Second AP[*] Edition
— with GridWorld

Java

Methods

Object-Oriented Programming
and
Data Structures

Maria Litvin

Phillips Academy, Andover, Massachusetts

Gary Litvin

Skylight Software, Inc.

Height | **Chapter 16** | Ascent
Baseline
Descent

Graphics

16.1   Prologue

What you see on your computer screen is ultimately determined by the contents of the video memory (*VRAM*) on the graphics adapter card. The video memory represents a rectangular array of *pixels* (picture elements). Each pixel has a particular color, which can be represented as a mix of red, green, and blue components, each with its own intensity. A typical graphics adapter may use eight bits to represent each of the red, green, and blue intensities (in the range from 0 to 255), so each color is represented in 24 bits (that is, three bytes). This allows for 2^{24} = 16,777,216 different colors. With a typical screen resolution of 1680 by 1050 pixels, your adapter needs $1680 \cdot 1050 \cdot 3$ bytes — a little over 5 MB — to hold the picture for one screen. The picture is produced by setting the color of each pixel in VRAM. The video hardware scans the whole video memory continuously and refreshes the image on the screen.

A graphics adapter is what we call a *raster* device: each individual pixel is changed separately from other pixels. (This is different from a *vector* device, such as a plotter, which can draw a line directly from point *A* to point *B*.) To draw a red line or a circle on a raster device, you need to set just the right group of pixels to the red color. That's where a graphics package can help: you certainly don't want to program all those routines for setting pixels yourself.

A typical graphics package has functions for setting drawing attributes, such as color, line style and width, fill texture or pattern for filled shapes, and font for text, and another set of functions for drawing simple shapes: lines, arcs, circles and ovals, rectangles, polygons, text, and so on. Java's graphics capabilities are based on the `Graphics` class and the `Graphics2D` package. The `Graphics` class is pretty rudimentary: it lets you set the color and font attributes and draw lines, arcs, ovals (including circles), rectangles, rectangles with rounded corners, polygons, polylines (open polygons), images, and text. There are "draw" and "fill" methods for each basic shape (for example, `drawRect` and `fillRect`).

The `Graphics2D` class is derived from `Graphics` and inherits all its methods. It works with a number of related interfaces and classes:

- The `Shape` interface and classes that implement it (`Line2D`, `Rectangle2D`, `Ellipse2D`, etc.) define different geometric shapes.

- The `Stroke` interface and one implementation of it, `BasicStroke`, represent in a very general manner the line width and style for drawing lines.

- The `Paint` interface and its implementations `Color`, `GradientPaint`, and `TexturePaint` represent a color, a color gradient (gradually changing color), and a texture for filling in shapes, respectively.

`Graphics2D` also adds methods for various coordinate transformations, including rotations.

More importantly, `Graphics2D` adds a capability for treating shapes polymorphically. In the `Graphics` class, contrary to the OOP spirit, shapes are not represented by objects, and there is a separate special method for drawing each shape. Suppose you are working on a drawing editor program that allows you to add different shapes to a picture. You keep all the shapes already added to the picture in some kind of a list. To redraw the picture you need to draw all the shapes from the list. With `Graphics` you have to store each shape's identification (such as "circle," "rectangle," etc.) together with its dimensions and position and use a `switch` or an `if-else` statement to call the appropriate drawing method for each shape.

With `Graphics2D`, you can define different shapes (objects of classes that implement the `Shape` interface) and store references to them in your list. Each shape provides a "path iterator" which generates a sequence of points on that shape's contour. These points are used by `Graphics2D`'s `draw(Shape s)` and `fill(Shape s)` methods. Thus, shapes are treated in a polymorphic manner and at the same time are drawn using the currently selected `Paint` and `Stroke`. If your own class implements `Shape` and supplies a `getPathIterator` method for it, then your "shapes" will be drawn properly, too, due to polymorphism.

Like any package with very general capabilities, `Graphics2D` is not easy to use. We will stay mostly within the limits of the `Graphics` class, but, if you are adventurous, you can examine the `Graphics2D` API and learn to use some of its fancy features.

In the following sections we will examine Java's event-driven graphics model and review the basic drawing attributes and methods of the `Graphics` class. We will then use its capabilities to write a simple *Puzzle* program in which a player rearranges the pieces of a scrambled picture.

