Supplement 3G

Graphics (Optional)

Introduction

One of the most compelling reasons to learn about using objects is that they allow us to draw graphics in Java. Graphics are used for games, computer animations, and modern graphical user interfaces (GUIs), and to render complex images. Graphics are also a good way to practice the use of parameters as discussed in the previous chapter.

In this optional supplement we will examine a few of the basic classes from Java's graphical framework and use them to draw patterned two-dimensional figures of shapes and text onto the screen.

3G.1 Introduction to Graphics

Java's original graphical tools were collectively known as the Abstract Window Toolkit (AWT). The classes associated with the AWT reside in the package `java.awt`. In order to create graphical programs like the ones you'll see in this section, you'll need to include the following `import` declaration at the top of your programs:

```
import java.awt.*;   // for graphics
```

To keep things simple, we'll use a custom class called `DrawingPanel` that was written by the authors of this textbook to simplify some of the more esoteric details of Java graphics. Its core code is less than a page long, so we aren't hiding much. You won't need to import `DrawingPanel`, but you will need to place the file `DrawingPanel.java` in the same folder as your program.

The drawing panel keeps track of the overall image, but the actual drawing will be done with an object of type `Graphics`. The `Graphics` class is also part of the `java.awt` library.

DrawingPanel

You can create a graphical window on your screen by constructing a `DrawingPanel` object. You must specify the width and height of the drawing area. When the following line executes, a window appears immediately on the screen:

```
DrawingPanel <name> = new DrawingPanel(<width>, <height>);
```

`DrawingPanel` objects have two public methods, listed in Table 3G.1.

The typical way that you'll use `DrawingPanel` will be to construct a panel of a particular height and width, set its background color (if you don't want the default white background), and then draw something on it using its `Graphics` object. The `DrawingPanel` appears on the screen at the time that you construct it.

All coordinates are specified as integers. Each (x, y) position corresponds to a different pixel on the computer screen. The word *pixel* is shorthand for "picture element" and represents a single dot on the computer screen.

> **Pixel**
>
> A single small dot on a computer screen.

Table 3G.1 Useful Methods for `DrawingPanel`

Method	Description
`getGraphics()`	returns a `Graphics` object that can be used to draw onto the panel
`setBackground(color)`	sets the background color of the panel to the given color (the default is white)

The coordinate system assigns the upper-left corner of a panel the position (0, 0). As you move to the right of this position, the x value increases. As you move down from this position, the y value increases. For example, suppose that you construct a `DrawingPanel` object with a width of 200 pixels and a height of 100 pixels. The upper-left corner will have the coordinates (0, 0), and the lower-right corner will have the coordinates (199, 99) as shown in Figure 3G.1.

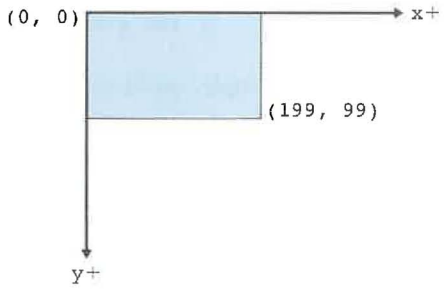


Figure 3G.1 The (x, y) Coordinate Space

This is likely to be confusing at first, because you're probably used to coordinate systems where y values decrease as you move down. However, you'll soon get the hang of it.

Drawing Lines and Shapes

So, how do you actually draw something? To draw shapes and lines, you don't talk directly to the `DrawingPanel`, but rather to a related object of type `Graphics`. Think of the `DrawingPanel` as a canvas and the `Graphics` object as the paintbrush. The `DrawingPanel` class has a method called `getGraphics` that returns its `Graphics` object:

```
Graphics g = <panel>.getGraphics();
```

One of the simplest drawing commands is `drawLine`, which takes four integer arguments.

For example, the method:

```
g.drawLine(<x1>, <y1>, <x2>, <y2>);
```

draws a line from the point ($x1$, $y1$) to the point ($x2$, $y2$). The `drawLine` method is just one of many commands a `Graphics` object understands; see Table 3G.2 for others. The `Graphics` object has many more methods in addition to the ones discussed here. You can read about them in the Java API documentation.

Here is a sample program that puts these pieces together:

```
1 // Draws a line onto a DrawingPanel.
2
3 import java.awt.*;  // for graphics
4
```

Table 3G.2 Some Useful Methods of `Graphics` Objects

Method	Description
`drawLine(x1, y1, x2, y2)`	draws a line between the points ($x1$, $y1$) and ($x2$, $y2$)
`drawOval(x, y, width, height)`	draws the outline of the largest oval that fits within the specified rectangle
`drawRect(x, y, width, height)`	draws the outline of the specified rectangle
`drawString(message, x, y)`	draws the given text with its lower-left corner at (x, y)
`fillOval(x, y, width, height)`	fills the largest oval that fits within the specified rectangle using the current color
`fillRect(x, y, width, height)`	fills the specified rectangle using the current color
`setColor(color)`	sets this graphics context's current color to the specified color (all subsequent graphics operations using this graphics context use this specified color)
`setFont(font)`	sets this graphics context's current font to the specified font (all subsequent strings drawn using this graphics context use this specified font)

```
5  public class DrawLine1 {
6      public static void main(String[] args) {
7          // create the drawing panel
8          DrawingPanel panel = new DrawingPanel(200, 100);
9
10         // draw a line on the panel using
11         // the Graphics paintbrush
12         Graphics g = panel.getGraphics();
13         g.drawLine(25, 75, 175, 25);
14     }
15 }
```

When you run this program, the window shown in Figure 3G.2 appears. Though it isn't text on the console, as in previous chapters, we'll still refer to this as the "output" of the program.

(Java can be run on a variety of systems. Depending on your operating system, your output may differ slightly from the screenshots in this chapter.)

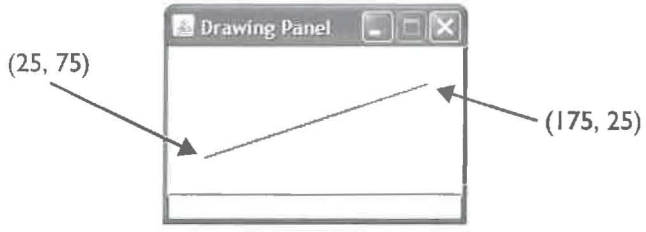


Figure 3G.2 Output of `DrawLine1`

The first statement in `main` constructs a `DrawingPanel` with a width of 200 and a height of 100. Once it has been constructed, the window will pop up on the screen. The second statement draws a line from (25, 75) to (175, 25). The first point is in the lower-left part of the window (25 over from the left, 75 down from the top). The second point is in the upper-right corner (175 over from the left, 25 down from the top).

Notice these particular lines of code:

```
Graphics g = panel.getGraphics();
g.drawLine(25, 75, 175, 25);
```

You might wonder why you can't just say:

```
panel.drawLine(25, 75, 175, 25);  // this is illegal
```

The problem is that there are two different objects involved in this program: the `DrawingPanel` itself (the canvas) and the `Graphics` object associated with the panel (the paintbrush). The panel doesn't know how to draw a line; only the `Graphics` object knows how to do this. You have to be careful to make sure that you are talking to the right object when you give a command.

This requirement can be confusing, but it is common in Java programs. In fact, in a typical Java program, there are hundreds (if not thousands) of objects interacting with each other. These interactions aren't so unlike interactions between people. If you want to schedule a meeting, a busy corporate executive might tell you, "Talk to my secretary about that." Or if you're asking difficult legal questions, a person might tell you, "Talk to my lawyer about that." In this case, the `DrawingPanel` doesn't know how to draw, so if it could talk it would say, "Talk to my `Graphics` object about that."

It's also legal to use the `Graphics` object without storing it in a variable, like this:

```
panel.getGraphics().drawLine(25, 75, 175, 25);    // also legal
```

But you'll often want to send several commands to the `Graphics` object, so it's more convenient to give it a name and store it in a variable.

Let's look at a more complicated example:

```
1  // Draws three lines to make a triangle.
2
3  import java.awt.*;
4
5  public class DrawLine2 {
6      public static void main(String[] args) {
7          DrawingPanel panel = new DrawingPanel(200, 100);
8
```

```
9           // draw a triangle on the panel
10          Graphics g = panel.getGraphics();
11          g.drawLine(25, 75, 100, 25);
12          g.drawLine(100, 25, 175, 75);
13          g.drawLine(25, 75, 175, 75);
14      }
15  }
```

This program draws three different lines to form a triangle, as shown in Figure 3G.3. The lines are drawn between three different points. In the lower-left corner we have the point (25, 75). In the middle at the top we have the point (100, 25). And in the lower-right corner we have the point (175, 75). The various calls on `drawLine` simply draw the lines that connect these three points.

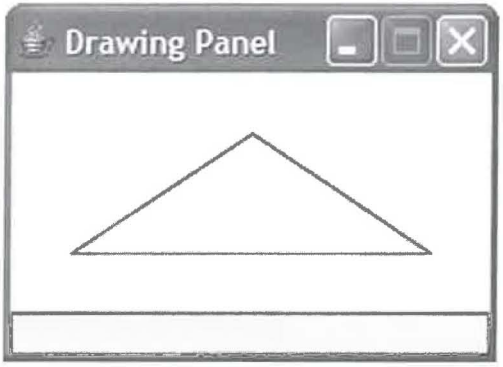


Figure 3G.3 Output of `DrawLine2`

The `Graphics` object also has methods for drawing particular shapes. For example, you can draw rectangles with the `drawRect` method:

```
g.drawRect(<x>, <y>, <width>, <height>);
```

This draws a rectangle with upper-left coordinates (x, y) and the given height and width.

