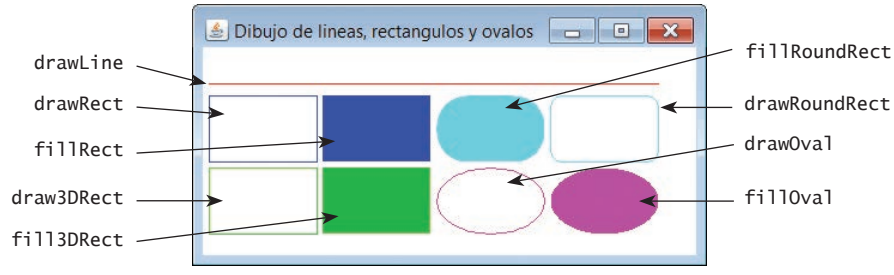

Fig. 13.19 | Prueba de `LineasRectsOvalosJPanel`.

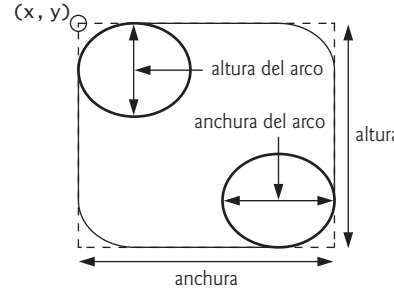

Fig. 13.20 | Anchura y altura del arco para los rectángulos redondeados.

un cuarto de círculo en cada esquina. Cuando la anchura y la altura del arco, así como la anchura y la altura del rectángulo tienen los mismos valores, el resultado es un círculo. Si los valores para `anchura` y `altura` son los mismos y los valores de `anchuraArco` y `alturaArco` son 0, el resultado es un cuadrado.

Los métodos **`draw3DRect`** (figura 13.18, línea 28) y **`fill3DRect`** (línea 29) reciben los mismos argumentos. Los primeros dos argumentos especifican la esquina *superior izquierda* del rectángulo. Los siguientes dos argumentos especifican la anchura y altura del rectángulo, respectivamente. El último argumento determina si el rectángulo está **con relieve** (`true`) o **sin relieve** (`false`). El efecto tridimensional de `draw3DRect` aparece como dos bordes del rectángulo en el color original y dos bordes en un color ligeramente más oscuro. El efecto tridimensional de `fill3DRect` aparece como dos bordes del rectángulo en el color del dibujo original y los otros dos bordes y el relleno en un color ligeramente más oscuro. Los rectángulos con relieve tienen los bordes de color original del dibujo en las partes superior e izquierda del rectángulo. Los rectángulos sin relieve tienen los bordes de color original del dibujo en las partes inferior y derecha del rectángulo. El efecto tridimensional es difícil de ver en ciertos colores.

Los métodos **`drawOval`** y **`fillOval`** (líneas 32 y 33) reciben los mismos cuatro argumentos. Los primeros dos argumentos especifican la coordenada superior izquierda del rectángulo delimitador que contiene el óvalo. Los últimos dos argumentos especifican la anchura y la altura del rectángulo delimitador, respectivamente. En la figura 13.21 se muestra un óvalo delimitado por un rectángulo. Observe que el óvalo toca el *centro* de los cuatro lados del rectángulo delimitador. (El rectángulo delimitador *no* se muestra en la pantalla).

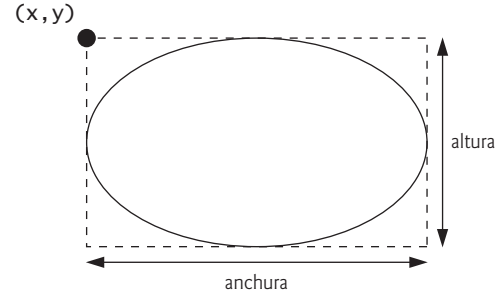

Fig. 13.21 | Óvalo delimitado por un rectángulo.

13.6 Dibujo de arcos

Un **arco** se dibuja como una porción de un óvalo. Los ángulos de los arcos se miden en grados. Los arcos se **extienden** (es decir, se mueven a lo largo de una curva) desde un **ángulo inicial**, con base en el número de grados especificados por el **ángulo del arco**. El ángulo inicial indica, en grados, dónde empieza el arco. El ángulo del arco especifica el número total de grados hasta los que se va a extender el arco. En la figura 13.22 se muestran dos arcos. El conjunto izquierdo de ejes muestra a un arco extendiéndose desde cero hasta aproximadamente 110 grados. Los arcos que se extienden en dirección *contraria a las manecillas del reloj* se miden en **grados positivos**. El conjunto derecho de ejes muestra a un arco extendiéndose desde cero hasta aproximadamente −110 grados. Los arcos que se extienden en dirección *a favor de las manecillas del reloj* se miden en **grados negativos**. Observe los cuadros punteados alrededor de los arcos en la figura 13.22. Cuando dibujamos un arco, debemos especificar un rectángulo delimitador para un óvalo. El arco se extenderá a lo largo de una parte del óvalo. Los métodos **`drawArc`** y **`fillArc`** de `Graphics` para dibujar arcos se sintetizan en la figura 13.23.

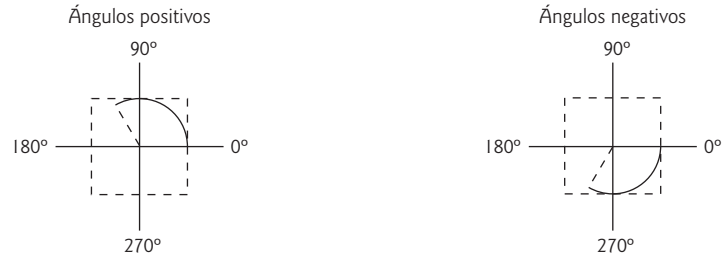

Fig. 13.22 | Ángulos positivos y negativos de un arco.

Método	Descripción
`public void drawArc(int x, int y, int anchura, int altura, int anguloInicial, int anguloArco)`	Dibuja un arco relativo a las coordenadas (x, y) de la esquina superior izquierda del rectángulo delimitador, con la `anchura` y `altura` especificadas. El segmento del arco se dibuja empezando en `anguloInicial` y se extiende hasta los grados especificados por `anguloArco`.
`public void fillArc(int x, int y, int anchura, int altura, int anguloInicial, int anguloArco)`	Dibuja un arco relleno (es decir, un sector) relativo a las coordenadas (x, y) de la esquina superior izquierda del rectángulo delimitador, con la `anchura` y `altura` especificadas. El segmento del arco se dibuja empezando en `anguloInicial` y se extiende hasta los grados especificados por `anguloArco`.

Fig. 13.23 | Métodos de `Graphics` para dibujar arcos.

Las figuras 13.24 y 13.25 demuestran el uso de los métodos para arcos de la figura 13.23. La aplicación dibuja seis arcos (tres sin rellenar y tres rellenos). Para ilustrar el rectángulo delimitador que ayuda a determinar en dónde aparece el arco, los primeros tres arcos se muestran dentro de un rectángulo rojo que tiene los mismos argumentos `x`, `y`, `anchura` y `altura` que los arcos.

```
1  // Fig. 13.24: ArcosJPanel.java
2  // Dibujo de arcos con drawArc y fillArc.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import javax.swing.JPanel;
6
7  public class ArcosJPanel extends JPanel
8  {
9     // dibuja rectángulos y arcos
10    @Override
11    public void paintComponent(Graphics g)
12    {
13       super.paintComponent(g);
14
```

Fig. 13.24 | Arcos mostrados con `drawArc` y `fillArc` (parte 1 de 2).

```
15         // empieza en 0 y se extiende hasta 360 grados
16         g.setColor(Color.RED);
17         g.drawRect(15, 35, 80, 80);
18         g.setColor(Color.BLACK);
19         g.drawArc(15, 35, 80, 80, 0, 360);
20
21         // empieza en 0 y se extiende hasta 110
22         g.setColor(Color.RED);
23         g.drawRect(100, 35, 80, 80);
24         g.setColor(Color.BLACK);
25         g.drawArc(100, 35, 80, 80, 0, 110);
26
27         // empieza en 0 y se extiende hasta -270 grados
28         g.setColor(Color.RED);
29         g.drawRect(185, 35, 80, 80);
30         g.setColor(Color.BLACK);
31         g.drawArc(185, 35, 80, 80, 0, -270);
32
33         // empieza en 0 y se extiende hasta 360 grados
34         g.fillArc(15, 120, 80, 40, 0, 360);
35
36         // empieza en 270 y se extiende hasta -90 grados
37         g.fillArc(100, 120, 80, 40, 270, -90);
38
39         // empieza en 0 y se extiende hasta -270 grados
40         g.fillArc(185, 120, 80, 40, 0, -270);
41      }
42   } // fin de la clase ArcosJPanel
```

Fig. 13.24 | Arcos mostrados con `drawArc` y `fillArc` (parte 2 de 2).

```
1  // Fig. 13.25: DibujarArcos.java
2  // Dibujo de arcos.
3  import javax.swing.JFrame;
4
5  public class DibujarArcos
6  {
7     // ejecuta la aplicación
8     public static void main(String[] args)
9     {
10       // crea marco para ArcosJPanel
11       JFrame marco = new JFrame("Dibujo de arcos");
12       marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14       ArcosJPanel arcosJPanel = new ArcosJPanel();
15       marco.add(arcosJPanel);
16       marco.setSize(300, 210);
17       marco.setVisible(true);
18    }
19 } // fin de la clase DibujarArcos
```

Fig. 13.25 | Dibujo de arcos (parte 1 de 2).

