10.14 (Optional) GUI and Graphics Case Study: Drawing with Polymorphism

You may have noticed in the drawing program created in GUI and Graphics Case Study Exercise 8.2 that shape classes have many similarities. Using inheritance, we can "factor out" the common features from classes `MyLine`, `MyRectangle` and `MyOval` and place them in a single shape *superclass*. Then, using variables of the superclass type, we can manipulate shape objects *polymorphically*. Removing the redundant code will result in a smaller, more flexible program that's easier to maintain.

GUI and Graphics Case Study Exercises

10.1 Modify the `MyLine`, `MyOval` and `MyRectangle` classes of GUI and Graphics Case Study Exercise 8.2 to create the class hierarchy in Fig. 10.16. Classes of the `MyShape` hierarchy should be "smart" shape classes that know how to draw themselves (if provided with a `GraphicsContext` object that tells them where to draw). Once the program creates an object from this hierarchy, it can manipulate it

polymorphically for the rest of its lifetime as a `MyShape`.

In your solution, class `MyShape` in Fig. 10.16 *must* be `abstract`. Since `MyShape` represents any shape in general, you cannot implement a `draw` method without knowing *specifically* what shape it is. The data representing the coordinates and color of the shapes in the hierarchy should be declared as `private` instance variables of class `MyShape`. In addition to the common data, class `MyShape` should declare the following methods:

1. A *no-argument constructor* that sets all the coordinates of the shape to `0` and the stroke color to `Color.BLACK`.
2. A constructor that initializes the coordinates and stroke color to the values of the arguments supplied.

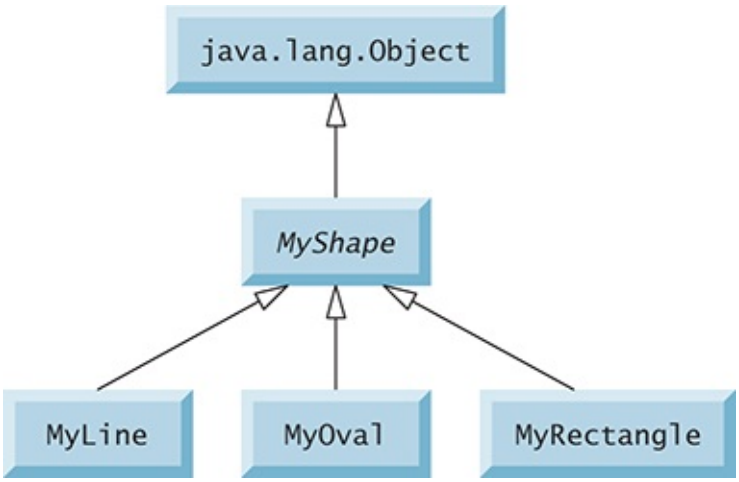

Fig. 10.16

`MyShape` hierarchy.

3. *Set* methods for the individual coordinates and color that allow the

programmer to set any piece of data independently for a shape in the hierarchy.

4. *Get* methods for the individual coordinates and color that allow the programmer to retrieve any piece of data independently for a shape in the hierarchy.
5. The `abstract` method

```
public abstract void draw(GraphicsContext gc);
```

which the program will call to tell a `MyShape` to draw itself on the screen.

To ensure proper encapsulation, all data in class `MyShape` must be `private`. This requires declaring proper *set* and *get* methods to manipulate the data. Class `MyLine` should provide a no-argument constructor and a constructor with arguments for the coordinates and stroke color. Classes `MyOval` and `MyRectangle` should provide a no-argument constructor and a constructor with arguments for the coordinates, stroke color, fill color and determining whether the shape is filled. The no-argument constructor should, in addition to setting the default values, set the shape's `filled` flag to `false` so that an unfilled shape is the default.