Another figure you'll often want to draw is a circle or, more generally, an oval. But how do you specify where it appears and how big it is? What you actually specify is what is known as the "bounding rectangle" of the circle or oval. Java will draw the largest oval possible that fits inside that rectangle. So, the method:

```
g.drawOval(<x>, <y>, <width>, <height>);
```

draws the largest oval that fits within the rectangle with upper-left coordinates (x, y) and the given height and width.

Notice that the first two values passed to `drawRect` and `drawOval` are coordinates, while the next two values are a width and a height. For example, here is a short program that draws two rectangles and two ovals:

```
1  // Draws several shapes.
2
3  import java.awt.*;
4
5  public class DrawShapes1 {
6      public static void main(String[] args) {
7          DrawingPanel panel = new DrawingPanel(200, 100);
8
9          Graphics g = panel.getGraphics();
10         g.drawRect(25, 50, 20, 20);
11         g.drawRect(150, 10, 40, 20);
12         g.drawOval(50, 25, 20, 20);
13         g.drawOval(150, 50, 40, 20);
14     }
15 }
```

Figure 3G.4 shows the output of the program.

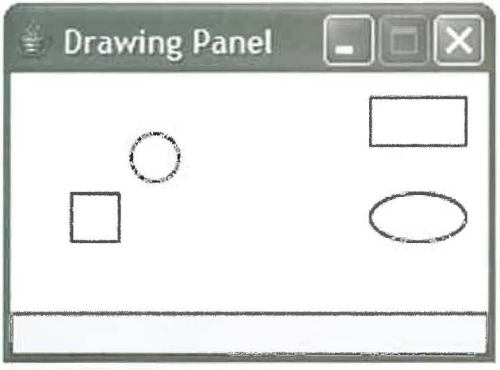


Figure 3G.4 Output of `DrawShapes1`

The first rectangle has its upper-left corner at the coordinates (25, 50). Its width and height are each 20, so this is a square. The coordinates of its lower-right corner would be (45, 70), or 20 more than the (x, y) coordinates of the upper-left corner. The program also draws a rectangle with its upper-left corner at (150, 10) that has a width of 40 and a height of 20 (wider than it is tall). The bounding rectangle of the first oval has upper-left coordinates (50, 25) and a width and height of 20. In other words, it's a circle. The bounding rectangle of the second oval has upper-left coordinates (150, 50), a width of 40, and a height of 20 (it's an oval that is wider than it is tall).

Sometimes you don't just want to draw the outline of a shape; you want to paint the entire area with a particular color. There are variations of the `drawRect` and `drawOval` methods known as `fillRect` and `fillOval` that do exactly that, drawing a rectangle or oval and filling it in with the current color of paint (the default is black). Let's change two of the calls in the previous program to be "fill" operations instead of "draw" operations:

```
1  // Draws and fills several shapes.
2
3  import java.awt.*;
4
5  public class DrawShapes2 {
6      public static void main(String[] args) {
7          DrawingPanel panel = new DrawingPanel(200, 100);
8
9          Graphics g = panel.getGraphics();
10         g.fillRect(25, 50, 20, 20);
11         g.drawRect(150, 10, 40, 20);
12         g.drawOval(50, 25, 20, 20);
13         g.fillOval(150, 50, 40, 20);
14     }
15 }
```

Now we get the output shown in Figure 3G.5 instead.

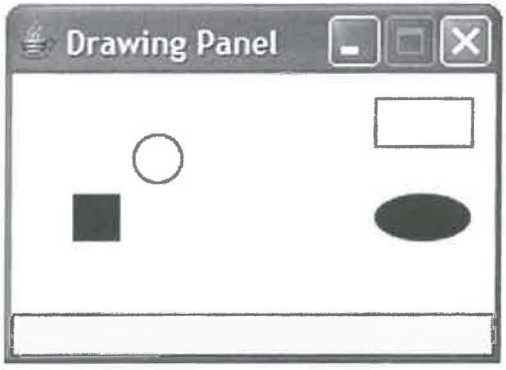


Figure 3G.5 Output of `DrawShapes2`

Colors

All of the shapes and lines drawn by the preceding programs were black, and all of the panels had a white background. These are the default colors, but you can change the background color of the panel, and you can change the color being used by the `Graphics` object as many times as you like. To change these colors, you use the standard `Color` class, which is part of the `java.awt` package.

Table 3G.3 `Color` Constants

`Color.BLACK`	`Color.GREEN`	`Color.RED`
`Color.BLUE`	`Color.LIGHT_GRAY`	`Color.WHITE`
`Color.CYAN`	`Color.MAGENTA`	`Color.YELLOW`
`Color.DARK_GRAY`	`Color.ORANGE`	
`Color.GRAY`	`Color.PINK`	

There are a number of predefined colors that you can refer to directly. They are defined as class constants in the `Color` class (a lot like the constants we used in Chapter 2). The names of these constants are all in uppercase and are self-explanatory. To refer to one of these colors, you have to precede it with the class name and a dot, as in `Color.GREEN` or `Color.BLUE`. The predefined `Color` constants are listed in Table 3G.3.

As mentioned earlier, the `DrawingPanel` object has a method that can be used to change the background color that covers the entire panel:

```
<panel>.setBackground(<color>);
```

Likewise, the `Graphics` object has a method that can be used to change the current color that draws or fills shapes and lines:

```
g.setColor(<color>);
```

Calling `setColor` is like dipping your paintbrush in a different color of paint. From that point on, all drawing and filling will be done in the specified color. For example, here is another version of the previous program that uses a cyan (light blue) background color and fills in the oval and square with white instead of black:

```
1  // Draws and fills shapes in different colors.
2
3  import java.awt.*;
4
5  public class DrawColoredShapes {
6      public static void main(String[] args) {
7          DrawingPanel panel = new DrawingPanel(200, 100);
8          panel.setBackground(Color.CYAN);
9
10         Graphics g = panel.getGraphics();
11         g.drawRect(150, 10, 40, 20);
12         g.drawOval(50, 25, 20, 20);
13         g.setColor(Color.WHITE);
14         g.fillOval(150, 50, 40, 20);
15         g.fillRect(25, 50, 20, 20);
16     }
17 }
```

This program produces the output shown in Figure 3G.6. (The figures shown in this textbook may not match the colors you would see on your screen.)

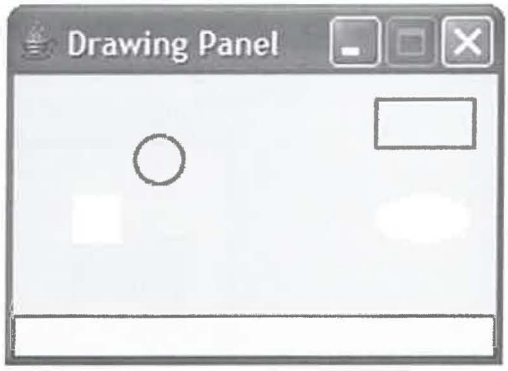


Figure 3G.6 Output of `DrawColoredShapes`

Notice that you tell the panel to set the background color, while you tell the `Graphics` object to set the foreground color. The reasoning is that the background color is a property of the entire window, while the foreground color affects only the particular shapes that you draw.

Notice also that the order of the calls has been rearranged. The two drawing commands appear first, then the call on `setColor` that changes the color to white, then the two filling commands. This ensures that the drawing is done in black and the filling is done in white. The order of operations is very important in these drawing programs, so you'll have to keep track of what your current color is each time you give a new command to draw or fill something.

Common Programming Error

Misunderstanding Draw vs. Fill

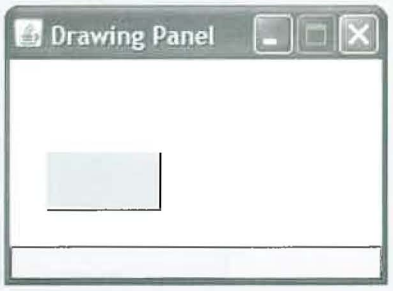
Some new programmers think that a shape must be drawn (such as with `drawRect`) before it can be filled in (such as with `fillRect`). This is not the case. In fact, when you are trying to draw an outlined shape, this is exactly the wrong thing to do. Suppose you want to draw a 60 × 30 green rectangle with a black border at (20, 50). You might write the following code:

```
g.setColor(Color.BLACK);   // incorrect code
g.drawRect(20, 50, 60, 30);
g.setColor(Color.GREEN);
g.fillRect(20, 50, 60, 30);
```

Continued on next page

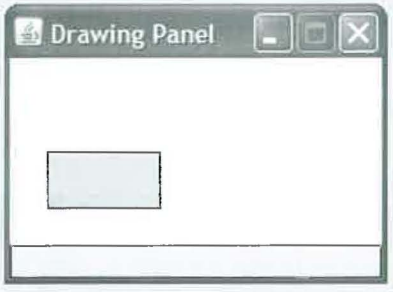
Continued from previous page

However, the fill command covers the same pixels as the draw command, and the green interior will be drawn over the black outline, leading to the following appearance:



Instead, the code should fill the interior of the rectangle first, then draw the black outline, to make sure that the outline shows on top of the filling. The following is the correct code and its output:

```
g.setColor(Color.GREEN);   // corrected code
g.fillRect(20, 50, 60, 30);
g.setColor(Color.BLACK);
g.drawRect(20, 50, 60, 30);
```



Drawing with Loops

In each of the preceding examples we used simple constants for the drawing and filling commands, but it is possible to use expressions. For example, suppose that we stick with our `DrawingPanel` size of 200 pixels wide and 100 pixels tall and we want to produce a diagonal series of four rectangles that extend from the upper-left corner to the lower-right corner, each with a white oval inside. In other words, we want to produce the output shown in Figure 3G.7.