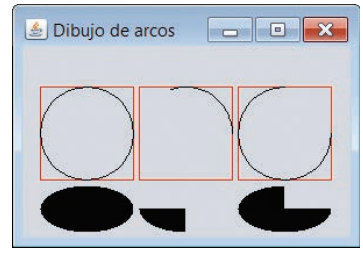

Fig. 13.25 | Dibujo de arcos (parte 2 de 2).

13.7 Dibujo de polígonos y polilíneas

Los **polígonos** son *figuras cerradas de varios lados*, compuestas por segmentos de línea recta. Las **polilíneas** son una *secuencia de puntos conectados*. En la figura 13.26 describimos los métodos para dibujar polígonos y polilíneas. Algunos métodos requieren un objeto `Polygon` (paquete `java.awt`). Los constructores de la clase `Polygon` se describen también en la figura 13.26. La aplicación de las figuras 13.27 y 13.28 dibuja polígonos y polilíneas.

Método	Descripción
Métodos de `Graphics` para dibujar polígonos	
`public void drawPolygon(int[] puntosX, int[] puntosY, int puntos)`	Dibuja un polígono. La coordenada *x* de cada punto se especifica en el arreglo `puntosX` y la coordenada *y* de cada punto se especifica en el arreglo `puntosY`. El último argumento especifica el número de `puntos`. Este método dibuja un *polígono cerrado*. Si el último punto es distinto del primero, el polígono se *cierra* mediante una línea que conecte el último punto con el primero.
`public void drawPolyline(int[] puntosX, int[] puntosY, int puntos)`	Dibuja una secuencia de líneas conectadas. La coordenada *x* de cada punto se especifica en el arreglo `puntosX` y la coordenada *y* de cada punto se especifica en el arreglo `puntosY`. El último argumento especifica el número de `puntos`. Si el último punto es distinto del primero, la polilínea *no* se cierra.
`public void drawPolygon(Polygon p)`	Dibuja el polígono especificado.
`public void fillPolygon(int[] puntosX, int[] puntosY, int puntos)`	Dibuja un polígono *relleno*. La coordenada *x* de cada punto se especifica en el arreglo `puntosX` y la coordenada *y* de cada punto se especifica en el arreglo `puntosY`. El último argumento especifica el número de `puntos`. Este método dibuja un *polígono cerrado*. Si el último punto es distinto del primero, el polígono se *cierra* mediante una línea que conecte el último punto con el primero.
`public void fillPolygon(Polygon p)`	Dibuja el polígono *relleno* especificado. El polígono es *cerrado*.

Fig. 13.26 | Métodos de `Graphics` para polígonos y métodos de la clase `Polygon` (parte 1 de 2).

Método	Descripción
Constructores y métodos de `Polygon`	
`public Polygon()`	Crea un nuevo objeto polígono. Este objeto no contiene ningún punto.
`public Polygon(int[] valoresX, int[] valoresY, int numeroDePuntos)`	Crea un nuevo objeto polígono. Este objeto tiene `numeroDePuntos` lados, en donde cada punto consiste de una coordenada *x* desde `valoresX`, y una coordenada *y* desde `valoresY`.
`public void addPoint(int x, int y)`	Agrega pares de coordenadas *x* y *y* al objeto `Polygon`.

Fig. 13.26 | Métodos de `Graphics` para polígonos y métodos de la clase `Polygon` (parte 2 de 2).

```
1  // Fig. 13.27: PoligonosJPanel.java
2  // Dibujo de polígonos.
3  import java.awt.Graphics;
4  import java.awt.Polygon;
5  import javax.swing.JPanel;
6
7  public class PoligonosJPanel extends JPanel
8  {
9     // dibuja polígonos y polilíneas
10    @Override
11    public void paintComponent(Graphics g)
12    {
13       super.paintComponent(g);
14
15       // dibuja polígono con objeto Polygon
16       int[] valoresX = {20, 40, 50, 30, 20, 15};
17       int[] valoresY = {50, 50, 60, 80, 80, 60};
18       Polygon poligono1 = new Polygon(valoresX, valoresY, 6);
19       g.drawPolygon(poligono1);
20
21       // dibuja polilíneas con dos arreglos
22       int[] valoresX2 = {70, 90, 100, 80, 70, 65, 60};
23       int[] valoresY2 = {100, 100, 110, 110, 130, 110, 90};
24       g.drawPolyline(valoresX2, valoresY2, 7);
25
26       // rellena polígono con dos arreglos
27       int[] valoresX3 = {120, 140, 150, 190};
28       int[] valoresY3 = {40, 70, 80, 60};
29       g.fillPolygon(valoresX3, valoresY3, 4);
30
31       // dibuja polígono relleno con objeto Polygon
32       Polygon polígono2 = new Polygon();
33       poligono2.addPoint(165, 135);
34       poligono2.addPoint(175, 150);
35       poligono2.addPoint(270, 200);
```

Fig. 13.27 | Polígonos mostrados con `drawPolygon` y `fillPolygon` (parte 1 de 2).

```
36          poligono2.addPoint(200, 220);
37          poligono2.addPoint(130, 180);
38          g.fillPolygon(poligono2);
39       }
40    } // fin de la clase PoligonosJPanel
```

Fig. 13.27 | Polígonos mostrados con `drawPolygon` y `fillPolygon` (parte 2 de 2).

```
1  // Fig. 13.28: DibujarPoligonos.java
2  // Dibujo de polígonos.
3  import javax.swing.JFrame;
4
5  public class DibujarPoligonos
6  {
7     // ejecuta la aplicación
8     public static void main(String[] args)
9     {
10       // crea marco para objeto PoligonosJPanel
11       JFrame marco = new JFrame("Dibujo de polígonos");
12       marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14       PoligonosJPanel poligonosJPanel = new PoligonosJPanel();
15       marco.add(poligonosJPanel);
16       marco.setSize(280, 270);
17       marco.setVisible(true);
18    }
19 } // fin de la clase DibujarPoligonos
```

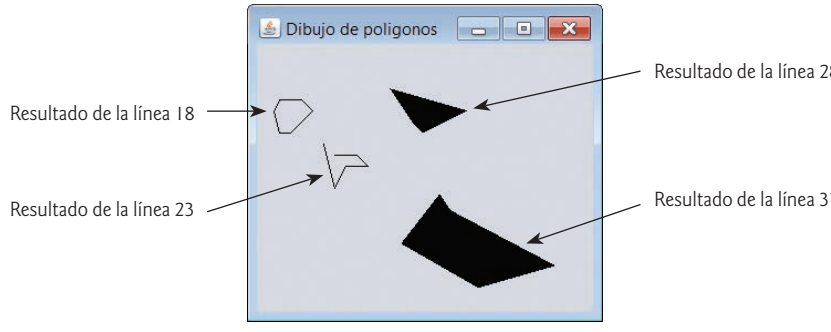

Fig. 13.28 | Dibujo de polígonos.

En las líneas 16 y 17 de la figura 13.27 se crean dos arreglos `int` y se utilizan para especificar los puntos del objeto `Polygon` llamado `poligono1`. La llamada al constructor de `Polygon` en la línea 18 recibe el arreglo `valoresX` que contiene la coordenada *x* de cada punto, el arreglo `valoresY` que contiene la coordenada *y* de cada punto y el número 6 (el número de puntos en el polígono). En la línea 19 se muestra `poligono1` al pasarlo como argumento para el método **`drawPolygon`** de `Graphics`.

En las líneas 22 y 23 se crean dos arreglos `int` y se utilizan para especificar los puntos de una serie de líneas conectadas. El arreglo `valoresX2` contiene la coordenada *x* de cada punto y el arreglo `valoresY2` contiene la coordenada *y* de cada punto. En la línea 24 se utiliza el método **`drawPolyline`** de `Graphics`

para mostrar la serie de líneas conectadas que se especifican mediante los argumentos `valoresX2`, `valoresY2` y `7` (el número de puntos).

