TUTORIAL 27

Drawing Shapes Application

Introduction to Polymorphism; an Expanded Discussion of Graphics

Objectives

In this tutorial, you will learn to:

- Use polymorphism to create an application that process related objects as though they are the same.
- Use additional `Graphics` methods such as `drawLine`.
- Create an application that allows users to draw shapes.

Outline

Polymorphism is an object-oriented programming concept that enables you to "program in the general" rather than having to "program in the specific." In particular, polymorphism makes it easy to write code to process a variety of related objects. The same method call is made on these objects and each of the objects will "do the right thing." If, for example, you ask an object to "talk" it will respond appropriately. If you tell a pig object to talk, it will respond with an "oink." If you tell a dog object to talk, it will respond with a "bark."

Polymorphic applications handle, in a simple and convenient manner, objects of many classes that belong to the same inheritance hierarchy. These applications focus on the similarities between these classes rather than the differences.

With polymorphism, it is also possible to design and implement systems that are easily extended with new capabilities. New classes can be added with little or no modification to the rest of the application, as long as those classes share the similarities of the classes that the application already processes. These new classes simply "plug right in."

In this tutorial, you will add polymorphic processing to the **Drawing Shapes** application. You will also learn additional methods of the `Graphics` class to outline and fill in different types of shapes.

27.1 Test-Driving the Drawing Shapes Application

In this tutorial, you will create a **Drawing Shapes** application that will allow students to draw lines, rectangles and ovals. The application must meet the following requirements:

Application Requirements

*The principal of the elementary school from Tutorial 21 has asked you to modify your **Painter** application. The user should now be able to choose a color from a `JColorChooser` dialog and a type of shape to be drawn from a `JComboBox`. The possible shapes include lines, rectangles and ovals. The user should be able to click a mouse button to create a shape and drag the mouse anywhere on the drawing area to resize that shape. Multiple shapes can be drawn on the drawing area, allowing the user to draw a picture by combining shapes.*

This application allows a user to draw three different kinds of shapes in a variety of colors. The user chooses the shape and color, then presses a mouse button and drags the mouse to create the shape. The user can draw as many shapes as desired. You begin by test-driving the completed application. Then, you will learn the additional Java technologies you will need to create your own version of this application.

Test-Driving the Drawing Shapes Application

1. ***Locating the completed application.*** Open the **Command Prompt** window by selecting **Start > Programs > Accessories > Command Prompt**. Change to your completed **Drawing Shapes** application directory by typing `cd C:\Examples\Tutorial27\CompletedApplication\DrawingShapes`.
2. ***Running the Drawing Shapes application.*** Type `java DrawingShapes` in the **Command Prompt** window to run the application (Fig. 27.1).

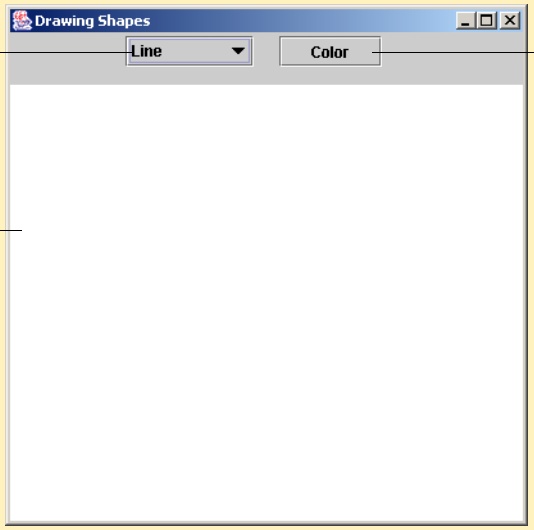

Figure 27.1 Running the completed **Drawing Shapes** application.

3. ***Changing the type of shape to draw.*** Click the `JComboBox` at the top of the application and select **Oval** (Fig. 27.2).

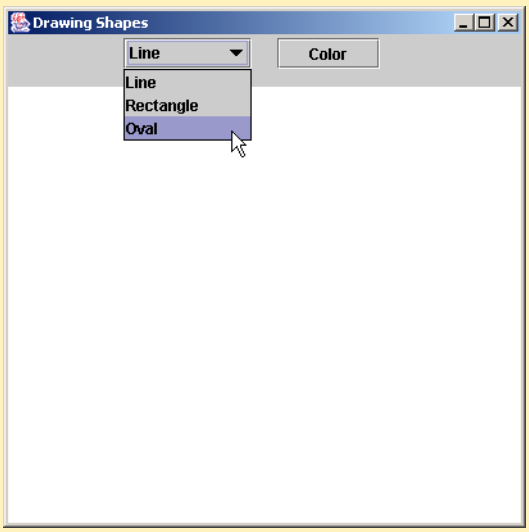

Figure 27.2 Running the completed **Drawing Shapes** application.

(cont.)

4. ***Changing the color of the shape to be drawn.*** Click the **Color** `JButton` at the top of the application. This will open the `JColorChooser` dialog which allows you to select a color for the shapes you will draw. The `JColorChooser` dialog will look identical to Fig. 22.3. Select a color and click the **OK** `JButton` in the `JColorChooser` dialog. Notice that when you select a new color, the color of the **Color** `JButton` changes to the newly selected color.
5. ***Drawing an oval.*** Once you have chosen a shape to draw and a color for your shape, move your mouse pointer to the drawing area (the white rectangle). Click and hold the left mouse button to create a new shape. One end of the shape will be positioned at the mouse cursor. Drag the mouse around to position the opposite end of the shape at the location you desire, then release the mouse.

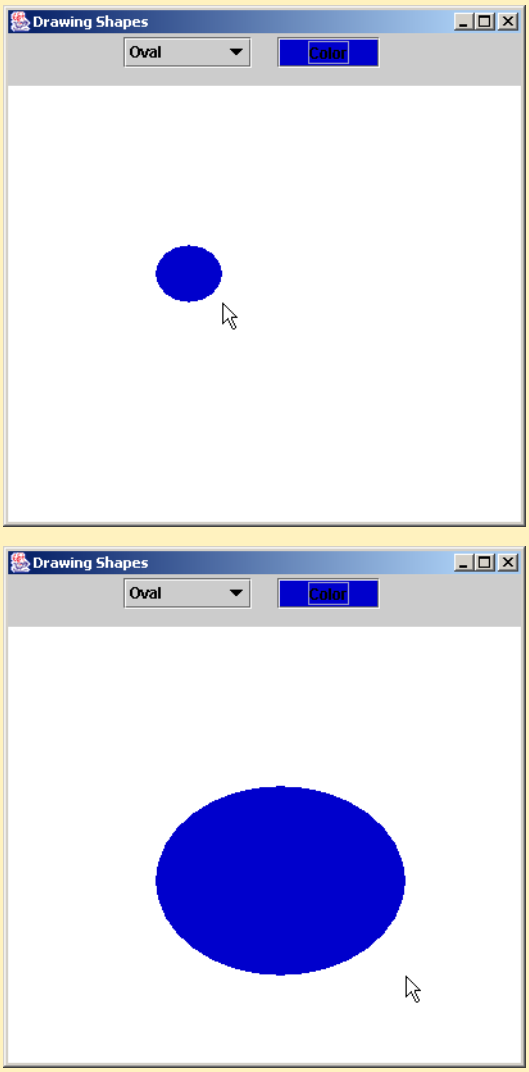

Figure 27.3 Drawing a shape on the application.

6. ***Closing the running application.*** Close your running application by clicking its close button.
7. ***Closing the Command Prompt window.*** Close the **Command Prompt** window by clicking its close button.

27.2 Polymorphism

You will now continue your study of object-oriented programming by learning about polymorphism with inheritance hierarchies. With polymorphism, the same method signature can be used to cause different actions to occur, depending on the type of the object on which the method is invoked.

As an example, suppose you design a video game that manipulates objects of many different types, including objects of classes `Bird`, `Fish` and `Snake`. Also, imagine that each of these classes inherits from a common superclass called `Animal`, which contains method `move`. Each subclass implements this method. Your video game application would maintain a collection (such as an `ArrayList`) of references to objects of the various classes. To move the animals, the application would periodically send each object the same message—namely `move`. Each object responds to this message in a unique way. For example, a `Bird` flies across the screen. A `Fish` swims through a lake. A `Snake` slithers through the grass. The same message (in this case, `move`) sent to a variety of objects would have "many forms" of results—hence the term polymorphism which means literally "many forms".

Consider another example—developing a simple payroll system for an `Employee` inheritance hierarchy. Every `Employee` has an `earnings` method that calculates the employee's weekly pay. These `earnings` methods vary by employee type—a `SalariedEmployee` is paid a fixed weekly salary regardless of the number of hours worked. An `HourlyEmployee` is paid by the hour and receives overtime pay. A `CommissionEmployee` receives a percentage of sales. The same message (in this case, `earnings`) sent to a variety of objects would have "many forms" of results—again, polymorphism.

For the **Drawing Shapes** application, you will develop a simple inheritance hierarchy. The `MyShape` class will declare the basic properties of a shape such as its color and location. Three other classes will extend `MyShape` and each of these classes will declare more specific shape information. These classes are `MyLine`, `MyRectangle` and `MyOval`. The UML class diagram of Fig. 27.4 demonstrates the inheritance hierarchy for your **Drawing Shapes** application.

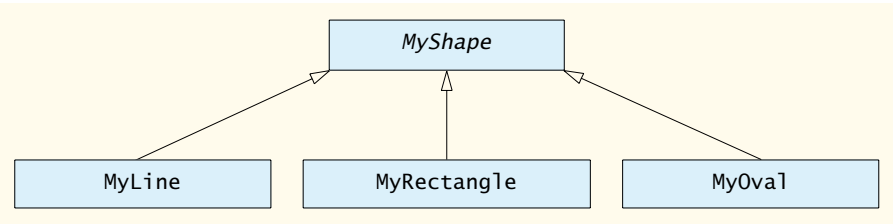

Figure 27.4 UML class diagram for the inheritance hierarchy in the **Drawing Shapes** application.

Calling the `draw` method on a `MyLine` object draws a line. Calling the `draw` method on a `MyRectangle` object draws a rectangle. [*Note:* The `MyOval` class is not included in the template application. You will declare it later in this tutorial.] The same message (in this case, `draw`) sent to a variety of objects would have "many forms" of results—again, polymorphism.

