

CHAPTER 14

JAVAFX BASICS

Objectives

- To distinguish between JavaFX, Swing, and AWT (§14.2).
- To write a simple JavaFX program and understand the relationship among stages, scenes, and nodes (§14.3).
- To create user interfaces using panes, groups, UI controls, and shapes (§14.4).
- To update property values automatically through property binding (§14.5).
- To use the common properties **style** and **rotate** for nodes (§14.6).
- To create colors using the **Color** class (§14.7).
- To create fonts using the **Font** class (§14.8).
- To create images using the **Image** class, and to create image views using the **ImageView** class (§14.9).
- To layout nodes using **Pane**, **StackPane**, **FlowPane**, **GridPane**, **BorderPane**, **HBox**, and **VBox** (§14.10).
- To display text using the **Text** class, and create shapes using the **Line**, **Circle**, **Rectangle**, **Ellipse**, **Arc**, **Polygon**, and **Polyline** classes (§14.11).
- To develop the reusable GUI component **ClockPane** for displaying an analog clock (§14.12).





14.1 Introduction

JavaFX is an excellent pedagogical tool for learning object-oriented programming.

JavaFX is a new framework for developing Java GUI programs. The JavaFX API is an excellent example of how the object-oriented principles are applied. This chapter serves two purposes. First, it presents the basics of JavaFX programming. Second, it uses JavaFX to demonstrate object-oriented design and programming. Specifically, this chapter introduces the framework of JavaFX and discusses JavaFX GUI components and their relationships. You will learn how to develop simple GUI programs using layout panes, groups, buttons, labels, text fields, colors, fonts, images, image views, and shapes.



14.2 JavaFX vs. Swing and AWT

Swing and AWT are replaced by the JavaFX platform for developing rich GUI applications.

AWT

Swing

JavaFX

why teaching JavaFX

When Java was introduced, the GUI classes were bundled in a library known as the *Abstract Windows Toolkit (AWT)*. AWT is fine for developing simple graphical user interfaces, but not for developing comprehensive GUI projects. In addition, AWT is prone to platform-specific bugs. The AWT user-interface components were replaced by a more robust, versatile, and flexible library known as *Swing*. Swing components are painted directly on canvases using Java code. Swing components depend less on the target platform, and use less of the native GUI resources. Swing is designed for developing desktop GUI applications. It is now replaced by a completely new GUI platform known as *JavaFX*. JavaFX incorporates modern GUI technologies to enable you to develop rich GUI applications. In addition, JavaFX provides a multitouch support for touch-enabled devices such as tablets and smart phones. JavaFX has a built-in 2D, 3D, animation support, and video and audio playback. Using third-party software, you can develop JavaFX programs to be deployed on devices running iOS or Android.

This book teaches Java GUI programming using JavaFX for three reasons. First, JavaFX is much simpler to learn and use for new Java programmers. Second, JavaFX is a better pedagogical tool for demonstrating object-oriented programming than Swing. Third, Swing is essentially dead because it will not receive any further enhancement. JavaFX is the new GUI tool for developing cross-platform rich GUI applications on desktop computers and on handheld devices.



14.2.1 Explain the evolution of Java GUI technologies.

14.2.2 Explain why this book teaches Java GUI using JavaFX.



14.3 The Basic Structure of a JavaFX Program

The `javafx.application.Application` class defines the essential framework for writing JavaFX programs.



VideoNote

Getting started with JavaFX

We begin by writing a simple JavaFX program that illustrates the basic structure of a JavaFX program. Every JavaFX program is defined in a class that extends `javafx.application.Application`, as shown in Listing 14.1.

LISTING 14.1 MyJavaFX.java

```

1 import javafx.application.Application;
2 import javafx.scene.Scene;
3 import javafx.scene.control.Button;
4 import javafx.stage.Stage;
```

```

5
6  public class MyJavaFX extends Application {
7      @Override // Override the start method in the Application class
8      public void start(Stage primaryStage) {
9          // Create a scene and place a button in the scene
10         Button btOK = new Button("OK");
11         Scene scene = new Scene(btOK, 200, 250);
12         primaryStage.setTitle("MyJavaFX"); // Set the stage title
13         primaryStage.setScene(scene); // Place the scene in the stage
14         primaryStage.show(); // Display the stage
15     }
16
17     /**
18      * The main method is only needed for the IDE with limited
19      * JavaFX support. Not needed for running from the command line.
20      */
21     public static void main(String[] args) {
22         Application.launch(args);
23     }
24 }
```

extend Application
override start
create a button
create a scene
set stage title
set a scene
display stage
main method
launch application

You can test and run your program from a command window or from an IDE such as NetBeans or Eclipse. A sample run of the program is shown in Figure 14.1. Since JDK 11, JavaFX has become a separate module. Supplements II.F–H give the tips for running JavaFX programs from a command window, NetBeans, and Eclipse using JDK 11.

JavaFX on NetBeans and Eclipse



FIGURE 14.1 A simple JavaFX displays a button in the window.

The **Launch** method (line 22) is a static method defined in the **Application** class for launching a stand-alone JavaFX application. The **main** method (lines 21–23) is not needed if you run the program from the command line. It may be needed to launch a JavaFX program from an IDE with a limited JavaFX support. When you run a JavaFX application without a main method, JVM automatically invokes the **Launch** method to run the application.

Launch

The main class overrides the **start** method defined in **javafx.application.Application** (line 8). After a JavaFX application is launched, the JVM constructs an instance of the class using its **no-arg** constructor and invokes its **start** method. The **start** method normally places UI controls in a scene and displays the scene in a stage, as shown in Figure 14.2a.

construct application

Line 10 creates a **Button** object and places it in a **Scene** object (line 11). A **Scene** object can be created using the constructor **Scene(node, width, height)**. This constructor specifies the width and height of the scene and places the node in the scene.

start application

A **Stage** object is a window. A **Stage** object called *primary stage* is automatically created by the JVM when the application is launched. Line 13 sets the scene to the primary stage and line 14 displays the primary stage. JavaFX names the **Stage** and **Scene** classes using the analogy from the theater. You may think of stage as the platform to support scenes, and nodes as actors to perform in the scenes.

scene

You can create additional stages if needed. The JavaFX program in Listing 14.2 displays two stages, as shown in Figure 14.2b.

primary stage

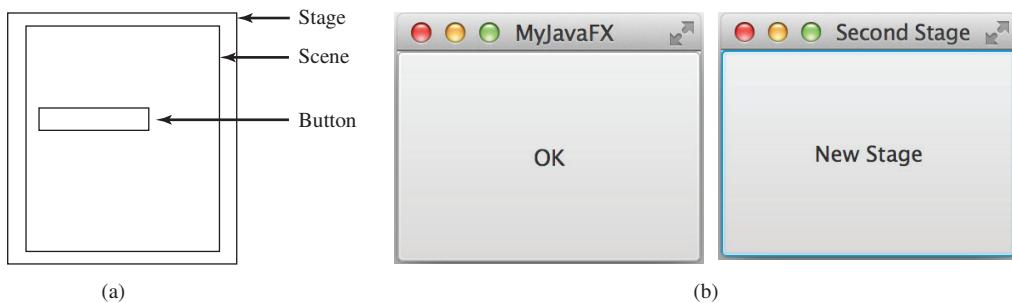


FIGURE 14.2 (a) Stage is a window for displaying a scene that contains nodes. (b) Multiple stages can be displayed in a JavaFX program.

LISTING 14.2 `MultipleStageDemo.java`

```

1 import javafx.application.Application;
2 import javafx.scene.Scene;
3 import javafx.scene.control.Button;
4 import javafx.stage.Stage;
5
6 public class MultipleStageDemo extends Application {
7     @Override // Override the start method in the Application class
8     public void start(Stage primaryStage) {
9         // Create a scene and place a button in the scene
10        Scene scene = new Scene(new Button("OK"), 200, 250);
11        primaryStage.setTitle("MyJavaFX"); // Set the stage title
12        primaryStage.setScene(scene); // Place the scene in the stage
13        primaryStage.show(); // Display the stage
14
15        Stage stage = new Stage(); // Create a new stage
16        stage.setTitle("Second Stage"); // Set the stage title
17        // Set a scene with a button in the stage
18        stage.setScene(new Scene(new Button("New Stage"), 200, 250));
19        stage.show(); // Display the stage
20    }
21 }
```

primary stage in start

display primary stage

create second stage

display second stage

main method omitted

prevent stage resizing



Note the main method is omitted in the listing since it is identical for every JavaFX application. From now on, we will not list the `main` method in our JavaFX source code for brevity.

By default, the user can resize the stage. To prevent the user from resizing the stage, invoke `stage.setResizable(false)`.

- 14.3.1** How do you define a JavaFX main class? What is the signature of the `start` method? What is a stage? What is a primary stage? Is a primary stage automatically created? How do you display a stage? Can you prevent the user from resizing the stage? Can you replace `Application.launch(args)` by `launch(args)` in line 22 in Listing 14.1?

- 14.3.2** Show the output of the following JavaFX program:

```

import javafx.application.Application;
import javafx.stage.Stage;

public class Test extends Application {
    public Test() {
        System.out.println("Test constructor is invoked");
    }
}
```

```

@Override // Override the start method in the Application class
public void start(Stage primaryStage) {
    System.out.println("start method is invoked");
}

public static void main(String[] args) {
    System.out.println("launch application");
    Application.launch(args);
}
}

```

14.4 Panes, Groups, UI Controls, and Shapes

Panes, Groups, UI controls, and shapes are subtypes of Node.

When you run MyJavaFX in Listing 14.1, the window is displayed as shown in Figure 14.1. The button is always centered in the scene and occupies the entire window no matter how you resize it. You can fix the problem by setting the position and size properties of a button. However, a better approach is to use container classes, called *panes*, for automatically laying out the nodes in a desired location and size. You place nodes inside a pane then place the pane into a scene. A *node* is a visual component such as a shape, an image view, a UI control, a group, or a pane. A *shape* refers to a text, line, circle, ellipse, rectangle, arc, polygon, polyline, and so on. A *UI control* refers to a label, button, check box, radio button, text field, text area, and so on. A *group* is a container that groups a collection of nodes. You can apply transformations or effects to a group, which automatically apply to all the children in the group. A scene can be displayed in a stage, as shown in Figure 14.3a. The relationship among **Stage**, **Scene**, **Node**, **Control**, **Group**, and **Pane** is illustrated in the UML diagram, as shown in Figure 14.3b. Note a **Scene** can contain a **Control**, **Group**, or a **Pane**, but not a **Shape** or an **ImageView**. A **Pane** or a **Group** can contain any subtype of **Node**. You can create a **Scene** using the constructor **Scene(Parent, width, height)** or **Scene(Parent)**. The dimension of the scene is automatically decided in the latter constructor. Every subclass of **Node** has a no-arg constructor for creating a default node.