En las líneas 27 y 28 se crean dos arreglos `int` y se utilizan para especificar los puntos de un polígono. El arreglo `valoresX3` contiene la coordenada *x* de cada punto y el arreglo `valoresY3` contiene la coordenada *y* de cada punto. En la línea 29 se muestra un polígono, al pasar al método **`fillPolygon`** de `Graphics` los dos arreglos (`valoresX3` y `valoresY3`) y el número de puntos a dibujar (4).

Error común de programación 13.1

Si el número de puntos especificados en el tercer argumento del método `drawPolygon` *o del método* `fillPolygon` *es mayor que el número de elementos en los arreglos de las coordenadas que especifican el polígono a mostrar, se lanzará una excepción* `ArrayIndexOutOfBoundsException`.

En la línea 32 se crea el objeto `Polygon` llamado `poligono2` sin puntos. En las líneas 32 a 36 se utiliza el método **`addPoint`** de `Polygon` para agregar pares de coordenadas *x* y *y* al objeto `Polygon`. En la línea 37 se muestra el objeto `Polygon` llamado `poligono2`, al pasarlo al método `fillPolygon` de `Graphics`.

13.8 La API Java 2D

La **API Java2D** proporciona herramientas avanzadas para gráficos bidimensionales para los programadores que requieren manipulaciones gráficas detalladas y complejas. La API incluye características para procesar arte lineal, texto e imágenes en los paquetes `java.awt`, `java.awt.image`, `java.awt.color`, `java.awt.font`, `java.awt.geom`, `java.awt.print` y `java.awt.image.renderable`. Las herramientas de la API son muy extensas como para cubrirlas todas en este libro. Para ver las especificaciones acerca de estas herramientas, visite `http://docs.oracle.com/javase/6/docs/technotes/guides/2d`. En esta sección veremos las generalidades de varias herramientas de Java2D.

El dibujo con la API Java2 se logra mediante el uso de una referencia **`Graphics2D`** (paquete `java.awt`) que es una *subclase abstracta* de la clase `Graphics`, por lo que tiene todas las herramientas para gráficos que se demostraron anteriormente en este capítulo. De hecho, el objeto utilizado para dibujar en todos los métodos `paintComponent` es una instancia de una *subclase* de `Graphics2D` que se pasa al método `paintComponent` y se utiliza mediante la *superclase* `Graphics`. Para acceder a las herramientas de `Graphics2D`, debemos convertir la referencia `Graphics (g)` que se pasa a `paintComponent` en una referencia `Graphics2D`, mediante una instrucción como:

```
Graphics2D g2d = (Graphics2D) g;
```

Los siguientes dos ejemplos utilizan esta técnica.

Líneas, rectángulos, rectángulos redondeados, arcos y elipses

En el siguiente ejemplo se muestran varias figuras de Java2D del paquete `java.awt.geom`, incluyendo a **`Line2D.Double`**, **`Rectangle2D.Double`**, **`RoundRectangle2D.Double`**, **`Arc2D.Double`** y **`Ellipse2D.Double`**. Observe la sintaxis de cada uno de los nombres de las clases. Cada una de estas clases representa una figura con las dimensiones especificadas como valores `double`. Hay una versión *separada* de cada figura, representada con valores `float` (como **`Ellipse2D.Float`**). En cada caso, `Double` es una clase `public static` anidada de la clase que se especifica a la izquierda del punto (por ejemplo, `Ellipse2D`). Para utilizar la clase `static` anidada, simplemente debemos calificar su nombre con el nombre de la clase externa.

En las figuras 13.29 y13.30, dibujamos figuras de Java2D y modificamos sus características de dibujo, como cambiar el grosor de una línea, rellenar figuras con patrones y dibujar líneas punteadas. Éstas son sólo algunas de las muchas herramientas que proporciona Java2D.

En la línea 25 de la figura 13.29 se convierte la referencia `Graphics` recibida por `paintComponent` a una referencia `Graphics2D` y se asigna a `g2d` para permitir el acceso a las características de Java2D.

```
1  // Fig. 13.29: FigurasJPanel.java
2  // Demostración de algunas figuras de Java 2D.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import java.awt.BasicStroke;
6  import java.awt.GradientPaint;
7  import java.awt.TexturePaint;
8  import java.awt.Rectangle;
9  import java.awt.Graphics2D;
10 import java.awt.geom.Ellipse2D;
11 import java.awt.geom.Rectangle2D;
12 import java.awt.geom.RoundRectangle2D;
13 import java.awt.geom.Arc2D;
14 import java.awt.geom.Line2D;
15 import java.awt.image.BufferedImage;
16 import javax.swing.JPanel;
17
18 public class FigurasJPanel extends JPanel
19 {
20    // dibuja figuras con la API Java 2D
21    @Override
22    public void paintComponent(Graphics g)
23    {
24       super.paintComponent(g);
25       Graphics2D g2d = (Graphics2D) g; // convierte a g en objeto Graphics2D
26
27       // dibuja una elipse en 2D, rellena con un gradiente color azul-amarillo
28       g2d.setPaint(new GradientPaint(5, 30, Color.BLUE, 35, 100,
29          Color.YELLOW, true));
30       g2d.fill(new Ellipse2D.Double(5, 30, 65, 100));
31
32       // dibuja rectángulo en 2D de color rojo
33       g2d.setPaint(Color.RED);
34       g2d.setStroke(new BasicStroke(10.0f));
35       g2d.draw(new Rectangle2D.Double(80, 30, 65, 100));
36
37       // dibuja rectángulo delimitador en 2D, con un fondo con búfer
38       BufferedImage imagenBuf = new BufferedImage(10, 10,
39          BufferedImage.TYPE_INT_RGB);
40
41       // obtiene objeto Graphics2D de imagenBuf y dibuja en él
42       Graphics2D gg = imagenBuf.createGraphics();
43       gg.setColor(Color.YELLOW);
44       gg.fillRect(0, 0, 10, 10);
45       gg.setColor(Color.BLACK);
46       gg.drawRect(1, 1, 6, 6);
47       gg.setColor(Color.BLUE);
48       gg.fillRect(1, 1, 3, 3);
49       gg.setColor(Color.RED);
```

Fig. 13.29 | Demostración de algunas figuras de Java 2D (parte 1 de 2).

```
50          gg.fillRect(4, 4, 3, 3); // dibuja un rectángulo relleno
51
52          // pinta a imagenBuf en el objeto JFrame
53          g2d.setPaint(new TexturePaint(imagenBuf,
54             new Rectangle(10, 10)));
55          g2d.fill(
56             new RoundRectangle2D.Double(155, 30, 75, 100, 50, 50));
57
58          // dibuja arco en forma de pastel en 2D, de color blanco
59          g2d.setPaint(Color.WHITE);
60          g2d.setStroke(new BasicStroke(6.0f));
61          g2d.draw(
62             new Arc2D.Double(240, 30, 75, 100, 0, 270, Arc2D.PIE));
63
64          // dibuja líneas 2D en verde y amarillo
65          g2d.setPaint(Color.GREEN);
66          g2d.draw(new Line2D.Double(395, 30, 320, 150));
67
68          // dibuja línea 2D usando el trazo
69          float[] guiones = {10}; // especifica el patrón de guiones
70          g2d.setPaint(Color.YELLOW);
71          g2d.setStroke(new BasicStroke(4, BasicStroke.CAP_ROUND,
72             BasicStroke.JOIN_ROUND, 10, guiones, 0));
73          g2d.draw(new Line2D.Double(320, 30, 395, 150));
74       }
75    } // fin de la clase FigurasJPanel
```

Fig. 13.29 | Demostración de algunas figuras de Java 2D (parte 2 de 2).

```
1  // Fig. 13.30: Figuras.java
2  // Prueba de FigurasJPanel.
3  import javax.swing.JFrame;
4
5  public class Figuras
6  {
7     // ejecuta la aplicación
8     public static void main(String[] args)
9     {
10       // crea marco para objeto FigurasJPanel
11       JFrame marco = new JFrame("Dibujo de figuras en 2D");
12       marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13
14       // crea objeto FigurasJPanel
15       FigurasJPanel figurasJPanel = new FigurasJPanel();
16
17       marco.add(figurasJPanel);
18       marco.setSize(425, 200);
19       marco.setVisible(true);
20    }
21 } // fin de la clase Figuras
```

Fig. 13.30 | Prueba de `FigurasJPanel` (parte 1 de 2).

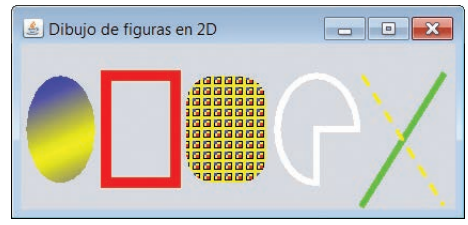

Fig. 13.30 | Prueba de `FigurasJPanel` (parte 2 de 2).

