

CAPITOLO I

Le immagini e Java2D

Obiettivi

- Comprendere i contesti e gli oggetti grafici
- Imparare a usare i colori
- Imparare a usare i tipi di caratteri
- Imparare a usare i metodi della classe **Graphics** per disegnare linee, rettangoli, rettangoli con angoli arrotondati, rettangoli tridimensionali, ovali, archi e poligoni
- Imparare a usare i metodi della classe **Graphics2D** di Java2D API, per disegnare linee, rettangoli, rettangoli con angoli arrotondati, ellissi, archi e altre forme
- Imparare a specificare le caratteristiche **Paint** e **Stroke** delle forme visualizzate con **Graphics2D**

1.1 Introduzione

In questo capitolo verranno prese in esame molte delle funzionalità Java riguardanti il disegno di forme bidimensionali, il controllo dei colori e il controllo dei tipi di carattere. Sin dall'inizio, Java ha permesso ai programmatori di perfezionare le proprie applet e applicazioni attraverso le immagini, e attualmente queste funzionalità grafiche sono state ulteriormente ampliate e perfezionate all'interno di *Java2D API*. Il capitolo parte con l'introduzione di molte delle capacità originarie di Java, per passare poi a presentare alcune delle nuove funzionalità, come il controllo dello stile delle linee utilizzate per disegnare e il controllo del riempimento delle figure disegnate (colore e trama).

La figura 1.1 mostra una parte della gerarchia di classi Java contenente alcune delle classi grafiche di base, oltre che le classi e interfacce Java2D API trattate in questo capitolo. La classe **Color** contiene i metodi e le costanti per gestire i colori; la classe **Font** contiene i metodi e le costanti per gestire i tipi di caratteri; la classe **FontMetrics** contiene i metodi necessari per ottenere informazioni riguardo ai tipi di caratteri; la classe **Polygon** contiene i metodi per creare dei poligoni; la classe **Graphics** contiene i metodi per disegnare stringhe, linee, rettangoli e altre forme.

Nella parte inferiore della figura 1.1 sono elencate alcune delle classi e delle interfacce di Java2D API. La classe **BasicStroke** permette di specificare le caratteristiche di disegno delle linee; la classe **GradientPaint** e la classe **TexturePaint** permettono di specificare le caratteristiche di riempimento delle forme, con trame e colori; le classi **GeneralPath**, **Arc2D**, **Ellipse2D**, **Line2D**, **Rectangle2D** e **RoundRectangle2D** definiscono un'ampia varietà di forme Java2D.

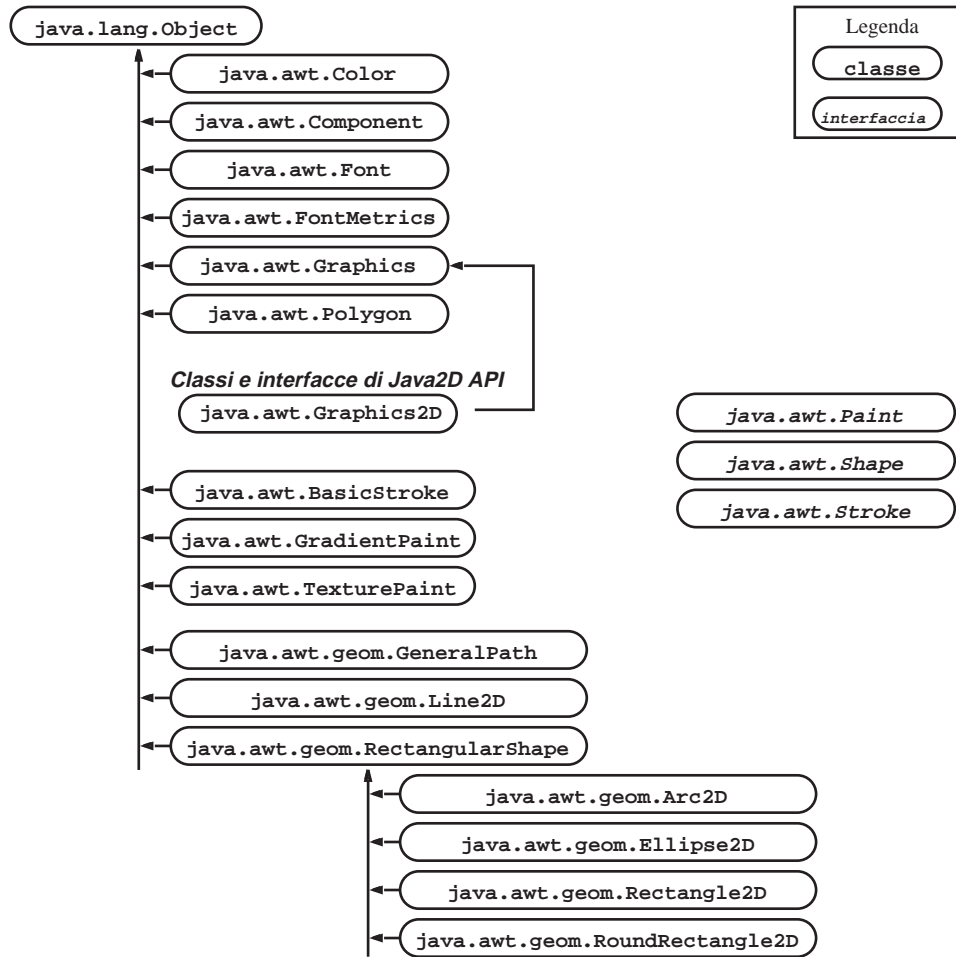


Figura 1.1 Alcune classi e interfacce utilizzate in questo capitolo per mostrare le capacità grafiche di Java 2

Per iniziare a disegnare con Java, è necessario prima capire cos'è il *sistema di coordinate* di Java (figura 1.2), ovvero uno schema per l'identificazione di ogni possibile punto presente sullo schermo. Per default, l'angolo superiore sinistro di un componente GUI (come nel caso di un'applet o di una finestra) corrisponde alle coordinate $(0, 0)$. Una coppia di coordinate è composta da una *coordinata x* (la *coordinata orizzontale*) e da una *coordinata y* (la *coordinata verticale*). La coordinata x è la distanza orizzontale che si misura verso destra partendo dall'angolo superiore sinistro; la coordinata y è la distanza verticale che si misura partendo dall'angolo superiore sinistro e andando verso il basso. L'asse x contiene ogni coordinata orizzontale, mentre l'asse y contiene ogni coordinata verticale.



Ingegneria del software 1.1

La coordinata dell'angolo superiore sinistro $(0, 0)$ di una finestra si trova in realtà dietro la barra del titolo della finestra stessa. Per questo motivo, le coordinate di disegno dovrebbero essere modificate affinché il disegno si trovi all'interno dei bordi della finestra.

La classe **Container** (una superclasse di tutte le finestre Java) possiede il metodo **getInsets**, che ritorna un oggetto **Insets** (package **java.awt**) proprio a questo scopo. Un oggetto **Insets** possiede quattro membri **public** (**top**, **bottom**, **left** e **right**), che rappresentano il numero di pixel presenti da ogni bordo della finestra fino alla sua area di disegno.

Specificando delle coordinate, i testi e le forme vengono visualizzati sullo schermo. Le unità delle coordinate sono i *pixel*, ovvero la più piccola unità di risoluzione di uno schermo.



Obiettivo portabilità 1.1

Schermi diversi hanno risoluzioni diverse (ovvero, varia la densità dei pixel). È questo il motivo per cui alcune immagini potrebbero apparire di diversa dimensione su schermi diversi.

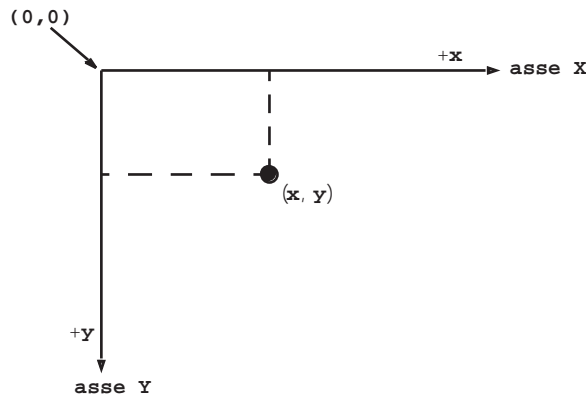


Figura I.2 Il sistema di coordinate Java. Le unità sono misurate in pixel.

I.2 I contesti e gli oggetti Graphics

Un contesto grafico Java consente di creare disegni sullo schermo, mentre un oggetto **Graphics** gestisce un contesto grafico controllando il modo in cui vengono disegnate le informazioni. Gli oggetti **Graphics** contengono i metodi necessari per il disegno, la gestione dei tipi di caratteri, la gestione dei colori e così via. Tra quelle viste finora, ogni applet che eseguiva dei disegni sullo schermo utilizzava l'oggetto **Graphics** chiamato **g** (l'argomento per il metodo **paint** dell'applet) per gestire il contesto grafico dell'applet. In questo capitolo, si parlerà del disegno nell'ambito delle applicazioni, ma tutte le tecniche che verranno prese in esame possono essere utilizzate anche con le applet.

La classe **Graphics** è una classe **abstract** (ovvero, gli oggetti **Graphics** non possono essere istanziati). Ciò favorisce la portabilità di Java, in quanto le operazioni di disegno vengono eseguite in modo diverso sulle varie piattaforme che supportano Java, e non ci può essere una sola classe che implementi le funzionalità di disegno per tutti i sistemi. In altre parole, le funzionalità che permettono a un PC Microsoft Windows di disegnare un rettangolo sono diverse da quelle che permettono a una workstation UNIX di disegnare un rettangolo. Quando Java viene implementato su ognuna di queste piattaforme, viene creata

una classe derivata di **Graphics**, che si occupa di implementare tutte le funzionalità di disegno. Questa implementazione viene tenuta nascosta dagli occhi dell'utente dalla classe **Graphics**, che fornisce l'interfaccia che permette di scrivere programmi in grado di supportare le immagini in modo indipendente dalla piattaforma.

La classe **Component** è la superclasse per molte delle classi del package **java.awt** (della classe **Component** si parlerà nel capitolo 2). Il metodo **paint** di **Component** prende come argomento un oggetto della classe **Graphics**, il quale viene passato al metodo **paint** dal sistema ogni volta che è necessaria un'operazione di disegno per un **Component**.

L'intestazione del metodo **paint** è

```
public void paint ( Graphics g )
```

L'oggetto **g** riceve un riferimento a un oggetto della classe derivata **Graphics** del sistema. Questa intestazione dovrebbe già esservi familiare, in quanto è la stessa che è stata utilizzata per le classi delle applet. La classe **Component** è in realtà una classe di base indiretta della classe **JApplet**, ovvero la superclasse di ognuna delle applet presentate in questo libro. Molte delle capacità della classe **JApplet** sono ereditate dalla classe **Component**. Il metodo **paint** definito nella classe **Component** non fa nulla per default, bensì deve essere sovrascritto dal programmatore.

Il metodo **paint** è raramente chiamato direttamente dal programmatore, in quanto il disegno di immagini è un *processo guidato dagli eventi*. Quando un'applet viene eseguita, il metodo **paint** viene automaticamente chiamato (a seguito di chiamate ai metodi **init** e **start** di **JApplet**). Perché **paint** venga nuovamente chiamato, deve avere luogo un *evento*; in modo simile, quando un qualsiasi **Component** viene visualizzato, è il metodo **paint** di quel **Component** a essere chiamato.

Se il programmatore necessita di chiamare **paint**, viene effettuata una chiamata al metodo **repaint** della classe **Component**. Il metodo **repaint** richiede subito una chiamata al metodo **update** della classe **Component**, per liberare lo sfondo (background) di quel **Component** da qualsiasi disegno precedente, dopodiché **update** chiama **paint** direttamente.

Il metodo **repaint** è frequentemente chiamato dal programmatore per “forzare” un'operazione **paint**, e non dovrebbe essere sovrascritto in quanto esegue delle attività direttamente dipendenti dal sistema. Il metodo **update** viene raramente chiamato direttamente, e qualche volta viene sovrascritto. La sovrascrittura del metodo **update** è utile per “perfezionare” le animazioni (per esempio, per ridurre lo sfarfallio sullo schermo), come vedrete nel capitolo 6.

Le intestazioni di **repaint** e **update** sono

```
public void repaint ()
public void update (Graphics g )
```

Il metodo **update** prende un oggetto **Graphics** come argomento, il quale è fornito automaticamente dal sistema quando **update** viene chiamato.

In questo capitolo verrà preso in esame il metodo **paint**, mentre nel prossimo capitolo si parlerà della natura guidata dagli eventi della grafica, prendendo in esame i metodi **repaint** e **update**.

1.3 Il controllo dei colori

I colori migliorano l'aspetto di un programma, aiutandolo a trasmettere in modo più chiaro il proprio messaggio.

La classe **Color** definisce i metodi e le costanti necessari per gestire i colori all'interno di un programma Java. Le costanti predefinite per i colori vengono mostrate nella figura 1.3, mentre molti altri metodi e costruttori si trovano nella figura 1.4. Notate come due dei metodi della figura 1.4 siano metodi **Graphics** specifici per i colori.