The overall width of 200 and overall height of 100 are divided evenly into four rectangles, which means that they must all be 50 pixels wide and 25 pixels high. So, width and height values for the four rectangles are the same, but the positions of their

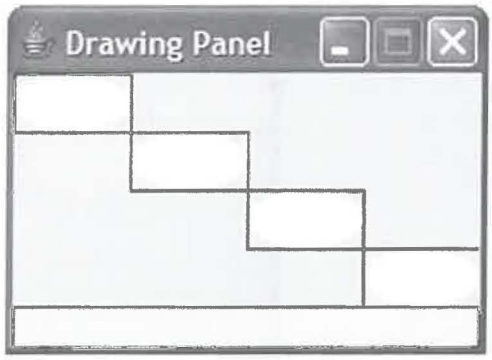


Figure 3G.7 Desired Output of `DrawLoop1`

upper-left corners are different. The first rectangle's upper-left corner is at (0, 0), the second is at (50, 25), the third is at (100, 50), and the fourth is at (150, 75). We need to write code to generate these different coordinates.

This is a great place to use a `for` loop. Using the techniques introduced in Chapter 2, we can make a table and develop a formula for the coordinates. In this case it is easier to have the loop start with 0 rather than 1, which will often be the case with drawing programs. Here is a program that makes a good first stab at generating the desired output:

```
1  // Draws boxed ovals using a for loop (flawed version).
2
3  import java.awt.*;
4
5  public class DrawLoop1 {
6      public static void main(String[] args) {
7          DrawingPanel panel = new DrawingPanel(200, 100);
8          panel.setBackground(Color.CYAN);
9
10         Graphics g = panel.getGraphics();
11         for (int i = 0; i < 4; i++) {
12             g.drawRect(i * 50, i * 25, 50, 25);
13             g.setColor(Color.WHITE);
14             g.fillOval(i * 50, i * 25, 50, 25);
15         }
16     }
17 }
```

This program produces the output shown in Figure 3G.8.

The coordinates and sizes are right, but not the colors. Instead of getting four black rectangles with white ovals inside, we're getting one black rectangle and three white rectangles. That's because we only have one call on `setColor` inside the loop. Initially the color will be set to black, which is why the first rectangle comes out

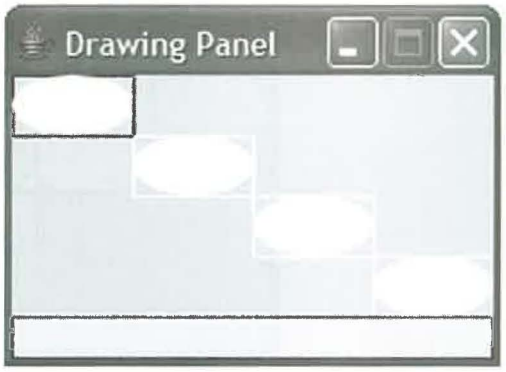


Figure 3G.8 Output of `DrawLoop1`

black. But once we make a call on `setColor` changing the color to white, every subsequent drawing and filling command is done in white, including the second, third, and fourth rectangles.

So, we need to include calls to set the color to black, to draw the rectangles, and to set the color to white to draw the filled ovals. While we're at it, it's a good idea to switch the order of these tasks. The rectangles and ovals overlap slightly, and we would rather have the rectangle drawn over the oval than the other way around. The following program produces the correct output:

```
1  // Draws boxed ovals using a for loop.
2
3  import java.awt.*;
4
5  public class DrawLoop2 {
6      public static void main(String[] args) {
7          DrawingPanel panel = new DrawingPanel(200, 100);
8          panel.setBackground(Color.CYAN);
9
10         Graphics g = panel.getGraphics();
11         for (int i = 0; i < 4; i++) {
12             g.setColor(Color.WHITE);
13             g.fillOval(i * 50, i * 25, 50, 25);
14             g.setColor(Color.BLACK);
15             g.drawRect(i * 50, i * 25, 50, 25);
16         }
17     }
18 }
```

It's also possible to create custom `Color` objects of your own, rather than using the constant colors provided in the `Color` class. Computer monitors use red, green, and blue (RGB) as their primary colors, so when you construct a `Color`

object you pass your own parameter values for the redness, greenness, and blueness of the color:

```
new Color(<red>, <green>, <blue>)
```

The red/green/blue components should be integer values between `0` and `255`. The higher the value, the more of that color is mixed in. All `0` values produce black, and all `255` values produce white. Values of `(0, 255, 0)` produce a pure green, while values of `(128, 0, 128)` make a dark purple color (because red and blue are mixed). Search for "RGB table" in your favorite search engine to find tables of many common colors.

The following program demonstrates the use of custom colors. It uses a class constant for the number of rectangles to draw and produces a blend of colors from black to white:

```
1  // Draws a smooth color gradient from black to white.
2
3  import java.awt.*;
4
5  public class DrawColorGradient {
6      public static final int RECTS = 32;
7
8      public static void main(String[] args) {
9          DrawingPanel panel = new DrawingPanel(256, 256);
10         panel.setBackground(new Color(255, 128, 0)); // orange
11
12         Graphics g = panel.getGraphics();
13
14         // from black to white, top left to bottom right
15         for (int i = 0; i < RECTS; i++) {
16             int shift = i * 256 / RECTS;
17             g.setColor(new Color(shift, shift, shift));
18             g.fillRect(shift, shift, 20, 20);
19         }
20     }
21 }
```

This program produces the output shown in Figure 3G.9.

It is also legal to store a `Color` object into a variable or pass it as a parameter. For example, we could have written the coloring code in the preceding program as follows:

```
Color c = new Color(shift, shift, shift);
g.setColor(c);
...
```

We will use this idea later when parameterizing colors in this chapter's Case Study.

Figure 3G.9 Output of `DrawColorGradient`

Text and Fonts

Another drawing command worth mentioning can be used to include text in your drawings. The `drawString` method of the `Graphics` object draws the given `String` with its lower-left corner at coordinates (x, y):

```
g.drawString(<message>, <x>, <y>);
```

This is a slightly different convention than we used for `drawRect`. With `drawRect`, we specified the coordinates of the upper-left corner. Here we specify the coordinates of the lower-left corner. By default, text is drawn approximately 10 pixels high. Here is a sample program that uses a loop to draw a particular `String` 10 different times, each time indenting it 5 pixels to the right and moving it down 10 pixels from the top:

```
1  // Draws a message several times.
2
3  import java.awt.*;
4
5  public class DrawStringMessage1 {
6      public static void main(String[] args) {
7          DrawingPanel panel = new DrawingPanel(200, 100);
```

```
8           panel.setBackground(Color.YELLOW);
9
10          Graphics g = panel.getGraphics();
11          for (int i = 0; i < 10; i++) {
12              g.drawString("There is no place like home",
13                           i * 5, 10 + i * 10);
14          }
15      }
16  }
```

This program produces the output shown in Figure 3G.10.

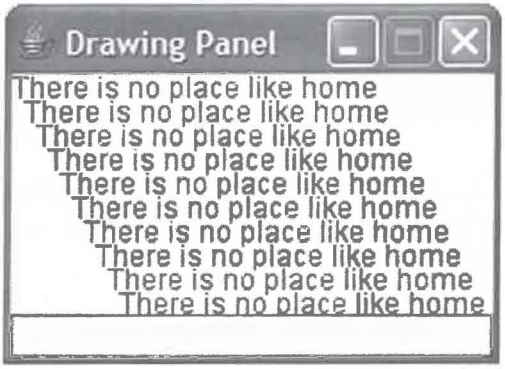


Figure 3G.10 Output of `DrawStringMessage`

Fonts are used to describe different styles for writing characters on the screen. If you'd like to change the style or size of the onscreen text, you can use the `setFont` method of the `Graphics` object.

> **Font**
>
> An overall design for a set of text characters, including the style, size, weight, and appearance of each character.

This method changes the text size and style in which strings are drawn.

The parameter to `setFont` is a `Font` object. A `Font` object is constructed by passing three parameters—the font's name as a `String`, its style (such as bold or italic), and its size as an integer:

```
new Font(<name>, <style>, <size>)
```

Common font styles such as bold are implemented as constants in the `Font` class. The available constants and some popular font names are listed in Tables 3G.4 and 3G.5.

Table 3G.4 Useful Constants of the `Font` Class

Constant	Displays
`Font.BOLD`	**Bold text**
`Font.ITALIC`	*Italic text*
`Font.BOLD + Font.ITALIC`	***Bold/Italic text***
`Font.PLAIN`	Plain text

Table 3G.5 Common Font Names

Name	Description
`"Monospaced"`	a typewriter font, such as `Courier New`
`"SansSerif"`	a font without curves (serifs) at letter edges, such as Arial
`"Serif"`	a font with curved edges, such as Times New Roman

As in the case of colors, setting the font affects only strings that are drawn after the font is set. The following program sets several fonts and uses them to draw strings:

```
1  // Draws several messages using different fonts.
2
3  import java.awt.*;
4
5  public class DrawFonts {
6      public static void main(String[] args) {
7          DrawingPanel panel = new DrawingPanel(200, 100);
8          panel.setBackground(Color.PINK);
9
10         Graphics g = panel.getGraphics();
11         g.setFont(new Font("Monospaced",
12                   Font.BOLD + Font.ITALIC, 36));
13         g.drawString("Too big", 20, 40);
14
15         g.setFont(new Font("SansSerif", Font.PLAIN, 10));
16         g.drawString("Too small", 30, 60);
17
18         g.setFont(new Font("Serif", Font.ITALIC, 18));
19         g.drawString("Just right", 40, 80);
20     }
21 }
```

This program produces the output shown in Figure 3G.11.