Óvalos, rellenos con degradado y objetos `Paint`

La primera figura que dibujamos es un óvalo relleno con colores que cambian gradualmente. En las líneas 28 y 29 se invoca el método **`setPaint`** de `Graphics2D` para establecer el objeto **`Paint`** que determina el color para la figura a mostrar. Un objeto `Paint` implementa a la interfaz `java.awt.Paint`. Puede ser algo tan simple como uno de los objetos `Color` que presentamos antes en la sección 13.3 (la clase `Color` implementa a Paint), o puede ser una instancia de las clases `GradientPaint`, **`SystemColor`**, `TexturePaint`, `LinearGradientPain` o `RadientGradientPaint` de la API Java2D. En este caso, utilizamos un objeto `GradientPaint`.

La clase `GradientPaint` ayuda a dibujar una figura en *colores que cambian gradualmente* (lo cual se conoce como **degradado**). El constructor de `GradientPaint` que se utiliza aquí requiere siete argumentos. Los primeros dos especifican la coordenada inicial del degradado. El tercer argumento especifica el `Color` inicial del degradado. El cuarto y quinto argumentos especifican la coordenada final del degradado. El sexto especifica el `Color` final del degradado. El último argumento especifica si el degradado es **cíclico** (`true`) o **acíclico** (`false`). Los dos conjuntos de coordenadas determinan la dirección del degradado. Como la segunda coordenada (35, 100) se encuentra hacia abajo y a la derecha de la primera coordenada (5, 30), el degradado va hacia abajo y a la derecha con cierto ángulo. Como este degradado es cíclico (`true`), el color empieza con azul, se convierte gradualmente en amarillo y luego regresa gradualmente a azul. Si el degradado es acíclico, el color cambia del primer color especificado (por ejemplo, azul) al segundo color (por ejemplo, amarillo).

En la línea 30 se utiliza el método **`fill`** de `Graphics2D` para dibujar un objeto **`Shape`** relleno (un objeto que implementa a la interfaz `Shape` del paquete `java.awt`). En este caso mostramos un objeto `Ellipse2D.Double`. El constructor de `Ellipse2D.Double` recibe cuatro argumentos que especifican el *rectángulo delimitador* para mostrar la elipse.

Rectángulos, trazos (objetos `Stroke`*)*

A continuación dibujamos un rectángulo rojo con un borde grueso. En la línea 33 se invoca a `setPaint` para establecer el objeto `Paint` en `Color.RED`. En la línea 34 se utiliza el método **`setStroke`** de `Graphics2D` para establecer las características del borde del rectángulo (o las líneas para cualquier otra figura). El método `setStroke` requiere como argumento un objeto que implemente a la interfaz **`Stroke`** (paquete `java.awt`). En este caso, utilizamos una instancia de la clase `BasicStroke`. Esta clase proporciona varios constructores para especificar la anchura de la línea, la manera en que ésta termina (lo cual se le conoce como **cofias**), la manera en que las líneas se unen entre sí (lo cual se le conoce como **uniones de línea**) y los atributos de los guiones de la línea (si es una línea punteada). El constructor aquí especifica que la línea debe tener una anchura de 10 píxeles.

En la línea 35 se utiliza el método **`draw`** de `Graphics2D` para dibujar un objeto `Shape`. En este caso se trata de una instancia de la clase `Rectangle2D.Double`. El constructor de `Rectangle2D.Double` recibe argumentos que especifican las coordenadas *x* y *y* de la esquina *superior izquierda*, así como la anchura y la altura del rectángulo.

Rectángulos redondeados, objetos `BufferedImage` y `TexturePaint`

A continuación dibujamos un rectángulo redondeado relleno con un patrón creado en un objeto **`BufferedImage`** (paquete `java.awt.image`). En las líneas 38 y 39 se crea el objeto `BufferedImage`. La clase `BufferedImage` puede usarse para producir imágenes en color y escala de grises. Este objeto `BufferedImage` en particular tiene una anchura y una altura de 10 píxeles (según lo especificado por los primeros dos argumentos del constructor). El tercer argumento del constructor, **`BufferedImage.TYPE_INT_RGB`**, indica que la imagen se almacena en color, utilizando el esquema de colores RGB.

Para crear el patrón de relleno para el rectángulo redondeado, debemos primero dibujar en el objeto `BufferedImage`. En la línea 42 se crea un objeto `Graphics2D` (con una llamada al método **`createGraphics`** de `BufferedImage`) que puede usarse para dibujar en el objeto `BufferedImage`. En las líneas 43 a 50 se utilizan los métodos `setColor`, `fillRect` y `drawRect` para crear el patrón.

En las líneas 53 y 54 se establece el objeto `Paint` en un nuevo objeto `TexturePaint` (paquete `java.awt`). Un objeto `TexturePaint` utiliza la imagen almacenada en su objeto `BufferedImage` asociado (el primer argumento del constructor) como la textura para rellenar una figura. El segundo argumento especifica el área `Rectangle` del objeto `BufferedImage` que se repetirá en toda la textura. En este caso, el objeto `Rectangle` es del mismo tamaño que el objeto `BufferedImage`. Sin embargo, puede utilizarse una porción más pequeña del objeto `BufferedImage`.

En las líneas 55 y 56 se utiliza el método `fill` de `Graphics2D` para dibujar un objeto `Shape` relleno; en este caso, es una instancia de la clase `RoundRectangle2D.Double`. El constructor de la clase `RoundRectangle2D.Double` recibe seis argumentos que especifican las dimensiones del rectángulo, así como la anchura y la altura del arco utilizado para redondear las esquinas.

Arcos

A continuación dibujamos un arco en forma de pastel con una línea blanca gruesa. En la línea 59 se establece el objeto `Paint` en `Color.WHITE`. En la línea 60 se establece el objeto `Stroke` en un nuevo objeto `BasicStroke` para una línea con 6 píxeles de anchura. En las líneas 61 y 62 se utiliza el método `draw` de `Graphics2D` para dibujar un objeto `Shape`; en este caso, un `Arc2D.Double`. Los primeros cuatro argumentos del constructor de `Arc2D.Double` especifican las coordenadas *x* y *y* de la esquina superior izquierda, así como la anchura y la altura del rectángulo delimitador para el arco. El quinto argumento especifica el ángulo inicial. El sexto especifica el ángulo del arco. El último argumento especifica cómo se *cierra* el arco. La constante **`Arc2D.PIE`** indica que el arco se *cierra* dibujando dos líneas: una que va desde el punto inicial del arco hasta el centro del rectángulo delimitador, y otra que va desde el centro del rectángulo delimitador hasta el punto final. La clase `Arc2D` proporciona otras dos constantes estáticas para especificar cómo se *cierra* el arco. La constante **`Arc2D.CHORD`** dibuja una línea que va desde el punto inicial hasta el punto final. La constante **`Arc2D.OPEN`** especifica que el arco *no* debe *cerrarse*.

Líneas

Por último, dibujamos dos líneas utilizando objetos **`Line2D`**: una sólida y una punteada. En la línea 65 se establece el objeto `Paint` en `Color.GREEN`. En la línea 66 se utiliza el método `draw` de `Graphics2D` para dibujar un objeto `Shape`; en este caso, una instancia de la clase `Line2D.Double`. Los argumentos del constructor de `Line2D.Double` especifican las coordenadas inicial y final de la línea.

En la línea 69 se declara un arreglo `float` de un elemento, el cual contiene el valor 10. Este arreglo describe los guiones en la línea punteada. En este caso, cada guion tendrá 10 píxeles de largo. Para crear guiones de diferentes longitudes en un patrón, simplemente debe proporcionar la longitud de cada guion como un elemento en el arreglo. En la línea 70 se establece el objeto `Paint` en `Color.YELLOW`. En las líneas 71 y 72 se establece el objeto `Stroke` en un nuevo objeto `BasicStroke`. La línea tendrá una anchura de 4 píxeles y extremos redondeados (**`BasicStroke.CAP_ROUND`**). Si las líneas se unen entre sí (como en un rectángulo en las esquinas), la unión de las líneas será redondeada (**`BasicStroke.`**

`JOIN_ROUND`). El argumento `guiones` especifica las longitudes de los guiones de la línea. El último argumento indica el índice inicial en el arreglo `guiones` para el primer guion en el patrón. En la línea 73 se dibuja una línea con el objeto `Stroke` actual.