Listing 14.3 gives a program that places a button in a pane, as shown in Figure 14.4.



pane
node
shape
UI control

group

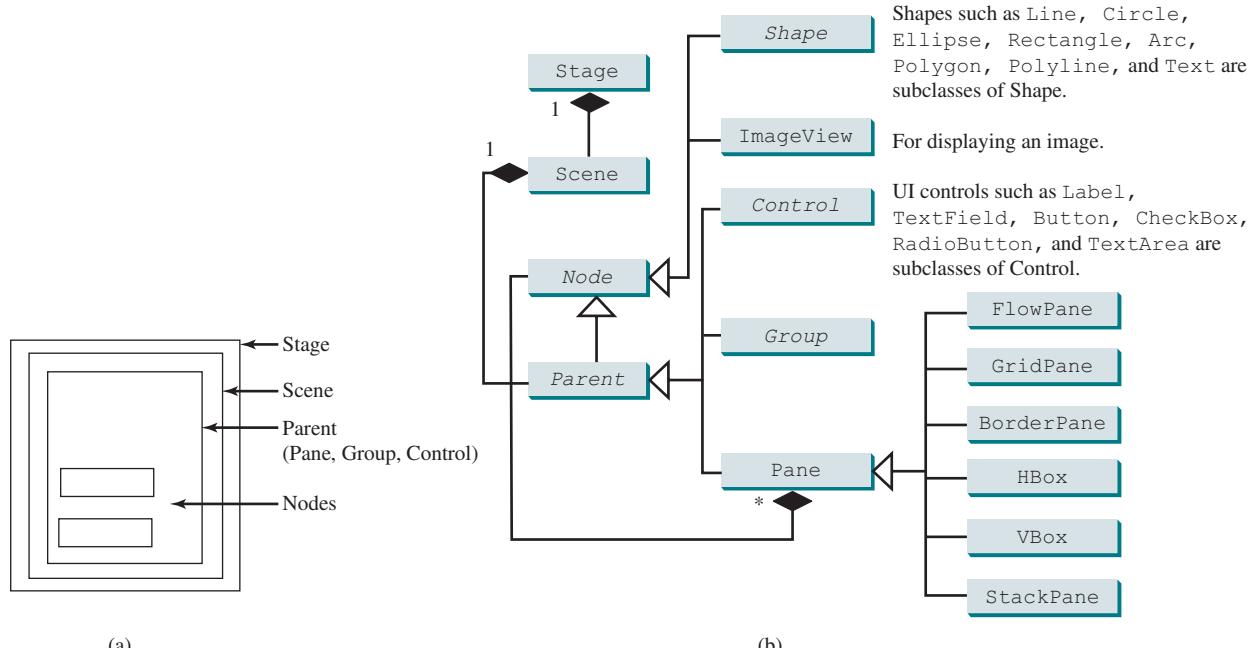


FIGURE 14.3 (a) Panes and groups are used to hold nodes. (b) Nodes can be shapes, image views, UI controls, groups, and panes.

LISTING 14.3 ButtonInPane.java

create a pane
add a button
add pane to scene

display stage

main method omitted

```

1 import javafx.application.Application;
2 import javafx.scene.Scene;
3 import javafx.scene.control.Button;
4 import javafx.stage.Stage;
5 import javafx.scene.layout.StackPane;
6
7 public class ButtonInPane extends Application {
8     @Override // Override the start method in the Application class
9     public void start(Stage primaryStage) {
10         // Create a scene and place a button in the scene
11         StackPane pane = new StackPane();
12         pane.getChildren().add(new Button("OK"));
13         Scene scene = new Scene(pane, 200, 50);
14         primaryStage.setTitle("Button in a pane"); // Set the stage title
15         primaryStage.setScene(scene); // Place the scene in the stage
16         primaryStage.show(); // Display the stage
17     }
18 }
```

**FIGURE 14.4** A button is placed in the center of the pane.

ObservableList

The program creates a **StackPane** (line 11) and adds a button as a child of the pane (line 12). The **getChildren()** method returns an instance of **javafx.collections.ObservableList**. **ObservableList** behaves very much like an **ArrayList** for storing a collection of elements. Invoking **add(e)** adds an element to the list. The **StackPane** places the nodes in the center of the pane on top of each other. Here, there is only one node in the pane. The **StackPane** respects a node's preferred size. Therefore, you see the button displayed in its preferred size.

Along with many other constructors, each pane and group have a no-arg constructor, and also a constructor that adds one or more children to the pane or group. Thus, the code in lines 11 and 12 can be replaced using one statement:

```
StackPane pane = new StackPane(new Button("OK"));
```

Listing 14.4 gives an example that displays a circle in the center of the pane, as shown in Figure 14.5a.

LISTING 14.4 ShowCircle.java

```

1 import javafx.application.Application;
2 import javafx.scene.Scene;
3 import javafx.scene.layout.Pane;
4 import javafx.scene.paint.Color;
5 import javafx.scene.shape.Circle;
6 import javafx.stage.Stage;
7
8 public class ShowCircle extends Application {
9     @Override // Override the start method in the Application class
10    public void start(Stage primaryStage) {
```

```

11  // Create a circle and set its properties
12  Circle circle = new Circle();
13  circle.setCenterX(100);
14  circle.setCenterY(100);
15  circle.setRadius(50);
16  circle.setStroke(Color.BLACK);
17  circle.setFill(Color.WHITE);
18
19  // Create a pane to hold the circle
20  Pane pane = new Pane();
21  pane.getChildren().add(circle);
22
23  // Create a scene and place it in the stage
24  Scene scene = new Scene(pane, 200, 200);
25  primaryStage.setTitle("ShowCircle"); // Set the stage title
26  primaryStage.setScene(scene); // Place the scene in the stage
27  primaryStage.show(); // Display the stage
28 }
29 }
```

create a circle
set circle properties

create a pane
add circle to pane

add pane to scene

display stage

main method omitted

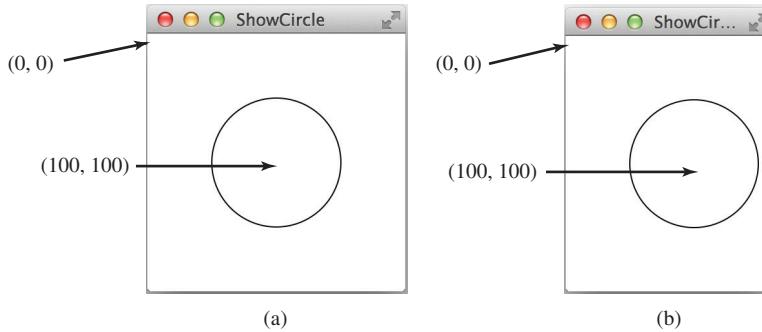


FIGURE 14.5 (a) A circle is displayed in the center of the scene. (b) The circle is not centered after the window is resized.

The program creates a **Circle** (line 12) and sets its center at (100, 100) (lines 13 and 14), which is also the center for the scene, since the scene is created with the width and height of 200 (line 24). The radius of the circle is set to 50 (line 15). Note the measurement units for graphics in Java are all in *pixels*.

pixels

The stroke color (i.e., the color to draw the circle) is set to black (line 16). The fill color (i.e., the color to fill the circle) is set to white (line 17). You may set the color to **null** to specify that no color is set.

set color

The program creates a **Pane** (line 20) and places the circle in the pane (line 21). Note the coordinates of the upper-left corner of the pane is **(0, 0)** in the Java coordinate system, as shown in Figure 14.6a, as opposed to the conventional coordinate system, where **(0, 0)** is at the center of the window, as shown in Figure 14.6b. The *x*-coordinate increases from left to right, and the *y*-coordinate increases downward in the Java coordinate system.

The pane is placed in the scene (line 24) and the scene is set in the stage (line 26). The circle is displayed in the center of the stage, as shown in Figure 14.5a. However, if you resize the window, the circle is not centered, as shown in Figure 14.5b. In order to display the circle centered as the window resizes, the *x*- and *y*-coordinates of the circle center need to be reset to the center of the pane. This can be done by using property binding, introduced in the next section.

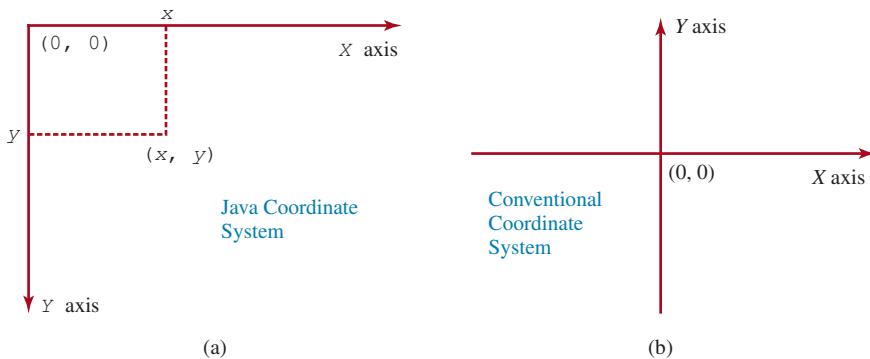


FIGURE 14.6 The Java coordinate system is measured in pixels, with **(0, 0)** at its upper-left corner.



- 14.4.1** How do you create a **Scene** object? How do you set a scene in a stage? How do you place a circle into a scene?
- 14.4.2** What is a pane? What is a node? How do you place a node in a pane? Can you directly place a **Shape** or an **ImageView** into a **Scene**? Can you directly place a **Control** or a **Pane** into a **Scene**?
- 14.4.3** How do you create a **Circle**? How do you set its center location and radius? How do you set its stroke color and fill color?
- 14.4.4** How do you replace the code in lines 20 and 21 in Listing 14.4 using one statement?

14.5 Property Binding



You can bind a target object to a source object. A change in the source object will be automatically reflected in the target object.

target object
source object
binding object
binding property
bindable object
observable object

JavaFX introduces a new concept called *property binding* that enables a *target object* to be bound to a *source object*. If the value in the source object changes, the target object is also automatically changed. The target object is called a *binding object* or a *binding property*, and the source object is called a *bindable object* or *observable object*. As discussed in Listing 14.4, ShowCircle.java, the circle is not centered after the window is resized. In order to display the circle centered as the window resizes, the x- and y-coordinates of the circle center need to be reset to the center of the pane. This can be done by binding the **centerX** with pane's **width/2** and **centerY** with pane's **height/2**, as given in lines 16–17 Listing 14.5.



VideoNote

Understand property binding

LISTING 14.5 ShowCircleCentered.java

```

1 import javafx.application.Application;
2 import javafx.scene.Scene;
3 import javafx.scene.layout.Pane;
4 import javafx.scene.paint.Color;
5 import javafx.scene.shape.Circle;
6 import javafx.stage.Stage;
7
8 public class ShowCircleCentered extends Application {
9     @Override // Override the start method in the Application class
10    public void start(Stage primaryStage) {

```

```

11 // Create a pane to hold the circle
12 Pane pane = new Pane();
13
14 // Create a circle and set its properties
15 Circle circle = new Circle();
16 circle.centerXProperty().bind(pane.widthProperty().divide(2));
17 circle.centerYProperty().bind(pane.heightProperty().divide(2));
18 circle.setRadius(50);
19 circle.setStroke(Color.BLACK);
20 circle.setFill(Color.WHITE);
21 pane.getChildren().add(circle); // Add circle to the pane
22
23 // Create a scene and place it in the stage
24 Scene scene = new Scene(pane, 200, 200);
25 primaryStage.setTitle("ShowCircleCentered"); // Set the stage title
26 primaryStage.setScene(scene); // Place the scene in the stage
27 primaryStage.show(); // Display the stage
28 }
29 }
```

create a pane
create a circle
bind properties
add circle to pane
add pane to scene
display stage
main method omitted

The **Circle** class has the **centerX** property for representing the *x*-coordinate of the circle center. This property like many properties in JavaFX classes can be used both as target and source in a property binding. A binding property is an object that can be bound to a source object. A target listens to the changes in the source and automatically updates itself once a change is made in the source. A target binds with a source using the **bind** method as follows:

```
target.bind(source);
```

The **bind** method is defined in the **javafx.beans.property.Property** interface. A binding property is an instance of **javafx.beans.property.Property**. An observable source object is an instance of the **javafx.beans.value.ObservableValue** interface. An **ObservableValue** is an entity that wraps a value and allows to observe the value for changes.