Costante Color	Colore	Valore RGB
<code>public final static Color orange</code>	arancione	255, 200, 0
<code>public final static Color pink</code>	rosa	255, 175, 175
<code>public final static Color cyan</code>	ciano	0, 255, 255
<code>public final static Color magenta</code>	magenta	255, 0, 255
<code>public final static Color yellow</code>	giallo	255, 255, 0
<code>public final static Color black</code>	nero	0, 0, 0
<code>public final static Color white</code>	bianco	255, 255, 255
<code>public final static Color gray</code>	grigio	128, 128, 128
<code>public final static Color lightGray</code>	grigio chiaro	192, 192, 192
<code>public final static Color darkGray</code>	grigio scuro	64, 64, 64
<code>public final static Color red</code>	rosso	255, 0, 0
<code>public final static Color green</code>	verde	0, 255, 0
<code>public final static Color blue</code>	blu	0, 0, 255

Figura 1.3 Le costanti static della classe Color e i loro valori RGB

Metodo	Descrizione
<code>public Color(int r, int g, int b)</code>	Crea un colore basato sulle quantità di rosso, verde e blu espresse sotto forma di numeri interi da 0 a 255.
<code>public Color(float r, float g, float b)</code>	Crea un colore basato sulle quantità di rosso, verde e blu espresse sotto forma di valori a virgola mobile da 0.0 a 1.0.
<code>public int getRed()</code> <code>// classe Color</code>	Ritorna un valore tra 0 e 255 rappresentante la quantità di rosso.

Figura 1.4 I metodi Color e i metodi Graphics relativi ai colori (continua)

Metodo	Descrizione
public int getGreen() // classe Color	Ritorna un valore tra 0 e 255 rappresentante la quantità di verde.
public int getBlue() // classe Color	Ritorna un valore tra 0 e 255 rappresentante la quantità di blu.
public Color getColor() // classe Graphics	Ritorna un oggetto Color che indica l'attuale colore del contesto grafico.
public void setColor(Color c) // classe Graphics	Imposta il colore attuale per disegnare con il contesto grafico.

Figura 1.4 I metodi **Color** e i metodi **Graphics** relativi ai colori

Ogni colore è creato da una quantità di rosso, una di verde e una di blu; insieme, queste quantità sono chiamate valori RGB. Tutti e tre i valori RGB possono essere numeri interi nell'intervallo da 0 a 255, oppure possono essere dei valori a virgola mobile nell'intervallo da 0.0 a 1.0. La prima parte del valore RGB definisce la quantità di rosso, la seconda la quantità di verde e la terza la quantità di blu; quanto più grande è il valore RGB, tanto più grande è la quantità di quel particolare colore. Java permette al programmatore di scegliere tra 256x256x256 (approssimativamente 16,7 milioni) di colori; non tutti i computer, però, sono in grado di mostrare tutti questi colori.



Errore tipico 1.1

*Scrivere una qualsiasi costante della classe **static Color** iniziando con una lettera maiuscola costituisce un errore di sintassi.*

Nella figura 1.4 vengono mostrati due costruttori di **Color**, di cui uno prende tre argomenti **int**, mentre l'altro prende tre argomenti **float**, e dove tutti gli argomenti specificano rispettivamente la quantità di rosso, verde e blu. I valori **int** devono essere compresi nell'intervallo tra 0 e 255, mentre i valori **float** nell'intervallo tra 0.0 e 1.0. Il nuovo oggetto **Color** avrà le quantità specificate di rosso, verde e blu.

I metodi **Color** **getRed**, **getGreen** e **getBlue** ritornano valori interi nell'intervallo da 0 a 255, a rappresentare rispettivamente la quantità di rosso, verde e blu. Il metodo **getColor** di **Graphics** ritorna un oggetto **Color** rappresentante il colore attualmente utilizzato per il disegno, mentre il metodo **setColor** di **Graphics** imposta il colore attuale.

L'applicazione della figura 1.5 mostra alcuni dei metodi della figura 1.4 disegnando dei rettangoli e delle stringhe di diversi colori.

```

1 // Fig. 1.5: ShowColors.java
2 // Dimostrazione di Colors
3 import java.awt.*;
4 import javax.swing.*;
5 import java.awt.event.*;
```

Figura 1.5 L'utilizzo di **Color** (continua)

```
6
7   public class ShowColors extends JFrame {
8       public ShowColors()
9       {
10          super( "Using colors" );
11
12          setSize( 400, 130 );
13          show();
14      }
15
16      public void paint( Graphics g )
17      {
18          // imposta un nuovo colore usando i numeri interi
19          g.setColor( new Color( 255, 0, 0 ) );
20          g.fillRect( 25, 25, 100, 20 );
21          g.drawString( "Current RGB: " + g.getColor(), 130, 40 );
22
23          // imposta un nuovo colore usando numeri a virgola mobile
24          g.setColor( new Color( 0.0f, 1.0f, 0.0f ) );
25          g.fillRect( 25, 50, 100, 20 );
26          g.drawString( "Current RGB: " + g.getColor(), 130, 65 );
27
28          // imposta un nuovo colore usando oggetti statici Color
29          g.setColor( Color.blue );
30          g.fillRect( 25, 75, 100, 20 );
31          g.drawString( "Current RGB: " + g.getColor(), 130, 90 );
32
33          // mostra i singoli valori RGB
34          Color c = Color.magenta;
35          g.setColor( c );
36          g.fillRect( 25, 100, 100, 20 );
37          g.drawString( "RGB values: " + c.getRed() + ", " +
38              c.getGreen() + ", " + c.getBlue(), 130, 115 );
39      }
40
41      public static void main( String args[] )
42      {
43          ShowColors app = new ShowColors();
44
45          app.addWindowListener(
46              new WindowAdapter() {
47                  public void windowClosing( WindowEvent e )
48                  {
49                      System.exit( 0 );
50                  }
51              }
52          );
53      }
54  }
```

Figura I.5 L'utilizzo di Color (continua)

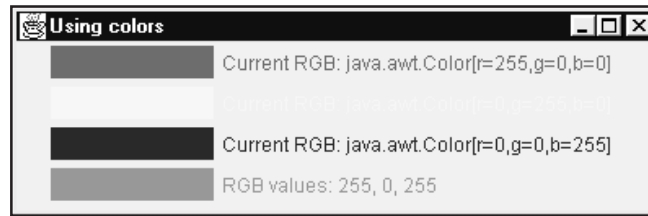


Figura 1.5 L'utilizzo di Color

Quando un'applicazione inizia l'esecuzione, viene chiamato il metodo **paint** della classe **ShowColors**. La riga 19

```
g.setColor( new Color( 255, 0, 0 ) );
```

utilizza il metodo **setColor** di **Graphics** per impostare l'attuale colore di disegno. Il metodo **setColor** riceve un oggetto **Color**. L'espressione **new Color** (255, 0, 0) crea un nuovo oggetto **Color** che rappresenta il rosso (255 per il valore del rosso e 0 per i valori del verde e del blu).

La riga 20

```
g.fillRect( 25, 25, 100, 20 );
```

utilizza il metodo **fillRect** di **Graphics** per disegnare un rettangolo riempito del colore attuale. Il metodo **fillRect** riceve gli stessi parametri del metodo **drawRect**.

La riga 21

```
g.drawString( "Current RGB: " + g.getColor(), 130, 40 );
```

utilizza il metodo **drawString** di **Graphics** per disegnare una **String** nel colore attuale. L'espressione **g.getColor()** richiama il colore attuale dall'oggetto **Graphics**. Il **Color** ritornato è concatenato con la stringa "Current RGB:", che dà vita a una chiamata implicita al metodo **toString** della classe **Color**. Notate come la rappresentazione **String** dell'oggetto **Color** contenga il nome della classe e il package (**java.awt.Color**), oltre che i valori del rosso, del verde e del blu.

Le righe dalla 24 alla 26 e dalla 29 alla 31 eseguono nuovamente queste stesse operazioni. La riga 24

```
g.setColor( new Color( 0.0f, 1.0f, 0.0f ) );
```

utilizza il costruttore **Color** con tre argomenti **float** per creare il colore verde (**0.0f** per il rosso, **1.0f** per il verde e **0.0f** per il blu). Notate la sintassi delle costanti: la lettera **f** dopo una costante a virgola mobile indica che la costante deve essere trattata come un tipo **float**; normalmente le costanti a virgola mobile vengono trattate come un tipo **double**.

La riga 29 imposta l'attuale colore di disegno su una delle costanti **Color** predefinite (**Color.blue**). Notate come l'operatore **new** non sia necessario per creare la costante; dato che le costanti **Color** sono **static**, vengono definite quando la classe **Color** è caricata in memoria in fase di esecuzione.

L'istruzione delle righe 37 e 38 mostra i metodi **Color** **getRed**, **getGreen** e **getBlue** sull'oggetto predefinito **Color.magenta**.



Ingegneria del software 1.2

*Per modificare il colore attuale, dovete creare un nuovo oggetto **Color** (oppure usare una delle costanti **Color** predefinite), dato che nella classe **Color** non ci sono metodi definiti per modificare le caratteristiche del colore attuale.*

Una delle caratteristiche nuove di Java è il componente GUI predefinito **JColorChooser** (package **javax.swing**), che consente di selezionare i colori. L'applicazione della figura 1.6 permette di premere un pulsante per visualizzare una finestra di dialogo **JColorChooser**; selezionando un colore e poi premendo il pulsante **OK**, il colore di sfondo della finestra dell'applicazione cambia.

```

1 // Fig. 1.6: ShowColors2.java
2 // Dimostrazione di JColorChooser
3 import java.awt.*;
4 import javax.swing.*;
5 import java.awt.event.*;
6
7 public class ShowColors2 extends JFrame {
8     private JButton changeColor;
9     private Color color = Color.lightGray;
10    private Container c;
11
12    public ShowColors2()
13    {
14        super( "Using JColorChooser" );
15
16        c = getContentPane();
17        c.setLayout( new FlowLayout() );
18
19        changeColor = new JButton( "Change Color" );
20        changeColor.addActionListener(
21            new ActionListener() {
22                public void actionPerformed((ActionEvent e) )
23                {
24                    color =
25                        JColorChooser.showDialog( ShowColors2.this,
26                                                  "Choose a color", color );
27
28                    if ( color == null )
29                        color = Color.lightGray;
30
31                    c.setBackground( color );
32                    c.repaint();
33                }
34            }
35        );
36        c.add( changeColor );

```

Figura 1.6 La finestra di dialogo **JColorChooser** (continua)

```
37
38     setSize( 400, 130 );
39     show();
40 }
41
42 public static void main( String args[] )
43 {
44     ShowColors2 app = new ShowColors2();
45
46     app.addWindowListener(
47         new WindowAdapter() {
48             public void windowClosing( WindowEvent e )
49             {
50                 System.exit( 0 );
51             }
52         }
53     );
54 }
55 }
```

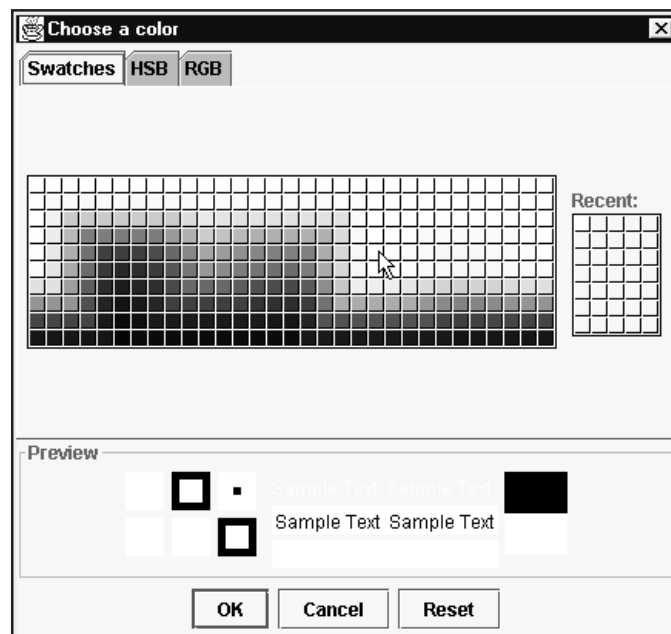
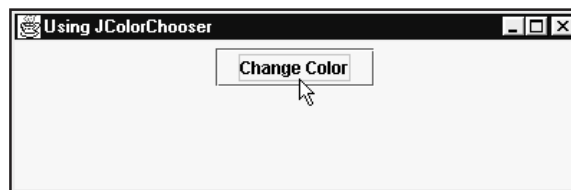


Figura I.6 La finestra di dialogo JColorChooser (continua)

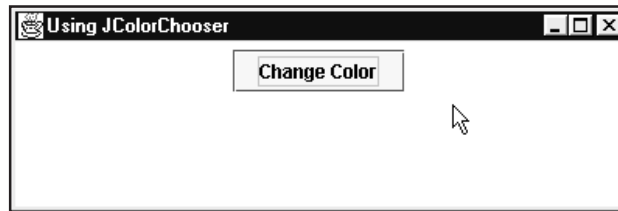


Figura 1.6 La finestra di dialogo `JColorChooser`

Le righe dalla 24 alla 26 del metodo `actionPerformed` per `changeColor`

```
color =
    JColorChooser.showDialog( ShowColors2.this,
        "Choose a color", color );
```

utilizzano il metodo `static showDialog` della classe `JColorChooser` per visualizzare la scelta dei colori. Questo metodo ritorna l'oggetto `Color` selezionato (oppure `null` se l'utente preme `Cancel` o chiude la finestra di dialogo senza premere `OK`).