You can draw lines, rectangles and ovals if you know two points in space. Lines require *x1*, *y1*, *x2* and *y2* coordinates. `GraphicsContext` method `strokeLine` will connect the two points supplied with a line. If you have the same four coordinate values (*x1*, *y1*, *x2* and *y2*) for ovals and rectangles, you can calculate the four arguments needed to draw them. Each requires an upper-left *x*-coordinate value (the smaller of

the two *x*-coordinate values), an upper-left *y*-coordinate value (the smaller of the two *y*-coordinate values), a *width* (the absolute value of the difference between the two *x*-coordinate values) and a *height* (the absolute value of the difference between the two *y*-coordinate values).

There should be no `MyLine`, `MyOval` or `MyRectangle` variables in the program—only `MyShape` variables that contain references to `MyLine`, `MyOval` and `MyRectangle` objects. The program should generate random shapes and store them in an `ArrayList<MyShape>`. When it's time to draw the `MyShape` objects, iterate through the `ArrayList<MyShape>` and polymorphically call every shape's `draw` method. Allow the user to specify the number of shapes to generate. The program will then generate and display the specified number of shapes. When creating the shapes, use random-number generation to select which shape to create and to specify the default values for that shape's coordinates, stroke color and, for rectangles and ovals, whether the shape is filled and its fill color.

10.2 ***(Drawing Application Modification)*** In the preceding exercise, you created a `MyShape` hierarchy in which classes `MyLine`, `MyOval` and `MyRectangle` extend `MyShape` directly. Notice the similarities between the `MyOval` and `MyRectangle` classes. Redesign and reimplement the code for the `MyOval` and `MyRectangle` classes to "factor out" the common features into the abstract class `MyBoundedShape` to produce the hierarchy in Fig. 10.17. Class `MyBoundedShape` should declare two constructors

that mimic those of class `MyShape`, but with an added parameter specifying whether the shape is filled. Class `MyBoundedShape` should also declare *get* and *set* methods for manipulating the `filled` flag and fill color, and methods that calculate the upper-left *x*-coordinate, upper-left *y*-coordinate, width and height. Remember, the values needed to draw an oval or a rectangle can be calculated from two (*x*, *y*) coordinates. If designed properly, the new `MyOval` and `MyRectangle` classes should each have only two constructors and a `draw` method.

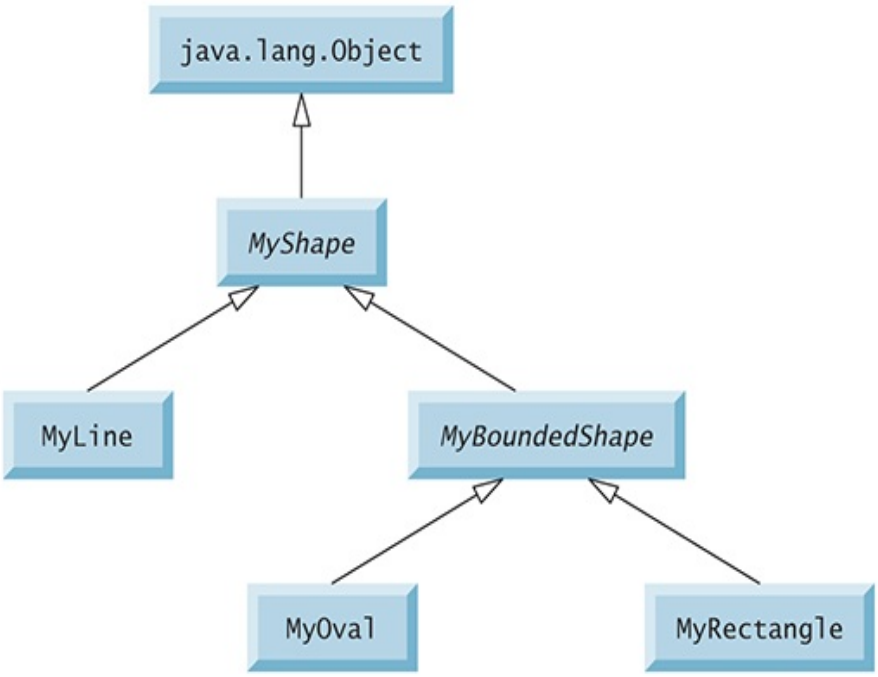

Fig. 10.17

`MyShape` hierarchy with `MyBoundedShape`.

Description