16.2 `paint`, `paintComponent`, and `repaint`

In Java, the hardest thing may be figuring out <u>when</u> and <u>where</u> to draw, rather than <u>how</u>. Java's graphics are necessarily event-driven because applets and applications run under multitasking operating systems. Suppose you are playing Solitaire when all of a sudden you decide to check your e-mail. You bring up your e-mail and its window overlaps a part of the Solitaire window. When you close or minimize your e-mail application, the operating system has to redisplay the Solitaire window. The operating system sends the *Solitaire* program a message that its window has been partially wiped out and now needs to be "repainted." *Solitaire* must be ready to dispatch a method in response to this "repaint" message.

In Java, this method is called `paint`. `paint` is a `void` method that receives one parameter of the type `Graphics`, usually named `g`:

```
public void paint(Graphics g)
{
  ...
}
```

`g` defines the graphics context: the size and position of the picture, the current attributes (color, font), the clipping area for the picture, and so on.

An object of the `JApplet` or `JFrame` type has a default `paint` method that calls the `paintComponent` method for each of the components (buttons, labels, text edit fields, etc.) in the object's "content pane" container. If you derive your main class from `JApplet` or `JFrame`, which is usually the case, and you want to do some drawing on your window, you can override the inherited `paint` method by defining your own. The first statement in your `paint` most likely will be a call to the base class's `paint`:

```
public void paint(Graphics g)
{
  super.paint(g);
  ...
}
```

After that you can add your own statements. That's exactly what we did in J_M\Ch02\HelloGui\HelloApplet.java:

```
public void paint(Graphics g)
{
  super.paint(g);    // call JApplet's paint method
                     //  to paint the background
  g.setColor(Color.RED);
  g.drawRect(25, 40, 150, 45);  // draw a rectangle 150 by 45
  g.setColor(Color.BLUE);
  g.drawString("Hello, Applet!", 60, 65);
}
```

Naturally, `paint` does not have to define all the drawing work in its own code. It can call other methods, passing `g` to them as one of the parameters.

`paint` is called automatically in response to certain messages received from the operating system. But sometimes your program needs to repaint the window itself after changing its appearance. It can't call `paint` directly because it does not have a valid `Graphics` parameter, a `g`, to pass to it. Instead your program calls the `repaint` method, which does not take any parameters. `repaint` places a request to repaint the window into the event queue, and in due time the `paint` method is invoked.

`paint` is the central drawing method for your application window, where all drawing originates. Therefore, it must handle all the different drawing requirements for all the different situations in which the application might find itself. This is not easy. Fortunately, in Java's *Swing* GUI package, you can redefine the painting of individual GUI components. Each type of component (a `JButton` object, a `JTextField` object, etc.) has its own default `paintComponent` method. `paintComponent` also takes one parameter, `Graphics g`. You can derive a class from any of these classes and redefine `paintComponent` in it. It is not very common, though, to draw on top of buttons or text edit fields. But your class can extend `JComponent` or `JPanel`. The default `paintComponent` method in `JPanel` just paints its background. You can derive your own class from `JPanel` and add your drawing in your own `paintComponent` method. Again, it will usually start by calling the base class's `paintComponent`.

That is precisely what we did in *Hello Graphics*, *Craps*, *Snack Bar*, *Chomp*, *Dance Studio* and other programs. In *Craps*, for instance, we derived a class `CrapsTable` from `JPanel` (J_M\Ch07\Craps\CrapsTable.java):

```
public class CrapsTable extends JPanel
  ...
```

We then provided our own `paintComponent` method for it:

```
public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  die1.draw(g);
  die2.draw(g);
}
```

When you need to repaint a component, you call its `repaint` method. (A component's `repaint` is different from `JFrame`'s `repaint`: it calls `paintComponent` only for this particular component.) In *Craps*, when the dice roll, we adjust their positions and call `table`'s `repaint`:

```
// Processes timer events
public void actionPerformed(ActionEvent e)
{
  if (diceAreRolling())
  {
    ...
  }
  else
  {
    ...
  }

  repaint();
}
```

> **Note that `repaint` just sends a request message to repaint the window or a component, and this request goes into the event queue. The actual painting may be postponed until your program finishes processing the current event and other events that are earlier in the event queue.**

By painting individual components in *Swing* you can implement smoother animations. Without this capability you would have to repaint the whole window when just one small part of it has changed. We will return to the subject of using `JPanel` and see another example later in this chapter, in the Pieces of the Puzzle case study.