Il metodo `showDialog` prende tre argomenti: un riferimento al suo `Component` padre, una `String` da visualizzare nella barra del titolo della finestra di dialogo, e il `Color` inizialmente selezionato per la finestra di dialogo. Il componente padre è la finestra da cui parte la visualizzazione di `JColorChooser`; mentre la finestra di dialogo `JColorChooser` si trova sullo schermo, l'utente non può interagire con il componente padre. (Di questo tipo di finestra di dialogo si parlerà nel capitolo 3.) Notate come la speciale sintassi `ShowColors2.this` viene utilizzata nell'istruzione precedente; quando si usa una classe interna, è possibile accedere al riferimento `this` dell'oggetto della classe esterna qualificando `this` con il nome della classe esterna e con l'operatore punto (`.`). Dopo che l'utente ha selezionato un colore, le righe 28 e 29 determinano se `color` è `null` e, se così è, impostano `color` sul default `Color.LightGray`. La riga 31

```
c.setBackground( color );
```

utilizza il metodo `setBackground` per modificare il colore di sfondo del riquadro dei contenuti (in questo programma rappresentato da `Container c`). Il metodo `setBackground` è uno dei tanti metodi `Component` che possono essere utilizzati con la maggior parte dei componenti GUI. La riga 32

```
c.repaint ();
```

assicura che il colore dello sfondo sia modificato, chiamando `repaint` per il riquadro dei contenuti. In questo modo viene programmata una chiamata al metodo `update` del riquadro dei contenuti, che ne colora lo sfondo utilizzando il colore di sfondo attuale.

La seconda immagine della figura 1.6 mostra la finestra di dialogo `JColorChooser` di default, che permette all'utente di selezionare un colore da una tavolozza composta di molti colori. Come potete notare, ci sono tre etichette nella parte superiore della finestra di dialogo (`Swatches`, `HSB` e `RGB`); queste tre etichette rappresentano tre modalità diverse per selezionare un colore. L'etichetta `HSB` permette di selezionare un colore in base alla sua tinta, alla sua saturazione e alla sua luminosità; l'etichetta `RGB` permette di selezionare un colore utilizzando un cursore per scegliere le quantità di rosso, verde e blu del colore in questione. Le etichette `HSB` e `RGB` vengono mostrate nella figura 1.7.

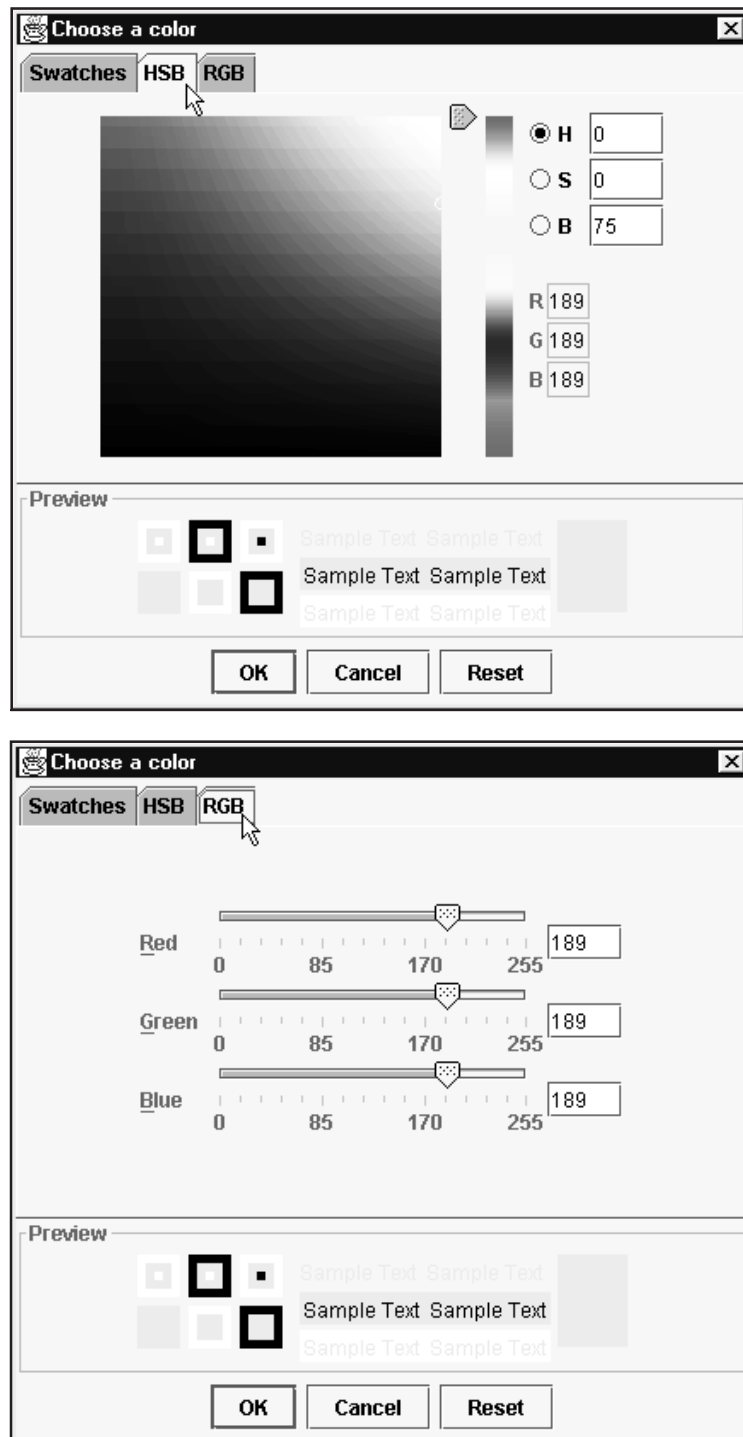


Figura I.7 Le etichette HSB e RGB della finestra di dialogo JColorChooser

1.4 Il controllo dei tipi di caratteri

Questa sezione introduce i metodi e le costanti per il controllo dei tipi di caratteri, la maggior parte dei quali fa parte della classe **Font**. Alcuni dei metodi della classe **Font** e della classe **Graphics** sono riassunti nella figura 1.8.

Il costruttore della classe **Font** prende tre argomenti: il nome, lo stile e la dimensione del tipo di carattere. Il nome del tipo di carattere è costituito da un qualsiasi tipo di carattere supportato dal sistema su cui viene eseguito il programma, come nel caso dei tipi di caratteri standard di Java: **Monospaced**, **SansSerif** e **Serif**.

Lo stile del carattere è **Font.PLAIN**, **Font.ITALIC** o **Font.BOLD** (costanti **static** della classe **Font**); gli stili dei caratteri possono essere usati in combinazione tra di loro (per esempio, **Font.ITALIC + Font.BOLD**). La dimensione del carattere è misurata in punti, dove un punto corrisponde a 1/72 di pollice.

Il metodo **setFont** di **Graphics** imposta il tipo di carattere di disegno attuale (il tipo di carattere che verrà usato per visualizzare il testo) in base al suo argomento **Font**.

Metodo o costante	Descrizione
public final static int PLAIN // classe Font	Una costante che rappresenta uno stile di carattere normale.
public final static int BOLD // classe Font	Una costante che rappresenta uno stile di carattere grassetto.
public final static int ITALIC // classe Font	Una costante che rappresenta uno stile di carattere corsivo.
public Font(String name, int style, int size)	Crea un oggetto Font con il tipo di carattere, lo stile e la dimensione specificati.
public int getStyle() // classe Font	Ritorna un valore intero che indica lo stile del carattere attuale.
public int getSize() // classe Font	Ritorna un valore intero con la dimensione del carattere.
public String getName() // classe Font	Ritorna il nome del carattere attuale sotto forma di stringa.

Figura 1.8 I metodi e le costanti Fonts e i metodi Graphics per i tipi di caratteri (continua)

Metodo o costante	Descrizione
public String getFamily() // classe Font	Ritorna il nome della famiglia del carattere attuale sotto forma di stringa.
public boolean isPlain() // classe Font	Controlla se un carattere è di stile normale. Ritorna true se il carattere è normale.
public boolean isBold() // classe Font	Controlla se un carattere è di stile grassetto. Ritorna true se il carattere è grassetto.
public boolean isItalic() // classe Font	Controlla se un carattere è di stile corsivo. Ritorna true se il carattere è corsivo.
public Font getFont() // classe Graphics	Ritorna un riferimento all'oggetto Font rappresentante il tipo di carattere attuale.
public void setFont(Font f) // classe Graphics	Imposta il tipo di carattere attuale sul tipo di carattere, lo stile e la dimensione specificati dal riferimento f all'oggetto Font .

Figura 1.8 I metodi e le costanti **Font**s e i metodi **Graphics** per i tipi di caratteri



Obiettivo portabilità 1.2

Il numero di tipi di caratteri varia da un sistema all'altro, ma il JDK garantisce che i tipi di caratteri **Serif**, **Monospaced**, **SansSerif**, **Dialog** e **DialogInput** siano sempre disponibili.



Errore tipico 1.2

Specificare un tipo di carattere che non è disponibile all'interno di un sistema costituisce un errore di logica. In questi casi, Java sostituisce quel tipo di carattere con il tipo di carattere di default del sistema.

Il programma della figura 1.9 visualizza il testo in quattro tipi di caratteri diversi, dove ogni tipo di carattere ha una differente dimensione. Il programma utilizza il costruttore **Font** per inizializzare gli oggetti **Font** delle righe 20, 25, 30 e 37 (ognuno in una chiamata al metodo **setFont** di **Graphics** per modificare il tipo di carattere di disegno). Ogni chiamata al costruttore **Font** passa un nome di tipo di carattere (**Serif**, **Monospaced** o **SansSerif**) come una **String**, uno stile di carattere (**Font.PLAIN**, **Font.ITALIC** o **Font.BOLD**) e una dimensione di carattere. Una volta invocato il metodo **setFont** di **Graphics**, tutto il testo visualizzato a seguito della chiamata apparirà con il nuovo tipo di carattere, fino a

che questo non viene nuovamente modificato. Notate come la riga 35 cambi il colore di disegno in rosso, affinché la stringa successiva appaia nel colore rosso.



Ingegneria del software 1.3

*Per modificare il tipo di carattere, dovete creare un nuovo oggetto **Font**, dato che nella classe **Font** non ci sono metodi set che permettono di modificare le caratteristiche del tipo di carattere attuale.*

```

1 // Fig. 1.9: Fonts.java
2 // L'uso dei tipi di caratteri
3 import java.awt.*;
4 import javax.swing.*;
5 import java.awt.event.*;
6
7 public class Fonts extends JFrame {
8     public Fonts()
9     {
10         super( "Using fonts" );
11
12         setSize( 420, 125 );
13         show();
14     }
15
16     public void paint( Graphics g )
17     {
18         // imposta il tipo di carattere a Serif (Times),
19         // grassetto, 12 pt e disegna una stringa
20         g.setFont( new Font( "Serif", Font.BOLD, 12 ) );
21         g.drawString( "Serif 12 point bold.", 20, 50 );
22
23         // imposta il tipo di carattere attuale a Monospaced
24         // (Courier), corsivo, 24pt e disegna una stringa
25         g.setFont( new Font( "Monospaced", Font.ITALIC, 24 ) );
26         g.drawString( "Monospaced 24 point italic.", 20, 70 );
27
28         // imposta il tipo di carattere come SansSerif
29         // (Helvetica), normale, 14pt e disegna una stringa
30         g.setFont( new Font( "SansSerif", Font.PLAIN, 14 ) );
31         g.drawString( «SansSerif 14 point plain.», 20, 90 );
32
33         // imposta il tipo di carattere come Serif (times),
34         // grassetto/corsivo, 18pt e disegna una stringa
35         g.setColor( Color.red );
36         g.setFont(
37             new Font( "Serif", Font.BOLD + Font.ITALIC, 18 ) );
38         g.drawString( g.getFont().getName() + " " +

```

Figura I.9 L'uso del metodo setFont per modificare i Font (continua)

```

39             g.getFont().getSize() +
40             " point bold italic.", 20, 10 );
41     }
42
43     public static void main( String args[] )
44     {
45         Fonts app = new Fonts();
46
47         app.addWindowListener(
48             new WindowAdapter() {
49                 public void windowClosing( WindowEvent e )
50                 {
51                     System.exit( 0 );
52                 }
53             }
54         );
55     }
56 }

```

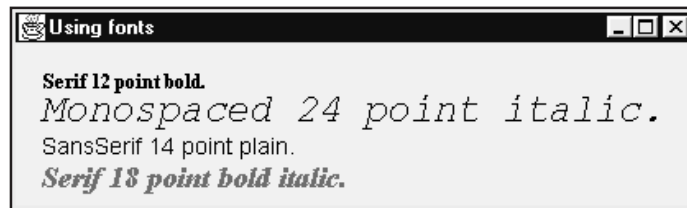


Figura 1.9 L'uso del metodo `setFont` per modificare i Font

Spesso è necessario ottenere informazioni circa il tipo di carattere attuale, come per esempio il suo nome, il suo stile o la sua dimensione. Molti dei metodi **Font** utilizzati per ottenere questo tipo di informazioni sono riassunti nella figura 1.8. Il metodo `getStyle` ritorna un valore intero rappresentante lo stile attuale; questo valore può essere **Font.PLAIN**, **Font.ITALIC**, **Font.BOLD** o una qualsiasi combinazione di **Font.PLAIN**, **Font.ITALIC** e **Font.BOLD**.