A binding property is an object. JavaFX defines binding properties for primitive types and strings. For a **double/float/long/int/boolean** value, its binding property type is **DoubleProperty/FloatProperty/LongProperty/IntegerProperty/BooleanProperty**, respectively. For a string, its binding property type is **StringProperty**. These properties are also subtypes of **ObservableValue**. Therefore, they can be used as both source and target in a binding.

By convention, each binding property (e.g., **centerX**) in a JavaFX class (e.g., **Circle**) has a getter (e.g., **getCenterX()**) and setter (e.g., **setCenterX(double)**) method for returning and setting the property's value. It also has a getter method for returning the property itself. The naming convention for this method is the property name followed by the word **Property**. For example, the property getter method for **centerX** is **centerXProperty()**. We call the **getCenterX()** method as the *value getter method*, the **setCenterX(double)** method as the *value setter method*, and **centerXProperty()** as the *property getter method*. Note **getCenterX()** returns a **double** value, and **centerXProperty()** returns an object of the **DoubleProperty** type. Figure 14.7a shows the convention for defining a binding property in a class, and Figure 14.7b shows a concrete example in which **centerX** is a binding property of the type **DoubleProperty**.

The program in Listing 14.5 is the same as in Listing 14.4 except that it binds **circle**'s **centerX** and **centerY** properties to half of **pane**'s width and height (lines 16 and 17). Note **circle.centerXProperty()** returns **centerX** and **pane.widthProperty()** returns

the **Property** interface

the **ObservableValue** interface

common binding properties

common **ObservableValue** objects

value getter method

value setter method

property getter method

```

public class SomeClassName {
    private PropertyType x;

    /** Value getter method */
    public propertyValueType getX() { ... }

    /** Value setter method */
    public void setX(propertyValueType value) { ... }

    /** Property getter method */
    public PropertyType xProperty() { ... }
}

```

(a) x is a binding property

```

public class Circle {
    private DoubleProperty centerX;

    /** Value getter method */
    public double getCenterX() { ... }

    /** Value setter method */
    public void setCenterX(double value) { ... }

    /** Property getter method */
    public DoubleProperty centerXProperty() { ... }
}

```

(b) centerX is binding property in the Circle class

FIGURE 14.7 A binding property has a value getter method, setter method, and property getter method.

width. Both **centerX** and **width** are binding properties of the **DoubleProperty** type. The numeric binding property classes such as **DoubleProperty** and **IntegerProperty** contain the **add**, **subtract**, **multiply**, and **divide** methods for adding, subtracting, multiplying, and dividing a value in a binding property and returning a new observable property. Therefore, **pane.widthProperty().divide(2)** returns a new observable property that represents half of the **pane**'s width. The statement

```
circle.centerXProperty().bind(pane.widthProperty().divide(2));
```

is the same as

```

DoubleProperty centerX = circle.centerXProperty();
DoubleProperty width = pane.widthProperty();
centerX.bind(width.divide(2));

```

Since **centerX** is bound to **width.divide(2)**, when **pane**'s width is changed, **centerX** automatically updates itself to match **pane**'s width / 2.

Listing 14.6 gives another example that demonstrates bindings.

LISTING 14.6 BindingDemo.java

create a DoubleProperty
create a DoubleProperty
bind property

set a new source value

```

1 import javafx.beans.property.DoubleProperty;
2 import javafx.beans.property.SimpleDoubleProperty;
3
4 public class BindingDemo {
5     public static void main(String[] args) {
6         DoubleProperty d1 = new SimpleDoubleProperty(1);
7         DoubleProperty d2 = new SimpleDoubleProperty(2);
8         d1.bind(d2);
9         System.out.println("d1 is " + d1.getValue()
10            + " and d2 is " + d2.getValue());
11         d2.setValue(70.2);
12         System.out.println("d1 is " + d1.getValue()
13            + " and d2 is " + d2.getValue());
14     }
15 }

```



```
d1 is 2.0 and d2 is 2.0
d1 is 70.2 and d2 is 70.2
```

The program creates an instance of `DoubleProperty` using `SimpleDoubleProperty(1)` (line 6). Note that `DoubleProperty`, `FloatProperty`, `LongProperty`, `IntegerProperty`, and `BooleanProperty` are abstract classes. Their concrete subclasses `SimpleDoubleProperty`, `SimpleFloatProperty`, `SimpleLongProperty`, `SimpleIntegerProperty`, and `SimpleBooleanProperty` are used to create instances of these properties. These classes are very much like wrapper classes `Double`, `Float`, `Long`, `Integer`, and `Boolean` with additional features for property binding.

The program binds `d1` with `d2` (line 8). Now the values in `d1` and `d2` are the same. After setting `d2` to `70.2` (line 11), `d1` also becomes `70.2` (line 13).

The binding demonstrated in this example is known as *unidirectional binding*. Occasionally, it is useful to synchronize two properties so a change in one property is reflected in another object, and vice versa. This is called a *bidirectional binding*. If the target and source are both binding properties and observable properties, they can be bound bidirectionally using the `bindBidirectional` method.

unidirectional binding

bidirectional binding



- 14.5.1** What is a binding property? What interface defines a binding property? What interface defines a source object? What are the binding object types for `int`, `long`, `float`, `double`, and `boolean`? Are `Integer` and `Double` binding properties? Can `Integer` and `Double` be used as source objects in a binding?
- 14.5.2** Following the JavaFX binding property naming convention, for a binding property named `age` of the `IntegerProperty` type, what is its value getter method, value setter method, and property getter method?
- 14.5.3** Can you create an object of `IntegerProperty` using `new IntegerProperty(3)`? If not, what is the correct way to create it? What will be the output if line 8 is replaced by `d1.bind(d2.multiply(2))` in Listing 14.6? What will be the output if line 8 is replaced by `d1.bind(d2.add(2))` in Listing 14.6?
- 14.5.4** What is unidirectional binding and what is bidirectional binding? Are all binding properties capable of bidirectional binding? Write a statement to bind property `d1` with property `d2` bidirectionally.

14.6 Common Properties and Methods for Nodes

The Node class defines many properties and methods that are common to all nodes.

Nodes share many common properties. This section introduces two such properties: `style` and `rotate`.



JavaFX CSS

JavaFX style properties are similar to cascading style sheets (CSS) used to specify the styles for HTML elements in a Web page. Therefore, the style properties in JavaFX are called *JavaFX CSS*. In JavaFX, a style property is defined with a prefix `-fx-`. Each node has its own style properties. You can find these properties at docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html. For information on HTML and CSS, see Supplements V.A and V.B. If you are not familiar with HTML and CSS, you can still use JavaFX CSS.

The syntax for setting a style is `styleName:value`. Multiple style properties for a node can be set together separated by semicolon (`;`). For example, the following statement:

```
circle.setStyle("-fx-stroke: black; -fx-fill: red;");
```

setStyle

sets two JavaFX CSS properties for a circle. This statement is equivalent to the following two statements:

```
circle.setStroke(Color.BLACK);
circle.setFill(Color.RED);
```

If an incorrect JavaFX CSS is used, your program will still compile and run, but the style will be ignored.

The `rotate` property enables you to specify an angle in degrees for rotating a node from its center. If the degree is positive, the rotation is performed clockwise; otherwise, it is performed counterclockwise. For example, the following code rotates a button 80 degrees:

```
button.setRotate(80);
```

Listing 14.7 gives an example that creates a button, sets its style, and adds it to a pane. It then rotates the pane 45 degrees and sets its style with border color red and background color light gray, as shown in Figure 14.8.

LISTING 14.7 NodeStyleRotateDemo.java

```

1 import javafx.application.Application;
2 import javafx.scene.Scene;
3 import javafx.scene.control.Button;
4 import javafx.stage.Stage;
5 import javafx.scene.layout.StackPane;
6
7 public class NodeStyleRotateDemo extends Application {
8     @Override // Override the start method in the Application class
9     public void start(Stage primaryStage) {
10         // Create a scene and place a button in the scene
11         StackPane pane = new StackPane();
12         Button btOK = new Button("OK");
13         btOK.setStyle("-fx-border-color: blue;");
14         pane.getChildren().add(btOK);
15
16         pane.setRotate(45);
17         pane.setStyle(
18             "-fx-border-color: red; -fx-background-color: lightgray;");
19
20         Scene scene = new Scene(pane, 200, 250);
21         primaryStage.setTitle("NodeStyleRotateDemo"); // Set the stage title
22         primaryStage.setScene(scene); // Place the scene in the stage
23         primaryStage.show(); // Display the stage
24     }
25 }
```

rotate the pane
set style for pane

main method omitted

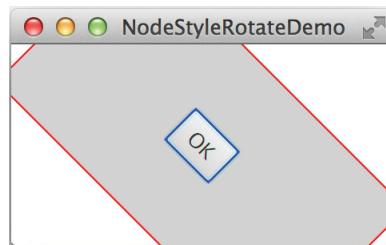


FIGURE 14.8 A pane's style is set and the pane is rotated 45 degrees.

As seen in Figure 14.8, rotating a pane causes all its containing nodes rotated as well.

The `Node` class contains many useful methods that can be applied to all nodes. For example, you can use the `contains(double x, double y)` method to test whether a point (x, y) is inside the boundary of a node and use the `setScaleX(double scale)` and `setScaleY(double scale)` methods to scale a node.

contains method
setScaleX method
setScaleY method



- 14.6.1** How do you set a style of a node with border color red? Modify the code to set the text color for the button to red.
- 14.6.2** Can you rotate a pane, a text, or a button? Modify the code to rotate the button 15 degrees counterclockwise? How do you test if a point is inside a node? How do you scale up or down a node?

14.7 The Color Class

The **Color** class can be used to create colors.

JavaFX defines the abstract **Paint** class for painting a node. The **javafx.scene.paint.Color** is a concrete subclass of **Paint**, which is used to encapsulate colors, as shown in Figure 14.9.

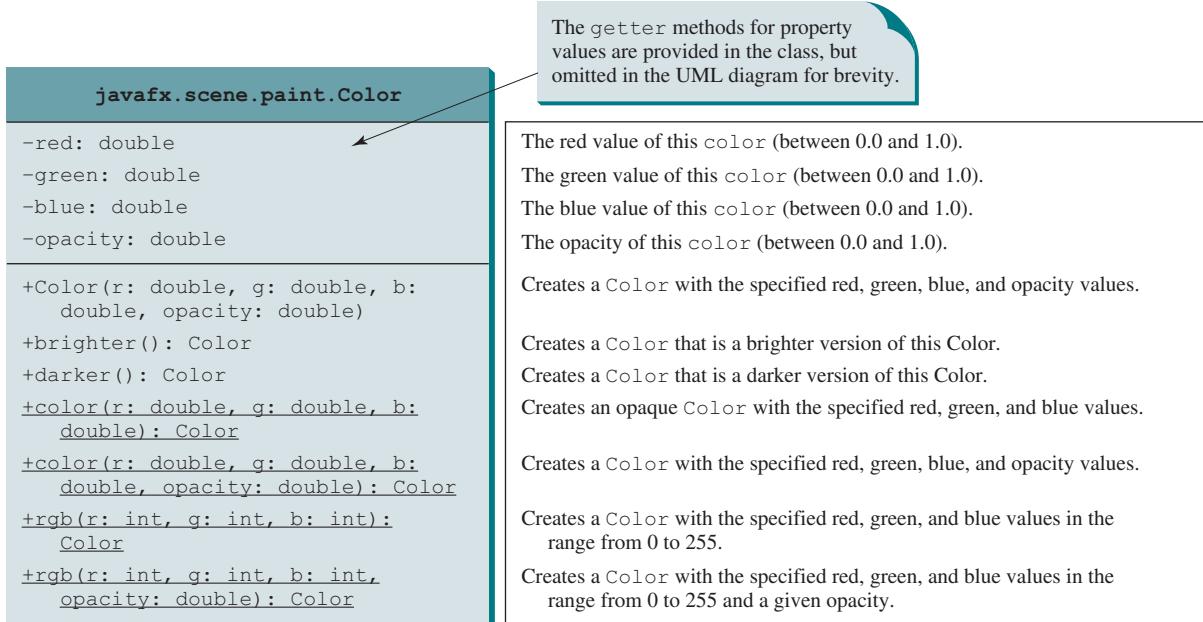


FIGURE 14.9 **Color** encapsulates information about colors.