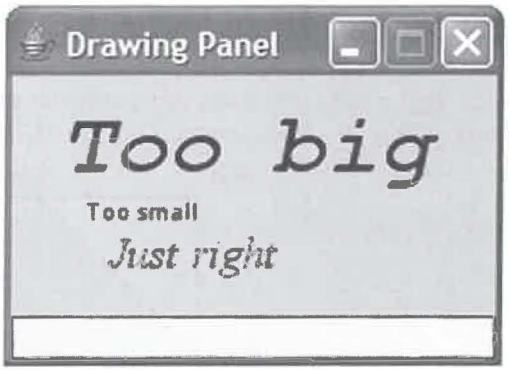


Figure 3G.11 Output of `DrawFonts`

3G.2 Procedural Decomposition with Graphics

If you write complex drawing programs, you will want to break them down into several static methods to structure the code and to remove redundancy. When you do this, you'll have to pass the `Graphics` object to each static method that you introduce. For a quick example, the `DrawStringMessage1` program from the previous section could be split into a `main` method and a `drawText` method, as follows:

```
1  // Draws a message several times using a static method.
2
3  import java.awt.*;
4
5  public class DrawStringMessage2 {
6      public static void main(String[] args) {
7          DrawingPanel panel = new DrawingPanel(200, 100);
8          panel.setBackground(Color.YELLOW);
9
10         Graphics g = panel.getGraphics();
11         drawText(g);
12     }
13
14     public static void drawText(Graphics g) {
15         for (int i = 0; i < 10; i++) {
16             g.drawString("There is no place like home",
17                          i * 5, 10 + i * 10);
18         }
19     }
20 }
```

This program produces the same output as the original program (Figure 3G.10).

The program wouldn't compile without passing `Graphics g` to the `drawText` method, because `g` is needed to call drawing methods such as `drawString` and `fillRect`.

A Larger Example: `DrawDiamonds`

Now let's consider a slightly more complicated task: drawing the largest diamond figure that will fit into a box of a particular size. The largest diamond that can fit into a box of size 50×50 is shown in Figure 3G.12.

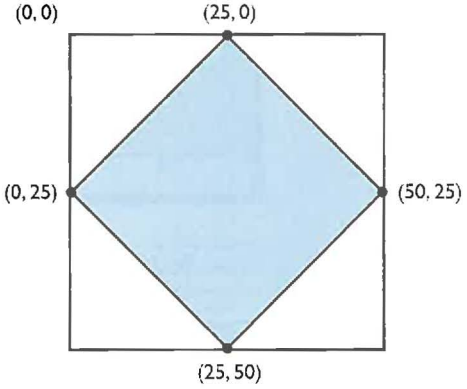


Figure 3G.12 Diamond

The code to draw such a diamond would be the following:

```
g.drawRect(0, 0, 50, 50);
g.drawLine(0, 25, 25, 0);
g.drawLine(25, 0, 50, 25);
g.drawLine(50, 25, 25, 50);
g.drawLine(25, 50, 0, 25);
```

Now imagine that we wish to draw three such 50×50 diamonds at different locations. We can turn our diamond-drawing code into a `drawDiamond` method that we'll call three times. Since each diamond will be in a different position, we can pass the x- and y-coordinates as parameters to our `drawDiamond` method.

A diamond enclosed by a box with top-left corner at the location (78, 22) is shown in Figure 3G.13.

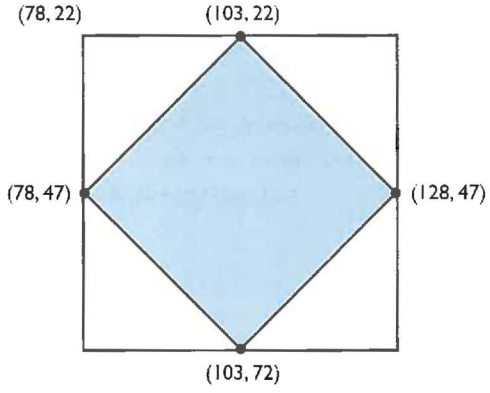


Figure 3G.13 Diamond at (78, 22)

The code to draw this diamond would be the following:

```
g.drawRect(78, 22, 50, 50);
g.drawLine(78, 47, 103, 22);
g.drawLine(103, 22, 128, 47);
g.drawLine(128, 47, 103, 72);
g.drawLine(103, 72, 78, 47);
```

As you can see, the parameter values passed to the `drawRect` and `drawLine` methods are very similar to those of the first diamond, except that they're shifted by 78 in the x-direction and 22 in the y-direction (except for the third and fourth parameters to `drawRect`, since these are the rectangle's width and height). This (78, 22) shift is called an *offset*.

We can generalize the coordinates to pass to `Graphics g`'s drawing commands so that they'll work with any diamond if we pass that diamond's top-left x- and y-offset. For example, we'll generalize the line from (0, 25) to (25, 0) in the first diamond and from (78, 47) to (103, 22) in the second diamond by saying that it is a line from $(x, y + 25)$ to $(x + 25, y)$, where (x, y) is the offset of the given diamond.

The following program uses the `drawDiamond` method to draw three diamonds without redundancy:

```
1  // This program draws several diamond figures of size 50x50.
2
3  import java.awt.*;
4
5  public class DrawDiamonds {
6      public static void main(String[] args) {
7          DrawingPanel panel = new DrawingPanel(250, 150);
8          Graphics g = panel.getGraphics();
9
10         drawDiamond(g, 0, 0);
11         drawDiamond(g, 78, 22);
12         drawDiamond(g, 19, 81);
13     }
14
15     // draws a diamond in a 50x50 box
16     public static void drawDiamond(Graphics g, int x, int y) {
17         g.drawRect(x, y, 50, 50);
18         g.drawLine(x, y + 25, x + 25, y);
19         g.drawLine(x + 25, y, x + 50, y + 25);
20         g.drawLine(x + 50, y + 25, x + 25, y + 50);
21         g.drawLine(x + 25, y + 50, x, y + 25);
22     }
23 }
```

This program produces the output shown in Figure 3G.14.

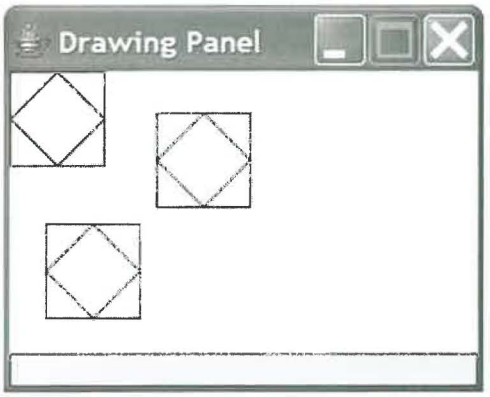


Figure 3G.14 Output of `DrawDiamonds`

It's possible to draw patterned figures in loops and to have one drawing method call another. For example, if we want to draw five diamonds, starting at (12, 15) and spaced 60 pixels apart, we just need a `for` loop that repeats five times and shifts the *x*-coordinate by 60 each time. Here's an example loop:

```
for (int i = 0; i < 5; i++) {
    drawDiamond(g, 12 + 60 * i, 15);
}
```

If we created another method to draw the line of five diamonds, we could call it from `main` to draw many lines of diamonds. Here's a modified version of the `DrawDiamonds` program with two graphical methods:

```
1  // This program draws several diamond figures of size 50x50.
2
3  import java.awt.*;
4
5  public class DrawDiamonds2 {
6      public static void main(String[] args) {
7          DrawingPanel panel = new DrawingPanel(360, 160);
8          Graphics g = panel.getGraphics();
9
10         drawManyDiamonds(g, 12, 15);
11         g.setColor(Color.RED);
12         drawManyDiamonds(g, 55, 100);
13     }
14
15     // draws five diamonds in a horizontal line
```

```
16      public static void drawManyDiamonds(Graphics g,
17                                          int x, int y) {
18          for (int i = 0; i < 5; i++) {
19              drawDiamond(g, x + 60 * i, y);
20          }
21      }
22
23      // draws a diamond in a 50x50 box
24      public static void drawDiamond(Graphics g, int x, int y) {
25          g.drawRect(x, y, 50, 50);
26          g.drawLine(x, y + 25, x + 25, y);
27          g.drawLine(x + 25, y, x + 50, y + 25);
28          g.drawLine(x + 50, y + 25, x + 25, y + 50);
29          g.drawLine(x + 25, y + 50, x, y + 25);
30      }
31  }
```

This program produces the output shown in Figure 3G.15.

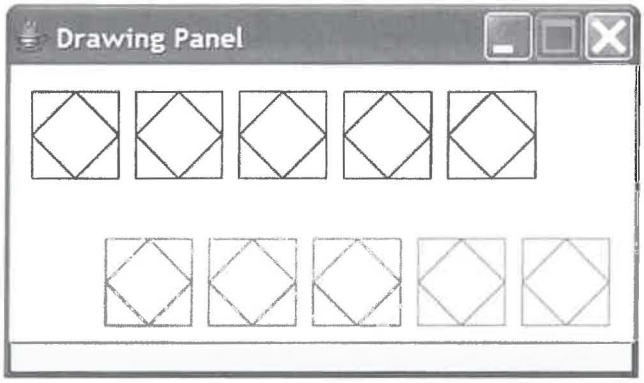


Figure 3G.15 Output of `DrawDiamonds2`

3G.3 Case Study: Pyramids

Imagine that you've been asked to write a program that will draw the images in Figure 3G.16 onto a `DrawingPanel`.

