T U T O R I A L

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 processes 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
27.1 Test-Driving the Drawing Shapes Application
27.2 Polymorphism
27.3 More Graphics Methods
27.4 Adding to the MyShape Inheritance Hierarchy
27.5 Wrap-Up

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.

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](image1.png)

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](image2.png)
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](image)

**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.

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**

<table>
<thead>
<tr>
<th>Action</th>
<th>Component/Object</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>When the user presses the mouse button:</td>
<td></td>
<td>User presses a mouse button</td>
</tr>
<tr>
<td>If Line is selected in JComboBox</td>
<td>painterJPanel</td>
<td></td>
</tr>
<tr>
<td>Create a line</td>
<td>shapeJComboBox</td>
<td></td>
</tr>
<tr>
<td>If Rectangle is selected in JComboBox</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create a rectangle</td>
<td>shapeJComboBox</td>
<td></td>
</tr>
<tr>
<td>If Oval is selected in JComboBox</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create an oval</td>
<td>shapeJComboBox</td>
<td></td>
</tr>
<tr>
<td>When the user clicks the Color JButton:</td>
<td></td>
<td>User clicks Color JButton</td>
</tr>
<tr>
<td>Display a JColorChooser dialog</td>
<td>colorJButton</td>
<td></td>
</tr>
<tr>
<td>Update the JButton’s color with the selected color</td>
<td>JColorChooser</td>
<td></td>
</tr>
<tr>
<td>Set the current shape color to the selected color</td>
<td>colorJButton</td>
<td></td>
</tr>
<tr>
<td>Set the current shape type to the selected item</td>
<td>paintJPanel</td>
<td></td>
</tr>
<tr>
<td>Get the shape type selected</td>
<td>shapeJComboBox</td>
<td>User selects an item in the JComboBox</td>
</tr>
<tr>
<td>Set the current shape type to the selected item</td>
<td>paintJPanel</td>
<td></td>
</tr>
<tr>
<td>Resize the shape</td>
<td>currentShape</td>
<td>User drags the mouse</td>
</tr>
<tr>
<td>Repaint the application</td>
<td>painterJPanel</td>
<td></td>
</tr>
</tbody>
</table>

**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](image)

   **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.
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.

<table>
<thead>
<tr>
<th>Graphics Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>drawLine(x1, y1, x2, y2)</code></td>
<td>Draws a line from the point ((x1, y1)) to the point ((x2, y2)).</td>
</tr>
<tr>
<td><code>drawRect(x, y, width, height)</code></td>
<td>Draws a rectangle of the specified width and height. The top-left corner of the rectangle is at the point ((x, y)).</td>
</tr>
<tr>
<td><code>fillRect(x, y, width, height)</code></td>
<td>Draws a solid rectangle of the specified width and height. The top-left corner of the rectangle is at the point ((x, y)).</td>
</tr>
<tr>
<td><code>drawOval(x, y, width, height)</code></td>
<td>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)).</td>
</tr>
<tr>
<td><code>fillOval(x, y, width, height)</code></td>
<td>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)).</td>
</tr>
<tr>
<td><code>setColor(color)</code></td>
<td>Sets the drawing color to the specified color.</td>
</tr>
</tbody>
</table>

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` super-class, 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.
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.

### 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.

### 3. Saving the application
Save your modified source code file.
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](image1)

**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](image2)

**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.
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.

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.

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.
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.

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.

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.
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.

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.

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.

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

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.
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.

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.

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.

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.

   ![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.

   ![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`. 
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.

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 27.28–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.

```java
// Tutorial 27: DrawingShapes.java
// Application allows user to draw lines, rectangles and ovals and
// choose the color of the drawn shape.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DrawingShapes extends JFrame {

    // JPanel for the shape and color controls
    private JPanel controlsJPanel;

    // JComboBox to allow selection of a shape
    private JComboBox shapeJComboBox;

    // JButton to select the color
    private JButton colorJButton;

    // array of shape types
    private String[] shapeTypes = { "Line", "Rectangle", "Oval" };

    // PaintJPanel instance variable
    private PaintJPanel painterPaintJPanel;

    // array of shape names
}
```

**Figure 27.29** Drawing Shapes code. (Part 1 of 3.)
public DrawingShapes()
{
    createUserInterface();
}

private void createUserInterface()
{
    Container contentPane = getContentPane();

    contentPane.setLayout( null );

    controlsJPanel = new JPanel();
    controlsJPanel.setBounds( 0, 0, 400, 40 );
    controlsJPanel.setLayout( null );
    contentPane.add( controlsJPanel );

    shapeJComboBox = new JComboBox( shapeTypes );
    shapeJComboBox.setBounds( 90, 2, 100, 24 );
    controlsJPanel.add( shapeJComboBox );

    colorJButton = new JButton();
    colorJButton.setBounds( 210, 2, 80, 24 );
    colorJButton.setText( "Color" );
    controlsJPanel.add( colorJButton );

    painterPaintJPanel = new PaintJPanel();
    painterPaintJPanel.setBounds( 0, 40, 400, 340 );
    painterPaintJPanel.setBackground( Color.WHITE );
    contentPane.add( painterPaintJPanel );