A **color** instance can be constructed using the following constructor:

```
public Color(double r, double g, double b, double opacity);
```

in which **r**, **g**, and **b** specify a color by its red, green, and blue components with values in the range from **0.0** (darkest shade) to **1.0** (lightest shade). The **opacity** value defines the transparency of a color within the range from **0.0** (completely transparent) to **1.0** (completely opaque). This is known as the RGBA model, where RGBA stands for red, green, blue, and alpha. The alpha value indicates the opacity. For example,

```
Color color = new Color(0.25, 0.14, 0.333, 0.51);
```

For an interactive demo, see liveexample.pearsoncmg.com/dsanimation/FigureSection14_7.html.

RGBA model

The **Color** class is immutable. Once a **Color** object is created, its properties cannot be changed. The **brighter()** method returns a new **Color** with a larger red, green, and blue values, and the **darker()** method returns a new **Color** with a smaller red, green, and blue values. The **opacity** value is the same as in the original **Color** object.

You can also create a **Color** object using the static methods **color(r, g, b)**, **color(r, g, b, opacity)**, **rgb(r, g, b)**, and **rgb(r, g, b, opacity)**.

Alternatively, you can use one of the many standard colors such as **BEIGE**, **BLACK**, **BLUE**, **BROWN**, **CYAN**, **DARKGRAY**, **GOLD**, **GRAY**, **GREEN**, **LIGHTGRAY**, **MAGENTA**, **NAVY**, **ORANGE**, **PINK**, **RED**, **SILVER**, **WHITE**, and **YELLOW** defined as constants in the **Color** class. The following code, for instance, sets the fill color of a circle to red:

```
circle.setFill(Color.RED);
```



- 14.7.1** How do you create a color? What is wrong about creating a `Color` using `new Color(1.2, 2.3, 3.5, 4)`? Which of two colors is darker, `new Color(0, 0, 0, 1)` or `new Color(1, 1, 1, 1)`? Does invoking `c.darker()` change the color value in `c`?
- 14.7.2** How do you create a `Color` object with a random color?
- 14.7.3** How do you set a circle object `c` with blue fill color using the `setFill` method and the `setStyle` method?

14.8 The Font Class



A `Font` describes font name, weight, and size.

You can set fonts for rendering the text. The `javafx.scene.text.Font` class is used to create fonts, as shown in Figure 14.10.

A `Font` instance can be constructed using its constructors or using its static methods. A `Font` is defined by its name, weight, posture, and size. Times New Roman, Courier, and Arial are examples of font names. You can obtain a list of available font family names by invoking the static `getFontNames()` method. This method returns `List<String>`. `List` is an interface that defines common methods for lists. `ArrayList`, introduced in Section 11.11, is a concrete class that implements `List`. There are two font postures defined as constants in the `FontPosture` class: `FontPosture.ITALIC` and `FontPosture.REGULAR`.

```
Font font1 = new Font("SansSerif", 16);
Font font2 = Font.font("Times New Roman", FontWeight.BOLD,
    FontPosture.ITALIC, 12);
```

Listing 14.8 gives a program that displays a label using the font (Times New Roman, bold, italic, and size 20), as shown in Figure 14.11.

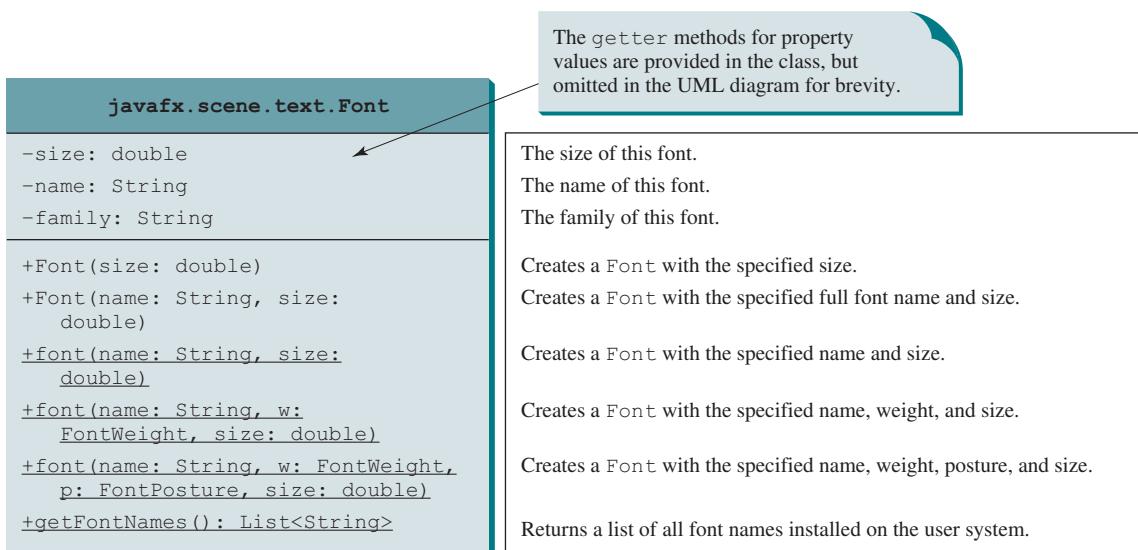


FIGURE 14.10 `Font` encapsulates information about fonts.

LISTING 14.8 FontDemo.java

```
1 import javafx.application.Application;
2 import javafx.scene.Scene;
3 import javafx.scene.layout.*;
4 import javafx.scene.paint.Color;
```

```

5 import javafx.scene.shape.Circle;
6 import javafx.scene.text.*;
7 import javafx.scene.control.*;
8 import javafx.stage.Stage;
9
10 public class FontDemo extends Application {
11     @Override // Override the start method in the Application class
12     public void start(Stage primaryStage) {
13         // Create a pane to hold the circle
14         Pane pane = new StackPane();                                create a StackPane
15
16         // Create a circle and set its properties
17         Circle circle = new Circle();                                create a Circle
18         circle.setRadius(50);
19         circle.setStroke(Color.BLACK);
20         circle.setFill(new Color(0.5, 0.5, 0.5, 0.1));          create a Color
21         pane.getChildren().add(circle); // Add circle to the pane      add circle to the pane
22
23         // Create a label and set its properties
24         Label label = new Label("JavaFX");                          create a label
25         label.setFont(Font.font("Times New Roman",           create a font
26             FontWeight.BOLD, FontPosture.ITALIC, 20));
27         pane.getChildren().add(label);                                add label to the pane
28
29         // Create a scene and place it in the stage
30         Scene scene = new Scene(pane);
31         primaryStage.setTitle("FontDemo"); // Set the stage title
32         primaryStage.setScene(scene); // Place the scene in the stage
33         primaryStage.show(); // Display the stage
34     }
35 }                                         main method omitted

```

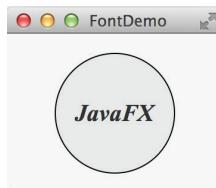


FIGURE 14.11 A label is on top of a circle displayed in the center of the scene.

The program creates a **StackPane** (line 14) and adds a circle and a label to it (lines 21 and 27). These two statements can be combined using the following one statement:

```
pane.getChildren().addAll(circle, label);
```

A **StackPane** places the nodes in the center and nodes are placed on top of each other. A custom color is created and set as a fill color for the circle (line 20). The program creates a label and sets a font (line 25) so that the text in the label is displayed in Times New Roman, bold, italic, and 20 pixels.

As you resize the window, the circle and label are displayed in the center of the window because the circle and label are placed in the stack pane. Stack pane automatically places nodes in the center of the pane.

A **Font** object is immutable. Once a **Font** object is created, its properties cannot be changed.

14.8.1 How do you create a **Font** object with font name **Courier**, size **20**, and weight **bold**?

14.8.2 How do you find all available fonts on your system?



14.9 The Image and ImageView Classes



The **Image** class represents a graphical image, and the **ImageView** class can be used to display an image.



VideoNote

Use Image and ImageView

The `javafx.scene.image.Image` class represents a graphical image and is used for loading an image from a specified filename or a URL. For example, `new Image("image/us.gif")` creates an **Image** object for the image file `us.gif` under the directory `image` in the Java class directory and `new Image("http://liveexample.pearsoncmg.com/book/image/us.gif")` creates an **Image** object for the image file in the URL on the Web.

The `javafx.scene.image.ImageView` is a node for displaying an image. An **ImageView** can be created from an **Image** object. For example, the following code creates an **ImageView** from an image file:

```
Image image = new Image("image/us.gif");
ImageView imageView = new ImageView(image);
```

Alternatively, you can create an **ImageView** directly from a file or a URL as follows:

```
ImageView imageView = new ImageView("image/us.gif");
```

The UML diagrams for the **Image** class and **ImageView** classes are illustrated in Figures 14.12 and 14.13.

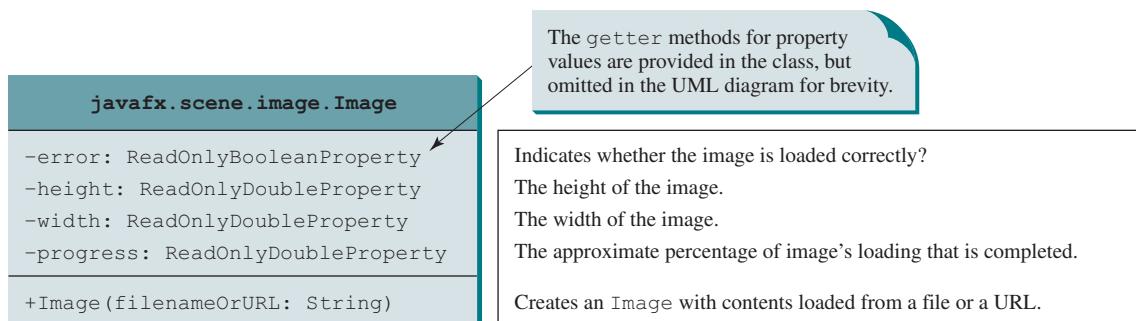


FIGURE 14.12 **Image** encapsulates information about images.