The overall drawing panel has a size of 350 × 250. Each pyramid is 100 pixels high and 100 pixels wide. The pyramids consist of centered flights of colored stairs

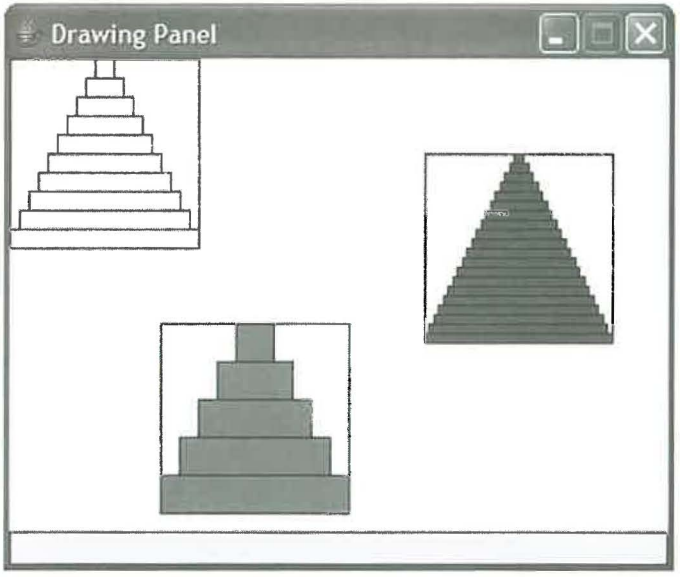


Figure 3G.16 Desired Pyramids Output

that widen toward the bottom, with black outlines around each stair. Table 3G.6 lists the attributes of each pyramid.

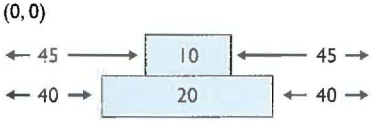
Table 3G.6 Pyramid Attributes

Fill color	Top-left corner	Number of stairs	Height of each stair
white	(0, 0)	10 stairs	10 pixels
red	(80, 140)	5 stairs	20 pixels
blue	(220, 50)	20 stairs	5 pixels

Unstructured Partial Solution

When trying to solve a larger and more complex problem like this, it's important to tackle it piece by piece and make iterative enhancements toward a final solution. Let's begin by trying to draw the top-left white pyramid correctly.

Each stair is centered horizontally within the pyramid. The top stair is 10 pixels wide. Therefore, it is surrounded by $\frac{90}{2}$ or 45 pixels of empty space on either side. That means that the 10×10 rectangle's top-left corner is at (45, 0). The second stair is 20 pixels wide, meaning that it's surrounded by $\frac{80}{2}$ or 40 pixels on each side:



The following program draws the white pyramid in the correct position:

```
1  import java.awt.*;
2
3  // Draws the first pyramid only, with a lot of redundancy.
4  public class Pyramids1 {
5      public static void main(String[] args) {
6          DrawingPanel panel = new DrawingPanel(350, 250);
7          Graphics g = panel.getGraphics();
8
9          // draws the border rectangle
10         g.drawRect(0, 0, 100, 100);
11
12         // draws the 10 "stairs" in the white pyramid
13         g.drawRect(45,  0,  10, 10);
14         g.drawRect(40, 10,  20, 10);
15         g.drawRect(35, 20,  30, 10);
16         g.drawRect(30, 30,  40, 10);
17         g.drawRect(25, 40,  50, 10);
18         g.drawRect(20, 50,  60, 10);
19         g.drawRect(15, 60,  70, 10);
20         g.drawRect(10, 70,  80, 10);
21         g.drawRect( 5, 80,  90, 10);
22         g.drawRect( 0, 90, 100, 10);
23     }
24 }
```

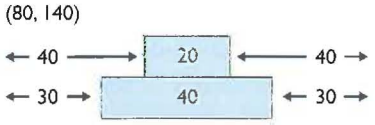
Looking at the code, it's clear that there's a lot of redundancy among the 10 lines to draw the stairs. Examining the patterns of numbers in each column reveals that the x value decreases by 5 each time, the y value increases by 10 each time, the width increases by 10 each time, and the height stays the same.

Another way of describing a stair's x value is to say that it is half of the overall 100 minus the stair's width. With that in mind, the following `for` loop draws the 10 stairs without the previous redundancy:

```
for (int i = 0; i < 10; i++) {
    int stairWidth = 10 * (i + 1);
    int stairHeight = 10;
    int stairX = (100 - stairWidth) / 2;
    int stairY = 10 * i;
    g.drawRect(stairX, stairY, stairWidth, stairHeight);
}
```

Generalizing the Drawing of Pyramids

Next let's add code to draw the bottom (red) pyramid. Its (x, y) position is (80, 140) and it has only five stairs. That means each stair is twice as tall and wide as those in the white pyramid.



Given this information, we can determine that the top stair's upper-left corner is at (120, 140) and its size is 20×20, the second stair's upper-left corner is at (110, 160) and its size is 40×20, and so on.

For the moment, let's focus on getting the coordinates of the stairs right and not worry about the red fill color. Here is a redundant bit of code to draw the red pyramid's stairs, without the coloring:

```
// draws the border rectangle
g.drawRect(80, 140, 100, 100);
// draws the 5 "stairs" of the red pyramid
g.drawRect(120, 140, 20, 20);
g.drawRect(110, 160, 40, 20);
g.drawRect(100, 180, 60, 20);
g.drawRect( 90, 200, 80, 20);
g.drawRect( 80, 220, 100, 20);
```

Again we have redundancy among the five lines to draw the stairs, so let's look for a pattern. We'll use a loop to eliminate the redundancy like we did for the last pyramid, but with appropriate modifications. Each stair's height is now 20 pixels, and each stair's width is now 20 times the number for that stair. The x- and y-coordinates are a bit trickier. The x-coordinate formula is similar to the `(100 - stairWidth) / 2` from before, but this time it must be shifted right by 80 to account for the position of its bounding box's top-left corner. The y-coordinate must similarly be shifted downward by 140 pixels. Here's the correct loop:

```
// draws the 5 "stairs" of the red pyramid
for (int i = 0; i < 5; i++) {
    int stairWidth = 20 * (i + 1);
    int stairHeight = 20;
    int stairX = 80 + (100 - stairWidth) / 2;
    int stairY = 140 + 20 * i;
    g.drawRect(stairX, stairY, stairWidth, stairHeight);
}
```

Can you spot the pattern between the two `for` loops used to draw the stairs of the pyramids? The x- and y-coordinates differ only in the addition of the offset from (0, 0) in the second loop. The stairs' widths and heights differ only in that one pyramid's stairs are 20 pixels tall and the other pyramid's stairs are 10 pixels tall (the result of dividing the overall size of 100 by the number of stairs).

Using the preceding information, let's turn the code for drawing a pyramid into a method that we can call three times to avoid redundancy. The parameters will be the (x, y) coordinates of the top-left corner of the pyramid's bounding box and the number of stairs in the pyramid. We'll also need to pass `Graphics g` as a parameter so that we can draw onto the `DrawingPanel`. We'll modify the `for` loop to compute the stair height first, then use the height to compute the stair width, and finally use the width and height to help compute the (x, y) coordinates of the stair. Here's the code:

```
public static void drawPyramid(Graphics g, int x,
                               int y, int stairs) {
    // draws the border rectangle
    g.drawRect(x, y, 100, 100);

    // draws the stairs of the pyramid
    for (int i = 0; i < stairs; i++) {
        int stairHeight = 100 / stairs;
        int stairWidth = stairHeight * (i + 1);
        int stairX = x + (100 - stairWidth) / 2;
        int stairY = y + stairHeight * i;
        g.drawRect(stairX, stairY, stairWidth, stairHeight);
    }
}
```

The preceding code is now generalized to draw a pyramid at any location with any number of stairs. But one final ingredient is missing: the ability to give a different color to each pyramid.

Complete Structured Solution

The preceding code is correct except that it doesn't allow us to draw the pyramids in the proper colors. Let's add an additional parameter, a `Color`, to our method and use it to fill the pyramid stairs as needed. We'll pass `Color.WHITE` as this parameter's value for the first white pyramid; it'll fill the stairs with white, even though this isn't necessary.

The way to draw a filled shape with an outline of a different color is to first fill the shape, then use the outline color to draw the same shape. For example, to get red rectangles with black outlines, first we'll use `fillRect` with red, then we'll use `drawRect` with black with the same parameters.