Creación de sus propias figuras mediante las rutas generales

A continuación presentamos una **ruta general**, que es una figura compuesta de líneas rectas y curvas complejas. Una ruta general se representa con un objeto de la clase `GeneralPath` (paquete `java.awt.geom`). La aplicación de las figuras 13.31 y 13.32 demuestra cómo dibujar una ruta general, en forma de una estrella de cinco puntas.

```
1  // Fig. 13.31: Figuras2JPanel.java
2  // Demostración de una ruta general.
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import java.awt.Graphics2D;
6  import java.awt.geom.GeneralPath;
7  import java.security.SecureRandom;
8  import javax.swing.JPanel;
9
10 public class Figuras2JPanel extends JPanel
11 {
12    // dibuja rutas generales
13    @Override
14    public void paintComponent(Graphics g)
15    {
16       super.paintComponent(g);
17       SecureRandom aleatorio = new SecureRandom();
18
19       int[] puntosX = {55, 67, 109, 73, 83, 55, 27, 37, 1, 43};
20       int[] puntosY = {0, 36, 36, 54, 96, 72, 96, 54, 36, 36};
21
22       Graphics2D g2d = (Graphics2D) g;
23       GeneralPath estrella = new GeneralPath();
24
25       // establece la coordenada inicial de la ruta general
26       estrella.moveTo(puntosX[0], puntosY[0]);
27
28       // crea la estrella; esto no la dibuja
29       for (int cuenta = 1; cuenta < puntosX.length; cuenta++)
30          estrella.lineTo(puntosX[cuenta], puntosY[cuenta]);
31
32       estrella.closePath(); // cierra la figura
33
34       g2d.translate(200, 200); // traslada el origen a (200, 200)
35
36       // gira alrededor del origen y dibuja estrellas en colores aleatorios
37       for (int cuenta = 1; cuenta <= 20; cuenta++)
38       {
39          g2d.rotate(Math.PI / 10.0); // gira el sistema de coordenadas
40
```

Fig. 13.31 | Rutas generales de Java 2D (parte 1 de 2).

```
41            // establece el color de dibujo al azar
42            g2d.setColor(new Color(aleatorio.nextInt(256),
43               aleatorio.nextInt(256), aleatorio.nextInt(256)));
44
45            g2d.fill(estrella); // dibuja estrella rellena
46         }
47      }
48   } // fin de la clase Figuras2JPanel
```

Fig. 13.31 | Rutas generales de Java 2D (parte 2 de 2).

```
1  // Fig. 13.32: Figuras2.java
2  // Demostración de una ruta general.
3  import java.awt.Color;
4  import javax.swing.JFrame;
5
6  public class Figuras2
7  {
8     // ejecuta la aplicación
9     public static void main(String[] args)
10    {
11       // crea marco para Figuras2JPanel
12       JFrame marco = new JFrame("Dibujo de figuras en 2D");
13       marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14
15       Figuras2JPanel figuras2JPanel = new Figuras2JPanel();
16       marco.add(figuras2JPanel);
17       marco.setBackground(Color.WHITE);
18       marco.setSize(400, 400);
19       marco.setVisible(true);
20    }
21 } // fin de la clase Figuras2
```

Fig. 13.32 | Demostración de una ruta general.

En las líneas 19 y 20 (figura 13.31) se declaran dos arreglos `int` que representan las coordenadas *x* y *y* de los puntos en la estrella. En la línea 23 se crea el objeto `GeneralPath` llamado `estrella`. En la línea 26 se utiliza el método **`moveTo`** de `GeneralPath` para especificar el primer punto en la `estrella`. La instrucción

`for` de las líneas 29 y 30 utiliza el método **`lineTo`** de `GeneralPath` para dibujar una línea al siguiente punto en la `estrella`. Cada nueva llamada a `lineTo` dibuja una línea del punto anterior al punto actual. En la línea 32 se utiliza el método **`closePath`** de `GeneralPath` para dibujar una línea del último punto hasta el punto especificado en la última llamada a `moveTo`. Esto completa la ruta general.

En la línea 34 se utiliza el método **`translate`** de `Graphics2D` para desplazar el origen del dibujo hasta la ubicación (200, 200). Todas las operaciones de dibujo utilizarán ahora la ubicación (200, 200) como si fuera (0, 0).

La instrucción `for` de las líneas 37 a 46 dibujan la estrella 20 veces, girándola alrededor del nuevo punto del origen. En la línea 39 se utiliza el método **`rotate`** de `Graphics2D` para girar la siguiente figura a mostrar. El argumento especifica el ángulo de giro en radianes (con $360° = 2\pi$ radianes). En la línea 45 se utiliza el método `fill` de `Graphics2D` para dibujar una versión rellena de la estrella.

13.9 Conclusión

En este capítulo aprendió a utilizar las herramientas de gráficos de Java para producir dibujos a colores. Aprendió a especificar la ubicación de un objeto usando el sistema de coordenadas de Java y a dibujar en una ventana usando el método `paintComponent`. Vio una introducción a la clase `Color` y aprendió a utilizar esta clase para especificar distintos colores usando sus componentes RGB. Utilizó el cuadro de diálogo `JColorChooser` para permitir a los usuarios seleccionar colores en un programa. Después aprendió a trabajar con los tipos de letra al dibujar texto en una ventana. Aprendió a crear un objeto `Font` a partir de un nombre, estilo y tamaño de tipo de letra, así como acceder a la métrica de un tipo de letra. De ahí, aprendió a dibujar varias figuras en una ventana, como rectángulos (regulares, redondeados y en 3D), óvalos y polígonos, así como líneas y arcos. Después utilizó la API Java 2D para crear figuras más complejas y rellenarlas con degradados o patrones. El capítulo concluyó con una explicación sobre las rutas generales, que se utilizan para construir figuras a partir de líneas rectas y curvas complejas. En el siguiente capítulo analizaremos la clase `String` y sus métodos. Presentaremos las expresiones regulares para asociar patrones en cadenas de texto, y demostraremos cómo validar la entrada del usuario mediante expresiones regulares.

Resumen

Sección 13.1 Introducción

- El sistema de coordenadas de Java (pág. 556) es un esquema para identificar todos los posibles puntos en la pantalla (pág. 567).
- Un par de coordenadas (pág. 556) está compuesto de una coordenada *x* (horizontal) y una coordenada *y* (vertical).
- Las coordenadas se utilizan para indicar en dónde deben mostrarse los gráficos en una pantalla.
- Las unidades de las coordenadas se miden en píxeles (pág. 556). Un píxel es la unidad más pequeña de resolución de un monitor de computadora.

Sección 13.2 Contextos y objetos de gráficos

- Un contexto de gráficos en Java (pág. 558) permite dibujar en la pantalla.
- La clase `Graphics` (pág. 558) contiene métodos para dibujar cadenas, líneas, rectángulos y otras figuras. También se incluyen métodos para manipular tipos de letra y colores.
- Un objeto `Graphics` administra un contexto de gráficos y dibuja píxeles en la pantalla, los cuales representan a otros objetos gráficos como líneas, elipses, rectángulos y otros polígonos (pág. 558).
- La clase `Graphics` es una clase `abstract`. Cada implementación de Java tiene una subclase de `Graphics`, la cual proporciona herramientas de dibujo. Esta implementación se oculta de nosotros mediante la clase `Graphics`, la cual proporciona la interfaz que nos permite utilizar los gráficos en forma independiente de la plataforma.

- El método `paintComponent` puede usarse para dibujar gráficos en cualquier componente `JComponent`.
- El método `paintComponent` recibe como argumento un objeto `Graphics` que el sistema pasa al método cuando un componente ligero de Swing necesita volver a pintarse.
- Cuando se ejecuta una aplicación, el contenedor de ésta llama al método `paintComponent`. Para que `paintComponent` sea llamado otra vez debe ocurrir un evento.
- Cuando se muestra un objeto `JComponent`, se hace una llamada a su método `paintComponent`.
- Al llamar al método `repaint` (pág. 559) en un componente, se actualizan los gráficos que se dibujan en ese componente.

Sección 13.3 Control de colores

- La clase `Color` (pág. 559) declara métodos y constantes para manipular los colores en un programa de Java.
- Todo color se crea a partir de un componente rojo, verde y azul. En conjunto estos componentes se llaman valores RGB (pág. 560). Los componentes RGB especifican la cantidad de rojo, verde y azul en un color, respectivamente. Entre mayor sea el valor RGB, mayor será la cantidad de ese color específico.
- Los métodos `getRed`, `getGreen` y `getBlue` de `Color` (pág. 560) devuelven valores enteros de 0 a 255, los cuales representan la cantidad de rojo, verde y azul, respectivamente.
- El método `getColor` de `Graphics` (pág. 560) devuelve un objeto `Color` que representa el color actual para dibujar.
- El método `setColor` de `Graphics` (pág. 560) establece el color actual para dibujar.
- El método `fillRect` de `Graphics` (pág. 560) dibuja un rectángulo relleno por el color actual del objeto `Graphics`.
- El método `drawString` de `Graphics` (pág. 562) dibuja un objeto `String` en el color actual.
- El componente de GUI `JColorChooser` (pág. 563) permite a los usuarios de una aplicación seleccionar colores.
- El método `static showDialog` de `JColorChooser` (pág. 564) muestra un cuadro de diálogo modal `JColorChooser`.