27.3 More Graphics Methods

Before you begin building your **Drawing Shapes** application, you should review its functionality. The following pseudocode describes the basic operation of the **Drawing Shapes** application:

```
When the user presses the mouse button:

    If Line is selected in the JComboBox
        Create a line

    If Rectangle is selected in the JComboBox
        Create a rectangle

    If Oval is selected in the JComboBox
        Create an oval

When the user clicks the Color JButton:
    Display a JColorChooser dialog
    Update the JButton's color with the selected color
    Set the current shape color to the selected color

When the user selects an item in the JComboBox:
    Get the shape type selected
    Set the current shape type to the selected item

When the user drags the mouse:
    Resize the shape
    Repaint the application
```

Now that you have test-driven the **Drawing Shapes** application and studied its pseudocode representation, you will use an ACE table to help you convert the pseudocode to Java. Figure 27.5 lists the actions, components and events that will help you complete your own version of the application.

Action/Component/ Event (ACE) Table for the Drawing Shapes Application

Action	Component/Object	Event
	`painterJPanel`	User presses a mouse button
If Line is selected in JComboBox	`shapeJComboBox`	
Create a line	`currentShape` (`MyShape`)	
If Rectangle is selected in JComboBox	`shapeJComboBox`	
Create a rectangle	`currentShape` (`MyShape`)	
If Oval is selected in JComboBox	`shapeJComboBox`	
Create an oval	`currentShape` (`MyShape`)	
	`colorJButton`	User clicks **Color** `JButton`
Display a JColorChooser dialog	`JColorChooser`	
Update the JButton's color	`colorJButton`	
Set the current shape color to the selected color	`paintJPanel`	
	`shapeJComboBox`	User selects an item in the `JComboBox`
Get the shape type selected	`shapeJComboBox`	
Set the current shape type to the selected item	`paintJPanel`	
	`painterJPanel`	User drags the mouse
Resize the shape	`currentShape` (`MyShape`)	
Repaint the application	`painterPaintJPanel`	

Figure 27.5 **Drawing Shapes** application ACE table.

When you think of a class type, you assume that applications will create objects of that type. However, there are cases in which it is useful to declare classes for which the programmer never intends to instantiate objects. Such classes are called **abstract classes**. Because abstract classes are used only as superclasses in inheritance hierarchies, those classes are often called **abstract superclasses**. These classes cannot be used to instantiate objects, because, as you will see, abstract classes are incomplete. Subclasses must declare the "missing pieces." Abstract superclasses are often used in polymorphic applications which is why polymorphism is sometimes called programming "in the abstract."

The purpose of an abstract class is to provide an appropriate superclass from which other classes can inherit. Classes that can be used to instantiate objects are called **concrete classes**. Abstract superclasses are too generic to create real objects—they specify only what is common among their subclasses. You need to be more specific before you can create objects. Concrete classes provide the specifics that make it possible to instantiate objects.

In the `MyShape` inheritance hierarchy described previously, `MyShape` is an abstract superclass. It declares a `draw` method, but does not provide an implementation of that method. If someone tells you to "draw the shape," your response would likely be "what shape should I draw?" This `draw` method is the missing piece that makes it impossible to instantiate a `MyShape` object. If instead you were told to "draw a line" or "draw a rectangle," you could do so. The `MyLine` class is a concrete subclass of `MyShape` because the `MyLine` class includes an implementation of the `draw` method which specifically draws a line. The `MyRectangle` class is a concrete subclass of `MyShape` because the `MyRectangle` class includes an implementation of the `draw` method which specifically draws a rectangle.

You will finish the `MyShape` inheritance hierarchy by declaring the `MyShape` class `abstract` and adding a `draw` method. You will then provide an implementation of the `draw` method in classes `MyLine` and `MyRectangle`.

Declaring an `abstract` Method

1. ***Copying the template to your working directory.*** Copy the `C:\Examples\Tutorial27\TemplateApplication\DrawingShapes` directory to your `C:\SimplyJava` directory.
2. ***Opening the `MyShape` template file.*** Open the template file `MyShape.java` in your text editor.
3. ***Declaring the `MyShape` class `abstract`.*** Modify line 5 as shown in Fig. 27.6. This line declares the `MyShape` class **`abstract`**. By declaring this class `abstract`, instances of this class cannot be created. In this application, you will create instances of `MyShape`'s subclasses—`MyLine`, `MyRectangle` and `MyOval`.

Declaring `MyShape` `abstract`

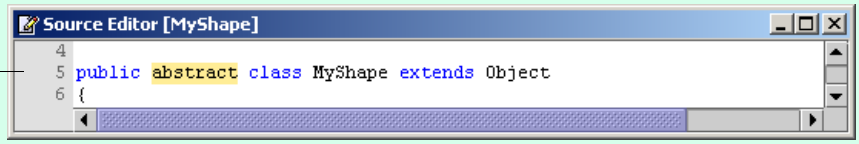

Figure 27.6 Declaring the `MyShape` class `abstract`.

4. ***Declaring an `abstract` method.*** Insert lines 95–96 of Fig. 27.7 after method `getColor`. These lines declare `abstract` method `draw`, but provide no implementation for it. Abstract methods are declared by writing a method header followed by a semicolon—no method body is provided. This `draw` method is the missing piece of the `MyShape` class that makes it impossible to instantiate. If any method in a class is declared `abstract`, then the whole class must be declared `abstract` (as you did in the previous step). The concrete subclasses of `MyShape` must provide an implementation of the `draw` method.

(cont.)

Declaring `abstract` method `draw`

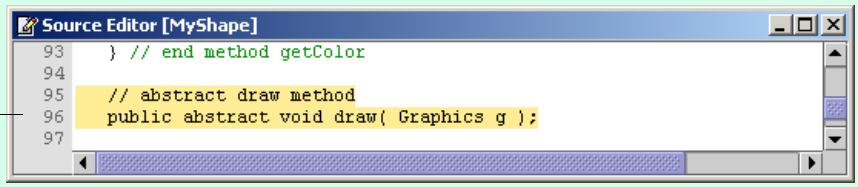

Figure 27.7 Declaring `abstract` method `draw`.

5. ***Saving the application.*** Save your modified source code file.

In Tutorial 20, you learned to set the color of drawn shapes using the `setColor` method and to draw a filled rectangle using the `fillRect` method. In Tutorial 21, you learned to draw a filled oval using the `fillOval` method. Each of these methods belongs to class `Graphics`. Now, you will learn about `Graphics` methods for drawing lines, rectangles and ovals. Figure 27.8 summarizes the `Graphics` methods you have learned and introduces several new ones.

Graphics Method	Description
`drawLine(x1, y1, x2, y2)`	Draws a line from the point `(x1, y1)` to the point `(x2, y2)`.
`drawRect(x, y, width, height)`	Draws a rectangle of the specified `width` and `height`. The top-left corner of the rectangle is at the point `(x, y)`.
`fillRect(x, y, width, height)`	Draws a solid rectangle of the specified `width` and `height`. The top-left corner of the rectangle is at the point `(x, y)`.
`drawOval(x, y, width, height)`	Draws an oval inside a rectangular area of the specified `width` and `height`. The top-left corner of the rectangular area is at the point `(x, y)`.
`fillOval(x, y, width, height)`	Draws a filled oval inside a rectangular area of the specified `width` and `height`. The top-left corner of the rectangular area is at the point `(x, y)`.
`setColor(color)`	Sets the drawing color to the specified `color`.

Figure 27.8 `Graphics` methods that draw lines, rectangles and ovals.

The `MyLine` class `extends` the `abstract` class `MyShape` which contains `abstract` method `draw`. To declare `MyLine` as a concrete subclass, you must provide an implementation for the `draw` method. If you extend an `abstract` superclass, you must provide an implementation for each of its `abstract` methods or else the subclass must be declared `abstract` as well.

You will now provide an implementation of the `draw` method in the `MyLine` class. This method should draw a line starting at one of the endpoints specified in the `MyLine` object and ending at the other one.

Implementing the draw Method in Class MyLine

1. ***Opening the `MyLine` template file.*** Open the template file `MyLine.java` in your text editor.
2. ***Implementing the `draw` method in the `MyLine` class.*** Insert lines 18–19 of Fig. 27.9 into the `draw` method. Line 18 calls method `getColor` to get the color of the `MyShape`. The return value is passed to method `setColor` to set the color of the `Graphics` object (`g`) for drawing. Line 19 calls the **`drawLine`** method on the `Graphics` object. This method takes four `int` values; the first two are the *x*- and *y*-coordinates of the first endpoint of the line and the second two are the *x*- and *y*-coordinates of the second endpoint of the line.

(cont.)

Implementing the `draw` method to draw a line

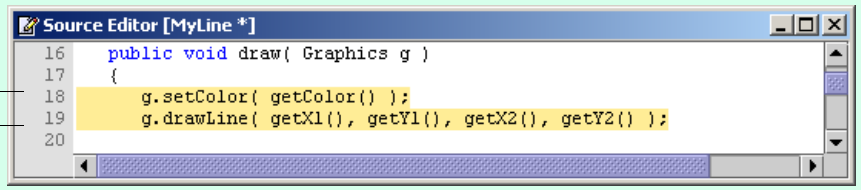

Figure 27.9 Implementing the `draw` method in class `MyLine`.

3. ***Saving the application.*** Save your modified source code file.

Your application receives input from the user in the form of two points on the screen—the location at which the user originally clicks the mouse button and the location to which the user drags the mouse cursor. Drawing a line between these two points is simple; the `drawLine` method of class `Graphics` takes the location of two points as arguments. Drawing a rectangle based on these two points is more complicated though. The drawn rectangle will have one corner located at one of the points and the diagonally opposite, corner located on the other point. In the `MyRectangle` class's `draw` method, you will need to use these two points to calculate the *x*- and *y*-coordinates of the upper-left corner of the rectangle along with the rectangle's width and height.

You will now implement the `draw` method in the `MyRectangle` class to make `MyRectangle` a concrete subclass of `MyShape`. This method should draw a rectangle on the screen with one corner at one point of the `MyRectangle` object and the opposite corner at the other point.

Implementing the draw Method in Class MyRectangle

1. ***Opening the `MyRectangle` template file.*** Open the template file `MyRectangle.java` in your text editor.
2. ***Calculating the coordinates of the upper-left corner.*** Insert lines 18–19 of Fig. 27.10 into the `draw` method. As you learned in Tutorial 20, the `fillRect` method takes as arguments the *x*- and *y*-coordinates of the upper-left point of the rectangle along with the width and the height. The `MyRectangle` class stores its data in instance variables `x1`, `x2`, `y1` and `y2`. Your `draw` method will need to convert the information stored in the `MyRectangle` class to the correct information to pass to the `fillRect` method.