Il metodo `getSize` ritorna la dimensione del carattere in punti, mentre `getName` ritorna il nome del tipo di carattere attuale sotto forma di **String**. Il metodo `getFamily` ritorna il nome della famiglia di tipi di caratteri a cui appartiene il tipo di carattere attuale (i nomi delle famiglie di tipi di caratteri sono specifici per ogni piattaforma).



Obiettivo portabilità 1.3

Ai fini della portabilità, Java usa dei nomi di tipi di caratteri standard riconducendoli (mappandoli) ai nomi di tipi di caratteri specifici per ogni sistema.

Anche i metodi della classe **Font** (riassunti nella figura 1.8) possono essere utilizzati per controllare lo stile del carattere attuale. Il metodo `isPlain` ritorna **true** se lo stile del carattere attuale è normale; il metodo `isBold` ritorna **true** se lo stile del carattere attuale è grassetto; il metodo `isItalic`, invece, ritorna **true** se lo stile del carattere attuale è corsivo.

A volte è necessario conoscere delle informazioni precise riguardo alla metrica di un carattere, come per esempio la sua *altezza*, oppure il suo *discendente* (la parte di carattere che si estende al di sotto della linea di base del carattere stesso), il suo *ascendente* (la parte di carattere che si estende al di sopra della linea di base) e l'*interlinea* (la differenza tra l'altezza e l'ascendente). La figura 1.10 mostra alcune delle metriche più comuni; come potete notare, la coordinata passata a **drawString** corrisponde all'angolo inferiore sinistro della linea di base del carattere.



Figura 1.10 Le metriche dei caratteri

La classe **FontMetrics** definisce vari metodi per ottenere le metriche di un carattere; questi metodi, insieme con il metodo **getFontMetrics** di **Graphics**, sono riassunti nella figura 1.11.

Metodo	Descrizione
public int getAscent()	// classe FontMetrics Ritorna un valore rappresentante l'ascendente di un carattere in punti.
public int getDescent()	// classe FontMetrics Ritorna un valore rappresentante il discendente di un carattere in punti.
public int getLeading()	// classe FontMetrics Ritorna un valore rappresentante l'interlinea di un carattere in punti.
public int getHeight()	// classe FontMetrics Ritorna un valore rappresentante l'altezza di un carattere in punti.
public FontMetrics getFontMetrics()	// classe Graphics Ritorna l'oggetto FontMetrics per l'attuale Font .
public FontMetrics getFontMetrics(Font f)	// classe Graphics Ritorna l'oggetto FontMetrics per l'argomento Font specificato.

Figura 1.11 I metodi FontMetrics e Graphics per ottenere le metriche dei caratteri

Il programma della figura 1.12 utilizza i metodi della figura 1.11 per ottenere le informazioni relative alle metriche di due caratteri.

```

1 // Fig. 1.12: Metrics.java
2 // Dimostrazione dei metodi della classe FontMetrics e
3 // della classe Graphics
4 import java.awt.*;
5 import java.awt.event.*;
6 import javax.swing.*;
7
8 public class Metrics extends JFrame {
9     public Metrics()
10    {
11        super( "Demonstrating FontMetrics" );
12
13        setSize( 510, 210 );
14        show();
15    }
16
17    public void paint( Graphics g )
18    {
19        g.setFont( new Font( "SansSerif", Font.BOLD, 12 ) );
20        FontMetrics fm = g.getFontMetrics();
21        g.drawString( "Current font: " + g.getFont(), 10, 40 );
22        g.drawString( "Ascent: " + fm.getAscent(), 10, 55 );
23        g.drawString( "Descent: " + fm.getDescent(), 10, 70 );
24        g.drawString( "Height: " + fm.getHeight(), 10, 85 );
25        g.drawString( "Leading: " + fm.getLeading(), 10, 100 );
26
27        Font font = new Font( «Serif», Font.ITALIC, 14 );
28        fm = g.getFontMetrics( font );
29        g.setFont( font );
30        g.drawString( «Current font: « + font, 10, 130 );
31        g.drawString( «Ascent: « + fm.getAscent(), 10, 145 );
32        g.drawString( "Descent: " + fm.getDescent(), 10, 160 );
33        g.drawString( "Height: " + fm.getHeight(), 10, 175 );
34        g.drawString( "Leading: " + fm.getLeading(), 10, 190 );
35    }
36
37    public static void main( String args[] )
38    {
39        Metrics app = new Metrics();
40
41        app.addWindowListener(
42            new WindowAdapter() {
43                public void windowClosing( WindowEvent e )
44                {
45                    System.exit( 0 );
46                }
47            }
48        );

```

Figura I.12 Come ottenere le informazioni riguardanti la metrica di un carattere (continua)

```

49     }
50     }

```

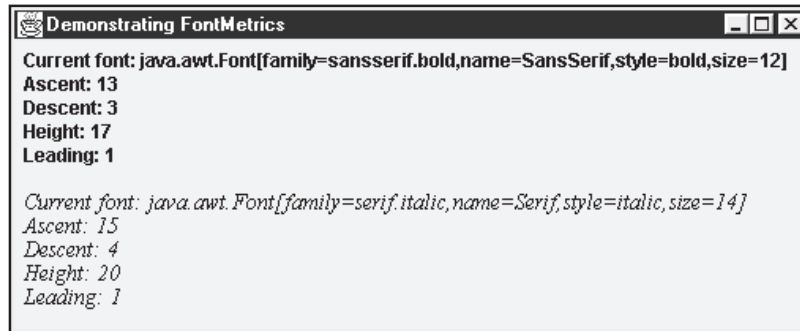


Figura I.12 Come ottenere le informazioni riguardanti la metrica di un carattere

La riga 19 crea e imposta il tipo di carattere di disegno attuale su **SansSerif**, grassetto, 12 punti. La riga 20 utilizza il metodo **getFontMetrics** di **Graphics** per ottenere l'oggetto **FontMetrics** relativo al tipo di carattere attuale.

La riga 21 utilizza una chiamata implicita al metodo **toString** della classe **Font** per mostrare la rappresentazione del tipo di carattere sotto forma di stringa. Le righe dalla 22 alla 25 utilizzano i metodi **FontMetric** per ottenere l'ascendente, il discendente, l'altezza e l'interlinea del tipo di carattere.

La riga 27 crea e imposta un nuovo tipo di carattere **Serif**, corsivo, a 14 punti. La riga 28 utilizza una seconda versione del metodo **getFontMetrics** di **Graphics**, il quale riceve un argomento **Font** e ritorna un oggetto **FontMetrics** corrispondente. Le righe dalla 31 alla 34 ottengono l'ascendente, il discendente, l'altezza e l'interlinea del carattere. Come potete notare, le metriche sono leggermente diverse per i due tipi di caratteri.

I.5 Disegnare linee, rettangoli e ovali

Questa sezione presenta una varietà di metodi **Graphics** per disegnare linee, rettangoli e ovali. Questi metodi, con i loro parametri, sono riassunti nella figura 1.13. Per ogni metodo che richiede un parametro **width** e **height**, i valori **width** e **height** non devono essere negativi, altrimenti la figura desiderata non verrà visualizzata.

Metodo	Descrizione
public void drawLine(int x1, int y1, int x2, int y2)	Disegna una linea tra il punto (x1, y1) e il punto (x2, y2).
public void drawRect(int x, int y, int width, int height)	Disegna un rettangolo dell'altezza e larghezza specificate. L'angolo superiore sinistro del rettangolo ha le coordinate (x, y).

Figura I.13 I metodi Graphics per disegnare linee, rettangoli e ovali (continua)

Metodo	Descrizione
public void fillRect(int x, int y, int width, int height)	Disegna un rettangolo pieno dell'altezza e larghezza specificate. L'angolo superiore sinistro del rettangolo ha le coordinate (x, y) .
public void clearRect(int x, int y, int width, int height)	Disegna un rettangolo pieno dell'altezza e larghezza specificate, mantenendo l'attuale colore di sfondo. L'angolo superiore sinistro del rettangolo ha le coordinate (x, y) .
public void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)	Disegna un rettangolo con angoli arrotondati nel colore attuale, e con l'altezza e la larghezza specificate. arcWidth e arcHeight determinano l'arrotondamento degli angoli (vedi figura 1.15).
public void fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)	Disegna un rettangolo pieno con angoli arrotondati nel colore attuale, e con l'altezza e la larghezza specificate. arcWidth e arcHeight determinano l'arrotondamento degli angoli (vedi figura 1.15).
public void draw3DRect(int x, int y, int width, int height, boolean b)	Disegna un rettangolo tridimensionale nel colore attuale e con l'altezza e la larghezza specificate. L'angolo superiore sinistro del rettangolo ha le coordinate (x, y) . Il rettangolo appare sollevato quando b è true , e abbassato quando b è false .
public void fill3DRect(int x, int y, int width, int height, boolean b)	Disegna un rettangolo tridimensionale pieno nel colore attuale e con l'altezza e la larghezza specificate. L'angolo superiore sinistro del rettangolo ha le coordinate (x, y) . Il rettangolo appare sollevato quando b è true , e abbassato quando b è false .
public void drawOval(int x, int y, int width, int height)	Disegna un ovale nel colore attuale, con l'altezza e la larghezza specificate. L'angolo superiore sinistro del rettangolo che circonda l'ovale ha le coordinate (x, y) . L'ovale tocca tutti i lati del rettangolo in corrispondenza del centro di ogni lato (figura 1.16).

Figura 1.13 I metodi Graphics per disegnare linee, rettangoli e ovali (continua)

Metodo	Descrizione
public void fillOval(int x, int y, int width, int height)	Disegna un ovale pieno nel colore attuale e con l'altezza e la larghezza specificate. L'angolo superiore sinistro del rettangolo che circonda l'ovale ha le coordinate (x , y). L'ovale tocca tutti e quattro i lati del rettangolo in corrispondenza del centro di ogni lato (vedi figura 1.16).

Figura 1.13 I metodi Graphics per disegnare linee, rettangoli e ovali

L'applicazione che viene mostrata nella figura 1.14 offre un esempio di come sia possibile disegnare una varietà di linee, rettangoli, rettangoli tridimensionali, rettangoli arrotondati e ovali.

```

1 // Fig. 1.14: LinesRectsOvals.java
2 // Disegnare linee, rettangoli e ovali
3 import java.awt.*;
4 import java.awt.event.*;
5 import javax.swing.*;
6
7 public class LinesRectsOvals extends JFrame {
8     private String s = "Using drawString!";
9
10    public LinesRectsOvals()
11    {
12        super( "Drawing lines, rectangles and ovals" );
13
14        setSize( 400, 165 );
15        show();
16    }
17
18    public void paint( Graphics g )
19    {
20        g.setColor( Color.red );
21        g.drawLine( 5, 30, 350, 30 );
22
23        g.setColor( Color.blue );
24        g.drawRect( 5, 40, 90, 55 );
25        g.fillRect( 100, 40, 90, 55 );
26
27        g.setColor( Color.cyan );
28        g.fillRoundRect( 195, 40, 90, 55, 50, 50 );
29        g.drawRoundRect( 290, 40, 90, 55, 20, 20 );
30
31        g.setColor( Color.yellow );
32        g.draw3DRect( 5, 100, 90, 55, true );
33        g.fill3DRect( 100, 100, 90, 55, false );

```

Figura 1.14 Alcuni metodi di disegno della classe Graphics (continua)

```

34
35     g.setColor( Color.magenta );
36     g.drawOval( 195, 100, 90, 55 );
37     g.fillOval( 290, 100, 90, 55 );
38 }
39
40 public static void main( String args[] )
41 {
42     LinesRectsOvals app = new LinesRectsOvals();
43
44     app.addWindowListener(
45         new WindowAdapter() {
46             public void windowClosing( WindowEvent e )
47             {
48                 System.exit( 0 );
49             }
50         }
51     );
52 }
53 }

```

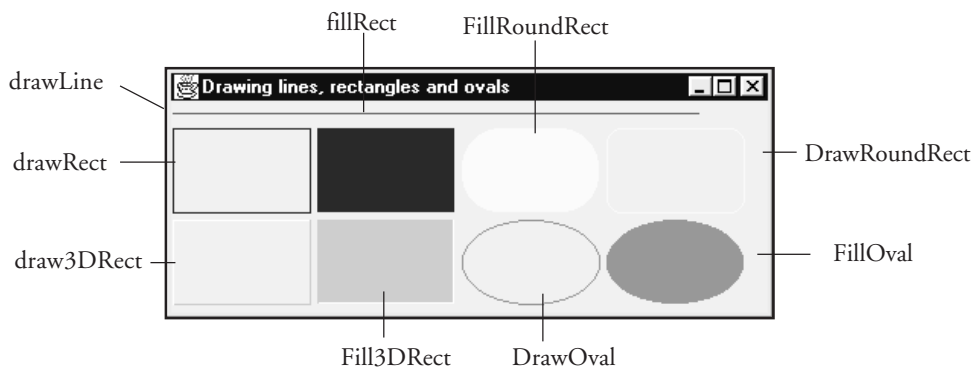


Figura I.14 Alcuni metodi di disegno della classe Graphics

I metodi **fillRoundRect** (riga 28) e **drawRoundRect** (riga 29) disegnano dei rettangoli con gli angoli arrotondati. I loro primi due argomenti specificano le coordinate dell'angolo superiore sinistro del rettangolo che circonda la figura, ovvero l'area in cui il rettangolo arrotondato verrà disegnato.