Sección 13.4 Manipulación de tipos de letra

- La clase `Font` (pág. 566) contiene métodos y constantes para manipular tipos de letra.
- El constructor de la clase `Font` recibe tres argumentos: el nombre (pág. 567), el estilo y el tamaño del tipo de letra.
- El estilo de tipo de letra de un objeto `Font` puede ser `Font.PLAIN`, `Font.ITALIC` o `Font.BOLD` (cada uno es un campo `static` de la clase `Font`). Los estilos de tipos de letra pueden usarse combinados (por ejemplo, `Font.ITALIC + Font.BOLD`).
- El tamaño de un tipo de letra se mide en puntos. Un punto es 1/72 de una pulgada.
- El método `setFont` de `Graphics` (pág. 567) establece el tipo de letra para dibujar el texto que se va a mostrar.
- El método `getSize` de `Font` (pág. 567) devuelve el tamaño del tipo de letra, en puntos.
- El método `getName` de `Font` (pág. 567) devuelve el nombre del tipo de letra actual, como una cadena.
- El método `getStyle` de `Font` (pág. 569) devuelve un valor entero que representa el estilo actual del objeto `Font`.
- El método `getFamily` de `Font` (pág. 569) devuelve el nombre de la familia a la que pertenece el tipo de letra actual. El nombre de la familia del tipo de letra es específico de cada plataforma.
- La clase `FontMetrics` (pág. 569) contiene métodos para obtener información sobre los tipos de letra.
- La métrica de tipos de letra (pág. 569) incluye la altura, el descendente y el ascendente.

Sección 13.5 Dibujo de líneas, rectángulos y óvalos

- Los métodos `fillRoundRect` (pág. 573) y `drawRoundRect` (pág. 573) de `Graphics` dibujan rectángulos con esquinas redondeadas.
- Los métodos `draw3DRect` (pág. 575) y `fill3DRect` (pág. 575) de `Graphics` dibujan rectángulos tridimensionales.
- Los métodos `drawOval` (pág. 575) y `fillOval` (pág. 575) de `Graphics` dibujan óvalos.

Sección 13.6 Dibujo de arcos

- Un arco (pág. 575) se dibuja como una porción de un óvalo.
- Los arcos se extienden desde un ángulo inicial, según el número de grados especificados por el ángulo del arco (pág. 575).
- Los métodos `drawArc` (pág. 575) y `fillArc` (pág. 575) de `Graphics` se utilizan para dibujar arcos.

Sección 13.7 Dibujo de polígonos y polilíneas

- La clase `Polygon` contiene métodos para crear polígonos.
- Los polígonos son figuras cerradas con varios lados compuestas de segmentos de línea recta.
- Las polilíneas (pág. 578) son una secuencia de puntos conectados.
- El método `drawPolyline` de `Graphics` (pág. 580) muestra una serie de líneas conectadas.
- Los métodos `drawPolygon` (pág. 580) y `fillPolygon` (pág. 581) de `Graphics` se utilizan para dibujar polígonos.
- El método `addPoint` (pág. 581) de la clase `Polygon` agrega pares de coordenadas *x* y *y* a un objeto `Polygon`.

Sección 13.8 La API Java 2D

- La API Java 2D (pág. 581) proporciona herramientas avanzadas de gráficos bidimensionales.
- La clase `Graphics2D` (pág. 581), una subclase de `Graphics`, se utiliza para dibujar con la API Java 2D.
- La API Java 2D contiene varias clases para dibujar figuras, incluyendo `Line2D.Double`, `Rectangle2D.Double`, `Rectangle2D.Double`, `Arc2D.Double` y `Ellipse2D.Double` (pág. 581).
- La clase `GradientPaint` (pág. 584) ayuda a dibujar una figura en colores que cambian en forma gradual; a esto se le conoce como degradado (pág. 584).
- El método `fill` de `Graphics2D` (pág. 584) dibuja un objeto relleno de cualquier tipo que implemente a la interfaz `Shape` (pág. 584).
- La clase `BasicStroke` (pág. 584) ayuda a especificar las características de dibujo de líneas.
- El método `draw` de `Graphics2D` (pág. 584) se utiliza para dibujar un objeto `Shape`.
- Las clases `GradientPaint` (pág. 584) y `TexturePaint` (pág. 585) ayudan a especificar las características para rellenar figuras con colores o patrones.
- Una ruta general (pág. 586) es una figura construida a partir de líneas rectas y curvas complejas; se representa mediante un objeto de la clase `GeneralPath` (pág. 586).
- El método `moveTo` de `GeneralPath` (pág. 587) especifica el primer punto en una ruta general.
- El método `lineTo` de `GeneralPath` (pág. 588) dibuja una línea hasta el siguiente punto en la ruta. Cada nueva llamada a `lineTo` dibuja una línea desde el punto anterior hasta el punto actual.
- El método `closePath` de `GeneralPath` (pág. 588) dibuja una línea desde el último punto hasta el punto especificado en la última llamada a `moveTo`. Esto completa la ruta general.
- El método `translate` de `Graphics2D` (pág. 588) se utiliza para mover el origen de dibujo hasta una nueva ubicación.
- El método `rotate` de `Graphics2D` (pág. 588) se utiliza para girar la siguiente figura a mostrar.

Ejercicios de autoevaluación

13.1 Complete las siguientes oraciones:

a) En Java2D, el método _____ de la clase _____ establece las características de una línea utilizada para dibujar una figura.

b) La clase _____ ayuda a especificar el relleno para una figura, de tal forma que el relleno cambie gradualmente de un color a otro.

c) El método _____ de la clase `Graphics` dibuja una línea entre dos puntos.

d) RGB son las iniciales en inglés de _____, _____ y _____.

e) Los tamaños de los tipos de letra se miden en unidades llamadas _____.
f) La clase _____ ayuda a especificar el relleno para una figura, utilizando un patrón dibujado en un objeto `BufferedImage`.

13.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) Los primeros dos argumentos del método `drawOval` de `Graphics` especifican la coordenada central del óvalo.
b) En el sistema de coordenadas de Java, las coordenadas *x* se incrementan de izquierda a derecha y las coordenadas *y* de arriba hacia abajo.
c) El método `fillPolygon` de `Graphics` dibuja un polígono sólido en el color actual.
d) El método `drawArc` de `Graphics` permite ángulos negativos.
e) El método `getSize` de `Graphics` devuelve el tamaño del tipo de letra actual, en centímetros.
f) La coordenada de píxel (0, 0) se encuentra exactamente en el centro del monitor.

13.3 Encuentre los errores en cada una de las siguientes instrucciones, y explique cómo corregirlos. Suponga que `g` es un objeto `Graphics`.
a)
```
g.setFont("SansSerif");
```
b)
```
g.erase(x, y, w, h);    // borrar rectángulo en (x, y)
```
c)
```
Font f = new Font("Serif", Font.BOLDITALIC, 12);
```
d)
```
g.setColor(255, 255, 0);  // cambiar color a amarillo
```

Respuestas a los ejercicios de autoevaluación

13.1 a) `setStroke`, `Graphics2D`. b) `GradientPath`. c) `drawLine`. d) rojo, verde, azul. e) puntos. f) `TexturePaint`.

13.2 a) Falso. Los primeros dos argumentos especifican la esquina superior izquierda del rectángulo delimitador.
b) Verdadero.
c) Verdadero.
d) Verdadero.
e) Falso. Los tamaños de los tipos de letra se miden en puntos.
f) Falso. La coordenada (0,0) corresponde a la esquina superior izquierda de un componente de la GUI, en el cual ocurre el dibujo.

13.3 a) El método `setFont` toma un objeto `Font` como argumento, no un `String`.
b) La clase `Graphics` no tiene un método erase. Debe utilizarse el método `clearRect`.
c) `Font.BOLDITALIC` no es un estilo de tipo de letra válido. Para obtener un tipo de letra en cursiva y negrita, use `Font.BOLD + Font.ITALIC`.
d) El método `setColor` toma un objeto `Color` como argumento, no tres enteros.

Ejercicios

13.4 Complete las siguientes oraciones:
a) La clase _____ de la API Java2D se utiliza para dibujar óvalos.
b) Los métodos `draw` y `fill` de la clase `Graphics2D` requieren un objeto de tipo _____ como su argumento.
c) Las tres constantes que especifican el estilo de los tipos de letra son _____, _____ y _____.
d) El método _____ de `Graphics2D` establece el color para pintar en figuras de Java2D.