   Line 18 uses the `Math` class's **`min`** method to determine the smaller of the two *x*-coordinates, which is the one farther left. This method call returns the left edge of the rectangle. Line 19 calls the `min` method to determine the smaller of the two *y*-coordinates, which is the one higher than the other. This method call returns the top of the rectangle.

Determining the *x*- and *y*-coordinates of the upper left corner

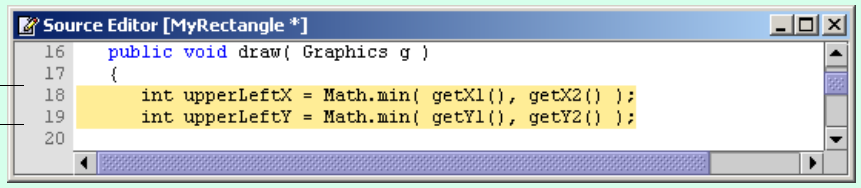

Figure 27.10 Calculating the coordinates of the upper-left corner.

(cont.)

3. ***Calculating the width and height***. Insert lines 20–24 of Fig. 27.11 into your code. The **abs** method of class `Math` returns the **absolute value** (the value of the number without the sign of the number) of the expression it receives. Line 20 uses the `abs` method to determine the difference between the two *x*-coordinates, which is the width of the rectangle. Line 21 uses the `Math` class's `abs` method to determine the difference between the two *y*-coordinates, which is the height of the rectangle. Line 23 sets the color of the rectangle. Line 24 calls method `fillRect` using the *x*- and *y*-coordinates that you calculated in the previous step, along with the width and height that you calculated in this step.

Calculating the width and height of the rectangle

```
Source Editor [MyRectangle *]
19      int upperLeftY = Math.min( getY1(), getY2() );
20      int width = Math.abs( getX1() - getX2() );
21      int height = Math.abs( getY1() - getY2() );
22
23      g.setColor( getColor() );
24      g.fillRect( upperLeftX, upperLeftY, width, height );
25
```

Figure 27.11 Calculating the width and height and drawing the rectangle.

4. ***Saving the application.*** Save your modified source code file.

You will now finish the `PaintJPanel` class to allow the user to create and resize shapes.

Finishing the `PaintJPanel` Class

1. ***Opening the template file***. Open the template file `PaintJPanel.java` in your text editor.
2. ***Declaring a `MyShape` instance variable.*** Add lines 13–14 of Fig. 27.12 into your code. These lines declare a `MyShape` instance variable to hold the current shape. The `MyShape` class is an `abstract` class and cannot be instantiated, but references of the `MyShape` class can be created. This is one of the keys to polymorphism. This reference is used to resize a shape after it has been created. With polymorphism, you do not need to know what type of shape is stored in the `MyShape` reference.

Declaring a `myShape` instance variable

```
Source Editor [PaintJPanel *]
11    private ArrayList shapesArrayList = new ArrayList();
12
13    // current shape that is being drawn
14    private MyShape currentShape;
15
```

Figure 27.12 Declaring a new `MyShape` object.

3. ***Creating a new `MyLine` object.*** Insert lines 72–77 of Fig. 27.13 into method `paintJPanelMousePressed`. Line 73 tests whether the user selected **Line** in the `JComboBox`. If this is the case, lines 75–76 create a new `MyLine` object. These lines use methods `getX` and `getY` of `MouseEvent` to determine where the mouse is positioned. This `MyLine` object is created with the first endpoint the same as the second endpoint. This makes the length of the line `0` and it appears as a single colored pixel. When the user drags the mouse, the second endpoint will be repositioned, changing the size of the line.

(cont.)

Creating a `MyLine` object

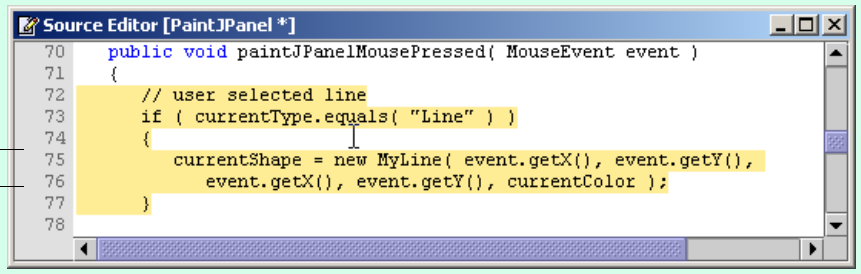

Figure 27.13 Creating new `MyLine` object.

4. ***Creating a new `MyRectangle` object.*** Insert lines 78–83 of Fig. 27.14 into method `paintJPanelMousePressed`. Line 79 tests whether the user selected **Rectangle** in the `JComboBox`. If this is the case, lines 81–82 create a new `MyRectangle` object. These lines use methods `getX` and `getY` of `MouseEvent` to determine where the mouse is positioned. This `MyRectangle` object is created with the first endpoint the same as the second endpoint. This makes the rectangle appear as a single colored pixel. When the user drags the mouse, the second endpoint will be repositioned, changing the size and shape of the rectangle.

Creating a `MyRectangle` object

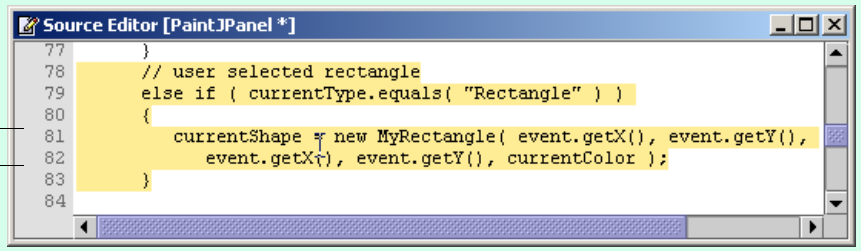

Figure 27.14 Creating new `MyRectangle` object.

Line 75 of Fig. 27.13 assigns a `MyLine` object to `MyShape` variable `currentShape` and line 81 of Fig. 27.14 assigns a `MyRectangle` object to `currentShape`. Java allows both of these assignments because the `MyLine` and `MyRectangle` classes have an "is-a" relationship with the `MyShape` class.

5. ***Adding the `MyShape` reference to the `ArrayList`.*** Add line 85 of Fig. 27.15 to method `paintJPanelMousePressed`. This line adds the new `MyShape` object to `shapesArrayList`.

Adding `currentShape` to the `shapes ArrayList`

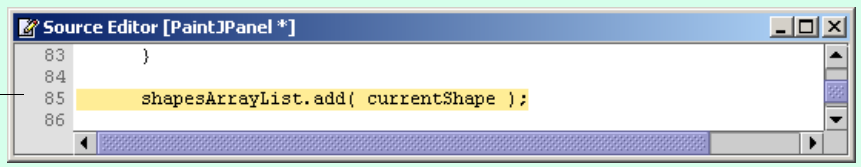

Figure 27.15 Adding the new `MyShape` to `shapesArrayList`.

6. ***Resizing the shape.*** Add lines 92–94 of Fig. 27.16 to method `paintJPanelMouseDragged`. When the user drags the mouse, `currentShape` must be resized. Lines 92–93 resize the shape by changing the x- and y-coordinates of the shape's second point. Recall that when the shape is constructed, the first and second points are at the same location. Changing the location of the second point resizes the shape, while keeping the first point in place. These lines use `MouseEvent` methods `getX` and `getY` to get the location of the mouse cursor.

(cont.)

Lines 92–93 use the `MyShape` variable `currentShape` without knowing exactly what type of shape is being affected. This is an example of polymorphic processing. The calls to methods `setX2` and `setY2` are allowed because these methods are declared in the `MyShape` class. All classes that extend `MyShape` contain these methods. Line 94 calls the `repaint` method, which will call the `paintComponent` method which you will declare next.

Setting the `currentShape`'s *x*- and *y*-coordinates

```
Source Editor [PaintJPanel *]
90   public void paintJPanelMouseDragged( MouseEvent event )
91   {
92      currentShape.setX2( event.getX() );
93      currentShape.setY2( event.getY() );
94      repaint();
95
```

Figure 27.16 Resizing the `MyShape` object.

7. ***Paint all the shapes.*** Add lines 103–112 of Fig. 27.17 to method `paintComponent`. Line 104 creates an `Iterator` to traverse through each element of `shapesArrayList`. Lines 107–112 iterate through the items in `shapesArrayList`. Line 110 calls method `next` to get a reference to the next object in `shapesArrayList`. This method returns an instance of type `Object` which is then cast to a `MyShape` reference and assigned to `nextShape`. Line 111 calls method `draw` on `nextShape`.

   At this point, you do not know which `draw` method will be called—the one in `MyLine` or the one in `MyRectangle`. The method call will be resolved only when the application is executed. Each shape in `shapesArrayList` knows how to draw itself. If `nextShape` is a `MyLine` object, the `draw` method from the `MyLine` class will be called. If `nextShape` is instead a `MyRectangle` object, the `draw` method from the `MyRectangle` class will be called.

Using a `while` statement to draw each shape

```
Source Editor [PaintJPanel *]
101      super.paintComponent( g );
102
103      MyShape nextShape;
104      Iterator shapesIterator = shapesArrayList.iterator();
105
106      // iterate through all the shapes
107      while ( shapesIterator.hasNext() )
108      {
109         // draw each shape
110         nextShape = ( MyShape ) shapesIterator.next();
111         nextShape.draw( g );
112      }
113
```

Figure 27.17 Drawing the shapes in `shapesArrayList` polymorphically.

8. ***Saving the application.*** Save your modified source code file.

You have now finished coding the `PaintJPanel` class. Next, you will instantiate an object of `PaintJPanel` and use it in your **Drawing Shapes** application to allow the user to draw shapes.

Adding a `PaintJPanel` to Your Application

1. ***Opening the template file.*** Open the template file `DrawingShapes.java` in your text editor.
2. ***Declaring a `PaintJPanel` instance variable.*** Add lines 19–20 of Fig. 27.18 to your code to declare a `PaintJPanel` instance variable. This `PaintJPanel` component listens for mouse events and uses them to draw shapes.