Here's the new version of the `drawPyramid` method that uses the fill color as a parameter:

```
public static void drawPyramid(Graphics g, Color c,
                               int x, int y, int stairs) {
    g.drawRect(x, y, 100, 100);

    for (int i = 0; i < stairs; i++) {
        int stairHeight = 100 / stairs;
        int stairWidth = stairHeight * (i + 1);
        int stairX = x + (100 - stairWidth) / 2;
        int stairY = y + stairHeight * i;

        g.setColor(c);
        g.fillRect(stairX, stairY, stairWidth, stairHeight);
        g.setColor(Color.BLACK);
        g.drawRect(stairX, stairY, stairWidth, stairHeight);
    }
}
```

Using this method, we can now draw all three pyramids easily by calling `drawPyramid` three times with the appropriate parameters:

```
drawPyramid(g, Color.WHITE, 0, 0, 10);
drawPyramid(g, Color.RED, 80, 140, 5);
drawPyramid(g, Color.BLUE, 220, 50, 20);
```

One last improvement we can make to our `Pyramids` program is to turn the overall pyramid size of 100 into a constant, so there aren't so many 100s lying around in the code. Here is the complete program:

```
1  // This program draws three colored pyramid figures.
2
3  import java.awt.*;
4
5  public class Pyramids {
6      public static final int SIZE = 100;
7
8      public static void main(String[] args) {
9          DrawingPanel panel = new DrawingPanel(350, 250);
10         Graphics g = panel.getGraphics();
11
12         drawPyramid(g, Color.WHITE, 0, 0, 10);
13         drawPyramid(g, Color.RED, 80, 140, 5);
14         drawPyramid(g, Color.BLUE, 220, 50, 20);
15     }
16
17     // draws one pyramid figure with the given
```

```
18      // number of stairs at the given (x, y) position
19      // with the given color
20      public static void drawPyramid(Graphics g, Color c,
21                                     int x, int y, int stairs) {
22
23          // draws the border rectangle
24          g.drawRect(x, y, SIZE, SIZE);
25
26          // draws the stairs of the pyramid
27          for (int i = 0; i < stairs; i++) {
28              int stairHeight = SIZE / stairs;
29              int stairWidth = stairHeight * (i + 1);
30              int stairX = x + (SIZE - stairWidth) / 2;
31              int stairY = y + stairHeight * i;
32
33              // fills the rectangles with the fill colors
34              g.setColor(c);
35              g.fillRect(stairX, stairY, stairWidth, stairHeight);
36
37              // draws the black rectangle outlines
38              g.setColor(Color.BLACK);
39              g.drawRect(stairX, stairY, stairWidth, stairHeight);
40          }
41      }
42  }
```

Chapter Summary

`DrawingPanel` is a custom class provided by the authors to easily show a graphical window on the screen. A `DrawingPanel` contains a `Graphics` object that can be used to draw lines, text, and shapes on the screen using different colors.

A `Graphics` object has many useful methods for drawing shapes and lines, such as `drawLine`, `fillRect`, and `setColor`. Shapes can be "drawn" (drawing only the outline) or "filled" (coloring the entire shape).

The `Graphics` object can write text on the screen with its `drawString` method. You can specify different font styles and sizes with the `setFont` method.

Graphical programs that are decomposed into methods must pass appropriate parameters to those methods (for example, the `Graphics` object, as well as any (x, y) coordinates, sizes, or other values that guide the figures to be drawn).

Self-Check Problems

Section 3G.1: Introduction to Graphics

1. Which of the following is the correct syntax to draw a rectangle?

   a. `Graphics g.drawRect(10, 20, 50, 30);`
   b. `g.drawRect(10, 20, 50, 30);`
   c. `g.draw.rectangle(10, 20, 50, 30);`
   d. `Graphics.drawRect(10, 20, 50, 30);`
   e. `g.drawRect(x = 10, y = 20, width = 50, height = 30);`

2. There are two mistakes in the following code, which attempts to draw a line from coordinates (50, 86) to (20, 35). What are they?

```
DrawingPanel panel = new DrawingPanel(200, 200);
panel.drawLine(50, 20, 86, 35);
```

3. The following code attempts to draw a black-filled outer rectangle with a white-filled inner circle inside it:

```
DrawingPanel panel = new DrawingPanel(200, 100);
Graphics g = panel.getGraphics();
g.setColor(Color.WHITE);
g.fillOval(10, 10, 50, 50);
g.setColor(Color.BLACK);
g.fillRect(10, 10, 50, 50);
```

   However, the graphical output looks like Figure 3G.17 instead. What must be changed for it to look as intended?

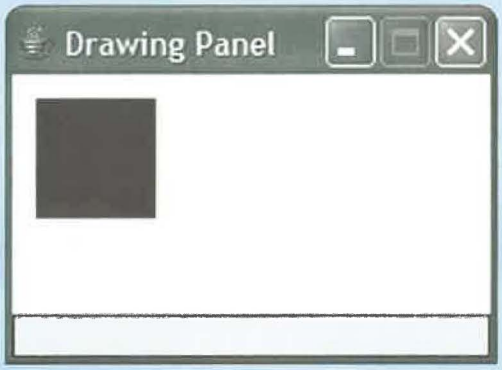


Figure 3G.17

4. The following code attempts to draw a black rectangle from (10, 20) to (50, 40) with a line across its diagonal:

```
DrawingPanel panel = new DrawingPanel(200, 100);
Graphics g = panel.getGraphics();
g.drawRect(10, 20, 50, 40);
g.drawLine(10, 20, 50, 40);
```

   However, the graphical output looks like Figure 3G.18 instead. What must be changed for it to look as intended?

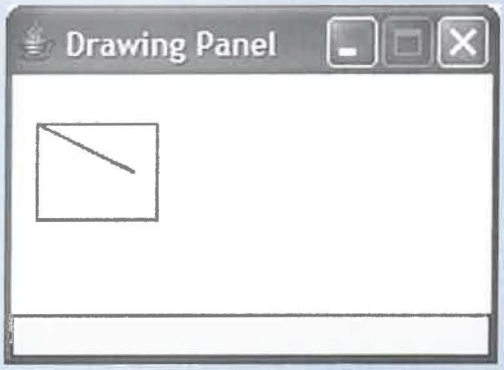


Figure 3G.18

5. What sort of figure will be drawn by the following program? Can you draw a picture that will approximately match its appearance without running it first?

```
1  import java.awt.*;
2
3  public class Draw7 {
4      public static void main(String[] args) {
5          DrawingPanel panel = new DrawingPanel(200, 200);
6          Graphics g = panel.getGraphics();
7          for (int i = 0; i < 20; i++) {
8              g.drawOval(i * 10, i * 10, 200 - (i * 10), 200 - (i * 10));
9          }
10     }
11 }
```

Exercises

1. Write a program that uses the `DrawingPanel` to draw Figure 3G.19.

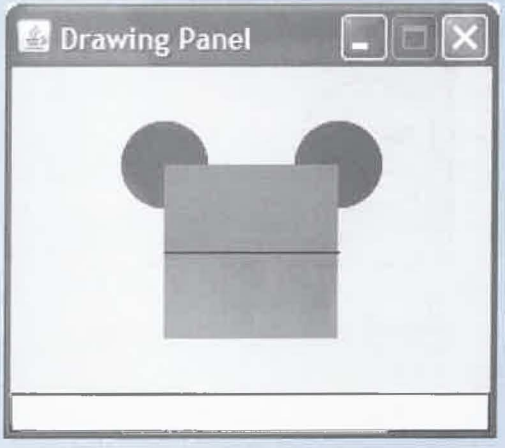


Figure 3G.19

The window is 220 pixels wide and 150 pixels tall. The background is yellow. There are two blue ovals of size 40 × 40 pixels. They are 80 pixels apart, and the left oval's top-left corner is located at position (50, 25). There is a red square whose top two corners exactly intersect the centers of the two ovals. Lastly, there is a black horizontal line through the center of the square.

2. Modify your program from the previous exercise to draw the figure by a method called `drawFigure`. The method should accept three parameters: the `Graphics g` of the `DrawingPanel` on which to draw, and a pair of (x, y) coordinates specifying the location of the top-left corner of the figure. Use the following heading for your method:

```
public static void drawFigure(Graphics g, int x, int y)
```

Set your `DrawingPanel`'s size to 450 × 150 pixels, and use your `drawFigure` method to place two figures on it, as shown in Figure 3G.20. One figure should be at position (50, 25) and the other should be at position (250, 45).

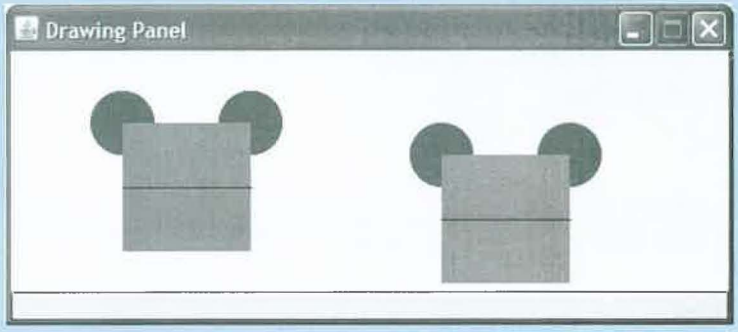


Figure 3G.20

3. Suppose you have the following existing program called `Face` that uses the `DrawingPanel` to draw the face figure shown in Figure 3G.21. Modify the program to draw the modified output shown in Figure 3G.22. Do so by writing a parameterized method that draws a face at different positions. The window size should be changed to 320 × 180 pixels, and the two faces' top-left corners are at (10, 30) and (150, 50).

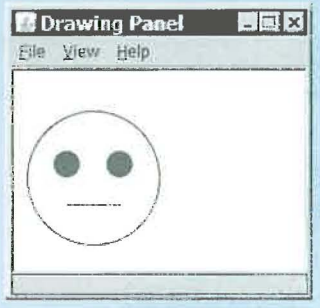


Figure 3G.21

```
1  public class Face {
2      public static void main(String[] args) {
3          DrawingPanel panel = new DrawingPanel(220, 150);
```

```
4           Graphics g = panel.getGraphics();
5
6           g.setColor(Color.BLACK);
7           g.drawOval(10, 30, 100, 100);    // face outline
8
9           g.setColor(Color.BLUE);
10          g.fillOval(30, 60, 20, 20);      // eyes
11          g.fillOval(70, 60, 20, 20);
12
13          g.setColor(Color.RED);           // mouth
14          g.drawLine(40, 100, 80, 100);
15      }
16  }
```

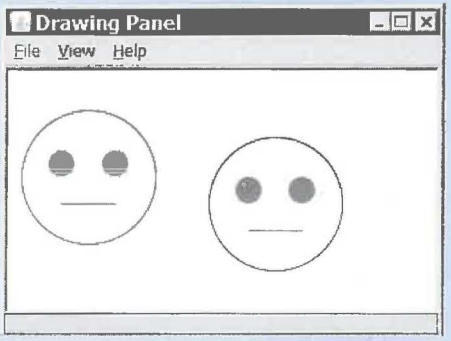


Figure 3G.22

4. Modify your previous Face program to draw the new output shown in Figure 3G.23. The window size should be changed to 520 × 180 pixels, and the faces' top-left corners are at (10, 30), (110, 30), (210, 30), (310, 30), and (410, 30).