13.5 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) El método `drawPolygon` de `Graphics` conecta en forma automática los puntos de los extremos del polígono.
b) El método `drawLine` de `Graphics` dibuja una línea entre dos puntos.
c) El método `fillArc` de `Graphics` utiliza grados para especificar el ángulo.
d) En el sistema de coordenadas de Java, los valores del eje *y* se incrementan de izquierda a derecha.
e) La clase `Graphics` hereda directamente de la clase `Object`.

f) `Graphics` es una clase `abstract`.
g) La clase `Font` hereda en forma directa de la clase `Graphics`.

13.6 ***(Círculos concéntricos mediante el uso del método `drawArc`)*** Escriba una aplicación que dibuje una serie de ocho círculos concéntricos. Los círculos deberán estar separados por 10 píxeles. Use el método `drawArc` de la clase `Graphics`.

13.7 ***(Círculos concéntricos mediante el uso de la clase `Ellipse2D.Double`)*** Modifique su solución al ejercicio 13.6, para dibujar los óvalos mediante el uso de instancias de la clase `Ellipse2D.Double` y el método `draw` de la clase `Graphics2D`.

13.8 ***(Líneas aleatorias mediante el uso de la clase `Line2D.Double`)*** Modifique su solución al ejercicio 13.7 para dibujar líneas aleatorias en colores aleatorios y grosores de línea aleatorios. Use la clase `Line2D.Double` y el método `draw` de la clase `Graphics2D` para dibujar las líneas.

13.9 ***(Triángulos aleatorios)*** Escriba una aplicación que muestre triángulos generados al azar en distintos colores. Cada triángulo deberá rellenarse con un color distinto. Use la clase `GeneralPath` y el método `fill` de la clase `Graphics2D` para dibujar los triángulos.

13.10 ***(Caracteres aleatorios)*** Escriba una aplicación que dibuje caracteres al azar, en distintos tamaños, colores y tipos de letra.

13.11 ***(Cuadrícula mediante el uso del método `drawLine`)*** Escriba una aplicación que dibuje una cuadrícula de 8 por 8. Use el método `drawLine` de `Graphics`.

13.12 ***(Cuadrícula mediante el uso de la clase `Line2D.Double`)*** Modifique su solución al ejercicio 13.11 para dibujar la cuadrícula utilizando instancias de la clase `Line2D.Double` y el método `draw` de la clase `Graphics2D`.

13.13 ***(Cuadrícula mediante el uso del método `drawRect`)*** Escriba una aplicación que dibuje una cuadrícula de 10 por 10. Use el método `drawRect` de `Graphics`.

13.14 ***(Cuadrícula mediante el uso de la clase `Rectangle2D.Double`)*** Modifique su solución al ejercicio 13.13 para dibujar la cuadrícula utilizando la clase `Rectangle2D.Double` y el método `draw` de la clase `Graphics2D`.

13.15 ***(Dibujo de tetraedros)*** Escriba una aplicación que dibuje un tetraedro (una figura tridimensional con cuatro caras triangulares). Use la clase `GeneralPath` y el método `draw` de la clase `Graphics2D`.

13.16 ***(Dibujo de cubos)*** Escriba una aplicación que dibuje un cubo. Use la clase `GeneralPath` y el método `draw` de la clase `Graphics2D`.

13.17 ***(Círculos mediante el uso de la clase `Ellipse2D.Double`)*** Escriba una aplicación que pida al usuario introducir el radio de un círculo como número de punto flotante y que dibuje el círculo, así como los valores del diámetro, la circunferencia y el área del círculo. Use el valor 3.14159 para π. [*Nota*: también puede usar la constante predefinida `Math.PI` para el valor de π. Esta constante es más precisa que el valor 3.14159. La clase `Math` se declara en el paquete `java.lang`, por lo que no necesita importarla]. Use las siguientes fórmulas (r es el radio):

$$\textit{diámetro} = 2r$$
$$\textit{circunferencia} = 2\pi r$$
$$\textit{área} = \pi r^2$$

El usuario debe introducir también un conjunto de coordenadas además del radio. Después dibuje el círculo y muestre su diámetro, circunferencia y área mediante el uso de un objeto `Ellipse2D.Double` para representar el círculo y el método `draw` de la clase `Graphics2D` para mostrarlo en pantalla.

13.18 ***(Protector de pantalla)*** Escriba una aplicación que simule un protector de pantalla. La aplicación deberá dibujar líneas al azar, utilizando el método `drawLine` de la clase `Graphics`. Después de dibujar 100 líneas, la aplicación deberá borrarse a sí misma y empezar a dibujar líneas nuevamente. Para permitir al programa dibujar en forma continua, coloque una llamada a `repaint` como la última línea en el método `paintComponent`. ¿Observó algún problema con esto en su sistema?

13.19 ***(Protector de pantalla mediante el uso de `Timer`)*** El paquete `javax.swing` contiene una clase llamada `Timer`, la cual es capaz de llamar al método `actionPerformed` de la interfaz `ActionListener` durante un intervalo de tiempo

fijo (especificado en milisegundos). Modifique su solución al ejercicio 13.18 para eliminar la llamada a `repaint` desde el método `paintComponent`. Declare su clase de manera que implemente a `ActionListener`. (El método `actionPerformed` deberá simplemente llamar a `repaint`). Declare una variable de instancia de tipo `Timer`, llamada `temporizador`, en su clase. En el constructor para su clase, escriba las siguientes instrucciones:

```
temporizador = new Timer(1000, this);
temporizador.start();
```

Esto crea una instancia de la clase `Timer` que llamará al método `actionPerformed` del objeto `this` cada `1000` milisegundos (es decir, cada segundo).

13.20 ***(Protector de pantalla para un número aleatorio de líneas)*** Modifique su solución al ejercicio 13.19 para permitir al usuario introducir el número de líneas aleatorias que deben dibujarse antes de que la aplicación se borre a sí misma y empiece a dibujar líneas otra vez. Use un objeto `JTextField` para obtener el valor. El usuario deberá ser capaz de escribir un nuevo número en el objeto `JTextField` en cualquier momento durante la ejecución del programa. Use una clase interna para realizar el manejo de eventos para el objeto `JTextField`.

13.21 ***(Protector de pantalla con figuras)*** Modifique su solución al ejercicio 13.20, de tal forma que utilice la generación de números aleatorios para seleccionar diferentes figuras a mostrar. Use métodos de la clase `Graphics`.

13.22 ***(Protector de pantalla mediante el uso de la API Java 2D)*** Modifique su solución al ejercicio 13.21 para utilizar clases y herramientas de dibujo de la API Java2D. Para las figuras como rectángulos y elipses, dibújelas con degradados generados al azar. Use la clase `GradientPaint` para generar el degradado.

13.23 ***(Gráficos de tortuga)*** Modifique su solución al ejercicio 7.21 (*Gráficos de tortuga*) para agregar una interfaz gráfica de usuario, mediante el uso de objetos `JTextField` y `JButton`. Dibuje líneas en vez de asteriscos (`*`). Cuando el programa de gráficos de tortuga especifique un movimiento, traduzca el número de posiciones en un número de píxeles en la pantalla, multiplicando el número de posiciones por 10 (o cualquier valor que usted elija). Implemente el dibujo con características de la API Java2D.

13.24 ***(Paseo del caballo)*** Produzca una versión gráfica del problema del Paseo del caballo (ejercicios 7.22, 7.23 y 7.26). A medida que se realice cada movimiento, la celda apropiada del tablero de ajedrez deberá actualizarse con el número de movimiento apropiado. Si el resultado del programa es un *paseo completo* o un *paseo cerrado*, el programa deberá mostrar un mensaje apropiado. Si lo desea puede utilizar la clase `Timer` (vea el ejercicio 13.19) para que le ayude con la animación del Paseo del caballo.

13.25 (***La tortuga y la liebre***) Produzca una versión gráfica de la simulación *La tortuga y la liebre* (ejercicio 7.28). Simule la montaña dibujando un arco que se extienda desde la esquina inferior izquierda de la ventana, hasta la esquina superior derecha. La tortuga y la liebre deberán correr hacia arriba de la montaña. Implemente la salida gráfica de manera que la tortuga y la liebre se impriman en el arco, en cada movimiento. [*Sugerencia*: extienda la longitud de la carrera de 70 a 300, para que cuente con un área de gráficos más grande].

13.26 ***(Dibujo de espirales)*** Escriba una aplicación que utilice el método `drawPolyline` de `Graphics` para dibujar una espiral similar a la que se muestra en la figura 13.33.