(cont.)

Declaring a `PaintJPanel` instance variable

```
Source Editor [DrawingShapes *]
17    private JButton colorJButton;
18
19    // PaintJPanel for drawing shapes
20    private PaintJPanel painterPaintJPanel;
21
```

Figure 27.18 Declaring a `PaintJPanel` instance variable.

3. ***Creating and customizing the `PaintJPanel`.*** Add lines 46–50 of Fig. 27.19 to your application. Line 47 instantiates a `PaintJPanel` object named `painterPaintJPanel`. Lines 48–49 set the *bounds* and *background* properties for the `painterPaintJPanel`, respectively. Line 50 adds `painterPaintJPanel` to the content pane to display the component and allow the user to interact with it.

```
Source Editor [DrawingShapes *]
44       contentPane.add( controlsJPanel );
45
46       // set up painterPaintJPanel
47       painterPaintJPanel = new PaintJPanel();
48       painterPaintJPanel.setBounds( 0, 40, 400, 340 );
49       painterPaintJPanel.setBackground( Color.WHITE );
50       contentPane.add( painterPaintJPanel );
51
```

Figure 27.19 Creating a new `PaintJPanel` object.

4. ***Setting the color for the next drawn `MyShape`.*** Add line 105 of Fig. 27.20 to method `colorJButtonActionPerformed`. This line sets the color of the shape to be drawn to the color the user selected in the `JColorChooser` dialog. Now, when the user selects a color, that color will be set as the current color of `painterPaintJPanel`.

Setting the `PaintJPanel`'s color

```
Source Editor [DrawingShapes *]
104         colorJButton.setBackground( selection );
105         painterPaintJPanel.setCurrentColor( selection );
106      }
```

Figure 27.20 Setting the color for the next `MyShape`.

5. ***Setting the type of the drawn `MyShape`.*** Add lines 113–114 of Fig. 27.21 to method `shapeJComboBoxActionPerformed`. These lines take the name of the shape that the user selected from `shapeJComboBox` and pass it to `painterPaintJPanel`. The `getSelectedItem` method returns the `Object` that is currently selected in `shapeJComboBox`, which is then cast to `String` and passed to method `setCurrentShapeType` of `PaintJPanel`. When the user drags the mouse on `painterPaintJPanel`, a shape of the user's selected type and color will appear.

Determining which shape to draw

```
Source Editor [DrawingShapes *]
111   private void shapeJComboBoxActionPerformed( ActionEvent event )
112   {
113      painterPaintJPanel.setCurrentShapeType(
114         ( String )shapeJComboBox.getSelectedItem() );
115
```

Figure 27.21 Changing the type of shape drawn.

6. ***Saving the application.*** Save your modified source code file.

(cont.)

7. ***Opening the Command Prompt window and changing directories.*** Open the **Command Prompt** window by selecting **Start > Programs > Accessories > Command Prompt**. Change to your working directory by typing `cd C:\SimplyJava\DrawingShapes`.
8. ***Compiling the application.*** Compile your application by typing `javac DrawingShapes.java PaintJPanel.java MyShape.java MyLine.java MyRectangle.java`.
9. ***Running the application.*** When your application compiles correctly, run it by typing `java DrawingShapes`. Figure 27.28 shows the completed application running. Users can now select and draw a line or a rectangle, but cannot select or draw an oval.

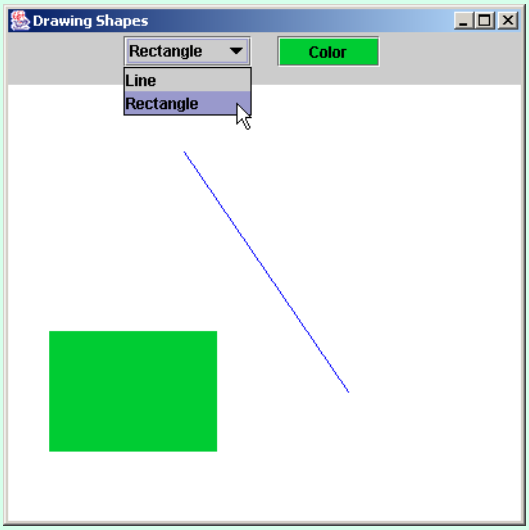

Figure 27.22 Completed **Drawing Shapes** application.

10. ***Closing the application.*** Close your running application by clicking its close button.
11. ***Closing the Command Prompt window.*** Close the **Command Prompt** window by clicking its close button.

SELF-REVIEW

1. The statement, _____, will draw a horizontal line.
   a) `drawLine( 0, 5, 5, 0 )` b) `drawLine( 0, 5, 5, 5 )`
   c) `drawLine( 5, 5, 5, 0 )` d) `drawLine( 5, 5, 5, 5 )`
2. The _____ method of class `Graphics` can draw the outline of a circle.
   a) `fillOval` b) `fillCircle`
   c) `drawOval` d) `drawCircle`

Answers: 1) b. 2) c.

27.4 Adding to the MyShape Inheritance Hierarchy

One of the benefits of polymorphism is that it makes it easy to add new types of objects to an existing application. In your **Drawing Shapes** application, the user can draw a line or a rectangle. Both the `MyLine` and the `MyRectangle` class extend the `MyShape` class and implement the `draw` method. You will now add to your application by declaring a `MyOval` class and adding it to the inheritance hierarchy. The `MyOval` class will also extend the `MyShape` class and declare a `draw` method. The application code will require only a few changes.

Adding Class MyOval to the Inheritance Hierarchy

1. ***Create the MyOval file.*** Create a new source code file. Name this new file `MyOval.java`. After you have created the file, open it in your text editor.
2. ***Declare the MyOval class.*** Add lines 1–8 of Fig. 27.23 to `MyOval.java`. Line 5 declares that class `MyOval` `extends` class `MyShape`. The class declaration ends with the right brace on line 8.

Class `MyOval` extends class `MyShape`

```
Source Editor [MyOval *]
 1 // Tutorial 27: MyOval.java
 2 // Class that declares an oval object.
 3 import java.awt.*;
 4
 5 public class MyOval extends MyShape
 6 {
 7
 8 } // end class MyOval
```

Figure 27.23 Declaring class `MyOval` to extend `MyShape`.

3. ***Adding a constructor.*** Add lines 7–13 of Fig. 27.24 to the class declaration. These lines declare a constructor for `MyOval` which takes four integer arguments and a `Color` argument. This constructor calls the superclass's constructor which also takes four `int` arguments and a `Color` argument.

`MyOval`'s constructor, which takes five arguments

```
Source Editor [MyOval *]
 6 {
 7    // constructor
 8    public MyOval( int firstX, int firstY, int secondX, int secondY,
 9       Color shapeColor )
10    {
11       super( firstX, firstY, secondX, secondY, shapeColor );
12
13    } // end constructor
14
```

Figure 27.24 Declaring a constructor in class `MyOval`.

4. ***Implementing the draw method.*** Add lines 15–26 of Fig. 27.25 after the constructor. These lines implement the `draw` method declared in class `MyShape` to draw an oval. Lines 18–21 calculate the dimensions of the oval to be drawn. These calculations are the same as those that were required for the `MyRectangle` class. Recall that the `min` method returns the smallest of the two values it receives and the `abs` method returns the absolute value of the expression it receives. Line 24 calls `Graphics` method `fillOval` to draw an oval in the application.

Implementing the `draw` method

```
Source Editor [MyOval *]
13    } // end constructor
14
15    // draw an oval
16    public void draw( Graphics g )
17    {
18       int upperLeftX = Math.min( getX1(), getX2() );
19       int upperLeftY = Math.min( getY1(), getY2() );
20       int width = Math.abs( getX1() - getX2() );
21       int height = Math.abs( getY1() - getY2() );
22
23       g.setColor( getColor() );
24       g.fillOval( upperLeftX, upperLeftY, width, height );
25
26    } // end method draw
27
```

Figure 27.25 Implementing method `draw` to draw a `MyOval` object.

5. ***Saving the application.*** Save your modified source code file.

Now that you have created class `MyOval`, you must modify some of the code in the application. First, you must add an option to the `JComboBox` allowing the user to select an oval to draw.

Allowing the User to Draw an Oval

1. ***Opening the template file.*** Open the template file `DrawingShapes.java` in your text editor.
2. ***Adding an oval option to the JComboBox.*** Modify line 23 of your source code file so it looks like line 23 of Fig. 27.26. This adds an "Oval" option to the `JComboBox` which allows the user to select an oval as the shape to draw.

Allow users to select "Oval" from `shapeTypes`

```
Source Editor [DrawingShapes *]
22    // array of shape types
23    private String[] shapeTypes = { "Line", "Rectangle", "Oval" };
24
```

Figure 27.26 Adding the oval option to the `String` array `shapeTypes`.

3. ***Saving the application.*** Save the modified source code file.

The user can now select an oval, but the application must also create a `MyOval` object.

Creating a MyOval Object

1. ***Opening the template file.*** Open the template file `PaintJPanel.java` in your text editor.
2. ***Creating a MyOval object.*** Add lines 84–89 of Fig. 27.27 to method `paintJPanelMousePressed`. Line 85 tests whether the current shape type is equal to `"Oval"`. If it is, lines 87–88 create a new `MyOval` object.

Drawing an oval if the user has selected this option

```
Source Editor [PaintJPanel *]
83        }
84        // user selected oval
85        else if ( currentType.equals( "Oval" ) )
86        {
87           currentShape = new MyOval( event.getX(), event.getY(),
88              event.getX(), event.getY(), currentColor );
89        }
90
```

Figure 27.27 Creating a `MyOval` object.

Notice that you do not need to make any changes to the method that resizes the shape (`paintJPanelMouseDragged`) or the method that draws the shape (`paintComponent`) because they handle the shapes polymorphically. Line 111 of Fig. 27.17 calls the `draw` method on `MyShape` reference `currentShape`. If `currentShape` actually refers to a `MyOval` object, the `draw` method declared in the `MyOval` class is called. The `MyOval` object knows how to draw itself.

3. ***Saving the application.*** Save your modified source code file.
4. ***Opening the Command Prompt window and changing directories.*** Open the **Command Prompt** window by selecting **Start > Programs > Accessories > Command Prompt**. Change to your working directory by typing `cd C:\SimplyJava\DrawingShapes`.
5. ***Compiling the application.*** Compile your application by typing `javac DrawingShapes.java PaintJPanel.java MyOval.java`.