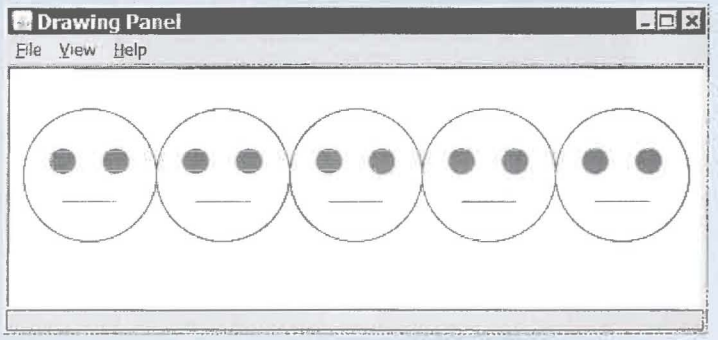


Figure 3G.23

5. Write a program called `ShowDesign` that uses the `DrawingPanel` to draw Figure 3G.24.

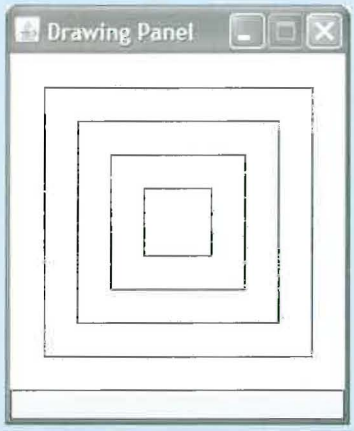


Figure 3G.24

The window is 200 pixels wide and 200 pixels tall. The background is white and the foreground is black. There are 20 pixels between each of the four rectangles, and the rectangles are concentric (their centers are at the same point). Use a loop to draw the repeated rectangles.

6. Modify your `ShowDesign` program from the previous exercise so that it has a method that accepts parameters for the window width and height and displays the rectangles at the appropriate sizes. For example, if your method was called with values of 300 and 100, the window would look like Figure 3G.25.

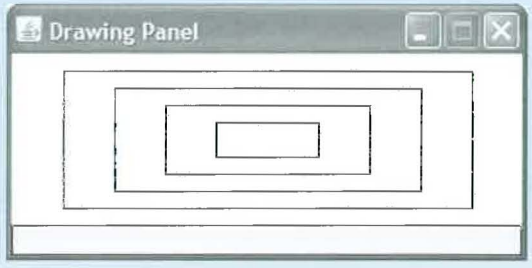


Figure 3G.25

7. Write a program called `Squares` that uses the `DrawingPanel` to draw the shape shown in Figure 3G.26.

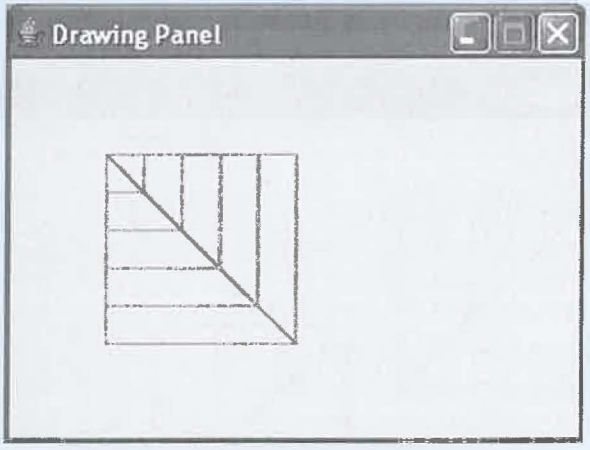


Figure 3G.26

The `DrawingPanel` is 300 pixels wide by 200 pixels high. Its background is cyan. The horizontal and vertical lines are drawn in red and the diagonal line is drawn in black. The upper-left corner of the diagonal line is at (50, 50). Successive horizontal and vertical lines are spaced 20 pixels apart.

8. Modify your code from the previous exercise to produce the pattern shown in Figure 3G.27.

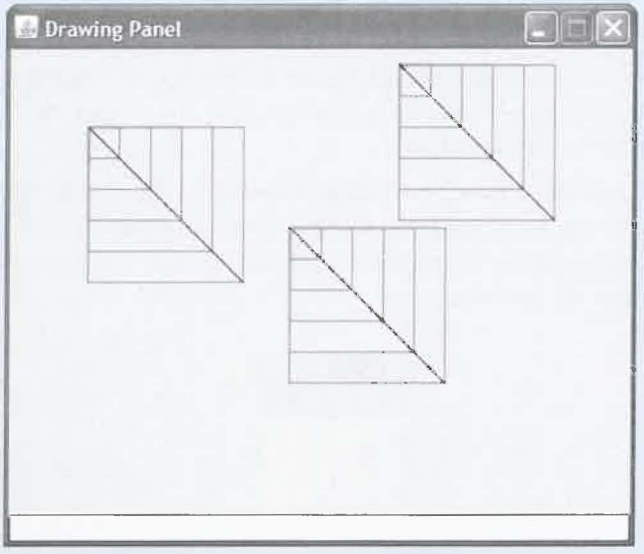


Figure 3G.27

The `DrawingPanel` is now 400 × 300 pixels in size. The first figure is at the same position, (50, 50). The other figures are at positions (250, 10) and (180, 115), respectively. Use one or more parameterized static methods to reduce the redundancy of your solution.

9. Modify your code from the previous exercise to produce the pattern shown in Figure 3G.28.

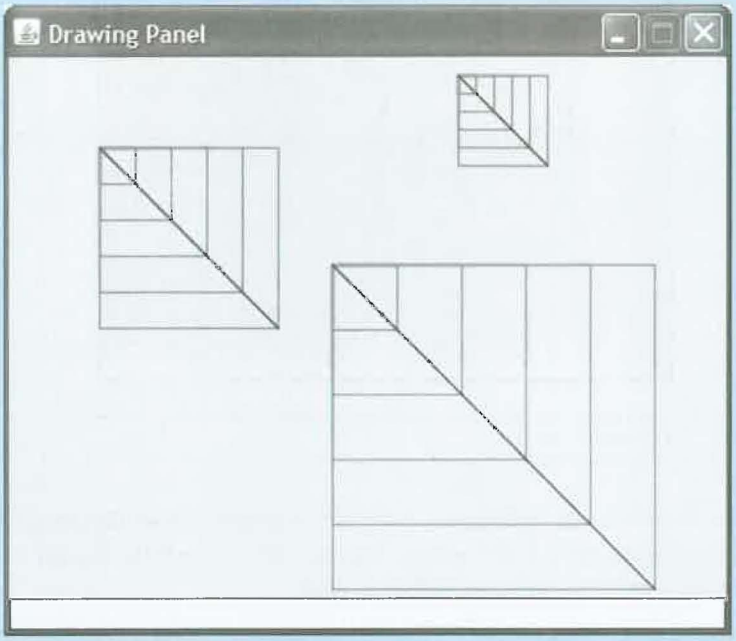


Figure 3G.28

The `DrawingPanel` is the same except that now each figure has a different size. The left figure has its original size of 100, the top-right figure has a size of 50, and the bottom-right figure has a size of 180. Use parameterized static methods to reduce the redundancy of your solution.

10. Write a program called `Stairs` that uses the `DrawingPanel` to draw the figure shown in Figure 3G.29. The first stair's top-left corner is at position (5, 5). The first stair is 10 × 10 pixels in size. Each stair is 10 pixels wider than the one above it. Make a table with the (*x*, *y*) coordinates and (*width* × *height*) sizes of the first five stairs. Note which values change and which ones stay the same.

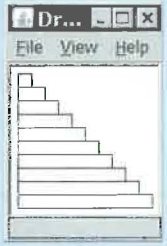


Figure 3G.29

11. Modify your previous `Stairs` program to draw each of the outputs shown in Figure 3G.30. Modify only the body of your loop. (You may want to make a new table to find the expressions for *x*, *y*, *width*, and *height* for each new output.)

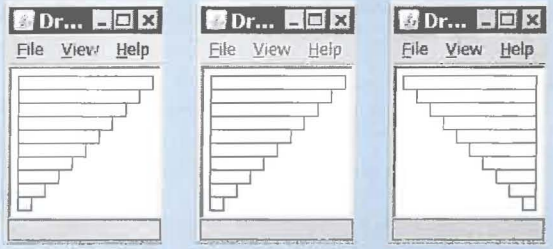


Figure 3G.30

12. Write a program called `Triangle` that uses the `DrawingPanel` to draw the figure shown in Figure 3G.31.

Figure 3G.31

The window is 600 × 200 pixels in size. The background is yellow and the lines are blue. The lines are 10 pixels apart vertically, and the diagonal lines intersect at the bottom of the figure in its horizontal center.