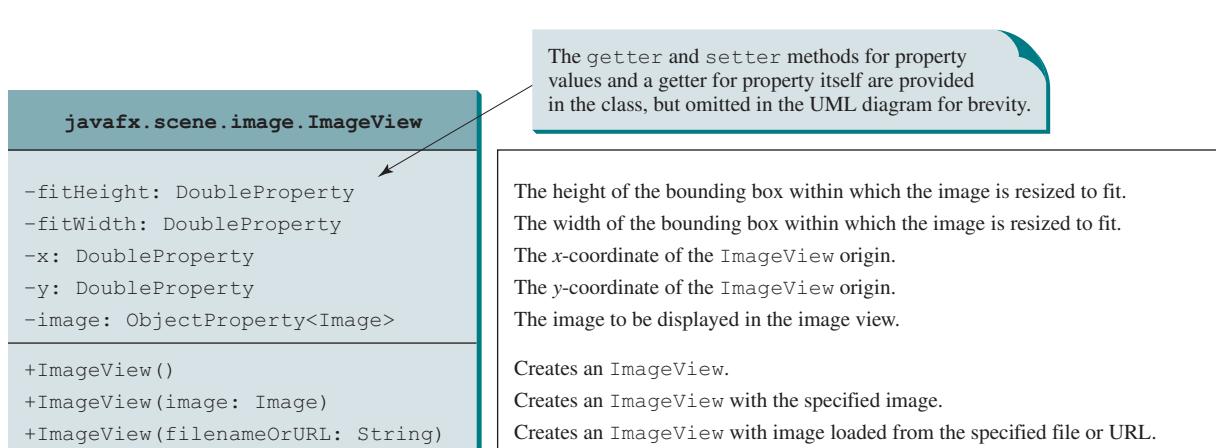


FIGURE 14.13 **ImageView** is a node for displaying an image.

Listing 14.9 displays an image in three image views, as shown in Figure 14.14.

LISTING 14.9 ShowImage.java

```

1 import javafx.application.Application;
2 import javafx.scene.Scene;
3 import javafx.scene.layout.HBox;
4 import javafx.scene.layout.Pane;
5 import javafx.geometry.Insets;
6 import javafx.stage.Stage;
7 import javafx.scene.image.Image;
8 import javafx.scene.image.ImageView;
9
10 public class ShowImage extends Application {
11     @Override // Override the start method in the Application class
12     public void start(Stage primaryStage) {
13         // Create a pane to hold the image views
14         Pane pane = new HBox(10);
15         pane.setPadding(new Insets(5, 5, 5, 5));
16         Image image = new Image("image/us.gif");
17         pane.getChildren().add(new ImageView(image));
18
19         ImageView imageView2 = new ImageView(image);
20         imageView2.setFitHeight(100);
21         imageView2.setFitWidth(100);
22         pane.getChildren().add(imageView2);
23
24         ImageView imageView3 = new ImageView(image);
25         imageView3.setRotate(90);
26         pane.getChildren().add(imageView3);
27
28         // Create a scene and place it in the stage
29         Scene scene = new Scene(pane);
30         primaryStage.setTitle("ShowImage"); // Set the stage title
31         primaryStage.setScene(scene); // Place the scene in the stage
32         primaryStage.show(); // Display the stage
33     }
34 }
```

main method omitted

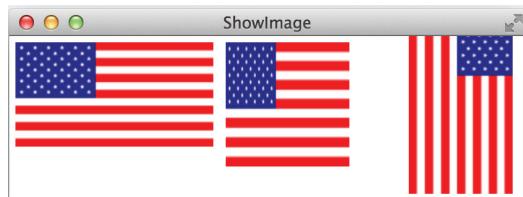
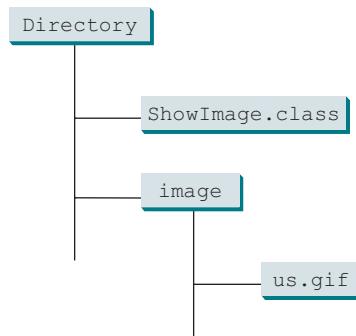


FIGURE 14.14 An image is displayed in three image views placed in a pane. *Source:* booka/Fotolia.

The program creates an **HBox** (line 14). An **HBox** is a pane that places all nodes horizontally in one row. The program creates an **Image**, then an **ImageView** for displaying the image, and places the **ImageView** in the **HBox** (line 17).

The program creates the second **ImageView** (line 19), sets its **fitHeight** and **fitWidth** properties (lines 20 and 21), and places the **ImageView** into the **HBox** (line 22). The program creates the third **ImageView** (line 24), rotates it 90 degrees (line 25), and places it into the **HBox** (line 26). The **setRotate** method is defined in the **Node** class and can be used for any node. Note an **Image** object can be shared by multiple nodes. In this case, it is shared by three **ImageView**. However, a node such as **ImageView** cannot be shared. You cannot place an **ImageView** multiple times into a pane or scene.

Note you must place the image file in the same directory as the class file, as shown in the following figure.



If you use the URL to locate the image file, the URL protocol `http://` must be present. Therefore, the following code is wrong:

```
new Image("liveexample.pearsoncmg.com/book/image/us.gif");
```

It must be replaced by

```
new Image("http://liveexample.pearsoncmg.com/book/image/us.gif");
```



- 14.9.1** How do you create an `Image` from a URL or a filename?
- 14.9.2** How do you create an `ImageView` from an `Image` or directly from a file or a URL?
- 14.9.3** Can you set an `Image` to multiple `ImageViews`? Can you display the same `ImageView` multiple times?



14.10 Layout Panes and Groups

JavaFX provides many types of panes for automatically laying out nodes in a desired location and size.



VideoNote

Use layout panes

Panes and groups are the containers for holding nodes. The `Group` class is often used to group nodes and to perform transformation and scale as a group. Panes and UI control objects are resizable, but group, shape, and text objects are not resizable. JavaFX provides many types of panes for organizing nodes in a container, as shown in Table 14.1. You have used the layout panes `Pane`, `StackPane`, and `HBox` in the preceding sections for containing nodes. This section introduces the panes in more details.

TABLE 14.1 Panes for Containing and Organizing Nodes

Class	Description
Pane	Base class for layout panes. It contains the <code>getChildren()</code> method for returning a list of nodes in the pane.
StackPane	Places the nodes on top of each other in the center of the pane.
FlowPane	Places the nodes row-by-row horizontally or column-by-column vertically.
GridPane	Places the nodes in the cells in a two-dimensional grid.
BorderPane	Places the nodes in the top, right, bottom, left, and center regions.
HBox	Places the nodes in a single row.
VBox	Places the nodes in a single column.

You have used the **Pane** in Listing 14.4, ShowCircle.java. A **Pane** is usually used as a canvas for displaying shapes. **Pane** is the base class for all specialized panes. You have used a specialized pane **StackPane** in Listing 14.3, ButtonInPane.java. Nodes are placed in the center of a **StackPane**. Each pane contains a list for holding nodes in the pane. This list is an instance of **ObservableList**, which can be obtained using pane's **getChildren()** method. You can use **add(node)** to add an element to the list and **addAll(node1, node2, ...)** to add a variable number of nodes.

ObservableList
getChildren()

14.10.1 FlowPane

FlowPane arranges the nodes in the pane horizontally from left to right, or vertically from top to bottom, in the order in which they were added. When one row or one column is filled, a new row or column is started. You can specify the way the nodes are placed horizontally or vertically using one of two constants: **Orientation.HORIZONTAL** or **Orientation.VERTICAL**. You can also specify the gap between the nodes in pixels. The class diagram for **FlowPane** is shown in Figure 14.15.

Data fields **alignment**, **orientation**, **hgap**, and **vgap** are binding properties. Recall that each binding property in JavaFX has a getter method (e.g., **getHgap()**) that returns its value, a setter method (e.g., **setHGap(double)**) for setting a value, and a getter method that returns the property itself (e.g., **hgapProperty()**). For a data field of **ObjectProperty<T>** type, the value getter method returns a value of type **T**, and the property getter method returns a property value of type **ObjectProperty<T>**.

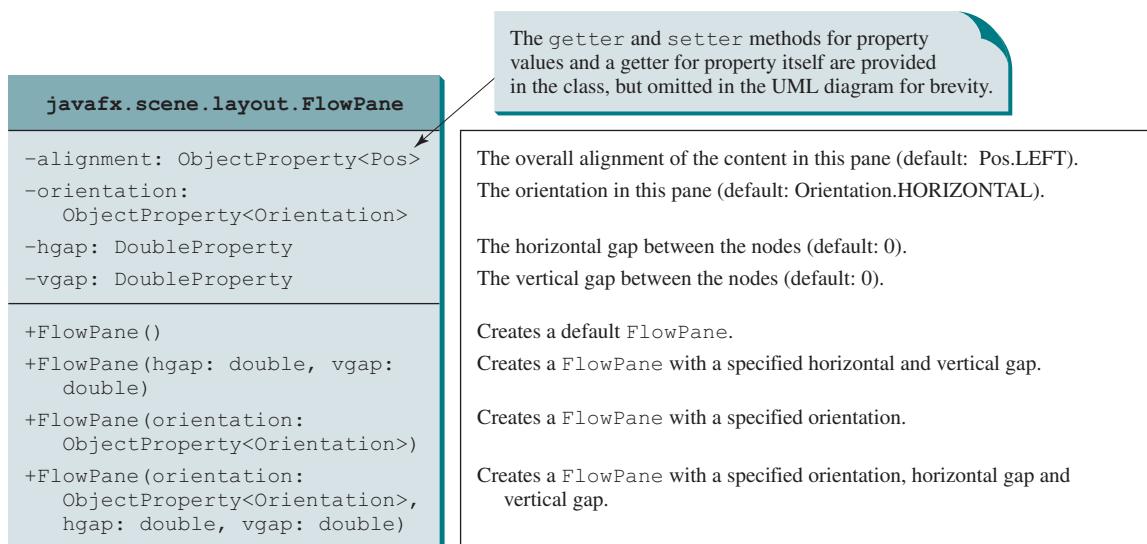


FIGURE 14.15 **FlowPane** lays out nodes row-by-row horizontally or column-by-column vertically.

Listing 14.10 gives a program that demonstrates **FlowPane**. The program adds labels and text fields to a **FlowPane**, as shown in Figure 14.16.

LISTING 14.10 ShowFlowPane.java

```

1 import javafx.application.Application;
2 import javafx.geometry.Insets;
3 import javafx.scene.Scene;
4 import javafx.scene.control.Label;
5 import javafx.scene.control.TextField;
6 import javafx.scene.layout.FlowPane;
7 import javafx.stage.Stage;

```

```

8
9  public class ShowFlowPane extends Application {
10    @Override // Override the start method in the Application class
11    public void start(Stage primaryStage) {
12      // Create a pane and set its properties
13      FlowPane pane = new FlowPane();
14      pane.setPadding(new Insets(11, 12, 13, 14));
15      pane.setHgap(5);
16      pane.setVgap(5);
17
18      // Place nodes in the pane
19      pane.getChildren().addAll(new Label("First Name:") ,
20          new TextField(), new Label("MI:"));
21      TextField tfMi = new TextField();
22      tfMi.setPrefColumnCount(1);
23      pane.getChildren().addAll(tfMi, new Label("Last Name:") ,
24          new TextField());
25
26      // Create a scene and place it in the stage
27      Scene scene = new Scene(pane, 200, 250);
28      primaryStage.setTitle("ShowFlowPane"); // Set the stage title
29      primaryStage.setScene(scene); // Place the scene in the stage
30      primaryStage.show(); // Display the stage
31    }
32  }

```

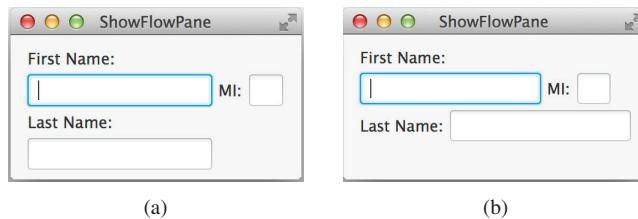


FIGURE 14.16 The nodes fill in the rows in the `FlowPane` one after another.