(cont.)

6. ***Running the application.*** When your application compiles correctly, run it by typing `java DrawingShapes`. Figure 27.28 shows the completed application running. Users can now select and draw an oval.

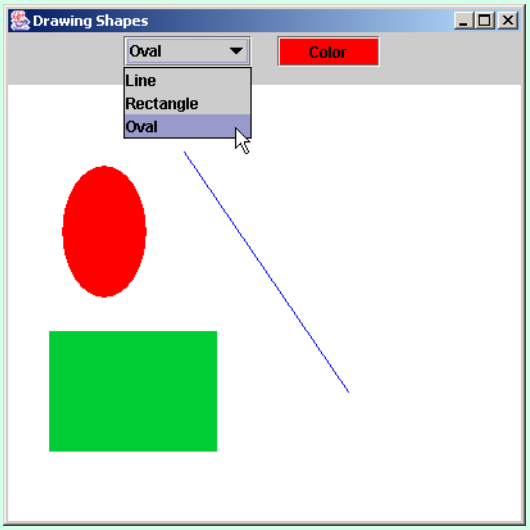

Figure 27.28 Completed **Drawing Shapes** application.

7. ***Closing the application.*** Close your running application by clicking its close button.
8. ***Closing the Command Prompt window.*** Close the **Command Prompt** window by clicking its close button.

Figure 27.29–Fig. 27.30 present the source code for the **Drawing Shapes** application. The lines of code that you added, viewed or modified in this tutorial are highlighted.

```
1  // Tutorial 27: DrawingShapes.java
2  // Application allows user to draw lines, rectangles and ovals and
3  // choose the color of the drawn shape.
4  import java.awt.*;
5  import java.awt.event.*;
6  import javax.swing.*;
7
8  public class DrawingShapes extends JFrame
9  {
10    // JPanel for the shape and color controls
11    private JPanel controlsJPanel;
12
13    // JComboBox to allow selection of a shape
14    private JComboBox shapeJComboBox;
15
16    // JButton to select the color
17    private JButton colorJButton;
18
19    // PaintJPanel for drawing shapes
20    private PaintJPanel painterPaintJPanel;
21
22    // array of shape types
23    private String[] shapeTypes = { "Line", "Rectangle", "Oval" };
24
```

`PaintJPanel` instance variable

Array of shape names

Figure 27.29 **Drawing Shapes** code. (Part 1 of 3.)

```
25    // no-argument constructor
26    public DrawingShapes()
27    {
28       createUserInterface();
29    }
30
31    // create and position GUI components; register event handlers
32    private void createUserInterface()
33    {
34       // get content pane for attaching GUI components
35       Container contentPane = getContentPane();
36
37       // enable explicit positioning of GUI components
38       contentPane.setLayout( null );
39
40       // set up controlsJPanel
41       controlsJPanel = new JPanel();
42       controlsJPanel.setBounds( 0, 0, 400, 40 );
43       controlsJPanel.setLayout( null );
44       contentPane.add( controlsJPanel );
45
46       // set up painterPaintJPanel
47       painterPaintJPanel = new PaintJPanel();
48       painterPaintJPanel.setBounds( 0, 40, 400, 340 );
49       painterPaintJPanel.setBackground( Color.WHITE );
50       contentPane.add( painterPaintJPanel );
51
52       // set up shapeJComboBox
53       shapeJComboBox = new JComboBox( shapeTypes );
54       shapeJComboBox.setBounds( 90, 2, 100, 24 );
55       controlsJPanel.add( shapeJComboBox );
56       shapeJComboBox.addActionListener(
57
58          new ActionListener() // anonymous inner class
59          {
60             // event method called when shapeJComboBox is selected
61             public void actionPerformed( ActionEvent event )
62             {
63                shapeJComboBoxActionPerformed( event );
64             }
65
66          } // end anonymous inner class
67
68       ); // end call to addActionListener
69
70       // set up colorJButton
71       colorJButton = new JButton();
72       colorJButton.setBounds( 210, 2, 80, 24 );
73       colorJButton.setText( "Color" );
74       controlsJPanel.add( colorJButton );
75       colorJButton.addActionListener(
76
77          new ActionListener() // anonymous inner class
78          {
79             // event handler called when colorJButton is pressed
80             public void actionPerformed( ActionEvent event )
81             {
82                colorJButtonActionPerformed( event );
```

Figure 27.29 **Drawing Shapes** code. (Part 2 of 3.)

```
83             }
84
85          } // end anonymous inner class
86
87       ); // end call to addActionListener
88
89       // set properties of application's window
90       setTitle( "Drawing Shapes" ); // set title bar string
91       setSize( 408, 407 );          // set window size
92       setVisible( true );           // display window
93
94    } // end method createUserInterface
95
96    // select a new color for the shape
97    private void colorJButtonActionPerformed( ActionEvent event )
98    {
99       Color selection = JColorChooser.showDialog( null,
100         "Select a Color", Color.BLACK );
101
102      if ( selection != null )
103      {
104         colorJButton.setBackground( selection );
105         painterPaintJPanel.setCurrentColor( selection );
106      }
107
108   } // end method colorJButtonActionPerformed
109
110   // set the selected shape in the painting panel
111   private void shapeJComboBoxActionPerformed( ActionEvent event )
112   {
113      painterPaintJPanel.setCurrentShapeType(
114         ( String )shapeJComboBox.getSelectedItem() );
115
116   } // end method shapeJComboBoxActionPerformed
117
118   // main method
119   public static void main( String args[] )
120   {
121      DrawingShapes application = new DrawingShapes();
122      application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
123
124   } // end method main
125
126 } // end class DrawingShapes
```

Setting the color of the `PaintJPanel` (line 105)

Setting the shape to draw (lines 113–114)

Figure 27.29 **Drawing Shapes** code. (Part 3 of 3.)

```
1  // Tutorial 27: PaintJPanel.java
2  // Panel allows user to create a shape.
3  import java.awt.*;
4  import java.awt.event.*;
5  import java.util.*;
6  import javax.swing.*;
7
8  public class PaintJPanel extends JPanel {
9
10    // ArrayList to hold the shapes
11    private ArrayList shapesArrayList = new ArrayList();
```

Figure 27.30 **PaintJPanel** code. (Part 1 of 3.)

```
12
13    // current shape that is being drawn
14    private MyShape currentShape;
15
16    // currently selected shape type
17    private String currentType = "Line";
18
19    // currently selected color
20    private Color currentColor = new Color( 204, 204, 204 );
21
22    // no-argument constructor
23    public PaintJPanel()
24    {
25       addMouseListener(
26
27          new MouseAdapter() // anonymous inner class
28          {
29             // event handler called when mouse button is pressed
30             public void mousePressed( MouseEvent event )
31             {
32                paintJPanelMousePressed( event );
33             }
34
35          } // end anonymous inner class
36
37       ); // end call to addMouseListener
38
39       addMouseMotionListener(
40
41          new MouseMotionAdapter() // anonymous inner class
42          {
43             // event handler called when the mouse is dragged
44             public void mouseDragged( MouseEvent event )
45             {
46                paintJPanelMouseDragged( event );
47             }
48
49          } // end anonymous inner class
50
51       ); // end call to addMouseMotionListener
52
53    } // end constructor
54
55    // change the current shape type
56    public void setCurrentShapeType( String shape )
57    {
58       currentType = shape;
59
60    } // end method setCurrentShapeType
61
62    // change the current color
63    public void setCurrentColor( Color shapeColor )
64    {
65       currentColor = shapeColor;
66
67    } // end method setCurrentColor
68
```

Figure 27.30 **PaintJPanel** code. (Part 2 of 3.)

```
69       // create a new shape
70       public void paintJPanelMousePressed( MouseEvent event )
71       {
72          // user selected line
73          if ( currentType.equals( "Line" ) )
74          {
75             currentShape = new MyLine( event.getX(), event.getY(),
76                event.getX(), event.getY(), currentColor );
77          }
78          // user selected rectangle
79          else if ( currentType.equals( "Rectangle" ) )
80          {
81             currentShape = new MyRectangle( event.getX(), event.getY(),
82                event.getX(), event.getY(), currentColor );
83          }
84          // user selected oval
85          else if ( currentType.equals( "Oval" ) )
86          {
87             currentShape = new MyOval( event.getX(), event.getY(),
88                event.getX(), event.getY(), currentColor );
89          }
90
91          shapesArrayList.add( currentShape );
92
93       } // end method paintJPanelMousePressed
94
95       // reset the second point for the shape
96       public void paintJPanelMouseDragged( MouseEvent event )
97       {
98          currentShape.setX2( event.getX() );
99          currentShape.setY2( event.getY() );
100         repaint();
101
102      } // end method paintJPanelMouseDragged
103
104      // paint all the shapes
105      public void paintComponent( Graphics g )
106      {
107         super.paintComponent( g );
108
109         MyShape nextShape;
110         Iterator shapesIterator = shapesArrayList.iterator();
111
112         // iterate through all the shapes
113         while ( shapesIterator.hasNext() )
114         {
115            // draw each shape
116            nextShape = ( MyShape ) shapesIterator.next();
117            nextShape.draw( g );
118         }
119
120      } // end method paintComponent
121
122   } // end class PaintJPanel
```

Creating a `MyLine` object and assigning it to a `MyShape` variable

Creating a `MyRectangle` object and assigning it to a `MyShape` variable

Creating a `MyOval` object and assigning it to a `MyShape` variable

Adding `currentShape` to `shapesArrayList`

Setting the `currentShape`'s x- and y- coordinates

Using a `while` statement to draw each shape

Figure 27.30 **PaintJPanel** code. (Part 3 of 3.)