Notate come le coordinate dell'angolo superiore sinistro non corrispondano ai margini del rettangolo arrotondato, bensì alle coordinate in cui i margini si troverebbero se il rettangolo avesse degli angoli squadrati. Il terzo e il quarto argomento specificano i valori **width** e **height** del rettangolo; i loro due ultimi argomenti (**arcWidth** e **arcHeight**) determinano, invece, i diametri orizzontali e verticali degli archi utilizzati per rappresentare gli angoli.

I metodi **draw3DRect** (riga 32) e **fill3DRect** (riga 33) prendono gli stessi argomenti. I primi due argomenti specificano l'angolo superiore sinistro del rettangolo, mentre i due argomenti successivi specificano rispettivamente **width** e **height**. L'ultimo argomento determina se il rettangolo è sollevato (**true**) o abbassato (**false**).

L'effetto tridimensionale di **draw3DRect** appare come due margini del rettangolo nel colore originale e due margini in un colore leggermente più scuro. I rettangoli sollevati hanno i margini superiore e sinistro nel colore originale, mentre i rettangoli abbassati hanno i margini inferiore e destro nel colore originale. L'effetto tridimensionale è difficile da vedere con determinati colori.

La figura 1.15 mostra la larghezza e l'altezza di un rettangolo arrotondato, così come la larghezza e l'altezza dei suoi archi. Utilizzando lo stesso valore per **arcWidth** e **arcHeight** viene prodotto un quarto di cerchio ad ogni angolo. Quando **width**, **height**, **arcWidth** e **arcHeight** hanno gli stessi valori, il risultato è un cerchio. Se i valori di **width** e **height** sono gli stessi, mentre i valori di **arcWidth** e **arcHeight** sono 0, il risultato è un quadrato.

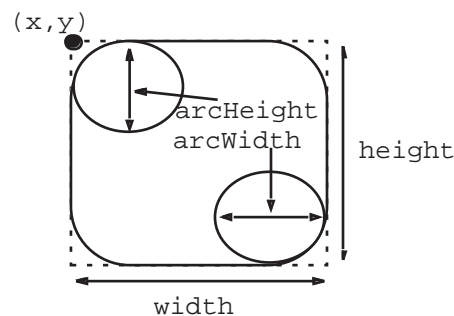


Figura 1.15 La larghezza e l'altezza degli archi dei rettangoli arrotondati

Entrambi i metodi **drawOval** e **fillOval** prendono gli stessi quattro argomenti. I primi due argomenti specificano la coordinata superiore sinistra del rettangolo che circonda l'ovale. Gli ultimi due argomenti specificano rispettivamente **width** e **height** del rettangolo che circonda l'ovale.

La figura 1.16 mostra un ovale circondato da un rettangolo; notate come l'ovale tocchi il centro di tutti e quattro i lati del rettangolo (il rettangolo non appare sullo schermo).

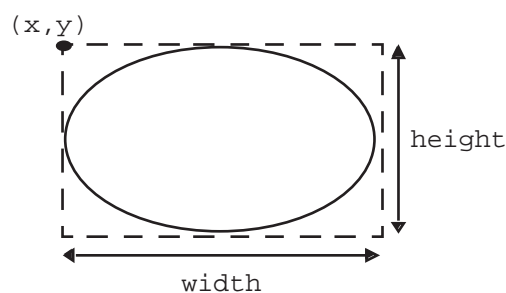


Figura 1.16 Un ovale circondato da un rettangolo

1.6 Disegnare degli archi

Un arco è una parte di un ovale, e i suoi angoli vengono misurati in gradi. L'arco si estende da un angolo di partenza per un numero determinato di gradi. L'angolo dell'arco specifica il numero totale di gradi per cui l'arco si estende.

La figura 1.17 mostra due archi; il gruppo di assi a sinistra mostra un arco che si estende da zero gradi a circa 110 gradi.

Gli archi che si estendono in direzione antioraria sono misurati in gradi positivi. Il gruppo di assi a destra mostra un arco che si estende da zero gradi a circa 110 gradi. Gli archi che si estendono in direzione oraria sono misurati in gradi negativi.

Osservate i riquadri tratteggiati intorno agli archi della figura 1.17: quando si disegna un arco, viene specificato un rettangolo che circonda un ovale. L'arco si estende lungo parte di quell'ovale. I metodi **drawArc** e **fillArc** di **Graphics** sono riassunti nella figura 1.18.

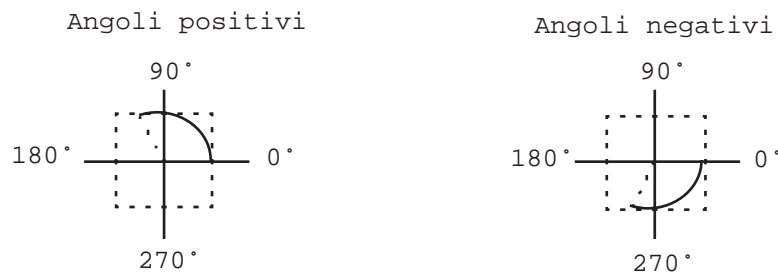


Figura 1.17 Angoli di archi positivi e negativi

Metodo	Descrizione
<pre>public void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)</pre>	<p>Disegna un arco corrispondente ai margini delle coordinate superiori sinistre (x, y) del rettangolo che circonda la figura, con l'altezza e la larghezza specificate. Il segmento dell'arco è disegnato partendo da startAngle, e si estende per arcAngle gradi.</p>
<pre>public void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)</pre>	<p>Disegna un arco pieno (ovvero un settore) corrispondente ai margini delle coordinate superiori sinistre (x, y) del rettangolo che circonda la figura, con l'altezza e la larghezza specificate. Il segmento dell'arco è disegnato partendo da startAngle, e si estende per arcAngle gradi.</p>

Figura 1.18 I metodi Graphics per disegnare degli archi

Il programma della figura 1.19 mostra i metodi della figura 1.18. Questo programma disegna sei archi (tre vuoti e tre pieni). Per mostrare il rettangolo che aiuta a determinare il posizionamento degli archi, i primi tre archi vengono visualizzati all'interno di un rettangolo giallo con gli stessi argomenti **x**, **y**, **width** e **height** degli archi.

```
1 // Fig. 1.19: DrawArcs.java
2 // Disegnare archi
3 import java.awt.*;
4 import javax.swing.*;
5 import java.awt.event.*;
6
7 public class DrawArcs extends JFrame {
8     public DrawArcs()
9     {
10         super( "Drawing Arcs" );
11
12         setSize( 300, 170 );
13         show();
14     }
15
16     public void paint( Graphics g )
17     {
18         // parte da 0 e si estende per 360 gradi
19         g.setColor( Color.yellow );
20         g.drawRect( 15, 35, 80, 80 );
21         g.setColor( Color.black );
22         g.drawArc( 15, 35, 80, 80, 0, 360 );
23
24         // parte da 0 e si estende per 10 gradi
25         g.setColor( Color.yellow );
26         g.drawRect( 100, 35, 80, 80 );
27         g.setColor( Color.black );
28         g.drawArc( 100, 35, 80, 80, 0, 10 );
29
30         // parte da 0 e si estende per -270 gradi
31         g.setColor( Color.yellow );
32         g.drawRect( 185, 35, 80, 80 );
33         g.setColor( Color.black );
34         g.drawArc( 185, 35, 80, 80, 0, -270 );
35
36         // parte da 0 e si estende per 360 gradi
37         g.fillArc( 15, 120, 80, 40, 0, 360 );
38
39         // parte da 270 e si estende per -90 gradi
40         g.fillArc( 100, 120, 80, 40, 270, -90 );
41
42         // parte da 0 e si estende per -270 gradi
43         g.fillArc( 185, 120, 80, 40, 0, -270 );
44     }
45
46     public static void main( String args[] )
47     {
48         DrawArcs app = new DrawArcs();
```

Figura I.19 La dimostrazione di drawArc e fillArc (continua)

```

49
50     app.addWindowListener(
51         new WindowAdapter() {
52             public void windowClosing( WindowEvent e )
53             {
54                 System.exit( 0 );
55             }
56         }
57     );
58 }
59 }

```

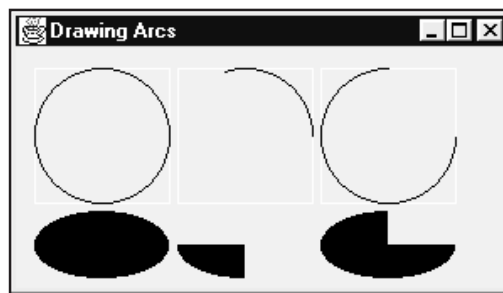


Figura 1.19 La dimostrazione di `drawArc` e `fillArc`

1.7 Disegnare poligoni e polilinee

I poligoni sono delle forme con più lati, mentre le polilinee sono una serie di punti collegati tra di loro. I metodi **Graphics** per disegnare poligoni e polilinee vengono presi in esame nella figura 1.20. Notate come alcuni metodi richiedano un oggetto **Polygon** (package **java.awt**). Anche i costruttori della classe **Polygon** vengono descritti nella figura 1.20. Il programma della figura 1.21 disegna dei poligoni e delle polilinee utilizzando i metodi e i costruttori della figura 1.20.

Le righe dalla 18 alla 20 creano due array di **int**, utilizzandoli per specificare i punti di **Polygon poly1**. Il costruttore **Polygon** chiamato alla riga 20 riceve l'array **xValues**, che contiene la coordinata **x** di ogni punto, l'array **yValue**, che contiene la coordinata **y** di ogni punto, e 6 (il numero di punti presenti nel poligono). La riga 22 visualizza **poly1** passandolo come argomento al metodo **drawPolygon** di **Graphics**.

Metodo	Descrizione
<pre>public void drawPolygon(int xPoints[], int yPoints[], int points)</pre>	<p>Disegna un poligono. La coordinata <i>x</i> di ogni punto è specificata nell'array xPoints, mentre la coordinata <i>y</i> di ogni punto è specificata nell'array yPoints. L'ultimo argomento specifica il numero di punti. Il metodo disegna un poligono chiuso, anche se l'ultimo punto è diverso dal primo.</p>

Figura 1.20 I metodi **Graphics** per i poligoni e i costruttori della classe **Polygon** (continua)

Metodo	Descrizione
public void drawPolyline(int xPoints[], int yPoints[], int points)	Disegna una serie di linee collegate tra di loro. La coordinata <i>x</i> di ogni punto è specificata nell'array xPoints , mentre la coordinata <i>y</i> di ogni punto è specificata nell'array yPoints . L'ultimo argomento specifica il numero di punti. Se l'ultimo punto è diverso dal primo, la polilinea non è chiusa.
public void drawPolygon(Polygon p)	Disegna il poligono chiuso specificato.
public void fillPolygon(int xPoints[], int yPoints[], int points)	Disegna un poligono pieno. La coordinata <i>x</i> di ogni punto è specificata nell'array xPoints , mentre la coordinata <i>y</i> di ogni punto è specificata nell'array yPoints . L'ultimo argomento specifica il numero di punti. Questo metodo disegna un poligono chiuso, anche se l'ultimo punto è diverso dal primo.
public void fillPolygon(Polygon p)	Disegna il poligono pieno specificato. Il poligono è chiuso.
public Polygon()	// Polygon class Costruisce un nuovo oggetto poligono, che non contiene punti.
public Polygon(int xValues[], int yValues[], int numberOfPoints)	// Polygon class Costruisce un nuovo oggetto poligono. Il poligono ha numberOfPoints lati, dove ogni punto consiste di una coordinata <i>x</i> di xValues e una coordinata <i>y</i> di yValues .