❖   ❖   ❖

An insightful reader may wonder at this point: how can we call `Graphics2D` methods if all we get in `paint` or `paintComponent` is a reference to `Graphics g`? The truth is that `g` is a `Graphics2D` reference in disguise. It is presented as `Graphics` simply for compatibility with earlier versions of Java. To use it for

calling `Graphics2D` methods you simply have to cast it into `Graphics2D`. For example:

```
Graphics2D g2D = (Graphics2D)g;
g2D.setPaint(new GradientPaint(0, 0, Color.RED,
                               100, 100, Color.BLUE, true));
< ... etc. >
```

16.3   Coordinates

The graphics context `g`, passed to `paint` and `paintComponent`, defines the coordinate system for the drawing. As in most computer graphics packages, the *y*-axis points down, not up as in math (Figure 16-1).

> **By default, the `Graphics` class places the origin at the <u>upper</u>-left corner of the content area of the application window (for `paint`) or in the upper-left corner of the component (for `paintComponent`). The coordinates are <u>integers</u>, and the units are pixels.**

The `translate(x, y)` method shifts the origin to the point (x, y). In the `Graphics2D` class, there are methods to scale and rotate the coordinates.
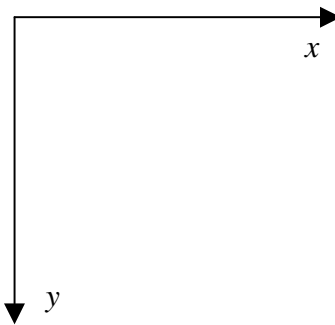

Figure 16-1. Graphics coordinates: *y*-axis points down

`Graphics` also sets the *clipping rectangle* to the window's drawing area or the component's drawing area. Anything outside the clipping rectangle does not show up on the screen. Therefore, one component's `paintComponent` method usually

can't paint over other components that do not overlap with it. `Graphics` has a method `setClip` for redefining the clipping rectangle.

What about scaling? As a programmer, you decide what happens when the application window is resized. In some applications (like *Craps* or *Snack Bar*) you may want to simply disallow resizing the window (by calling `JFrame`'s method `setResizable(false)`). In other programs (like *Lipogrammer*) you may rely on Java's component layout manager to adjust the positions of the components on the window. You may choose to adjust the <u>positions</u> of some of the graphics elements in the picture, but not their sizes. Or you may want everything scaled, as in the *Puzzle* program later in this chapter.

You may lose some precision when scaling coordinates, but the high resolution (number of pixels per unit length) of modern graphics adapters makes these inaccuracies hardly visible.

> **Each component provides `getWidth` and `getHeight` methods that return the current width and height of the component in pixels. You can scale the coordinates based on these values.**

Suppose you want to draw a filled red rectangle with its center at the center of the panel and its size equal to 75 percent of the panel's size. On top of it you want to draw a filled blue oval inscribed into the rectangle (Figure 16-2). This can be accomplished as follows:

```
public void paintComponent(Graphics g)
{
  super.paintComponent(g);  // call JPanel's paintComponent

  int width = getWidth();
  int height = getHeight();
  int xSize = (int)(.75 * width);
  int ySize = (int)(.75 * height);
  int x0 = width/2 – xSize/2;     // Coordinates of the
  int y0 = height/2 – ySize/2;    //   upper-left corner

  g.setColor(Color.RED);
  g.fillRect(x0, y0, xSize, ySize);
  g.setColor(Color.BLUE);
  g.fillOval(x0, y0, xSize, ySize);
}
```
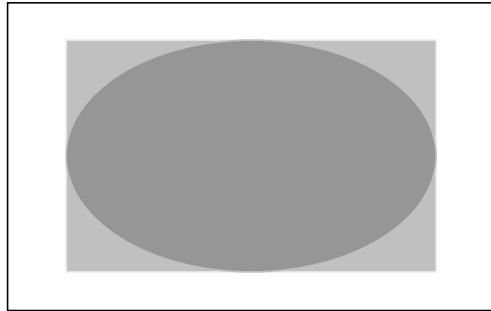
Figure 16-2.   A filled oval inscribed into a filled rectangle

The above example shows not only how a drawing can be scaled, but also how the positions of simple shapes (rectangles, ovals) are passed to drawing methods. The position and size of a rectangle are described by the *x* and *y* coordinates of its upper-left corner and by its width and height. In the above example we subtract half the size of the rectangle from the coordinates of the center of the panel to determine where the upper-left corner should be.