The program creates a `FlowPane` (line 13) and sets its `padding` property with an `Insets` object (line 14). An `Insets` object specifies the size of the border of a pane. The constructor `Insets(11, 12, 13, 14)` creates an `Insets` with the border sizes for top (11), right (12), bottom (13), and left (14) in pixels, as shown in Figure 14.17. You can also use the constructor `Insets(value)` to create an `Insets` with the same value for all four sides. The `hGap` and `vGap` properties are in lines 15 and 16 to specify the horizontal gap and vertical gap, respectively, between two nodes in the pane, as shown in Figure 14.17.

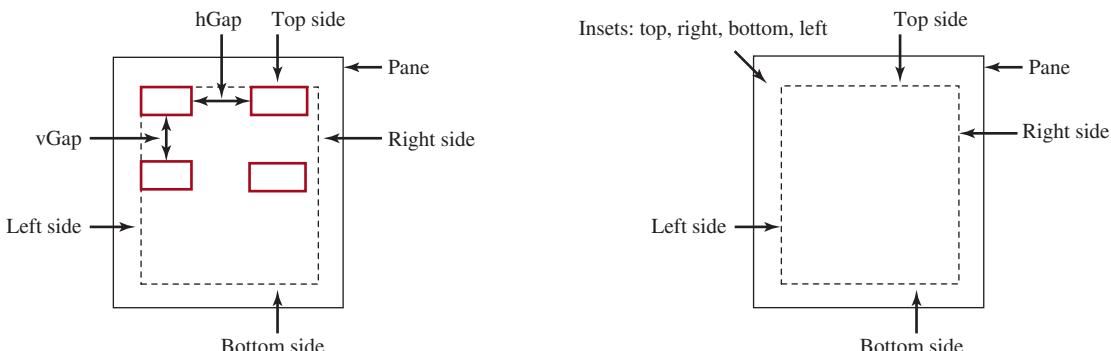


FIGURE 14.17 You can specify `hGap` and `vGap` between the nodes in a `FlowPane`.

Each **FlowPane** contains an object of **ObservableList** for holding the nodes. This list can be obtained using the **getChildren()** method (line 19). To add a node into a **FlowPane** is to add it to this list using the **add(node)** or **addAll(node1, node2, ...)** method. You can also remove a node from the list using the **remove(node)** method, or use the **removeAll()** method to remove all nodes from the pane. The program adds the labels and text fields into the pane (lines 19–24). Invoking **tfMi.setPrefColumnCount(1)** sets the preferred column count to 1 for the MI text field (line 22). The program declares an explicit reference **tfMi** for a **TextField** object for MI. The explicit reference is necessary because we need to reference the object directly to set its **prefColumnCount** property.

The program adds the pane to the scene (line 27), sets the scene in the stage (line 29), and displays the stage (line 30). Note if you resize the window, the nodes are automatically rearranged to fit in the pane. In Figure 14.16a, the first row has three nodes, but in Figure 14.16b, the first row has four nodes because the width has been increased.

Suppose you wish to add the object **tfMi** to a pane 10 times; will 10 text fields appear in the pane? No, a node such as a text field can be added to only one pane and once. Adding a node to a pane multiple times or to different panes will cause a runtime error.

14.10.2 GridPane

A **GridPane** arranges nodes in a grid (matrix) formation. The nodes are placed in the specified column and row indices. The class diagram for **GridPane** is shown in Figure 14.18.

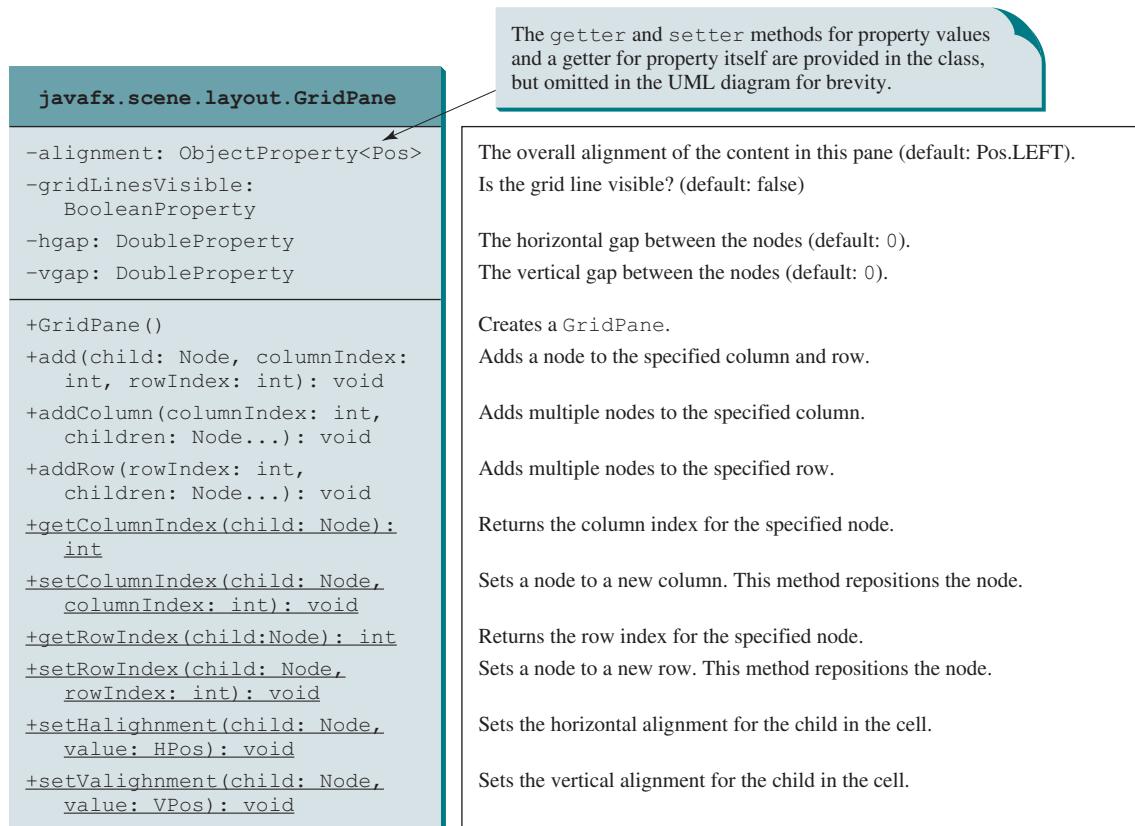


FIGURE 14.18 **GridPane** lays out nodes in the specified cell in a grid.

Listing 14.11 gives a program that demonstrates **GridPane**. The program is similar to the one in Listing 14.10, except that it adds three labels and three text fields, and a button to the specified location in a grid, as shown in Figure 14.19.

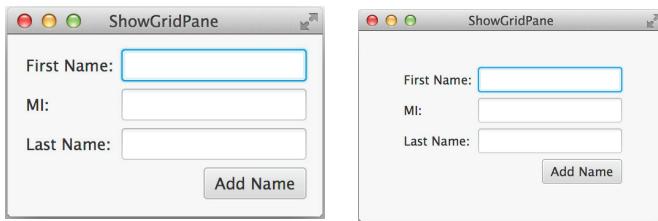


FIGURE 14.19 The **GridPane** places the nodes in a grid with a specified column and row indices.

LISTING 14.11 ShowGridPane.java

```

1 import javafx.application.Application;
2 import javafx.geometry.HPos;
3 import javafx.geometry.Insets;
4 import javafx.geometry.Pos;
5 import javafx.scene.Scene;
6 import javafx.scene.control.Button;
7 import javafx.scene.control.Label;
8 import javafx.scene.control.TextField;
9 import javafx.scene.layout.GridPane;
10 import javafx.stage.Stage;
11
12 public class ShowGridPane extends Application {
13     @Override // Override the start method in the Application class
14     public void start(Stage primaryStage) {
15         // Create a pane and set its properties
16         GridPane pane = new GridPane();
17         pane.setAlignment(Pos.CENTER);
18         pane.setPadding(new Insets(11.5, 12.5, 13.5, 14.5));
19         pane.setHgap(5.5);
20         pane.setVgap(5.5);
21
22         // Place nodes in the pane
23         pane.add(new Label("First Name:"), 0, 0);
24         pane.add(new TextField(), 1, 0);
25         pane.add(new Label("MI:"), 0, 1);
26         pane.add(new TextField(), 1, 1);
27         pane.add(new Label("Last Name:"), 0, 2);
28         pane.add(new TextField(), 1, 2);
29         Button btAdd = new Button("Add Name");
30         pane.add(btAdd, 1, 3);
31         GridPane.setAlignment(btAdd, HPos.RIGHT);
32
33         // Create a scene and place it in the stage
34         Scene scene = new Scene(pane);
35         primaryStage.setTitle("ShowGridPane"); // Set the stage title
36         primaryStage.setScene(scene); // Place the scene in the stage
37         primaryStage.show(); // Display the stage
38     }
39 }
```

create a grid pane
set properties

add label
add text field

add button
align button right

create a scene

display stage

main method omitted

The program creates a **GridPane** (line 16) and sets its properties (line 17–20). The alignment is set to the center position (line 17), which causes the nodes to be placed in the center of the grid pane. If you resize the window, you will see the nodes remained in the center of the grid pane.

The program adds the label in column **0** and row **0** (line 23). The column and row index starts from **0**. The **add** method places a node in the specified column and row. Not every cell in the grid needs to be filled. A button is placed in column 1 and row 3 (line 30), but there are no nodes placed in column 0 and row 3. To remove a node from a **GridPane**, use **pane.getChildren().remove(node)**. To remove all nodes, use **pane.getChildren().removeAll()**.

remove nodes

The program invokes the static **setHalignment** method to align the button right in the cell (line 31).

Note the scene size is not set (line 34). In this case, the scene size is automatically computed according to the sizes of the nodes placed inside the scene.

By default, the grid pane will resize rows and columns to the preferred sizes of its contents, even if the grid pane is resized larger than its preferred size. You may purposely set a large value for the preferred width and height of its contents by invoking the **setPrefWidth** and **setPrefHeight** methods, so the contents will be automatically stretched to fill in the grid pane when the grid pane is enlarged (see Programming Exercise 14.8).

remove nodes
setPrefWidth
setPrefHeight

14.10.3 BorderPane

A **BorderPane** can place nodes in five regions: top, bottom, left, right, and center, using the **setTop(node)**, **setBottom(node)**, **setLeft(node)**, **setRight(node)**, and **setCenter(node)** methods. The class diagram for **BorderPane** is shown in Figure 14.20.