Figura I.20 I metodi Graphics per i poligoni e i costruttori della classe Polygon

```

1 // Disegnare poligoni
2 import java.awt.*;
3 import java.awt.event.*;
4 import javax.swing.*;
5
6 public class DrawPolygons extends JFrame {
7     public DrawPolygons()
8     {
9         super( "Drawing Polygons" );
10
11         setSize( 275, 230 );
12         show();

```

Figura I.21 La dimostrazione di drawPolygon e fillPolygon (continua)

```
13     }
14
15     public void paint( Graphics g )
16     {
17         int xValues[] = { 20, 40, 50, 30, 20, 15 };
18         int yValues[] = { 50, 50, 60, 80, 80, 60 };
19         Polygon poly1 = new Polygon( xValues, yValues, 6 );
20
21         g.drawPolygon( poly1 );
22
23         int xValues2[] = { 70, 90, 100, 80, 70, 65, 60 };
24         int yValues2[] = { 100, 100, 110, 110, 130, 110, 90 };
25
26         g.drawPolyline( xValues2, yValues2, 7 );
27
28         int xValues3[] = { 120, 140, 150, 190 };
29         int yValues3[] = { 40, 70, 80, 60 };
30
31         g.fillPolygon( xValues3, yValues3, 4 );
32
33         Polygon poly2 = new Polygon();
34         poly2.addPoint( 165, 135 );
35         poly2.addPoint( 175, 150 );
36         poly2.addPoint( 270, 200 );
37         poly2.addPoint( 200, 220 );
38         poly2.addPoint( 130, 180 );
39
40         g.fillPolygon( poly2 );
41     }
42
43     public static void main( String args[] )
44     {
45         DrawPolygons app = new DrawPolygons();
46
47         app.addWindowListener(
48             new WindowAdapter() {
49                 public void windowClosing( WindowEvent e )
50                 {
51                     System.exit( 0 );
52                 }
53             }
54         );
55     }
56 }
```

Figura I.21 La dimostrazione di drawPolygon e fillPolygon (continua)

Le righe 24 e 25 creano due array di **int**, utilizzandoli per specificare i punti di una serie di linee collegate. L'array **xValues2** contiene la coordinata *x* di ogni punto, mentre l'array **yValues2** contiene la coordinata *y* di ogni punto.

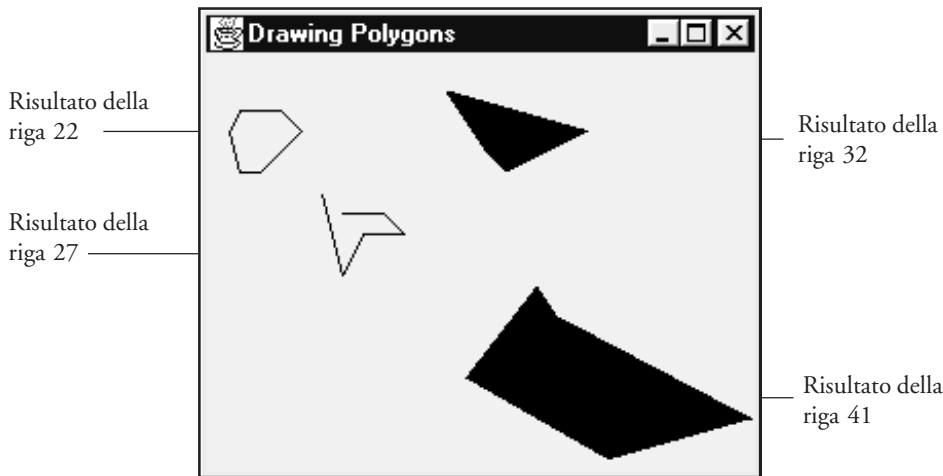


Figura 1.21 La dimostrazione di `drawPolygon` e `fillPolygon`

La riga 27 utilizza il metodo `drawPolyline` di `Graphics` per visualizzare la serie di linee collegate specificata dagli argomenti `xValues2`, `yValues2` e `7` (il numero di punti).

Le righe 29 e 30 creano due array di `int`, utilizzandoli per specificare i punti di un poligono. L'array `xValues3` contiene la coordinata x di ogni punto, mentre l'array `yValues3` contiene la coordinata y di ogni punto. La riga 32 visualizza un poligono passando al metodo `fillPolygon` di `Graphics` i due array (`xValues3` e `yValues3`), oltre al numero di punti da disegnare (`4`).



Errore tipico 1.3

Una `ArrayIndexOutOfBoundsException` viene lanciata se il numero di punti specificati nel terzo argomento del metodo `drawPolygon` o del metodo `fillPolygon` è maggiore del numero di elementi contenuti nell'array di coordinate che definiscono il poligono da visualizzare.

1.8 Java2D API

La nuova *Java2D API* offre due funzionalità avanzate di disegno bidimensionale, utili ai programmatori che utilizzano immagini complesse e molto dettagliate. L'API contiene delle funzionalità che permettono di elaborare le linee, i testi e le immagini dei package `java.awt`, `java.awt.image`, `java.awt.color`, `java.awt.font`, `java.awt.geom`, `java.awt.print` e `java.awt.image.renderable`. Le capacità di questa API non possono essere prese in esame completamente da questo libro, e si consiglia quindi di dare uno sguardo anche alla demo Java2D (capitolo 3 del volume Fondamenti) per una panoramica delle varie caratteristiche.

Java2D API utilizza un'istanza della classe `Graphics2D` (package `java.awt`). La classe `Graphics2D` è una sottoclasse della classe `Graphics`, quindi possiede tutte le capacità viste all'inizio di questo capitolo. In realtà, l'oggetto utilizzato per disegnare con ogni metodo `paint` era proprio un oggetto della classe `Graphics2D` passato al metodo `paint`, al quale si ha accesso attraverso il riferimento `g` della superclasse `Graphics`. Per accedere alle

funzionalità di **Graphics2D**, è necessario trasformare il riferimento **Graphics** passato a **paint** in un riferimento **Graphics2D** con un'istruzione di questo tipo

```
graphics2D g2d = ( Graphics2D ) g;
```

Il programma delle prossime sezioni utilizza proprio questa tecnica.

1.9 Le forme Java2D

In questa sezione verranno presentate alcune delle forme del package **java.awt.geom**, tra cui **Ellipse2D.Double**, **Rectangle2D.Double**, **RoundRectangle2D.Double**, **Arc2D.Double** e **Line2D.Double**. Notate la sintassi di ognuno di questi nomi. Ognuna di queste classi rappresenta una forma con le dimensioni specificate sotto forma di valori a virgola mobile con precisione doppia; esiste però anche una versione di ognuna rappresentata con valori a virgola mobile con precisione singola (come **Ellipse2D.Float**). In ogni caso, **Double** è una classe **static** interna della classe alla sinistra dell'operatore punto (per esempio, **Ellipse2D**). Per utilizzare la classe interna **static**, è sufficiente qualificarne il nome con il nome della classe esterna.

Il programma della figura 1.22 mostra varie forme Java2D e varie caratteristiche di disegno, come per esempio le linee spesse, il riempimento delle figure e le linee tratteggiate. Queste sono soltanto alcune delle tante capacità offerte da Java2D.

```

1 // Fig. 1.22: Shapes.java
2 // Alcune delle forme di Java2D
3 import javax.swing.*;
4 import java.awt.event.*;
5 import java.awt.*;
6 import java.awt.geom.*;
7 import java.awt.image.*;
8
9 public class Shapes extends JFrame {
10     public Shapes()
11     {
12         super( "Drawing 2D shapes" );
13
14         setSize( 425, 160 );
15         show();
16     }
17
18     public void paint( Graphics g )
19     {
20         // crea 2D convertendo g in Graphics2D
21         Graphics2D g2d = ( Graphics2D ) g;
22
23         // disegna ellissi 2D riempite con una tinta blu-gialla
24         g2d.setPaint(
25             new GradientPaint( 5, 30, // x1, y1

```

Figura 1.22 Alcune delle forme Java 2D (continua)

```
26             Color.blue,    // Color iniziale
27             35, 100,       // x2, y2
28             Color.yellow,  // Color finale
29             true );       // ciclico
30 g2d.fill( new Ellipse2D.Double( 5, 30, 65, 100 ) );
31
32 // disegna un rettangolo 2D rosso
33 g2d.setPaint( Color.red );
34 g2d.setStroke( new BasicStroke( 10.0f ) );
35 g2d.draw(
36     new Rectangle2D.Double( 80, 30, 65, 100 ) );
37
38 // disegna rettangolo 2D arrotondato con motivo di sfondo
39 BufferedImage buffImage =
40     new BufferedImage(
41         10, 10, BufferedImage.TYPE_INT_RGB );
42
43 Graphics2D gg = buffImage.createGraphics();
44 gg.setColor( Color.yellow ); // disegna in giallo
45 gg.fillRect( 0, 0, 10, 10 ); // disegna rettangolo pieno
46 gg.setColor( Color.black ); // disegna in nero
47 gg.drawRect( 1, 1, 6, 6 ); // disegna un rettangolo
48 gg.setColor( Color.blue ); // disegna in blu
49 gg.fillRect( 1, 1, 3, 3 ); // disegna rettangolo pieno
50 gg.setColor( Color.red ); // disegna in rosso
51 gg.fillRect( 4, 4, 3, 3 ); // disegna rettangolo pieno
52
53 // disegna buffImage sul JFrame
54 g2d.setPaint(
55     new TexturePaint(
56         buffImage, new Rectangle( 10, 10 ) ) );
57 g2d.fill(
58     new RoundRectangle2D.Double(
59         155, 30, 75, 100, 50, 50 ) );
60
61 // disegna un arco 2D in bianco
62 g2d.setPaint( Color.white );
63 g2d.setStroke( new BasicStroke( 6.0f ) );
64 g2d.draw(
65     new Arc2D.Double(
66         240, 30, 75, 100, 0, 270, Arc2D.PIE ) );
67
68 // disegna linee 2D in verde e giallo
69 g2d.setPaint( Color.green );
70 g2d.draw( new Line2D.Double( 395, 30, 320, 150 ) );
71
72 float dashes[] = { 10 };
73
```

Figura I.22 Alcune delle forme Java 2D (continua)

```

74         g2d.setPaint( Color.yellow );
75         g2d.setStroke(
76             new BasicStroke( 4,
77                 BasicStroke.CAP_ROUND,
78                 BasicStroke.JOIN_ROUND,
79                 10, dashes, 0 ) );
80         g2d.draw( new Line2D.Double( 320, 30, 395, 150 ) );
81     }
82
83     public static void main( String args[] )
84     {
85         Shapes app = new Shapes();
86
87         app.addWindowListener(
88             new WindowAdapter() {
89                 public void windowClosing( WindowEvent e )
90                 {
91                     System.exit( 0 );
92                 }
93             }
94         );
95     }
96 }

```

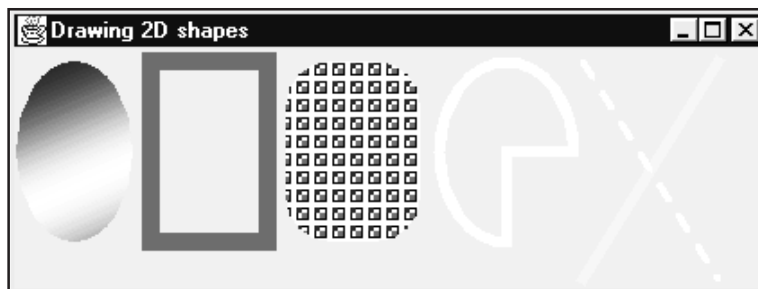


Figura I.22 Alcune delle forme Java 2D

La riga 21 trasforma il riferimento **Graphics** ricevuto da **paint** in un riferimento **Graphics2D**, e lo assegna a **g2d** per consentire l'accesso alle funzionalità di **Java2D**.

La prima forma che viene disegnata è un ovale pieno di colori che cambiano in modo graduale. Le righe 24-29

```

g2d.setPaint(
    new GradientPaint( 5, 30,          // x1, y1
                     Color.blue,     // Color iniziale
                     35, 100,        // x2, y2
                     Color.yellow,   // Color finale
                     true ) );       // ciclico

```

invocano il metodo **setPaint** di **Graphics2D** per impostare l'oggetto **Paint** che determina il colore della forma da visualizzare. Un oggetto **Paint** è un oggetto di una qualsiasi

classe che implementa l'interfaccia **java.awt.Paint**. L'oggetto **Paint** può essere semplice come uno degli oggetti **Color** predefiniti introdotti nella sezione 1.3 (la classe **Color** implementa **Paint**), oppure può anche essere un'istanza delle classi **GradientPaint**, **SystemColor** o **TexturePaint** di Java2D API. In questo caso, utilizziamo un oggetto **GradientPaint**.

La classe **GradientPaint** aiuta a disegnare una forma con colori che cambiano in modo graduale. Il costruttore **GradientPaint** qui utilizzato richiede sette argomenti; i primi due argomenti specificano la coordinata di partenza per la gradazione del colore, il terzo argomento specifica il **Color** di partenza della gradazione, il quarto e il quinto specificano la coordinata finale della gradazione, mentre il sesto argomento specifica il **Color** finale. L'ultimo argomento specifica se la gradazione è ciclica (**true**) o aciclica (**false**), mentre le due coordinate ne determinano la direzione.

Dato che la seconda coordinata (35, 100) è in basso e a destra rispetto alla prima coordinata (5, 30), la gradazione si estenderà verso il basso e verso destra. Dato che questa gradazione è ciclica (**true**), il colore parte con il blu e diventa gradatamente giallo, per poi gradatamente ritornare blu. Se la gradazione fosse aciclica, il colore passerebbe soltanto dal primo colore specificato (blu) al secondo colore (giallo).