> **The position and size of a rounded rectangle, an oval, and even an arc are described by the position of the rectangle in which those shapes are inscribed (Figure 16-3).**

In the above code the same parameters are passed to `fillRect` and `fillOval` because the oval is inscribed into the rectangle.

Figure 16-3.   Positioning of ovals, arcs, and rounded rectangles

16.4   Colors

We have already used the `setColor` method whose parameter is a `Color` object. The `Color` class has thirteen predefined constants for colors (`WHITE`, `BLACK`, `GRAY`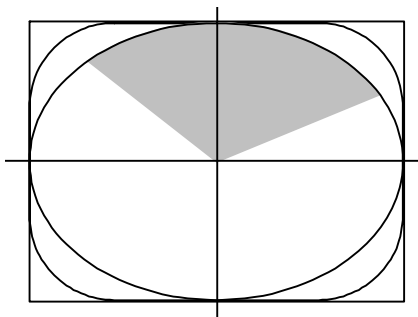, `LIGHT_GRAY`, `DARK_GRAY`, `BLUE`, `GREEN`, `CYAN`, `RED`, `MAGENTA`, `PINK`, `ORANGE`, and `YELLOW`). You can also construct your own color by specifying its red, green, and blue components:

```
int redValue = 18, greenValue = 50, blueValue = 255;
Color c = new Color(redValue, greenValue, blueValue);
g.setColor(c);
```

or simply:

```
g.setColor(new Color(18, 50, 255));
```

A `Color` object also has the methods `brighter` and `darker`, which return a new color made from the original color by adjusting its brightness. For example:

```
g.setColor(Color.ORANGE.darker().darker());
```

> **You can set the background color for a component by calling that component's `setBackground` method. This method only specifies the new background color but does not automatically repaint the component. If you set the background inside your `paintComponent` method, do it before calling `super.paintComponent`. If you do it elsewhere, call `repaint`.**

The Java default color for components is gray, and we often change it to white:

```
setBackground(Color.WHITE);
```

RGB numbers for different colors can be obtained from painting and imaging programs and color choosing tools on the web☞rgbcolors or from Java's own `JColorChooser` class.

16.5   Drawing Shapes

Figure 16-4 summarizes the drawing methods of the Graphics class.

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

g.clearRect(x, y, width, height);
g.drawRect(x, y, width, height);
g.fillRect(x, y, width, height);

g.drawOval(x, y, width, height);
g.fillOval(x, y, width, height);
g.drawRoundRect(x, y, width, height, horzDiam, vertDiam);
g.fillRoundRect(x, y, width, height, horzDiam, vertDiam);

g.draw3DRect(x, y, width, height, isRaised);
g.fill3DRect(x, y, width, height, isRaised);

g.drawArc(x, y, width, height, fromDegree, measureDegrees);
g.fillArc(x, y, width, height, fromDegree, measureDegrees);

g.drawPolygon(xCoords, yCoords, nPoints);
g.fillPolygon(xCoords, yCoords, nPoints);
g.drawPolyline(xCoords, yCoords, nPoints);

g.drawString(str, x, y);

g.drawImage(image, x, y, this);
```

Figure 16-4.   The drawing methods of the `Graphics` class

The drawLine(x1, y1, x2, y2) method draws a straight line from (x1, y1) to (x2, y2).

There are several methods for drawing and filling rectangles, ovals, and arcs: clearRect, drawRect, fillRect, drawRoundRect, fillRoundRect, draw3DRect, fill3DRect, drawOval, fillOval, drawArc, and fillArc. We have already used most of them in one project or another, so they should look familiar. The first four parameters in each of these methods are the same: the *x* and *y* coordinates of the upper-left corner, and the width and height of the bounding rectangle (as explained in Section 16.3). The clearRect method fills the rectangle

with the component's current background color.   The `drawRoundRect` and `fillRoundRect` methods take two additional parameters: the horizontal and vertical diameters of the oval used to round the corners. The `draw3DRect` and `fill3DRect` methods add a shadow on two sides to hint at a 3-D effect. Their fifth parameter can be either `true` for a "raised" rectangle or `false` for a "lowered" rectangle.

The `drawArc` and `fillArc` methods, respectively, draw and fill a fragment of an oval inscribed into the bounding rectangle. `fillArc` fills a sector of the oval (a slice of the pie) bound by the arc. The fifth and sixth parameters in these methods are the beginning angle (with the 0 at the easternmost point) and the measure of the arc in degrees (going counterclockwise).

❖   ❖   ❖

The `drawPolygon` and `fillPolygon` methods take three parameters: the array of *x*-coordinates of the vertices, the array of *y*-coordinates of the vertices, and the number of points:

```
drawPolygon(int[] xCoords, int[] yCoords, int n)
```

The number of points `n` should not exceed the smaller of `xCoords.length` and `yCoords.length`. As you can see, the `xCoords` and `yCoords` arrays do not have to be filled to capacity: they may hold fewer points than their size allows. This is convenient if you are adding points interactively or if you are reading them from a file and don't know in advance how many points you will end up with.

`drawPolygon` and `fillPolygon` automatically connect the last point to the first point and draw or fill a closed polygon, respectively. So

```
    g.drawPolygon(xCoords, yCoords, n);
```

is basically the same as:

```
    for (int i = 0; i < n – 1; i++)
    {
      g.drawLine(xCoords[i], yCoords[i], xCoords[i+1]; yCoords[i+1]);
    }
    g.drawLine(xCoords[n-1], yCoords[n-1], xCoords[0]; yCoords[0]);
```

The `drawPolyline` method works the same way as `drawPolygon`, but it does not connect the last point to the first.

16.6   Fonts and Text

The `setFont` method lets you set the font for drawing text. Java uses an object of the `Font` class to describe a font. `Font` objects are used for graphics text displayed with `g.drawString` and for text in various GUI components (`JLabel`, `JTextField`, `JTextArea`, `JButton`, etc.). The `Graphics` method for setting a font and the methods for setting a font in Swing components share the same name, `setFont`.

▎ **A font is described by its name, its style, and its size.**

Font names are system-dependent, but Java guarantees that at least three font names are always recognized:

- `"Serif"`, a proportional font, in which letters may have different widths, with serifs, or little decorative strokes, like Times Roman: ABCabc

- `"SansSerif"`, a `proportional` font without serifs, like Arial: ABCabc

- `"Monospaced"`, a fixed-`width` font where all characters have the same width, like Courier: ABCabc

The `GraphicsEnvironment` class has a `getAllFonts()` method that returns an array of all the fonts available in the system. For example:

```
GraphicsEnvironment env =
            GraphicsEnvironment.getLocalGraphicsEnvironment();
Font[] allFonts = env.getAllFonts();
```

Each font can come in four styles: `Font.PLAIN`, `Font.BOLD`, `Font.ITALIC`, or `Font.BOLD | Font.ITALIC` (the bit-wise combination of the two attributes, meaning both bold and italic).

The font size is specified in *points*. In typography, a point is 1/72 of an inch, but this measure loses its meaning when the text is scaled to a computer screen. For default coordinates, Java assumes that one point is equal to one pixel.

You can create all the fonts you need ahead of time (possibly, in the constructor for your drawing panel). For example:

```
Font font1 = new Font("Monospaced", Font.PLAIN, 20);
Font font2 = new Font("Serif", Font.BOLD, 30);
```

Then you set the font with setFont. For example:

```
g.setFont(font2);
```

If you intend to use a font only once, you can create an anonymous font on the fly:

```
g.setFont(new Font("Serif", Font.BOLD, 30));
```

For very precise text positioning, Graphics has a method getFontMetrics that returns a FontMetrics object. This object, in turn, has getAscent, getDescent, and getHeight methods that return the font's vertical measurements (Figure 16-5).

Figure 16-5.   Font metrics

The Graphics class's drawString(text, x, y) method draws the text string. This method positions the left end of the text's baseline at the point (*x*, *y*).

16.7   *Case Study and Lab:* **Pieces of the Puzzle**

In this section we will create a program *Puzzle* to implement a simple puzzle that involves rearranging pieces of a picture. The program first shows a picture made of nine pieces on a 3 by 3 grid. After two seconds, it scrambles the pieces randomly and shows the scrambled picture. The player has to restore the picture by moving the pieces around. There is an extra empty cell below the picture for holding a piece temporarily (Figure 16-6). The player can move a piece by "picking it up," then "dropping" it into the empty cell. To pick up a piece the player clicks on it. This is acknowledged by some feedback; for example, the picked piece gets a different background color. To "drop" the piece the player clicks on the empty cell.

Play with this program (click on J$_M$\Ch16\Puzzle\Puzzle.jar) to get a feel for how it works. In this version the initial picture is not very creative: it simply shows a circle and the numbers of the pieces (numbered from left to right and top to bottom like a telephone keypad).   Examine the code for the Puzzle class in J$_M$\Ch16\Puzzle. As you can see, the program processes one timer event and after that is driven by mouse events. The Puzzle class implements a MouseListener interface by providing its five required methods. Of them only mousePressed is used.   There is a way to not include unused methods (using a so-called *adapter class*), but we just use empty methods.
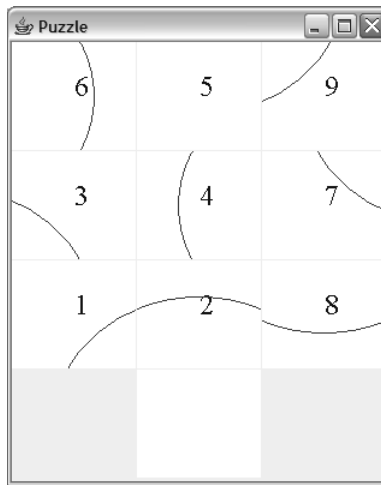

Figure 16-6.   The *Puzzle* program

The pieces of the puzzle are numbered from 1 to 9, and the number 0 represents the empty cell. Before scrambling, the pieces are arranged as follows:

```
1 2 3
4 5 6
7 8 9
  0
```

The program uses an array of cells to hold the pieces. Each cell in the puzzle "knows" which piece it is currently holding. That number is returned by the cell's getPieceNumber method and can be set by the cell's setPieceNumber(k)

method. In the initial non-scrambled picture, the index of each cell in the array matches the number of the piece displayed in it.

The logic for moving the pieces is pretty straightforward. When a mouse clicks on the program's window, the `mousePressed` method is called:

```
public void mousePressed(MouseEvent e)
{
  int x = e.getX();
  int y = e.getY();

  // Figure out the index of the cell that was clicked:
  int col = 3 * x / getWidth();
  int row = 4 * y / getHeight();
  int i = 3 * row + col;
  if (i >= 0 && i < 9)
    i++;
  else if (i == 10)
    i = 0;
  else
    return;

  if (pickedIndex < 0)
    pickPiece(i);
  else
    dropPiece(i);
}
```

It gets the coordinates of the click and figures out in which cell `i` it occurred. Then it can go two different ways, depending whether there is already a picked-up piece "hanging in the air" or not. The `Puzzle` class has a field `pickedIndex` that holds the index of the cell whose piece has been picked up. If there is no picked piece, `pickedIndex` is equal to –1. Then if the player has clicked on a non-empty cell, the puzzle piece is (logically) "lifted" from that cell. The cell is highlighted, and its index `i` is saved in `pickedIndex` for future use. The `pickPiece` method implements this (Figure 16-7).

If, on the other hand, there is already a piece "in the air" (`pickedIndex` ≥ 0) and the player has clicked on the empty cell (the cell holding the 0 piece), then the piece that was picked up earlier is "dropped" into that cell. The cell is updated to reflect that it now holds a piece with a particular number while the previously picked cell is set to empty. This is implemented in the `dropPiece` method (Figure 16-7).

Figure 16-8 shows a little state machine with two states that represents this logic. A *state machine* is a model that uses nodes to represent different possible states (of a system or a program) and connecting arrows to represent the rules for changing states.

```
...

private void pickPiece(int i)
{
  if (cells[i].getPieceNumber() != 0) // pick only non-empty cells
  {
    pickedIndex = i;
    cells[i].setPicked(true);
    cells[i].repaint();
  }
  else
  {
    bzz.play();
  }
}

private void dropPiece(int i)
{
  if (cells[i].getPieceNumber() == 0) // drop only on the empty cell
  {
    // Set the empty cell's number to the picked piece
    int k = cells[pickedIndex].getPieceNumber();
    cells[i].setPieceNumber(k);
    cells[i].repaint();