13.27 ***(Gráfico de pastel)*** Escriba un programa que reciba como entrada cuatro números y que los grafique en forma de gráfico de pastel. Use la clase `Arc2D.Double` y el método `fill` de la clase `Graphics2D` para realizar el dibujo. Dibuje cada pieza del pastel en un color distinto.

13.28 ***(Selección de figuras)*** Escriba una aplicación que permita al usuario seleccionar una figura de un objeto `JComboBox` y que la dibuje 20 veces, con ubicaciones y medidas aleatorias en el método `paintComponent`. El primer elemento en el objeto `JComboBox` debe ser la figura predeterminada a mostrar la primera vez que se hace una llamada a `paintComponent`.

13.29 ***(Colores aleatorios)*** Modifique el ejercicio 13.28 para dibujar cada una de las 20 figuras con tamaños aleatorios en un color seleccionado al azar. Use los 13 objetos `Color` predefinidos en un arreglo de objetos `Color`.

13.30 ***(Cuadro de diálogo `JColorChooser`)*** Modifique el ejercicio 13.28 para permitir al usuario seleccionar de un cuadro de diálogo `JcolorChooser` el color en el que deben dibujarse las figuras.

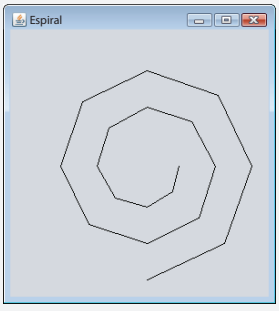

Fig. 13.33 | Dibujo de una espiral mediante el uso del método `drawPolyline`.

(Opcional) Ejemplo práctico de GUI y Gráficos: agregar Java2D

13.31 Java 2D presenta muchas nuevas herramientas para crear gráficos únicos e impresionantes. Agregaremos un pequeño subconjunto de estas características a la aplicación de dibujo que creó en el ejercicio 12.17. Esta versión de la aplicación de dibujo, permitirá al usuario especificar degradados para rellenar figuras y modificar las características de trazo para dibujar líneas y los contornos de las figuras. El usuario podrá elegir qué colores van a formar el degradado y también podrá establecer la anchura y longitud de guion del trazo.

Primero debe actualizar la jerarquía `MiFigura` para que soporte la funcionalidad de Java 2D. Haga las siguientes modificaciones en la clase `MiFigura`:

a) Cambie el tipo del parámetro del método `abstract draw`, de `Graphics` a `Graphics2D`.
b) Cambie todas las variables de tipo `Color` al tipo `Paint`, para habilitar el soporte para los degradados. [*Nota*: recuerde que la clase `Color` implementa a la interfaz `Paint`].
c) Agregue una variable de instancia de tipo `Stroke` en la clase `MiFigura` y un parámetro `Stroke` en el constructor para inicializar la nueva variable de instancia. El trazo predeterminado deberá ser una instancia de la clase `BasicStroke`.

Cada una de las clases `MiLinea`, `MiFiguraDelimitada`, `MiOvalo` y `MiRect` debe agregar un parámetro `Stroke` a sus constructores. En los métodos `draw` cada figura debe establecer los objetos `Paint` y `Stroke` antes de dibujar o rellenar una figura. Como `Graphics2D` es una subclase de `Graphics`, podemos seguir utilizando los métodos `drawLine`, `drawOval`, `fillOval`, y otros métodos más de `Graphics`, para dibujar las figuras. Al llamarlos, estos métodos dibujarán la figura apropiada usando las opciones especificadas para los objetos `Paint` y `Stroke`.

Después actualice el objeto `PanelDibujo` para manejar las herramientas de Java 2D. Cambie todas las variables `Color` a variables `Paint`. Declare una variable de instancia llamada `trazoActual` de tipo `Stroke` y proporcione un método *establecer* para esta variable. Actualice las llamadas a los constructores de cada figura para incluir los argumentos `Paint` y `Stroke`. En el método `paintComponent`, convierta la referencia `Graphics` al tipo `Graphics2D` y use la referencia `Graphics2D` en cada llamada al método `draw` de `MiFigura`.

A continuación, haga que se pueda tener acceso a las nuevas características de Java 2D mediante la GUI. Cree un objeto `JPanel` de componentes de GUI para establecer las opciones de Java2D. Agregue esos componentes a la parte superior del objeto `MarcoDibujo`, debajo del panel que actualmente contiene los controles de las figuras estándar (vea la figura 13.34). Estos componentes de GUI deben incluir lo siguiente:

a) Una casilla de verificación para especificar si se va a pintar usando un degradado.
b) Dos objetos `JButton`, cada uno de los cuales debe mostrar un cuadro de diálogo `JColorChooser`, para permitir al usuario elegir los colores primero y segundo en el degradado. (Éstos sustituirán al objeto `JComboBox` que se utiliza para extender el color en el ejercicio 12.17).
c) Un campo de texto para introducir la anchura del objeto `Stroke`.
d) Un campo de texto para introducir la longitud de guion del objeto `Stroke`.
e) Una casilla de verificación para seleccionar si se va a dibujar una línea punteada o sólida.

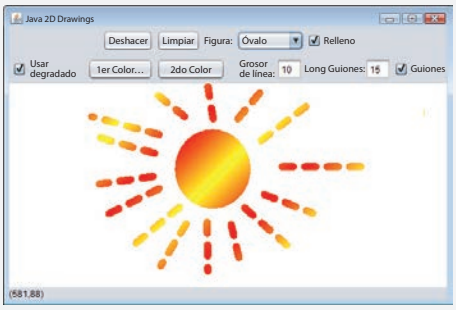

Fig. 13.34 | Dibujo con `Java2D`.

Si el usuario opta por dibujar con un degradado, establezca el objeto `Paint` en el `PanelDibujo` de forma que sea un degradado de los dos colores seleccionados por el usuario. La expresión

```
new GradientPaint(0, 0, color1, 50, 50, color2, true))
```

crea un objeto `GradientPath` que avanza diagonalmente en círculos, desde la esquina superior izquierda hasta la esquina inferior derecha, cada 50 píxeles. Las variables `color1` y `color2` representan los colores elegidos por el usuario. Si éste no elije usar un degradado, entonces simplemente establezca el objeto `Paint` en el `PanelDibujo` de manera que sea el primer `Color` elegido por el usuario.

Para los trazos, si el usuario elije una línea sólida entonces cree el objeto `Stroke` con la expresión

```
new BasicStroke(anchura, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND)
```

en donde la variable `anchura` es la anchura especificada por el usuario en el campo de texto de anchura de línea. Si el usuario selecciona una línea punteada, entonces cree el objeto `Stroke` con la expresión

```
new BasicStroke(anchura, BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND,
   10, guiones, 0)
```

en donde `anchura` es de nuevo la anchura en el campo de anchura de texto, y `guiones` es un arreglo con un elemento cuyo valor es la longitud especificada en el campo de longitud de guion. Los objetos `Panel` y `Stroke` deben pasarse al constructor del objeto figura cuando se cree la figura en el objeto `PanelDibujo`.

Marcando la diferencia

13.32 ***(Pantallas con tipos de letra grandes para personas con poca visión)*** La accesibilidad de las computadoras e Internet para todas las personas, sin importar las discapacidades, se está volviendo cada vez más importante a medida que estas herramientas desempeñan más roles en nuestras vidas personales y de negocios. De acuerdo con una estimación reciente de la Organización Mundial de la Salud (`www.who.int/mediacentre/factsheets/fs282/es`), 246 millones de personas en todo el mundo sufren de baja visión. Para aprender más sobre la baja visión, dé un vistazo a la simulación de baja visión basada en GUI en `www.webaim.org/simulations/lowvision.php`. Las personas con baja visión podrían preferir un tipo o un tamaño de letra más grandes para leer documentos electrónicos y páginas Web. Java cuenta con cinco tipos de letra "lógicos" integrados, que se garantiza estarán disponibles en cualquier implementación de Java, incluyendo `Serif`, `Sans-serif` y `Monospaced`. Escriba una aplicación con GUI que proporcione un objeto `JTextArea` en donde el usuario pueda escribir texto. Permita al usuario seleccionar `Serif`, `Sans-serif` o `Monospaced` de un `JComboBox`. Proporcione un objeto `JCheckBox` **Negrita**, que cuando se seleccione el texto se ponga en negrita. Incluya los objetos `JButton` **Aumentar tamaño de letra** y **Reducir tamaño de letra**, que permitan al usuario escalar el tamaño del tipo de letra para aumentarlo o reducirlo, respectivamente, un punto a la vez. Empiece con un tamaño de letra de 18 puntos. Para los fines de este ejercicio, establezca el tamaño del tipo de letra en los objetos `JComboBox`, `JButton` y `JCheckBox` en 20 puntos, de modo que una persona con baja visión pueda leer el texto en ellos.