La riga 30

```
g2d.fill( new Ellipse2D.Double( 5, 30, 65, 100 ) );
```

utilizza il metodo **fill** di **Graphics2D** per disegnare un oggetto **Shape** pieno. L'oggetto **Shape** è un'istanza di una qualsiasi classe che implementa l'interfaccia **Shape** (package **java.awt**); in questo caso, si tratta di un'istanza della classe **Ellipse2D.Double**. Il costruttore **Ellipse2D.Double** riceve quattro argomenti che specificano il rettangolo che circonda l'ellisse da visualizzare.

A questo punto, viene disegnato un rettangolo con un bordo spesso. La riga 33 usa **setPaint** per impostare l'oggetto **Paint** su **Color.red**. La riga 34

```
g2d.setStroke( new BasicStroke( 10.0f ) );
```

usa il metodo **setStroke** di **Graphics2D** per impostare le caratteristiche dei bordi del rettangolo (o delle linee nel caso di qualsiasi altra forma). Il metodo **setStroke** richiede un oggetto **Stroke** come argomento. L'oggetto **Stroke** è un'istanza di una qualsiasi classe che implementa l'interfaccia **Stroke** (package **java.awt**); in questo caso, si tratta di un'istanza della classe **BasicStroke**.

La classe **BasicStroke** fornisce una varietà di costruttori per specificare la larghezza della linea, il modo in cui questa termina, il modo in cui due linee si uniscono, e gli attributi della tratteggiatura della linea (nel caso si tratti di una linea tratteggiata). Il costruttore specifica qui che la linea dovrebbe essere larga 10 pixel.

Le righe 35 e 36

```
g2d.draw(
    new Rectangle2D.Double( 80, 30, 65, 100 ) );
```

usano il metodo **draw** di **Graphics2D** per disegnare un oggetto **Shape**; in questo caso, un'istanza della classe **Rectangle2D.Double**. Il costruttore **Rectangle2D.Double** riceve quattro argomenti che specificano la coordinata x superiore sinistra, la coordinata y superiore sinistra, la larghezza e l'altezza del rettangolo.

Per proseguire, viene disegnato un rettangolo arrotondato riempito con una trama creata in un oggetto **BufferedImage** (package **java.awt.image**). Le righe 39-41

```
BufferedImage buffImage =
    new BufferedImage(
        10, 10, BufferedImage.TYPE_INT_RGB );
```

creano l'oggetto **BufferedImage**. La classe **BufferedImage** può essere utilizzata per produrre delle immagini a colori e nella scala dei grigi. Questo particolare **BufferedImage** è largo 10 pixel e alto 10 pixel. L'argomento **BufferedImage.TYPE_INT_RGB** indica che l'immagine viene memorizzata a colori, utilizzando lo schema di colori RGB.

Per creare la trama di riempimento del rettangolo arrotondato, è necessario per prima cosa disegnare in **BufferedImage**. La riga 43

```
Graphics2D gg = buffImage.createGraphics();
```

crea un oggetto **Graphics2D** che può essere usato per disegnare all'interno del **BufferedImage**. Le righe dalla 44 alla 51 usano i metodi **setColor**, **fillRect** e **drawRect** (visti all'inizio del capitolo) per creare la trama.

Le righe 54-56

```
g2d.setPaint(
    new TexturePaint(
        buffImage, new Rectangle( 10, 10 ) ) );
```

impostano l'oggetto **Paint** su di un nuovo oggetto **TexturePaint** (package **java.awt**). Un oggetto **TexturePaint** utilizza quale riempimento l'immagine memorizzata nel suo **BufferedImage** associato. Il secondo argomento specifica l'area **Rectangle** del **BufferedImage** che verrà replicato all'interno del riempimento. In questo caso, il **Rectangle** ha la stessa dimensione del **BufferedImage**, ma è possibile anche utilizzarne una porzione più piccola.

Le righe dalla 57 alla 59

```
g2d.fill(
    new RoundRectangle2D.Double(
        155, 30, 75, 100, 50, 50 ) );
```

usano il metodo **fill** di **Graphics2D** per disegnare un oggetto **Shape** (in questo caso un'istanza della classe **RoundRectangle2D.Double**). Il costruttore **RoundRectangle2D.Double** riceve sei argomenti che specificano le dimensioni del rettangolo e la larghezza e altezza dell'arco usati per determinare l'arrotondamento degli angoli.

Per proseguire, viene disegnato un arco a forma di torta, con una linea bianca spessa. La riga 62 imposta l'oggetto **Paint** su **Color.white**. La riga 63 imposta l'oggetto **Stroke** su di un nuovo **BasicStroke** per una linea larga 6 pixel.

Le righe 64-66

```
g2d.draw(
    new Arc2D.Double(
        240, 30, 75, 100, 0, 270, Arc2D.PIE ) );
```

usano il metodo **draw** di **Graphics2D** per disegnare un oggetto **Shape**; in questo caso, un **Arc2D.Double**. I primi quattro argomenti del costruttore **Arc2D.Double** specificano la

coordinata x superiore sinistra, la coordinata y superiore sinistra, la larghezza e l'altezza del rettangolo che circonda l'arco. Il quinto argomento specifica l'angolo di partenza, mentre il sesto argomento specifica l'angolo dell'arco. L'ultimo argomento specifica il modo in cui l'arco si chiude. Le costanti **Arc2D.PIE** indicano che l'arco è chiuso da due linee; una linea va dal punto di partenza dell'arco al centro del rettangolo che circonda l'arco, mentre l'altra linea va dal centro del rettangolo fino al punto di fine.

La classe **Arc2D** fornisce due altre costanti **static** per specificare il modo in cui l'arco si chiude; la costante **Arc2D.CHORD** disegna una linea dal punto di partenza al punto di fine, mentre la costante **Arc2D.OPEN** specifica che l'arco non è chiuso.

Per finire, vengono disegnate due linee utilizzando gli oggetti **Line2D** (uno pieno e uno tratteggiato). La riga 69 imposta l'oggetto **Paint** su **Color.green**. La riga 70

```
g2d.draw( new Line2D.Double( 395, 30, 320, 150 ) );
```

usa il metodo **draw** di **Graphics2D** per disegnare un oggetto **Shape**; in questo caso un'istanza della classe **Line2D.Double**. Gli argomenti del costruttore **Line2D.Double** specificano le coordinate di partenza e di fine della linea.

La riga 72 definisce un array di **float** contenente il valore 10. Questo array sarà utilizzato per descrivere i tratteggi della linea tratteggiata; in questo caso, ogni tratteggio sarà lungo 10 pixel. Per creare tratteggi di diverse lunghezze all'interno di una trama, basta semplicemente fornire la lunghezza di ogni tratteggio quale elemento dell'array. La riga 74 imposta l'oggetto **Paint** su **Color.yellow**. Le righe 75-79

```
g2d.setStroke(
    new BasicStroke( 4,
                    BasicStroke.CAP_ROUND,
                    BasicStroke.JOIN_ROUND,
                    10, dashes, 0 ) );
```

impostano l'oggetto **Stroke** su di un nuovo **BasicStroke**. La linea sarà larga 4 pixel e avrà le estremità arrotondate (**BasicStroke.CAP_ROUND**). Se le linee si uniscono (come avviene agli angoli di un rettangolo), il loro punto di unione sarà arrotondato (**BasicStroke.JOIN_ROUND**). L'argomento **dashes** specifica la lunghezza dei tratteggi della linea. L'ultimo argomento indica il pedice di inizio dell'array **dashes** per il primo tratteggio della trama. La riga 80 disegna poi una linea con l'attuale **Stroke**.

Un *general path* è una forma costituita di linee diritte e curve complesse, ed è rappresentato con un oggetto della classe **GeneralPath** (package **java.awt.geom**). Il programma della figura 1.23 disegna un general path con la forma di una stella a cinque punte.

```
1 // Fig. 1.23: Shapes2.java
2 // Dimostrazione di un general path
3 import javax.swing.*;
4 import java.awt.event.*;
5 import java.awt.*;
6 import java.awt.geom.*;
7
8 public class Shapes2 extends JFrame {
```

Figura 1.23 Un esempio di **GeneralPath** (continua)

```

9      public Shapes2()
10     {
11         super( "Drawing 2D Shapes" );
12
13         setBackground( Color.yellow );
14         setSize( 400, 400 );
15         show();
16     }
17
18     public void paint( Graphics g )
19     {
20         int xPoints[] =
21             { 55, 67, 109, 73, 83, 55, 27, 37, 1, 43 };
22         int yPoints[] =
23             { 0, 36, 36, 54, 96, 72, 96, 54, 36, 36 };
24
25         Graphics2D g2d = ( Graphics2D ) g;
26
27         // crea una stella da una serie di punti
28         GeneralPath star = new GeneralPath();
29
30         // imposta la coordinata iniziale del General Path
31         star.moveTo( xPoints[ 0 ], yPoints[ 0 ] );
32
33         // crea la stella—non la disegna
34         for ( int k = 1; k < xPoints.length; k++ )
35             star.lineTo( xPoints[ k ], yPoints[ k ] );
36
37         // chiude la forma
38         star.closePath();
39
40         // traduce l'origine in (200, 200)
41         g2d.translate( 200, 200 );
42
43         // disegna le stelle con colori casuali intorno all'origine
44         for ( int j = 1; j <= 20; j++ ) {
45             g2d.rotate( Math.PI / 10.0 );
46             g2d.setColor(
47                 new Color( ( int ) ( Math.random() * 256 ),
48                             ( int ) ( Math.random() * 256 ),
49                             ( int ) ( Math.random() * 256 ) ) );
50             g2d.fill( star ); // draw a filled star
51         }
52     }
53
54     public static void main( String args[] )
55     {
56         Shapes2 app = new Shapes2();

```

Figura I.23 Un esempio di GeneralPath (continua)

```

57
58     app.addWindowListener(
59         new WindowAdapter() {
60             public void windowClosing( WindowEvent e )
61                 {
62                 System.exit( 0 );
63             }
64         }
65     );
66 }
67 }

```

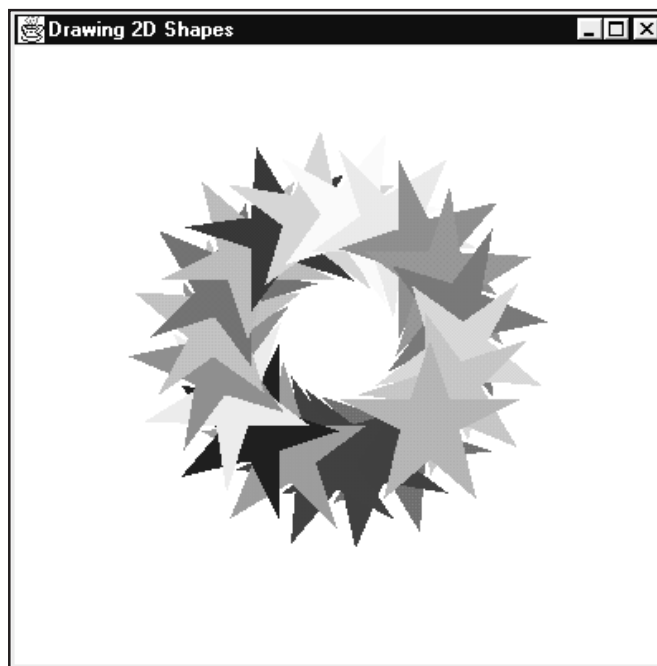


Figura I.23 Un esempio di `GeneralPath`

Le righe 20-23 definiscono due array di `int` rappresentanti le coordinate x e y dei punti della stella. La riga 28

```
GeneralPath star = new GeneralPath();
```

definisce l'oggetto `star` di `GeneralPath`.

La riga 31

```
star.moveTo( xPoints[ 0 ], yPoints[ 0 ] );
```

usa il metodo `moveTo` di `GeneralPath` per specificare il primo punto di `star`. La struttura `for` delle linee 34 e 35

```
for ( int k = 1; k < xPoints.length; k++ )
    star.lineTo( xPoints[ k ], yPoints[ k ] );
```

usa il metodo **lineTo** di **GeneralPath** per disegnare una linea verso il punto successivo di **star**. Ogni nuova chiamata a **LineTo** disegna una linea che parte dal punto precedente verso il punto attuale. La riga 38

```
star.closePath();
```

usa il metodo **closePath** di **GeneralPath** per disegnare una linea dall'ultimo punto al punto specificato nell'ultima chiamata a **moveTo**. Ora il general path è completo.

La riga 41

```
g2d.translate( 200, 200 );
```

usa il metodo **translate** di **Graphics2D** per spostare l'origine del disegno alla posizione (200, 200). Tutte le operazioni di disegno utilizzano ora la posizione (200, 200) come (0, 0).

La struttura **for** della riga 44 disegna questa **star** 20 volte, ruotandola intorno al nuovo punto di origine. La riga 45

```
g2d.rotate( Math.PI / 10.0 );
```

usa il metodo **rotate** di **Graphics2D** per ruotare la forma successiva. L'argomento specifica l'angolo di rotazione in radianti (dove $360^\circ = 2\pi$ radianti). La riga 50 usa il metodo **fill** di **Graphics2D** per disegnare una versione piena di questa **star**.