    // Set the piece number for the source cell to "empty"
    cells[pickedIndex].setPieceNumber(0);
    cells[pickedIndex].setPicked(false);
    cells[pickedIndex].repaint();

    pickedIndex = -1;    // nothing picked now
    if (allSet())
      bells.play();
    else
      drop.play();
  }
  else
  {
    bzz.play();
  }
}

...
```

Figure 16-7. `pickPiece` and `dropPiece` methods in the `Puzzle` class
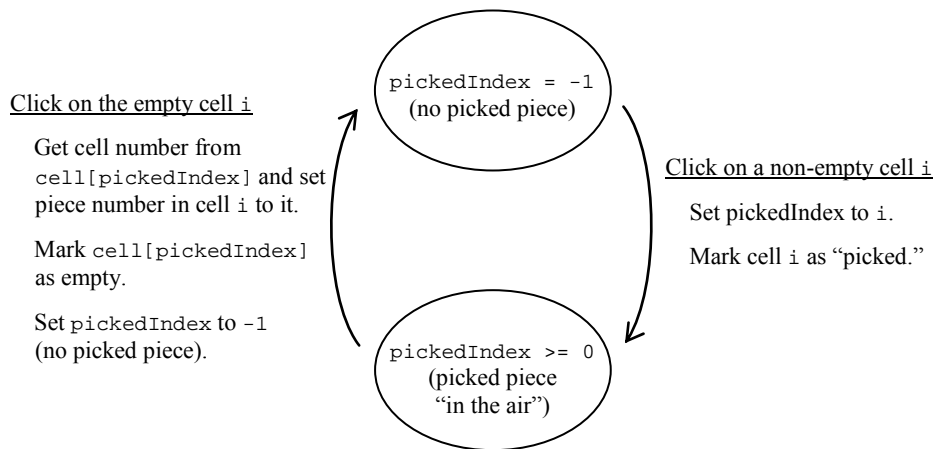**(J_M\Ch16\Puzzle\Puzzle.java)**

Figure 16-8. State machine diagram for moving pieces in the
***Puzzle* program**

❖   ❖   ❖

The trickier part is to display different pieces of the puzzle in different cells. The question is: How can we show a particular piece in a cell? One approach would be to have a separate method draw each of the nine pieces of the puzzle. This might work, but only for very simple pictures. In the real puzzle we want to use drawings whose lines cut across the grid lines; to draw separate fragments of them would be a nightmare.

A better approach would be to use <u>one</u> method to draw the whole picture, but show only the appropriate piece of it in each cell. We use a separate panel for each of the ten cells. Each panel can be an object of the same class that we derive from `JPanel`. Each panel will use the same `paintComponent` method to draw the whole big picture, but only part of it will be visible on the panel. All we have to do is <u>shift the origin</u> of the coordinate system appropriately for each panel, depending on what puzzle piece it currently holds. For example (Figure 16-9), in order to draw piece number 6 correctly, no matter where it is currently located, the origin of its coordinate system should be shifted up by one cell height and to the left by two cell widths (with minor adjustments for the gaps between cells).

`JFrame`'s default `paint` method will automatically paint all the panels for us.
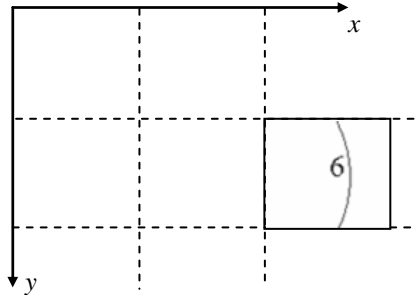
**Figure 16-9. The offset of relative coordinates
for a piece of the puzzle**

Now our plan of action is clear. Our program will use two classes. The first one, `Puzzle`, is the main class derived from `JFrame`. The second class, let's call it `PuzzleCell`, will be derived from `JPanel`. The program declares and initializes an array of ten `PuzzleCell` objects, one for each cell on the 3 by 3 grid and an extra empty cell. The cells are indexed 0, 1, 2, ..., 9. We first place cells 1 through 9 on the puzzle grid starting from the upper-left corner, then we place cell 0 in the middle of the fourth row (the program's layout actually uses a 4 by 3 grid).

Each `PuzzleCell` object holds a number representing the piece of the picture it currently displays. When a cell is created, the piece number is passed as a parameter to its constructor. At the beginning, before the picture is scrambled, the piece number is set to that cell's index. The code in `Puzzle`'s constructor creates the cells and adds them to the program's content pane:

```
public Puzzle()
{
  ...