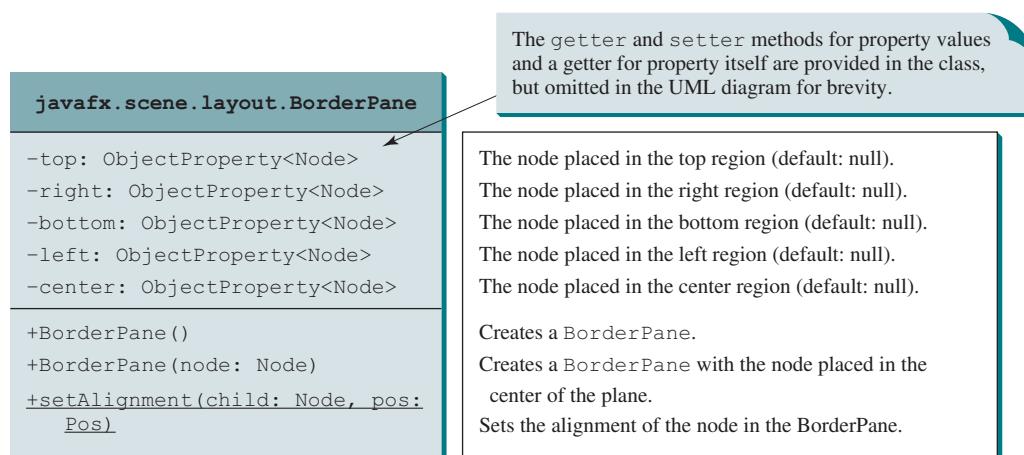


FIGURE 14.20 **BorderPane** places the nodes in top, bottom, left, right, and center regions.

Listing 14.12 gives a program that demonstrates **BorderPane**. The program places five buttons in the five regions of the pane, as shown in Figure 14.21.

LISTING 14.12 ShowBorderPane.java

```

1 import javafx.application.Application;
2 import javafx.geometry.Insets;
3 import javafx.scene.Scene;
4 import javafx.scene.control.Label;
5 import javafx.scene.layout.BorderPane;
6 import javafx.scene.layout.StackPane;
7 import javafx.stage.Stage;
8
9 public class ShowBorderPane extends Application {
10     @Override // Override the start method in the Application class

```

```

11  public void start(Stage primaryStage) {
12      // Create a border pane
13      BorderPane pane = new BorderPane();
14
15      // Place nodes in the pane
16      pane.setTop(new CustomPane("Top"));
17      pane.setRight(new CustomPane("Right"));
18      pane.setBottom(new CustomPane("Bottom"));
19      pane.setLeft(new CustomPane("Left"));
20      pane.setCenter(new CustomPane("Center"));
21
22      // Create a scene and place it in the stage
23      Scene scene = new Scene(pane);
24      primaryStage.setTitle("ShowBorderPane"); // Set the stage title
25      primaryStage.setScene(scene); // Place the scene in the stage
26      primaryStage.show(); // Display the stage
27  }
28 }
29
30 // Define a custom pane to hold a label in the center of the pane
31 class CustomPane extends StackPane {
32     public CustomPane(String title) {
33         getChildren().add(new Label(title));
34         setStyle("-fx-border-color: red");
35         setPadding(new Insets(11.5, 12.5, 13.5, 14.5));
36     }
37 }
```

create a border pane
add to top
add to right
add to bottom
add to left
add to center
define a custom pane
add a label to pane
set style
set padding
main method omitted

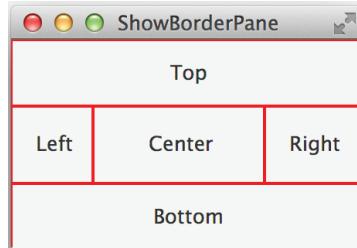


FIGURE 14.21 The **BorderPane** places the nodes in five regions of the pane.

The program defines **CustomPane** that extends **StackPane** (line 31). The constructor of **CustomPane** adds a label with the specified title (line 33), sets a style for the border color, and sets a padding using insets (line 35).

The program creates a **BorderPane** (line 13) and places five instances of **CustomPane** into five regions of the border pane (lines 16–20). Note a pane is a node. Therefore, a pane can be added into another pane. To remove a node from the top region, invoke **setTop(null)**. If a region is not occupied, no space will be allocated for this region.

14.10.4 HBox and VBox

An **HBox** lays out its children in a single horizontal row. A **VBox** lays out its children in a single vertical column. Recall that a **FlowPane** can lay out its children in multiple rows or multiple columns, but an **HBox** or a **VBox** can lay out children only in one row or one column. The class diagrams for **HBox** and **VBox** are shown in Figures 14.22 and 14.23.

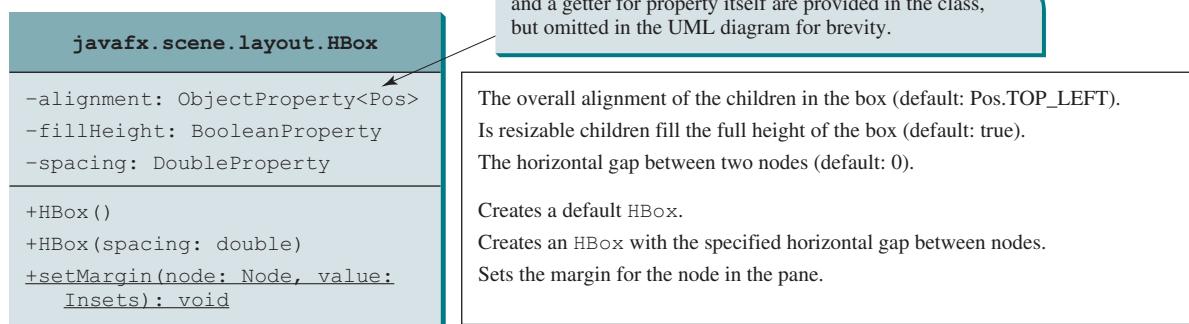


FIGURE 14.22 **HBox** places the nodes in one row.

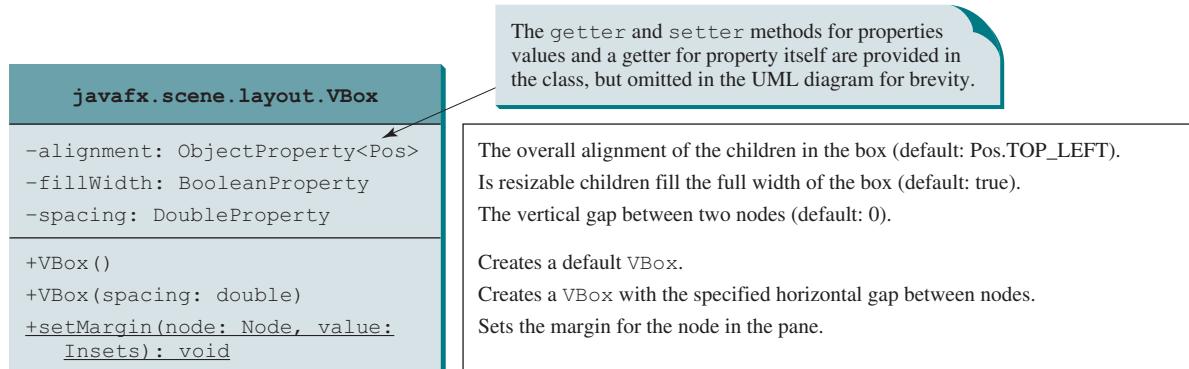


FIGURE 14.23 **VBox** places the nodes in one column.

Listing 14.13 gives a program that demonstrates **HBox** and **VBox**. The program places two buttons and an image view in an **HBox** and five labels in a **VBox**, as shown in Figure 14.24.

LISTING 14.13 ShowHBoxVBox.java

```

1 import javafx.application.Application;
2 import javafx.geometry.Insets;
3 import javafx.scene.Scene;
4 import javafx.scene.control.Button;
5 import javafx.scene.control.Label;
6 import javafx.scene.layout.BorderPane;
7 import javafx.scene.layout.HBox;
8 import javafx.scene.layout.VBox;
9 import javafx.stage.Stage;
10 import javafx.scene.image.Image;
11 import javafx.scene.image.ImageView;
12
13 public class ShowHBoxVBox extends Application {
14     @Override // Override the start method in the Application class
15     public void start(Stage primaryStage) {
16         // Create a border pane
17         BorderPane pane = new BorderPane();
18         create a border pane

```

```

19     // Place nodes in the pane
20     pane.setTop(getHBox());
21     pane.setLeft(getVBox());
22
23     // Create a scene and place it in the stage
24     Scene scene = new Scene(pane);
25     primaryStage.setTitle("ShowHBoxVBox"); // Set the stage title
26     primaryStage.setScene(scene); // Place the scene in the stage
27     primaryStage.show(); // Display the stage
28 }
29
30 private HBox getHBox() {
31     HBox hBox = new HBox(15);
32     hBox.setPadding(new Insets(15, 15, 15, 15));
33     hBox.setStyle("-fx-background-color: gold");
34     hBox.getChildren().add(new Button("Computer Science"));
35     hBox.getChildren().add(new Button("Chemistry"));
36     ImageView imageView = new ImageView(new Image("image/us.gif"));
37     hBox.getChildren().add(imageView);
38     return hBox;
39 }
40
41 private VBox getVBox() {
42     VBox vBox = new VBox(15);
43     vBox.setPadding(new Insets(15, 5, 5, 5));
44     vBox.getChildren().add(new Label("Courses"));
45
46     Label[] courses = {new Label("CSCI 1301"), new Label("CSCI 1302"),
47                         new Label("CSCI 2410"), new Label("CSCI 3720")};
48
49     for (Label course: courses) {
50         vBox.setMargin(course, new Insets(0, 0, 0, 15));
51         vBox.getChildren().add(course);
52     }
53
54     return vBox;
55 }
56 }
```

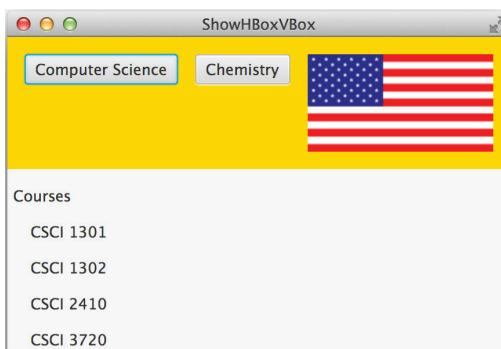


FIGURE 14.24 The `HBox` places the nodes in one row, and the `VBox` places the nodes in one column. *Source:* booka/Fotolia.

The program defines the `getHBox()` method. This method returns an `HBox` that contains two buttons and an image view (lines 30–39). The background color of the `HBox` is set to gold using Java CSS (line 33). The program defines the `getVBox()` method. This method returns a `VBox` that contains five labels (lines 41–55). The first label is added to the `VBox` in line 44 and

the other four are added in line 51. The `setMargin` method is used to set a node's margin when placed inside the `VBox` (line 50).

- 14.10.1** How do you add a node to a `Pane`, `StackPane`, `FlowPane`, `GridPane`, `BorderPane`, `HBox`, and `VBox`? How do you remove a node from these panes?
- 14.10.2** How do you set the alignment to right for nodes in a `FlowPane`, `GridPane`, `HBox`, and `VBox`?
- 14.10.3** How do you set the horizontal gap and vertical gap between nodes in 8 pixels in a `FlowPane` and `GridPane` and set spacing in 8 pixels in an `HBox` and `VBox`?
- 14.10.4** How do you get the column and row index of a node in a `GridPane`? How do you reposition a node in a `GridPane`?
- 14.10.5** What are the differences between a `FlowPane` and an `HBox` or a `VBox`?



14.11 Shapes

JavaFX provides many shape classes for drawing texts, lines, circles, rectangles, ellipses, arcs, polygons, and polylines.

The `Shape` class is the abstract base class that defines the common properties for all shapes. Among them are the `fill`, `stroke`, and `strokeWidth` properties. The `fill` property specifies a color that fills the interior of a shape. The `stroke` property specifies a color that is used to draw the outline of a shape. The `strokeWidth` property specifies the width of the outline of a shape. This section introduces the classes `Text`, `Line`, `Rectangle`, `Circle`, `Ellipse`, `Arc`, `Polygon`, and `Polyline` for drawing texts and simple shapes. All these are subclasses of `Shape`, as shown in Figure 14.25.