Declaring MyShape abstract

```
1  // Tutorial 27: MyShape.java
2  // Superclass for all shape objects.
3  import java.awt.*;
4
5  public abstract class MyShape extends Object
6  {
7     private int x1;
8     private int y1;
9     private int x2;
10    private int y2;
11    private Color color;
12
13    // constructor
14    public MyShape( int firstX, int firstY, int secondX, int secondY,
15       Color shapeColor )
16    {
17       setX1( firstX );
18       setY1( firstY );
19       setX2( secondX );
20       setY2( secondY );
21       setColor( shapeColor );
22
23    } // end constructor
24
25    // set x1 value
26    public void setX1( int x )
27    {
28       x1 = x;
29
30    } // end method setX1
31
32    // get x1 value
33    public int getX1()
34    {
35       return x1;
36
37    } // end method getX1
38
39    // set Y1 value
40    public void setY1( int y )
41    {
42       y1 = y;
43
44    } // end method setY1
45
46    // get Y1 value
47    public int getY1()
48    {
49       return y1;
50
51    } // end method getY1
52
53    // set x2 value
54    public void setX2( int x )
55    {
56       x2 = x;
57
58    } // end method setX2
```

Figure 27.31 **My Shape** code. (Part 1 of 2.)

```
59
60       // get x2 value
61       public int getX2()
62       {
63          return x2;
64
65       } // end method getX2
66
67       // set y2 value
68       public void setY2( int y )
69       {
70          y2 = y;
71
72       } // end method setY2
73
74       // get y2 value
75       public int getY2()
76       {
77          return y2;
78
79       } // end method getY2
80
81       // set color value
82       public void setColor( Color c )
83       {
84          color = c;
85
86       } // end method setColor
87
88       // get color value
89       public Color getColor()
90       {
91          return color;
92
93       } // end method getColor
94
95       // abstract draw method
96       public abstract void draw( Graphics g );
97
98    } // end class MyShape
```

Declaring `abstract` method `draw`

Figure 27.31 **My Shape** code. (Part 2 of 2.)

```
1  // Tutorial 27: MyLine.java
2  // Class that declares a line object.
3  import java.awt.*;
4
5  public class MyLine extends MyShape
6  {
7     // constructor
8     public MyLine( int firstX, int firstY, int secondX, int secondY,
9        Color shapeColor )
10    {
11       super( firstX, firstY, secondX, secondY, shapeColor );
12
13    } // end constructor
14
```

Figure 27.32 **My Line** code. (Part 1 of 2.)

```
15    // draw a line
16    public void draw( Graphics g )
17    {
18       g.setColor( getColor() );
19       g.drawLine( getX1(), getY1(), getX2(), getY2() );
20
21    } // end method draw
22
23 } // end class MyLine
```

Implementing the abstract `draw` method from `MyShape`

Figure 27.32 **My Line** code. (Part 2 of 2.)

```
1  // Tutorial 27: MyRectangle.java
2  // Class that declares a rectangle object.
3  import java.awt.*;
4
5  public class MyRectangle extends MyShape
6  {
7     // constructor
8     public MyRectangle( int firstX, int firstY, int secondX,
9        int secondY, Color shapeColor )
10    {
11       super( firstX, firstY, secondX, secondY, shapeColor );
12
13    } // end constructor
14
15    // draw a rectangle
16    public void draw( Graphics g )
17    {
18       int upperLeftX = Math.min( getX1(), getX2() );
19       int upperLeftY = Math.min( getY1(), getY2() );
20       int width = Math.abs( getX1() - getX2() );
21       int height = Math.abs( getY1() - getY2() );
22
23       g.setColor( getColor() );
24       g.fillRect( upperLeftX, upperLeftY, width, height );
25
26    } // end method draw
27
28 } // end class MyRectangle
```

Implementing the abstract `draw` method from `MyShape`

Calculating the *x*- and *y*-coordinates, width and height of the rectangle

Drawing a rectangle

Figure 27.33 **My Rectangle** code.

```
1  // Tutorial 27: MyOval.java
2  // Class that declares an oval object.
3  import java.awt.*;
4
5  public class MyOval extends MyShape
6  {
7     // constructor
8     public MyOval( int firstX, int firstY, int secondX, int secondY,
9        Color shapeColor )
10    {
11       super( firstX, firstY, secondX, secondY, shapeColor );
12
13    } // end constructor
14
```

Extending class `MyShape`

`MyOval`'s constructor takes five argurments

Figure 27.34 **My Oval** code. (Part 1 of 2.)

Implementing the abstract `draw` method from `MyShape`

Calculating the *x*- and *y*-coordinates, width and height of the rectangle

```
15    // draw an oval
16    public void draw( Graphics g )
17    {
18       int upperLeftX = Math.min( getX1(), getX2() );
19       int upperLeftY = Math.min( getY1(), getY2() );
20       int width = Math.abs( getX1() - getX2() );
21       int height = Math.abs( getY1() - getY2() );
22
23       g.setColor( getColor() );
24       g.fillOval( upperLeftX, upperLeftY, width, height );
25
26    } // end method draw
27
28 } // end class MyOval
```

Figure 27.34 **My Oval** code. (Part 2 of 2.)

SELF-REVIEW

1. The `min` and `abs` methods belong to the _____ class.
   a) `Calc` b) `Math`
   c) `Calculation` d) `Number`
2. The `drawLine`, `fillOval` and `setColor` methods belong to the _____ class.
   a) `Draw` b) `Graphics`
   c) `Drawing` d) `Graphic`

Answers: 1) b. 2) b.

27.5 Wrap-Up

In this tutorial, you learned about polymorphism. You created a **Drawing Shapes** application, which allows you to draw a picture by combining different colored shapes. You learned how to use additional `Graphics` methods to draw a line, a filled rectangle and a filled oval.

While building the **Drawing Shapes** application, you used an inheritance hierarchy consisting of the `MyShape` superclass and the `MyLine`, `MyRectangle` and `MyOval` subclasses. You also handled objects of the three subclasses polymorphically—by treating them as objects of the `MyShape` superclass.

In the next tutorial, you will learn about the Java Speech API which produces synthetic speech from text input. You will use this technology to create a phone book application that will speak a selected person's phone number.

SKILLS SUMMARY

Drawing a Rectangle

- Use the `Graphics` method `drawRect` to draw the rectangle specified by its *x*- and *y*-coordinates, width and height.

Drawing an Oval

- Use the `Graphics` method `drawOval` to draw the oval specified by its bounding box's *x*- and *y*-coordinates, width and height.

Drawing a Line

- Use the `Graphics` method `drawLine` to draw the line specified by its beginning and ending *x*- and *y*-coordinates.

KEY TERMS

`abs` method of the `Math` class—Returns the absolute value of a given value.

absolute value—The value of a number without the sign of the number.

abstract class—A class that cannot be instantiated. Often called an abstract superclass because it is usable only as the superclass in an inheritance hierarchy. These classes are incomplete; they are missing pieces necessary for instantiation which concrete subclasses must implement.

`abstract` keyword—Used to declare that a class or method is abstract.

abstract method—Contains a method header but no method body. Any class with an abstract method must be an abstract class.

concrete class—A class that can be instantiated.

`drawLine` method of the `Graphics` class—Draws a line using the given *x*- and *y*-coordinates.

`drawOval` method of the `Graphics` class—Draws an oval using the bounding box's upper-left *x*- and *y*-coordinates and the width and height.

`drawRect` method of the `Graphics` class—Draws a rectangle using the given *x*- and *y*-coordinates and the rectangle's width and height.

`min` method of the `Math` class—Returns the minimum of two values.

polymorphism—Concept that allows you to write applications that handle, in a more general manner, a wide variety of classes related by inheritance.

JAVA LIBRARY REFERENCE

`Graphics` The `Graphics` class provides methods to draw shapes of varying colors.

- ***Methods***

`drawLine`—Takes four arguments and draws a line at the specified beginning and ending *x*- and *y*-coordinates.

`drawOval`—Takes four arguments and draws an unfilled oval inside a bounding rectangular area. The first two arguments are the *x*- and *y*-coordinates of the top-left corner of the rectangular area and the second two are the width and height.

`drawRect`—Takes four arguments and draws an unfilled rectangle at the specified upper-left *x*- and *y*-coordinates and of the specified width and height.

`fillRect`—Takes four arguments and draws a solid rectangle at the specified upper-left *x*- and *y*-coordinates and of the specified width and height.

`fillOval`—Takes four arguments and draws a solid oval inside a bounding rectangular area. The first two arguments are the *x*- and *y*-coordinates of the top-left corner of the rectangular area and the second two are the width and height.

`setColor`—Sets the color of the `Graphics` object.

`Math` The `Math` class provides methods to perform different mathematical functions.

- ***Methods***

`abs`—Returns the absolute value of its argument.

`max`—Returns the greater of its two arguments.

`min`—Returns the lesser of its two arguments.

MULTIPLE-CHOICE QUESTIONS

27.1 The code _____ will draw a solid circle.

a) `drawCircle( 50, 50, 25 );`
b) `fillOval( 50, 25, 50, 25 );`
c) `fillOval( 50, 50, 25, 25 );`
d) `drawOval( 50, 50, 50, 50 );`

27.2 Because of polymorphism, using the same _____ can cause different actions to occur depending on the type of the object on which a method is invoked.

a) method return type
b) instance variable
c) local variable
d) method signature

27.3 The _____ method returns the absolute value of a number.

a) `abs`
b) `absolute`
c) `positive`
d) `positiveValue`

27.4 If `MyTruck` extends `MyCar`, _____.

a) an object of `MyTruck` can be assigned to a variable of type `MyCar`
b) an object of `MyCar` can be assigned to a variable of type `MyTruck`
c) objects of either class cannot be assigned to the opposite class
d) both a and b.

27.5 Polymorphism allows you to program _____ .

a) "in the abstract"
b) "in the general"
c) "in the specific"
d) Both a and b.