13. Write a program called `Spiral` that uses the `DrawingPanel` to draw the figure shown in Figure 3G.32.

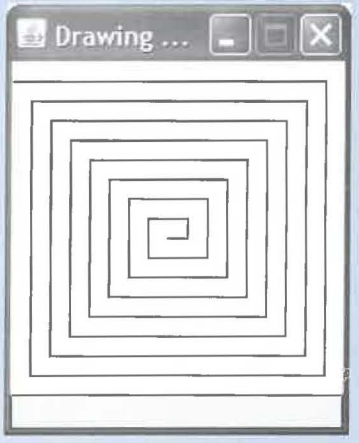


Figure 3G.32

The window is 170 × 170 pixels. The background is white and the foreground is black. The spiral line begins at point (0, 10) and spirals inward. There are 10 pixels between each arm of the spiral and eight spirals in all. The initial spiral touches points (0, 10), (160, 10), (160, 160), (10, 160), and (10, 20).

For an additional challenge, parameterize your program with parameters such as the window size and the number of spiral loops desired.

Programming Projects

1. Write a program that draws the patterns shown in Figure 3G.33 onto a `DrawingPanel`.

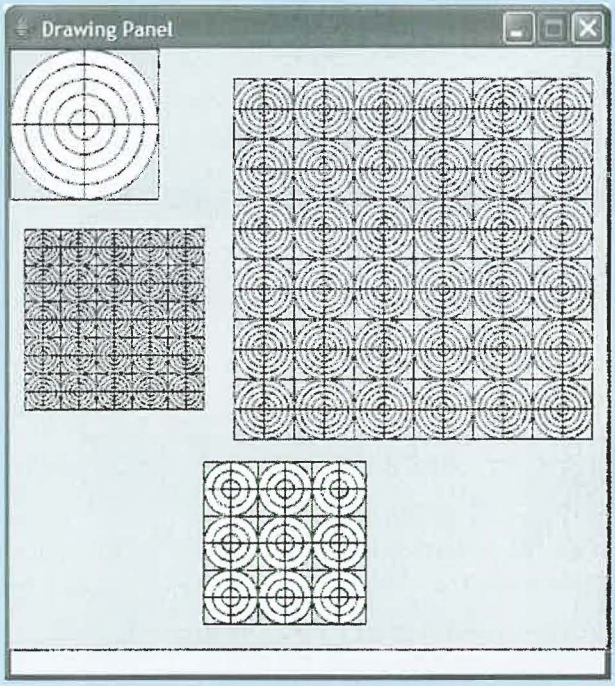


Figure 3G.33

The `DrawingPanel`'s size is 400 × 400 pixels and its background color is cyan. It contains four figures of concentric yellow circles with black outlines, all surrounded by a green rectangle with a black outline. The four figures on your `DrawingPanel` should have the properties shown in Table 3G.7.

Table 3G.7 Circle Figure Properties

Description	(*x*, *y*) position	Size of subfigures	Number of circles	Number of rows/cols
top left	(0, 0)	100 × 100	5	1 × 1
bottom left	(10, 120)	24 × 24	4	5 × 5
top right	(150, 20)	40 × 40	5	6 × 6
bottom right	(130, 275)	36 × 36	3	3 × 3

Break down your program into methods for drawing one subfigure as well as larger grids of subfigures, such as the 5 × 5 grid at (10, 120).

2. Write a program that draws the image shown in Figure 3G.34 onto a `DrawingPanel` of size 200 × 200. Each stamp is 50 × 50 pixels in size.

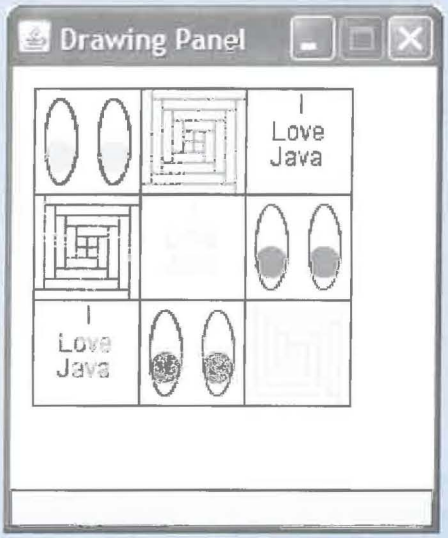


Figure 3G.34

3. Write a program that draws checkerboards like these shown in Figure 3G.35 onto a `DrawingPanel` of size 420 × 300.

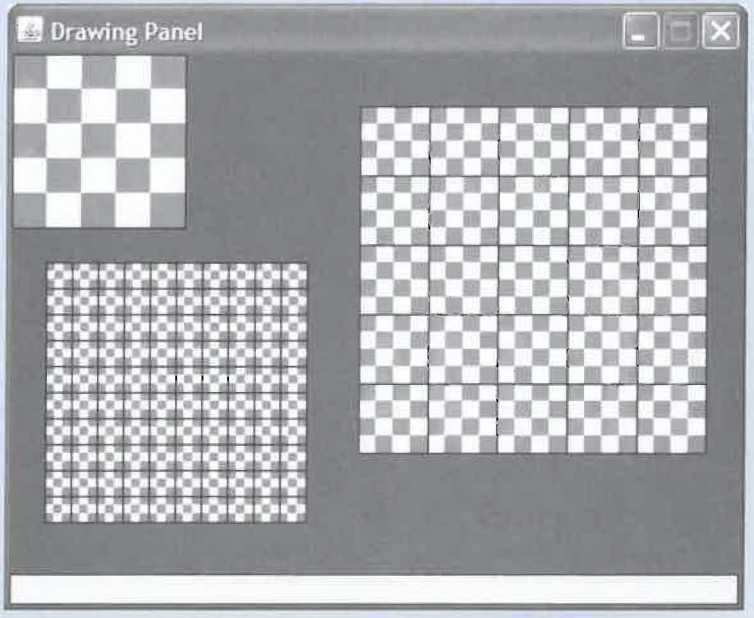


Figure 3G.35

4. Write a modified version of the `Projectile` case study program from Chapter 3 that draws a graph of the projectile's flight onto a `DrawingPanel` of size 420 × 220. For example, the panel shown in Figure 3G.36 draws a projectile with an initial velocity of 30 meters per second, an angle of 50 degrees, and 10 steps.

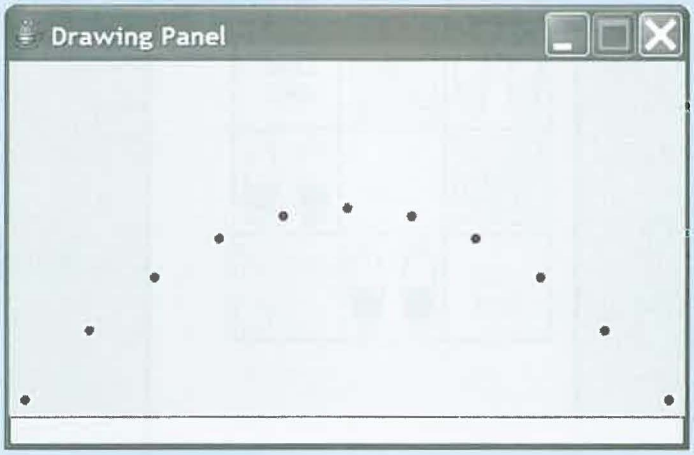


Figure 3G.36

5. Write a program that draws the image shown in Figure 3G.37 onto a `DrawingPanel` of size 650 × 400. The image represents a famous optical illusion called the "Cafe Wall," in which a series of straight squares appears to be slanted.

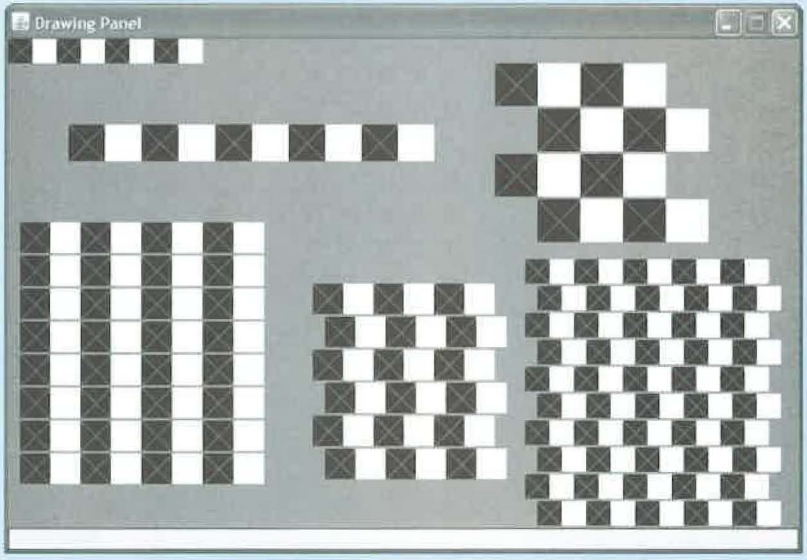


Figure 3G.37

The image has a gray background and many rows of black and white squares with a blue X drawn through each black square. The two free-standing rows in the diagram have the following properties:

Table 3G.8 Cafe Wall Row Properties

Description	(*x*, *y*) position	Number of pairs	Size of each box
upper-left	(0, 0)	4	20
mid-left	(50, 70)	5	30

The diagram has four grids of rows of squares, with 2 pixels of vertical space between adjacent rows. A key aspect of the optical illusion is that every other row is shifted horizontally by a particular offset. The four grids have the following properties:

Table 3G.9 Cafe Wall Grid Properties

Description	(*x*, *y*) position	Number of pairs	Size of each box	2nd row offset
lower-left	(10, 150)	4	25	0
lower-middle	(250, 200)	3	25	10
lower-right	(425, 180)	5	20	10
upper-right	(400, 20)	2	35	35