} // end call to addActionListener

new ActionListener() // anonymous inner class
{
    public void actionPerformed( ActionEvent event )
    {
        shapeJComboBoxActionPerformed( event );
    }

} // end anonymous inner class

new ActionListener() // anonymous inner class
{
    public void actionPerformed( ActionEvent event )
    {
        colorJButtonActionPerformed( event );
    }

} // end anonymous inner class

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

```java
public class PaintJPanel extends JPanel {

    // ArrayList to hold the shapes
    private ArrayList shapesArrayList = new ArrayList();

    // set the selected shape in the painting panel
    private void shapeJComboBoxActionPerformed(ActionEvent event) {
        painterPaintJPanel.setCurrentShapeType((String)shapeJComboBox.getSelectedItem());
    }

    // Tutorial 27: PaintJPanel.java
    // Panel allows user to create a shape.
    import java.awt.*;
    import java.awt.event.*;
    import java.util.*;
    import javax.swing.*;

    public class PaintJPanel extends JPanel {

        // ArrayList to hold the shapes
        private ArrayList shapesArrayList = new ArrayList();
    }

    // set the selected shape in the painting panel
    private void shapeJComboBoxActionPerformed(ActionEvent event) {
        painterPaintJPanel.setCurrentShapeType((String)shapeJComboBox.getSelectedItem());
    }

    // main method
    public static void main(String args[]) {
        DrawingShapes application = new DrawingShapes();
        application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    // end class DrawingShapes
```

Figure 27.30  PaintJPanel code. (Part 1 of 3.)
// current shape that is being drawn  
private MyShape currentShape;  

// currently selected shape type  
private String currentType = "Line";  

// currently selected color  
private Color currentColor = new Color(204,204,204);  

// no-argument constructor  
public PaintJPanel()  
{  
    addMouseListener(  
        new MouseAdapter() // anonymous inner class  
        {  
            // event handler called when mouse button is pressed  
            public void mousePressed( MouseEvent event )  
            {  
                paintJPanelMousePressed( event );  
            }  
        } // end anonymous inner class  
    ); // end call to addMouseListener  
}

addMouseMotionListener(  
    new MouseMotionAdapter() // anonymous inner class  
    {  
        // event handler called when the mouse is dragged  
        public void mouseDragged( MouseEvent event )  
        {  
            paintJPanelMouseDragged( event );  
        }  
    } // end anonymous inner class  
); // end call to addMouseMotionListener  

// change the current shape type  
public void setCurrentShapeType( String shape )  
{  
    currentType = shape;  
} // end method setCurrentShapeType  

// change the current color  
public void setCurrentColor( Color shapeColor )  
{  
    currentColor = shapeColor;  
} // end method setCurrentColor  

Figure 27.30  PaintJPanel code. (Part 2 of 3.)
// create a new shape
public void paintJPanelMousePressed( MouseEvent event )
{
    // user selected line
    if ( currentType.equals( "Line" ) )
    {
        currentShape = new MyLine( event.getX(), event.getY(),
                event.getX(), event.getY(), currentColor );
    }
    // user selected rectangle
    else if ( currentType.equals( "Rectangle" ) )
    {
        currentShape = new MyRectangle( event.getX(), event.getY(),
                event.getX(), event.getY(), currentColor );
    }
    // user selected oval
    else if ( currentType.equals( "Oval" ) )
    {
        currentShape = new MyOval( event.getX(), event.getY(),
                event.getX(), event.getY(), currentColor );
    }
    shapesArrayList.add( currentShape );
}
} // end method paintJPanelMousePressed

// reset the second point for the shape
public void paintJPanelMouseDragged( MouseEvent event )
{
    currentShape = new MyShape( event.getX(), event.getY(),
                event.getX(), event.getY(), currentColor );
    currentShape.setX2( event.getX() );
    currentShape.setY2( event.getY() );
    repaint();
}
} // end class PaintJPanel

// paint all the shapes
public void paintComponent( Graphics g )
{
    super.paintComponent( g );
    MyShape nextShape;
    Iterator shapesIterator = shapesArrayList.iterator();
    // iterate through all the shapes
    while ( shapesIterator.hasNext() )
    {
        // draw each shape
        nextShape = ( MyShape ) shapesIterator.next();
        nextShape.draw( g );
    }
} // end method paintComponent

Figure 27.30 PaintJPanel code. (Part 3 of 3.)
// Tutorial 27: MyShape.java
// Superclass for all shape objects.
import java.awt.*;

public abstract class MyShape extends Object {
    private int x1;
    private int y1;
    private int x2;
    private int y2;
    private Color color;