27.6 The first and third arguments taken by the `drawLine` method specify the line's _____ coordinates.

a) upper-left
b) *x*-
c) *y*-
d) none of the above

27.7 Methods such as `drawOval` and `drawRect` are declared in the _____ class.

a) `Drawing`
b) `Paint`
c) `Graphics`
d) `Images`

27.8 In applications that use polymorphism, the exact type of an object _____.

a) is known only during execution
b) is decided when the application is compiled
c) is known while you are coding
d) is never known

27.9 The code _____ will draw a straight, vertical line.

a) `drawLine( 50, 50, 25, 25 );`
b) `drawLine( 25, 25, 50, 25 );`
c) `drawLine( 50, 25, 50, 25 );`
d) `drawLine( 50, 25, 50, 50 );`

27.10 Polymorphism involves using a variable of a _____ type to invoke methods on superclass and subclass objects.

a) primitive
b) superclass
c) subclass
d) none of the above

EXERCISES

27.11 (***Advanced* Screen Saver *Application***) Write an application that mimics the behavior of a screen saver. It should draw random shapes onto a black background and the shapes should build up on top of each other until the screen saver resets (every 30 seconds). You have been provided with a **Screen Saver** application that does not yet display outlined shapes. It uses the `MyRectangle` and `MyOval` classes that you created in this tutorial. Add the code that will display random outlined shapes in your output. Your output should look like Fig. 27.35.

a) ***Copying the template to your working directory.*** Copy the directory `C:\Examples\Tutorial27\Exercises\AdvancedScreenSaver` to your `C:\SimplyJava` directory.

b) ***Opening the template file.*** Open the `MyRectangle.java` file in your text editor.

c) ***Adding an instance variable to the `MyRectangle` class.*** At line 7, add a comment indicating that the instance variable is a `boolean` and will indicate whether or not the rectangle is filled. At line 8, add a `private` instance variable named `filled` of type `boolean`.

d) ***Modifying the `MyRectangle` constructor.*** You will now modify the `MyRectangle` constructor so that it can accept an additional `boolean` argument. At line 12, add a `boolean` argument named `fill` to the end of the parameter list. At line 16, set the instance variable `filled` equal to the value of parameter `fill` and on the same line, add a comment indicating that `filled` will specify if the shape will be filled.

e) ***Modifying the `draw` method.*** At line 31, add comment indicating that an `if` statement will execute if the rectangle is filled. At line 32, add an `if` statement that checks if `filled` is `true`. If it is, then the application should call the `fillRect` method (which is on line 30 of the template).

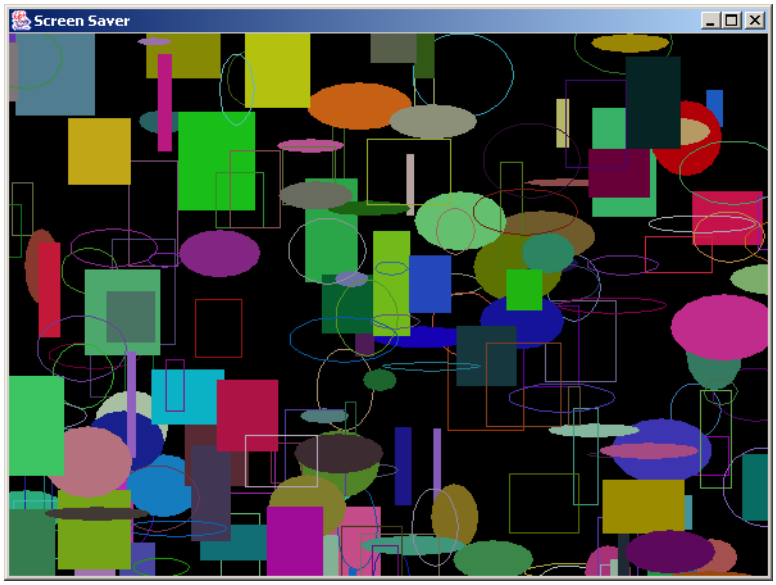

Figure 27.35 Advanced **Screen Saver** application.

f) ***Finishing the draw method.*** At line 37, add an `else` to the `if` statement from the previous step. If `filled` is `false`, the application should call the `drawRect` method.

g) ***Saving the application.*** Save your modified source code file.

h) ***Opening the template file.*** Open the `MyOval.java` file in your text editor.

i) ***Modifying the `MyOval` class.*** Apply *Steps c–f* to the `MyOval` class. The line numbers for `MyOval` will be the same as `MyRectangle`. Use the `fillOval` and `drawOval` methods in place of the `fillRect` and `drawRect` methods respectively.

j) ***Saving the application.*** Save your modified source code file.

k) ***Opening the template file.*** Open the `DrawJPanel.java` file in your text editor.

l) ***Modifying the shape constructor calls.*** You will now add a `boolean` argument to the statements that invoke the shape constructors. On line 117, add an additional argument to the end of the list of arguments. The statement being modified is creating an outlined oval, which means it should not be filled. So, the additional argument should be the keyword `false`. This will result in instance variable `filled`, of the `MyOval` class, being set to `false`. On line 123, add the additional argument, `true`, to the end of the list of arguments. Now, when this line of code is executed, a `MyOval` object with instance variable `filled` set to `true` will be created. On line 130, add the additional argument, `false`, to the end of the list of arguments. When this line of code is executed, a `MyRectangle` object with instance variable `filled` set to `false` will be created. Finally, on line 136, add the additional argument, `true`, to the end of the list of arguments. When this line of code is executed, a `MyRectangle` object with instance variable `filled` set to `true` will be created.

m) ***Saving the application.*** Save your modified source code file.

n) ***Opening the Command Prompt window and changing directories.*** Open the **Command Prompt** window by selecting **Start > Programs > Accessories > Command Prompt**. Change to your working directory by typing `cd C:\SimplyJava\Advanced-ScreenSaver`.

o) ***Compiling the application.*** Compile your application by typing `javac ScreenSaver.java DrawJPanel.java MyRectangle.java MyOval.java`.

p) ***Running the completed application.*** When your application compiles correctly, run it by typing `java ScreenSaver`. Test your application by ensuring that shapes appear and that the screen clears itself every thirty seconds.

q) ***Closing the application.*** Close your running application by clicking its close button.

r) ***Closing the Command Prompt window.*** Close the **Command Prompt** window by clicking its close button.

27.12 (***Logo Designer Application***) Write an application that allows users to design a company logo. It should be able to draw lines as well as both filled and empty rectangles and ovals with a simple coordinate input interface. Your GUI should look like Fig. 27.36.

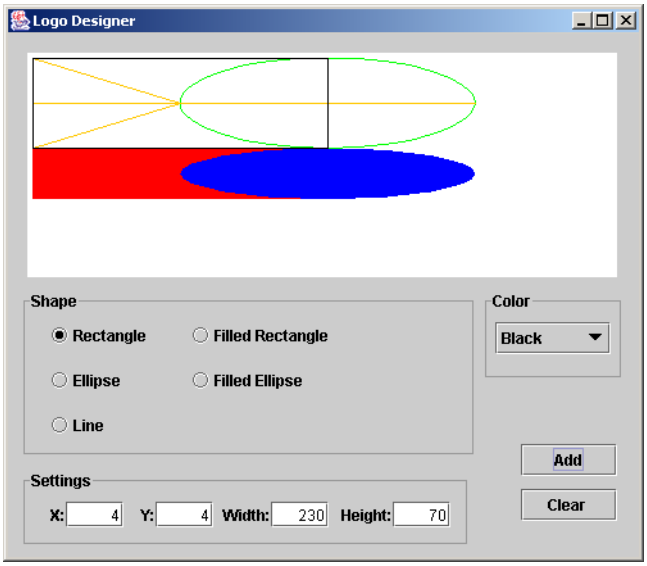

Figure 27.36 **Logo Designer** application.

a) ***Copying the template to your working directory.*** Copy the directory `C:\Examples\Tutorial27\Exercises\LogoDesigner` to your `C:\SimplyJava` directory.

b) ***Opening the template file.*** Open the `MyRectangle.java` and `MyOval.java` files in your text editor.

c) ***Modifying the `MyRectangle` and `MyOval` classes.*** Apply *Steps c–j* of the previous exercise (Exercise 27.11) to your `MyRectangle` and `MyOval` classes. This will add the ability to draw both filled and outlined shapes to your shape hierarchy.

d) ***Opening the template file.*** Open the `DrawJPanel.java` file in your text editor.

e) ***Adding the `addShape` method.*** At line 31, add a comment indicating that the method will add the shape to `shapeArray` and then repaint. On line 32, add the method header for the `addShape` method. This method does not return a value and takes an argument of type `MyShape` named `shape`. Add `shape` to `shapeArrayList` by calling the `add` method on `shapeArrayList` and passing it `shape`. Then, call the `repaint` method so that the newly added shape will be displayed. Be sure to end the method with a right brace on line 37.

f) ***Saving the application.*** Save your modified source code file.

g) ***Opening the template file.*** Open the `LogoDesigner.java` file in your text editor.

h) ***Invoking method `addShape` to draw a line.*** You will now invoke method `addShape` in order to display a new line on the `JPanel`. At lines 279–280, call method `addShape` on variable `drawingJPanel`. Pass it a new `MyLine` object created with the arguments `x`, `y`, `width`, `height` and `drawColor`.

i) ***Invoking method `addShape` to draw an oval.*** You will now invoke method `addShape` in order to display a new, outlined oval on the `JPanel`. On lines 284–285, call method `addShape` on variable `drawingJPanel`. Pass it a new `MyOval` object created with the arguments `x`, `y`, `x + width`, `y + height`, `drawColor` and `false`. On lines 289–290, call `addShape` again, but this time draw a filled oval instead of an outlined one by changing the `boolean` value at the end of the argument list to `true`.

j) ***Invoking method `addShape` to draw a rectangle.*** You will now invoke method `addShape` in order to display a new, outlined rectangle on the `JPanel`. On lines 294–295, call method `addShape` on variable `drawingJPanel`. Pass it a new `MyRectangle` object created with the arguments `x`, `y`, `x + width`, `y + height`, `drawColor` and `false`. On lines 299–300, call `addShape` again, but this time draw a filled rectangle instead of an outlined one by changing the `boolean` value at the end of the argument list to `true`.

k) ***Saving the application.*** Save your modified source code file.

l) ***Opening the Command Prompt window and changing directories.*** Open the **Command Prompt** by selecting **Start > Programs > Accessories > Command Prompt**. Change to your working directory by typing `cd C:\SimplyJava\LogoDesigner`.

m) ***Compiling the application.*** Compile your application by typing `javac LogoDesigner.java DrawJPanel.java MyRectangle.java MyOval.java`.

n) ***Running the completed application.*** When your application compiles correctly, run it by typing `java LogoDesigner`. Test your application by drawing different shapes using different *x*- and *y*- coordinates, heights and widths.

o) ***Closing the application.*** Close your running application by clicking its close button.

p) ***Closing the Command Prompt window.*** Close the **Command Prompt** window by clicking its close button.

27.13 (***Whack A Mole Application***) Create a **Whack A Mole**[1] game application that emulates its popular arcade counterpart. Allow players to start a new game by clicking a button. Then, a mole should appear randomly within a single cell of an outlined grid. Clicking on the mole before it moves will add 50 points to the score. Playing the game should result in output similar to Fig. 27.37.