  Container c = getContentPane();
  c.setLayout(new GridLayout(4, 3, 1, 1));
                    // 4 by 3; horz gap 1, vert gap 1
  cells = new PuzzleCell[10];

  for (int i = 1; i <= 9; i++)
  {
    cells[i] = new PuzzleCell(i);
    c.add(cells[i]);
  }

  ...
}
```

After showing the original picture we need to scramble it. To do that, we initialize a temporary array with values 1 through 9 and shuffle it, then assign the shuffled numbers to cells 1 through 9:

```
// Scramble the puzzle by setting shuffled numbers 1 through 9
//    to the puzzle cells:
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};

shuffle(numbers);
for (int i = 1; i <= 9; i++)
{
  int k = numbers[i-1];
  cells[i].setPieceNumber(k);
}
```

Set up a project with the three Java files from J_M\Ch16\Puzzle folder: Puzzle.java, PuzzleCell.java, and EasySound.java. Copy the three audio clip .wav files from J_M\Ch16\Puzzle to the folder that will hold your compiled class files.

As a warm-up exercise, write the code for the Puzzle class's shuffle method (see Question 12 in the exercises for Chapter 14). This method should rearrange the elements of a given array in random order. The algorithm is very similar to Selection Sort, only instead of choosing the largest among the first *n* elements, you choose an element randomly among the first *n* elements and swap it with the *n*-th element. This algorithm produces all possible arrangements with equal probabilities.

As another little exercise, write the Puzzle class's allSet method, which returns true if all pieces have been returned to their proper places and false otherwise. Call the getPieceNumber method for each of the cells and compare its returned value with the cell number.

And now to the serious business. Fill in the blanks in the PuzzleCell class's paintComponent method. First set the background color — white for a non-picked piece and yellow for a picked piece — and call super.paintComponent. Then shift the origin appropriately, based on the value of PuzzleCell's pieceNumber field, which represents the number of the piece this panel (cell) is supposed to show. Recall that the panel's getWidth and getHeight methods return the dimensions of the panel. You need to adjust them slightly to compensate for the gaps between panels. Finally, call a method that paints your picture.

Test your code first with the simple picture (a circle and cell numbers) provided. It will also help you test your `shuffle` method and your coordinate offsets. The purpose of the circle is to test how the pieces fit together — make sure your circle looks smooth. After you get it working, create a different picture of your choice for the puzzle. For instance, you can draw circles, polygons, or letters of different sizes and colors that intersect the grid. Make sure your picture is <u>fully scalable</u>, so that if the program's window shrinks or stretches, the picture shrinks or stretches with it. Find several five- to seven-year-olds and test your working puzzle on them.

16.8   Summary

Java provides a straightforward but limited class `Graphics` for drawing simple shapes and graphics text. The `Graphics2D` package is much more powerful but harder to use.

Since Java applets and applications are event-driven, all drawing must originate either in the `paint` method of the applet or application window or in the `paintComponent` method of one of the *Swing* components (usually an object of a class derived from `JComponent` or `JPanel`). `paint` and `paintComponent` take one parameter, `Graphics g`, which defines the graphics context for this component. If the application needs to repaint its component, it calls `repaint`, which takes no parameters. The repaint request is added to the events queue and eventually `paintComponent` is called.

The Java coordinate system has the origin in the upper-left corner of the window or panel, with the *y*-axis pointing down. The coordinates are integers, and their units are pixels. The `drawLine` method draws a line segment described by the coordinates of its beginning and end. A rectangle is described by the *x*, *y* coordinates of its upper-left corner, width, and height. An oval, a rounded rectangle, and even an arc is defined by the position of the bounding rectangle into which it is inscribed. Besides filled or hollow rectangles, rounded rectangles, ovals, and arcs, `Graphics` can draw polygons and polylines.

You can set the current drawing color by calling `g.setColor(…)`, which takes a `Color` object as its parameter. To set the background color, call the component's `setBackground` method.

You can display graphics text by calling the `drawString` method; to choose a desired font, call `setFont`.

📖 **Exercises** 📖

The exercises for this chapter are in the book (*Java Methods: Object-Oriented Programming and Data Structures, 2nd AP Edition*, ISBN 978-0-9824775-7-1, Skylight Publishing, 2011 [1]).