    // constructor
    public MyShape( int firstX, int firstY, int secondX, int secondY,
                    Color shapeColor )
    {
        setX1( firstX );
        setY1( firstY );
        setX2( secondX );
        setY2( secondY );
        setColor( shapeColor );
    } // end constructor

    // set x1 value
    public void setX1( int x )
    {
        x1 = x;
    } // end method setX1

    // get x1 value
    public int getX1()
    {
        return x1;
    } // end method getX1

    // set Y1 value
    public void setY1( int y )
    {
        y1 = y;
    } // end method setY1

    // get Y1 value
    public int getY1()
    {
        return y1;
    } // end method getY1

    // set x2 value
    public void setX2( int x )
    {
        x2 = x;
    } // end method setX2

    // get x2 value
    public int getX2()
    {
        return x2;
    } // end method getX2

    // get y2 value
    public int getY2()
    {
        return y2;
    } // end method getY2

    // set Y2 value
    public void setY2( int y )
    {
        y2 = y;
    } // end method setY2

    // get Y2 value
    public int getY2()
    {
        return y2;
    } // end method getY2

    // set shape color
    public void setColor( Color shapeColor )
    {
        color = shapeColor;
    } // end method setColor

    // get shape color
    public Color getColor()
    {
        return color;
    } // end method getColor

} // end class MyShape

Figure 27.31  My Shape code. (Part 1 of 2.)
// get x2 value
public int getX2()
{
    return x2;
} // end method getX2

// set y2 value
public void setY2( int y )
{
    y2 = y;
} // end method setY2

// get y2 value
public int getY2()
{
    return y2;
} // end method getY2

// set color value
public void setColor( Color c )
{
    color = c;
} // end method setColor

// get color value
public Color getColor()
{
    return color;
} // end method getColor

// abstract draw method
public abstract void draw( Graphics g );

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

// Tutorial 27: MyLine.java
// Class that declares a line object.
import java.awt.*;

public class MyLine extends MyShape
{
    // constructor
    public MyLine( int firstX, int firstY, int secondX, int secondY,
        Color shapeColor )
    {
        super( firstX, firstY, secondX, secondY, shapeColor );
    } // end constructor

Figure 27.32 My Line code. (Part 1 of 2.)
// draw a line
public void draw( Graphics g )
{
    g.setColor( getColor() );
    g.drawLine( getX1(), getY1(), getX2(), getY2() );
}

} // end method draw

} // end class MyLine

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

// Tutorial 27: MyRectangle.java
// Class that declares a rectangle object.
import java.awt.*;

public class MyRectangle extends MyShape {
    // constructor
    public MyRectangle( int firstX, int firstY, int secondX, int secondY, Color shapeColor )
    {
        super( firstX, firstY, secondX, secondY, shapeColor );
    }

    // draw a rectangle
    public void draw( Graphics g )
    {
        int upperLeftX = Math.min( getX1(), getX2() );
        int upperLeftY = Math.min( getY1(), getY2() );
        int width = Math.abs( getX1() - getX2() );
        int height = Math.abs( getY1() - getY2() );

        g.setColor( getColor() );
        g.fillRect( upperLeftX, upperLeftY, width, height );

    }

} // end class MyRectangle

Figure 27.33  My Rectangle code.

// Tutorial 27: MyOval.java
// Class that declares an oval object.
import java.awt.*;

public class MyOval extends MyShape {

    // constructor
    public MyOval( int firstX, int firstY, int secondX, int secondY, Color shapeColor )
    {
        super( firstX, firstY, secondX, secondY, shapeColor );
    }

} // end class MyOval

Figure 27.34  My Oval code. (Part 1 of 2.)
810  Introduction to Polymorphism; an Expanded Discussion of Graphics

Tutorial 27

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).
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.

![Logo Designer application](image)

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.
Tutorial 27

Drawing Shapes Application

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 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.

---

![Whack A Mole Application](image)

**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-`
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 Com-
mmand 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 WhackA-
Mole.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.

```java
private void drawJButtonActionPerformed( ActionEvent event )
{
    MyOval oval;
    for ( int i = 0; i <= 50; i += 10 )
    {
        oval = new MyOval( i, 20, 10, 10, Color.GREEN );
        shapes.add( oval );
    } // end for
    repaint();
} // 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.

```java
private void drawImageJButtonActionPerformed( ActionEvent event )
{
    // set shape
    MyShape rectangle = new MyRectangle( 2, 3, 40, 30 );
    // set color
    rectangle.setColor( Color.ORANGE );
    // add rectangle to shapesArrayList
    shapesArrayList.add( rectangle );
} // 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 like Fig. 27.38.

![Figure 27.38 Moving Shapes application.](image)

**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.