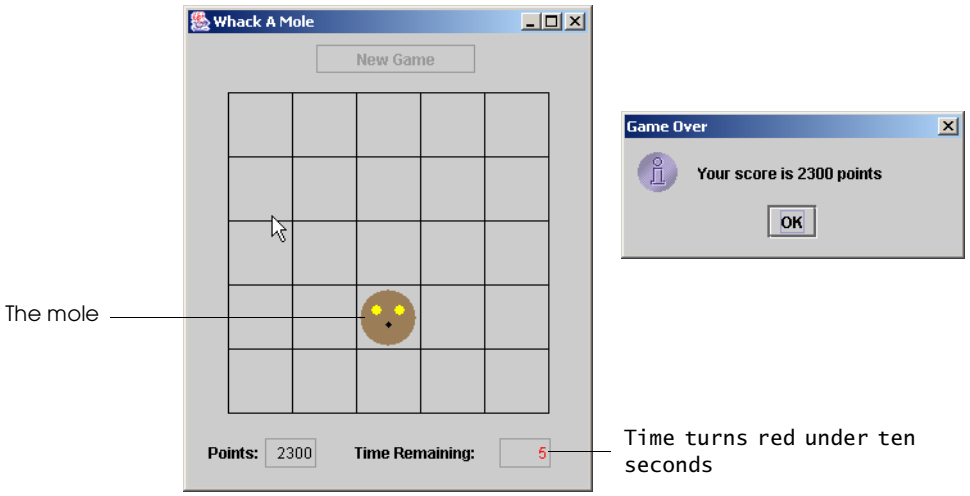

Figure 27.37 **Whack A Mole** application.

a) ***Copying the template to your working directory.*** Copy the `C:\Examples\Tutorial27\Exercises\WhackAMole` directory to your `C:\SimplyJava` directory.

b) ***Opening the template file.*** Open the `Mole.java` file in your text editor.

c) ***Declaring local variables in the `drawMole` method.*** At line 23, add a comment indicating that the cell dimensions will be calculated. At line 24, declare and initialize a local variable of type `int` named `x`. Set `x` equal to `moleColumn * 50`. Next, declare and initialize another local variable of type `int` named `y`. Set `y` equal to `moleRow * 50`. Variables `x` and `y` represent the *x*- and *y*-coordinates in pixels of each cell. These variables will be used in later calculations.

d) ***Drawing the mole's head in the `drawMole` method.*** At line 27, add a comment indicating that the mole's head color will be set. Now, notice that the parameter list of the `drawMole` method indicates that it will be passed an instance of `Graphics` named `g`. On line 28, call the `setColor` method on `g`. Pass a `new Color` to method `setColor`. Pass the integer values, `155`, `126`, and `87` to the `new Color` constructor. Next, add a comment indicating that the mole's head will be drawn, then call the `fillOval` method on `g`. Pass the following arguments to method `fillOval`: `x + 38`, `y + 72`, `44` and `44`.

e) ***Drawing the mole's eyes in the `drawMole` method.*** At line 33, call the `setColor` method on `g` to set the mole's eye color. Pass constant `Color.YELLOW` to the `set-`

1. Be careful before you download any **Whack A Mole** games from the Internet. For a while there was a virus-infected version that would read your hard drive while you were playing.

`Color` method. On line 35, add a comment indicating that the mole's eyes will be drawn, then, on line 36, call the `fillOval` method on `g`. Pass the following arguments to method `fillOval`: `x + 47`, `y + 84`, `8` and `8`. On line 37, call the `fillOval` method on `g`. Pass the following arguments to method `fillOval`: `x + 65`, `y + 84`, `8` and `8`.

f) ***Drawing the mole's nose in the drawMole method.*** At line 39, call the `setColor` method on `g`. Pass constant `Color.BLACK` to the `setColor` method. On line 40, call the `fillOval` method on `g`. Pass the following arguments to method `fillOval`: `x + 58`, `y + 97`, `5` and `5`.

g) ***Saving the application.*** Save your modified source code file.

h) ***Opening the Command Prompt window and changing directories.*** Open the **Command Prompt** by selecting **Start > Programs > Accessories > Command Prompt**. Change to your working directory by typing `cd C:\SimplyJava\WhackAMole`.

i) ***Compiling the application.*** Compile your application by typing `javac WhackAMole.java Mole.java`.

j) ***Running the completed application.*** When your application compiles correctly, run it by typing `java WhackAMole`. Test your application by playing the game a few times. Make sure that the mole looks as shown in Fig. 27.37.

k) ***Closing the application.*** Close your running application by clicking its close button.

l) ***Closing the Command Prompt window.*** Close the **Command Prompt** window by clicking its close button.

What does this code do? ▶

27.14 What is the result of the following code? Assume that the classes used are those from the **Drawing Shapes** application and that this method is in the `PainterJPanel` class.

```
1  private void drawJButtonActionPerformed( ActionEvent event )
2  {
3     MyOval oval;
4
5     for ( int i = 0; i <= 50; i += 10 )
6     {
7        oval = new MyOval( i, 20, 10, 10, Color.GREEN );
8        shapes.add( oval );
9
10    } // end for
11
12    repaint();
13
14 } // end method drawJButtonActionPerformed
```

What's wrong with this code? ▶

27.15 Find the error(s) in the following code. This is the definition for an `actionPerformed` event handler for a `JButton`. This event handler should draw a rectangle on a `JPanel`. Assume that the classes used are those from the **Drawing Shapes** application.

```
1  private void drawImageJButtonActionPerformed( ActionEvent event )
2  {
3     // set shape
4     MyShape rectangle = new MyRectangle( 2, 3, 40, 30 );
5
6     // set color
7     rectangle.setColor( Color.ORANGE );
8
9     // add rectangle to shapesArrayList
10    shapesArrayList.add( rectangle );
11
12 } // end method drawImageJButtonActionPerformed
```

Programming Challenge ▶

27.16 (***Moving Shapes Application***) Enhance the **Drawing Shapes** application that you created in this tutorial. Improve the application so that once you finish drawing a shape, the shape will be given a random velocity and begin to move, bouncing off the walls of the `PaintJPanel`. Your output should be capable of looking look like Fig. 27.38.

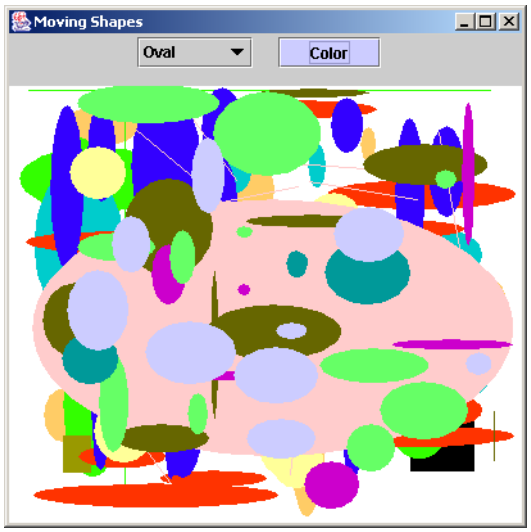

Figure 27.38 **Moving Shapes** application.

a) ***Copying the template to your working directory.*** Copy the directory `C:\Examples\Tutorial27\Exercises\MovingShapes` to your `C:\SimplyJava` directory.

b) ***Opening the template file.*** Open the `MyMovingShape.java` file in your text editor.

c) ***Adding a method to your `MyMovingShape` class to change the position of the shape.*** The abstract superclass for this inheritance hierarchy has been renamed `MyMovingShape`. Add a `public` method named `moveShape` to the class. It should take no arguments and have no return type. Two new instance variables, `dx` and `dy`, have been added to the `MyMovingShape` class for you. Variable `dx` holds the distance along the *x*-axis that the shape must travel in one move. Variable `dy` holds the distance along the *y*-axis that the shape must travel in one move. Add `dx` to the `x1` and `x2` values and add `dy` to the `y1` and `y2` values. Follow good programming practice by using the corresponding `get` and `set` methods instead of modifying the variables directly.

d) ***Finishing the `moveShape` method.*** Add two `if` statements to the `moveShape` method to reverse the direction of the shape if it has hit a wall. The first `if` statement should check if either *x*-coordinate (`x1` or `x2`) is less than `0` or greater than `400`. If this is `true` then set the value of `dx` equal to the negative of itself. Make sure that you use the correct `get` or `set` methods to do this. The second `if` statement should check if either *y*-coordinate (`y1` or `y2`) is less than `0` or greater than `340`. If this is true then set the value of `dy` equal to the negative of itself. Again, make sure that you use the correct `get` or `set` methods to do this.

e) ***Saving the application.*** Save your modified source code file.

f) ***Opening the template file.*** Open the `PaintJPanel.java` file in your text editor.

g) ***Modifying the `moveTimerActionPerformed` method.*** The `moveTimerActionPerformed` method will iterate through every shape in `shapeArrayList` to call the `moveShape` method of each shape. To do this, first declare a local variable of type `MyMovingShape` named `nextShape`. Declare another local variable of type `Iterator` named, `shapesIterator` and initialize it to the value returned by calling the `iterator` method on `shapeArrayList`. Then, create a `while` loop whose condition is the `boolean` returned by calling the `hasNext` method of `shapesIterator`. Within the `while` loop, set `nextShape` equal to the reference returned by the `next` method of `shapesIterator`. The `next` method will return the next indexed object in `shapeArrayList`, which may be of type `MyLine`, `MyRectangle`, or `MyOval`. This means that you will have to cast the returned object to a `MyMovingShape` object before storing it

in a variable of type `MyMovingShape`. Before ending the `while` loop, call the `moveShape` method on `nextShape`. The `while` loop you have created will now iterate through every shape in `shapeArrayList` to call the `moveShape` method of each shape.

h) ***Saving the application.*** Save your modified source code file.

i) ***Opening the Command Prompt window and changing directories.*** Open the **Command Prompt** by selecting **Start > Programs > Accessories > Command Prompt**. Change to your working directory by typing `cd C:\SimplyJava\MovingShapes`.

j) ***Compiling the application.*** Compile your application by typing `javac MovingShapes.java PaintJPanel.java MyMovingShape.java`.

k) ***Running the completed application.*** When your application compiles correctly, run it by typing `java MovingShapes`. Test your application by drawing each of the three shapes and pick a different color for each of them. Make sure that the shapes move around and bounce off all of the walls.

l) ***Closing the application.*** Close your running application by clicking its close button.

m) ***Closing the Command Prompt window.*** Close the **Command Prompt** window by clicking its close button.