Esercizi di autovalutazione

- 1.1 Completate le seguenti frasi:
- In Java2D, il metodo _____ della classe _____ imposta le caratteristiche di una linea utilizzata per disegnare una forma.
 - La classe _____ aiuta a definire il riempimento di una forma, come per esempio un riempimento che passa gradatamente da un colore ad un altro.
 - Il metodo _____ della classe **Graphics** disegna una linea tra due punti.
 - RGB è l'abbreviazione di _____, _____ e _____.
 - Le dimensioni dei caratteri sono misurate in unità chiamate _____.
 - La classe _____ aiuta a definire il riempimento di una forma usando una trama disegnata in un **BufferedImage**.
- 1.2 Determinate se le seguenti affermazioni sono *vere* o *false*. Se sono *false*, spiegate il perché.
- I primi due argomenti del metodo **drawOval** di **Graphics** specificano la coordinata centrale dell'ovale.
 - Nel sistema di coordinate Java, i valori x aumentano da sinistra verso destra.
 - Il metodo **fillPolygon** disegna un poligono pieno con il colore attuale.
 - Il metodo **drawArc** permette di avere angoli negativi.
 - Il metodo **getSize** ritorna la dimensione in centimetri del carattere attuale.
 - Le coordinate dei pixel (0, 0) si trovano al centro esatto dello schermo.
- 1.3 Trovate gli errori in ognuna delle seguenti righe, e spiegate come sia possibile correggerli. Immaginate che **g** sia un oggetto **Graphics**.
- `g.setFont("SansSerif");`
 - `g.erase(x, y, w, h); // clear rectangle at (x, y)`
 - `Font f = new Font("Serif", Font.BOLDITALIC, 12);`
 - `g.setColor(Color.Yellow); // change color to yellow`

Risposte agli esercizi di autovalutazione

- 1.1 a) **setStroke**, **Graphics2D**. b) **GradientPaint**. c) **drawLine**. d) red, green, blue. e) punti. f) **TexturePaint**.
- 1.2 a) Falso. I primi due argomenti specificano l'angolo superiore sinistro del rettangolo che circonda la forma.
 b) Vero.
 c) Vero.
 d) Vero.
 e) Falso. Le dimensioni dei caratteri sono misurate in punti.
 f) Falso. La coordinata $(0, 0)$ corrisponde all'angolo superiore sinistro di un componente GUI all'interno del quale avviene il disegno.
- 1.3 a) Il metodo **setFont** prende un oggetto **Font** come argomento, e non un oggetto **String**.
 b) La classe **Graphics** non possiede un metodo **erase**. In questi casi, è necessario utilizzare il metodo **clearRect**.
 c) **Font.BOLDITALIC** non è uno stile valido. Per avere un carattere corsivo grassetto, è necessario utilizzare **Font.BOLD + Font.ITALIC**.
 d) **Yellow** dovrebbe iniziare con la lettera minuscola: **g.setColor (Color.yellow);**.

Esercizi

- 1.4 Completate le seguenti frasi:
 a) La classe _____ di Java2D API è usata per definire degli ovali.
 b) I metodi **draw** e **fill** della classe **Graphics2D** richiedono un oggetto di tipo _____ quale argomento.
 c) Le tre costanti che specificano lo stile del carattere sono _____, _____ e _____.
 d) Il metodo _____ di **Graphics2D** imposta il colore di disegno per le forme Java2D.
- 1.5 Determinate se le seguenti affermazioni sono *vere* o *false*. Se sono *false*, spiegate il perché.
 a) Il metodo **drawPolygon** collega automaticamente due estremità del poligono.
 b) Il metodo **drawLine** disegna una linea tra due punti.
 c) Il metodo **fillArc** utilizza i gradi per specificare l'angolo.
 d) Nel sistema di coordinate Java, i valori y aumentano dall'alto verso il basso.
 e) La classe **Graphics** eredita direttamente dalla classe **Object**.
 f) La classe **Graphics** è una classe **abstract**.
 g) La classe **Font** eredita direttamente dalla classe **Graphics**.
- 1.6 Scrivete un programma che disegni una serie di otto cerchi concentrici. I cerchi devono essere separati da 10 pixel. Utilizzate il metodo **drawOval** della classe **Graphics**.
- 1.7 Scrivete un programma che disegni una serie di otto cerchi concentrici. I cerchi devono essere separati da 10 pixel. Utilizzate il metodo **drawArc**.
- 1.8 Modificate la soluzione dell'esercizio 1.6 al fine di disegnare i cerchi utilizzando delle istanze della classe **Ellipse2D.Double** e il metodo **draw** della classe **Graphics2D**.
- 1.9 Scrivete un programma che disegni delle linee di lunghezza casuale, con colori casuali.
- 1.10 Modificate la soluzione dell'esercizio 1.9 per disegnare delle linee casuali con colori e spessori casuali. Usate la classe **Line2D.Double** e il metodo **draw** della classe **Graphics2D** per disegnare le linee.

1.11 Scrivete un programma che visualizzi dei triangoli generati casualmente in colori diversi. Ogni triangolo deve essere riempito con un colore diverso. Usate la classe **GeneralPath** e il metodo **Fill** della classe **Graphics2D** per disegnare i triangoli.

1.12 Scrivete un programma che disegni casualmente dei caratteri di diverse dimensioni e di diversi colori.

1.13 Scrivete un programma che disegni una griglia 8x8. Usate il metodo **drawLine**.

1.14 Modificate la soluzione dell'esercizio 1.13 al fine di disegnare la griglia usando delle istanze della classe **Line2D.Double** e il metodo **draw** della classe **Graphics2D**.

1.15 Scrivete un programma che disegni una griglia 10x10. Usate il metodo **drawRect**.

1.16 Modificate la soluzione dell'esercizio 1.15 al fine di disegnare la griglia utilizzando delle istanze della classe **Line2D.Double** e il metodo **draw** della classe **Graphics2D**.

1.17 Scrivete un programma che disegni un tetraedro (una piramide). Utilizzate le classi **GeneralPath** e il metodo **draw** della classe **Graphics2D**.

1.18 Scrivete un programma che disegni un cubo. Utilizzate la classe **GeneralPath** e il metodo **draw** della classe **Graphics2D**.

1.19 Nell'esercizio 3.9 del volume Fondamenti, avete realizzato un'applet che, in base al raggio di un cerchio definito dall'utente, visualizzava il diametro, la circonferenza e l'area del cerchio. Modificate la soluzione di questo esercizio al fine di leggere un gruppo di coordinate oltre che il raggio. Disegnate poi un cerchio e visualizzatene il diametro, la circonferenza e l'area utilizzando l'oggetto **Ellipse2D.Double**, così da rappresentare il cerchio e il metodo **draw** della classe **Graphics2D** per visualizzare questo cerchio.

1.20 Scrivete un'applicazione che simuli uno screen saver (salvaschermo). L'applicazione dovrebbe disegnare delle linee a caso, utilizzando il metodo **drawLine** della classe **Graphics**. Dopo avere disegnato 100 linee, l'applicazione dovrebbe pulire lo schermo (svuotarlo) e cominciare nuovamente a disegnare delle linee. Perché il programma possa disegnare in continuazione, definite una chiamata a **repaint** quale ultima riga del metodo **paint**. Notate dei problemi con il vostro sistema?

1.21 Il package **javax.swing** contiene una classe chiamata **Timer**, che è in grado di chiamare il metodo **actionPerformed** dell'interfaccia **ActionListener** con un intervallo di tempo predefinito (specificato in millisecondi). Modificate la soluzione dell'esercizio 1.20 al fine di rimuovere la chiamata a **repaint** dal metodo **paint**. Definite la classe in modo che implementi **ActionListener** (il metodo **actionPerformed** dovrebbe semplicemente chiamare **repaint**). Definite, all'interno della vostra classe, una variabile di istanza di tipo **Timer** chiamata **timer**. Nel costruttore della classe, scrivete le seguenti istruzioni:

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

così da creare un'istanza della classe **Timer** che chiami il metodo **actionPerformed** dell'oggetto **this** ogni 1000 millisecondi (ovvero ogni secondo).

1.22 Modificate la soluzione dell'esercizio 1.21 al fine di permettere all'utente di inserire il numero di linee casuali che devono essere disegnate prima che l'applicazione svuoti lo schermo e riparta a disegnare nuove linee. Utilizzate un **TextField** per ottenere il valore. L'utente dovrebbe essere in grado di digitare un nuovo numero nel **TextField** in qualsiasi momento durante l'esecuzione del programma. *Nota:* La combinazione dei componenti Swing GUI e delle operazioni di disegno dà vita a problemi interessanti, dei quali si parlerà nei capitoli 2 e 3.

Per il momento, la prima riga del metodo **paint** dovrebbe essere

```
super.paint( g );
```

così da assicurarsi che i componenti GUI vengano visualizzati correttamente. Noterete che alcune delle linee disegnate casualmente oscureranno il **JTextField**. Usate una definizione di classe interna per eseguire la gestione degli eventi di **JTextField**.

1.23 Modificate la soluzione dell'esercizio 1.21 al fine di scegliere a caso forme diverse da visualizzare (utilizzate i metodi della classe **Graphics**).

1.24 Modificate la soluzione dell'esercizio 1.23 al fine di utilizzare le classi e le funzionalità di disegno di Java2D API. Nel caso di forme quali i rettangoli e le ellissi, disegnateli con gradazioni generate in modo casuale (usate la classe **GradientPaint** per generare le gradazioni di colori).

1.25 Scrivete una versione grafica della soluzione dell'esercizio 6.37 del volume Fondamenti (*Le torri di Hanoi*). Dopo avere studiato il capitolo 6, sarete in grado di implementare una versione di questo esercizio utilizzando le funzionalità grafiche, audio e di animazione di Java.

1.26 Modificate il programma della figura 7.9 del volume Fondamenti al fine di aggiornare il conteggio di ogni lato dei dati dopo ogni lancio. Convertite l'applicazione in un'applicazione a finestre (ovvero in una sottoclasse di **JFrame**) e utilizzate il metodo **drawString** di **Graphics** per mostrare i totali risultanti.

1.27 Modificate la soluzione dell'esercizio 7.21 del volume Fondamenti al fine di aggiungere un'interfaccia utente grafica usando **JTextField** e **JButton**. Disegnate inoltre delle linee al posto degli asterischi (*). Quando il programma specifica una mossa, traducete il numero di posizioni in un numero di pixel sullo schermo, moltiplicando il numero di posizioni per 10 (o per un qualsiasi valore di vostra scelta). Implementate il disegno con le funzionalità di Java2D API. *Nota:* La combinazione dei componenti Swing GUI e delle operazioni di disegno dà vita a problemi interessanti, dei quali si parlerà nei capitoli 2 e 3. Per il momento, la prima riga del metodo **paint** dovrebbe essere

```
super.paint( g );
```

così da assicurarsi che i componenti GUI vengano visualizzati correttamente.

1.28 Realizzate una versione grafica del problema degli esercizi 7.22, 7.23 e 7.26 del volume Fondamenti. Ogni volta che viene effettuata una mossa, la corrispondente casella della scacchiera dovrebbe essere aggiornata con il corrispondente numero di mossa. Se il risultato del programma è un giro completo o un giro chiuso, il programma dovrebbe visualizzare un messaggio appropriato. Se volete, potete usare la classe **Timer** (vedi esercizio 1.24) per facilitare l'animazione del percorso del cavallo. Ogni secondo, dovrebbe avvenire la mossa successiva.

1.29 Realizzate una versione grafica della simulazione dell'esercizio 7.41 del volume Fondamenti. Simulate la montagna disegnando un arco che si estende dalla parte inferiore sinistra della finestra verso la parte superiore destra. La tartaruga e la lepre dovrebbero correre sulla montagna. Implementate l'output grafico in modo che la tartaruga e la lepre vengano effettivamente disegnate sull'arco ad ogni mossa. [*Nota:* Estendete la lunghezza della gara da 70 a 300, così da avere un'area di disegno più ampia.]

1.30 Realizzate una versione grafica degli esercizi 7.38-7.40 del volume Fondamenti. Utilizzate i labirinti come guide per la creazione delle versioni grafiche. Mentre il labirinto viene risolto, al suo interno dovrebbe apparire un piccolo cerchio che indica la posizione attuale. Se volete, potete usare la classe **Timer** (vedi esercizio 1.24) per facilitare l'animazione dell'attraversamento del labirinto. Ogni secondo, dovrebbe avvenire la mossa successiva.

- 1.31 Realizzate una versione grafica dell'esercizio 7.28 del volume Fondamenti, che mostri ogni valore mentre viene posizionato nel bucket appropriato e mentre viene copiato nuovamente nell'array originario.
- 1.32 Scrivete un programma che utilizzi il metodo **drawPolyline** per disegnare una spirale.
- 1.33 Scrivete un programma che inserisca quattro numeri e poi li rappresenti sotto forma di un grafico a torta. Utilizzate la classe **Arc2D.Double** e il metodo **fill** della classe **Graphics2D** per eseguire il disegno. Disegnate ogni porzione della torta con un colore diverso. Scrivete un'applet che inserisca quattro numeri e poi li rappresenti sotto forma di un grafico a barre. Utilizzate la classe **Rectangle2D.Double** e il metodo **fill** della classe **Graphics2D** per eseguire il disegno. Disegnate ogni barra con un colore diverso.