Use shapes
fill property
stroke property
strokeWidth property

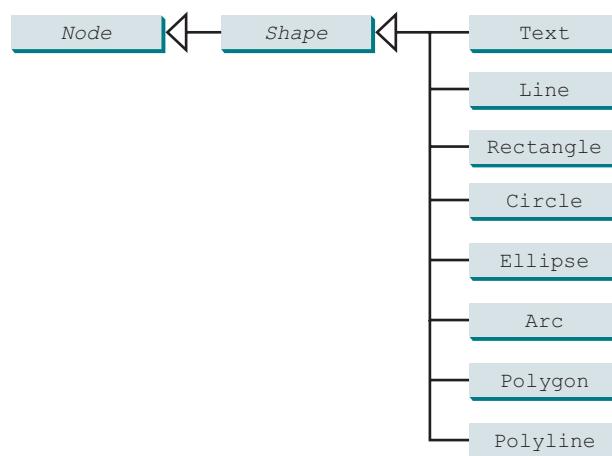
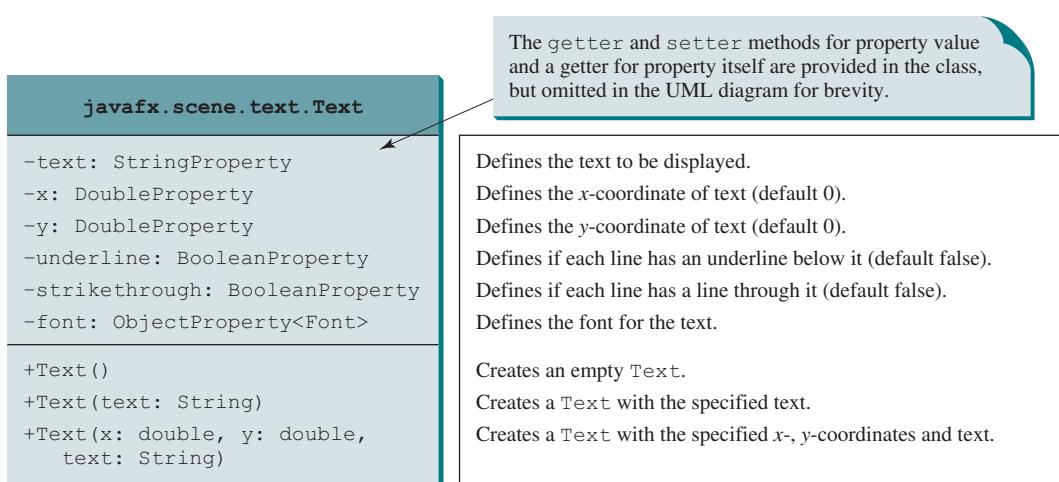
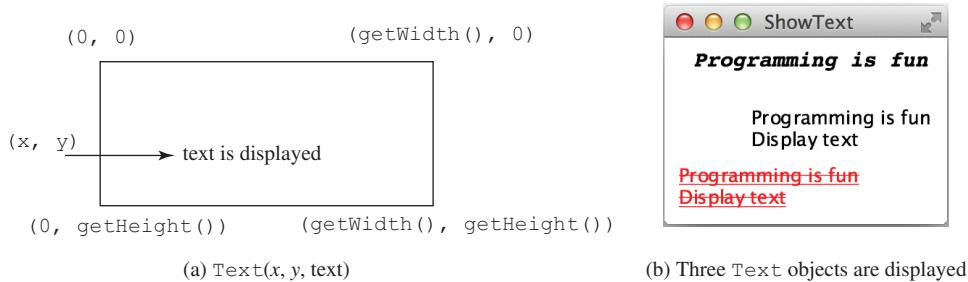


FIGURE 14.25 A shape is a node. The `Shape` class is the root of all shape classes.

14.11.1 Text

The `Text` class defines a node that displays a string at a starting point (`x`, `y`), as shown in Figure 14.27a. A `Text` object is usually placed in a pane. The pane's upper-left corner point is `(0, 0)` and the bottom-right point is `(pane.getWidth(), pane.getHeight())`. A string may be displayed in multiple lines separated by `\n`. The UML diagram for the `Text` class is shown in Figure 14.26. Listing 14.14 gives an example that demonstrates text, as shown in Figure 14.27b.

**FIGURE 14.26** `Text` defines a node for displaying a text.**FIGURE 14.27** A `Text` object is created to display a text.**LISTING 14.14** ShowText.java

```

1 import javafx.application.Application;
2 import javafx.scene.Scene;
3 import javafx.scene.layout.Pane;
4 import javafx.scene.paint.Color;
5 import javafx.geometry.Insets;
6 import javafx.stage.Stage;
7 import javafx.scene.text.Text;
8 import javafx.scene.text.Font;
9 import javafx.scene.text.FontWeight;
10 import javafx.scene.text.FontPosture;
11
12 public class ShowText extends Application {
13     @Override // Override the start method in the Application class
14     public void start(Stage primaryStage) {
15         // Create a pane to hold the texts
16         Pane pane = new Pane();
17         pane.setPadding(new Insets(5, 5, 5, 5));
18         Text text1 = new Text(20, 20, "Programming is fun");
19         text1.setFont(Font.font("Courier", FontWeight.BOLD,
20             FontPosture.ITALIC, 15));
21         pane.getChildren().add(text1);
    
```

create a pane

create a Text

set text font

add text to pane

```

22
23     Text text2 = new Text(60, 60, "Programming is fun\nDisplay text");
24     pane.getChildren().add(text2);
25
26     Text text3 = new Text(10, 100, "Programming is fun\nDisplay text");
27     text3.setFill(Color.RED);
28     text3.setUnderline(true);
29     text3.setStrikethrough(true);
30     pane.getChildren().add(text3);
31
32     // Create a scene and place it in the stage
33     Scene scene = new Scene(pane);
34     primaryStage.setTitle("ShowText"); // Set the stage title
35     primaryStage.setScene(scene); // Place the scene in the stage
36     primaryStage.show(); // Display the stage
37 }
38 }
```

main method omitted

The program creates a **Text** (line 18), sets its font (line 19), and places it to the pane (line 21). The program creates another **Text** with multiple lines (line 23) and places it to the pane (line 24). The program creates the third **Text** (line 26), sets its color (line 27), sets an underline and a strike through line (lines 28 and 29), and places it to the pane (line 30).

14.11.2 Line

A line connects two points with four parameters **startX**, **startY**, **endX**, and **endY**, as shown in Figure 14.29a. The **Line** class defines a line. The UML diagram for the **Line** class is shown in Figure 14.28. Listing 14.15 gives an example that demonstrates the line shape, as shown in Figure 14.29b.

LISTING 14.15 ShowLine.java

```

1 import javafx.application.Application;
2 import javafx.scene.Scene;
3 import javafx.scene.layout.Pane;
4 import javafx.scene.paint.Color;
5 import javafx.stage.Stage;
6 import javafx.scene.shape.Line;
7
8 public class ShowLine extends Application {
9     @Override // Override the start method in the Application class
10    public void start(Stage primaryStage) {
11        // Create a scene and place it in the stage
12        Scene scene = new Scene(new LinePane(), 200, 200);
13        primaryStage.setTitle("ShowLine"); // Set the stage title
14        primaryStage.setScene(scene); // Place the scene in the stage
15        primaryStage.show(); // Display the stage
16    }
17 }
18
19 class LinePane extends Pane {
20     public LinePane() {
21         Line line1 = new Line(10, 10, 10, 10);
22         line1.endXProperty().bind(widthProperty().subtract(10));
23         line1.endYProperty().bind(heightProperty().subtract(10));
24         line1.setStrokeWidth(5);
25         line1.setStroke(Color.GREEN);
26         getChildren().add(line1);
27     }
28 }
```

create a pane in scene

main method omitted

define a custom pane

create a line

set stroke width

set stroke

add line to pane

```

27
28     Line line2 = new Line(10, 10, 10, 10);
29     line2.startXProperty().bind(widthProperty().subtract(10));
30     line2.endYProperty().bind(heightProperty().subtract(10));
31     line2.setStrokeWidth(5);
32     line2.setStroke(Color.GREEN);
33     getChildren().add(line2);
34 }
35 }
```

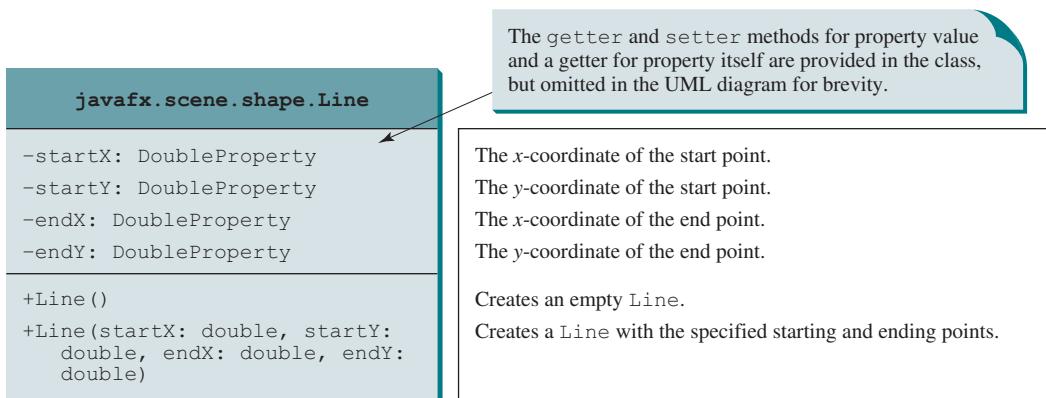


FIGURE 14.28 The `Line` class defines a line.

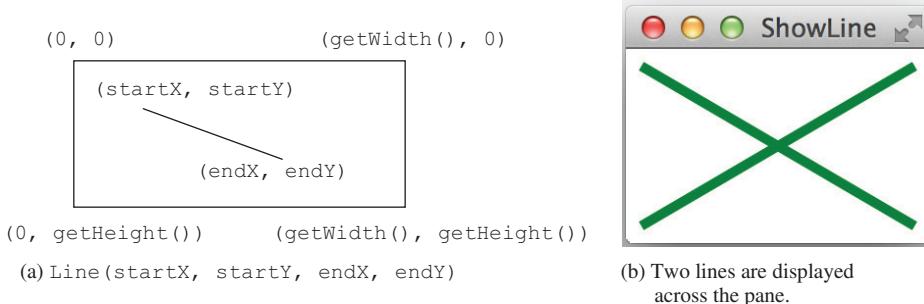


FIGURE 14.29 A `Line` object is created to display a line.

The program defines a custom pane class named `LinePane` (line 19). The custom pane class creates two lines and binds the starting and ending points of the line with the width and height of the pane (lines 22, 23, 29, and 30) so the two points of the lines are changed as the pane is resized.

14.11.3 Rectangle

A rectangle is defined by the parameters `x`, `y`, `width`, `height`, `arcWidth`, and `arcHeight`, as shown in Figure 14.31a. The rectangle's upper-left corner point is at `(x, y)`, parameter `aw` (`arcWidth`) is the horizontal diameter of the arcs at the corner, and `ah` (`arcHeight`) is the vertical diameter of the arcs at the corner.

The `Rectangle` class defines a rectangle. The UML diagram for the `Rectangle` class is shown in Figure 14.30. Listing 14.16 gives an example that demonstrates rectangles, as shown in Figure 14.31b.

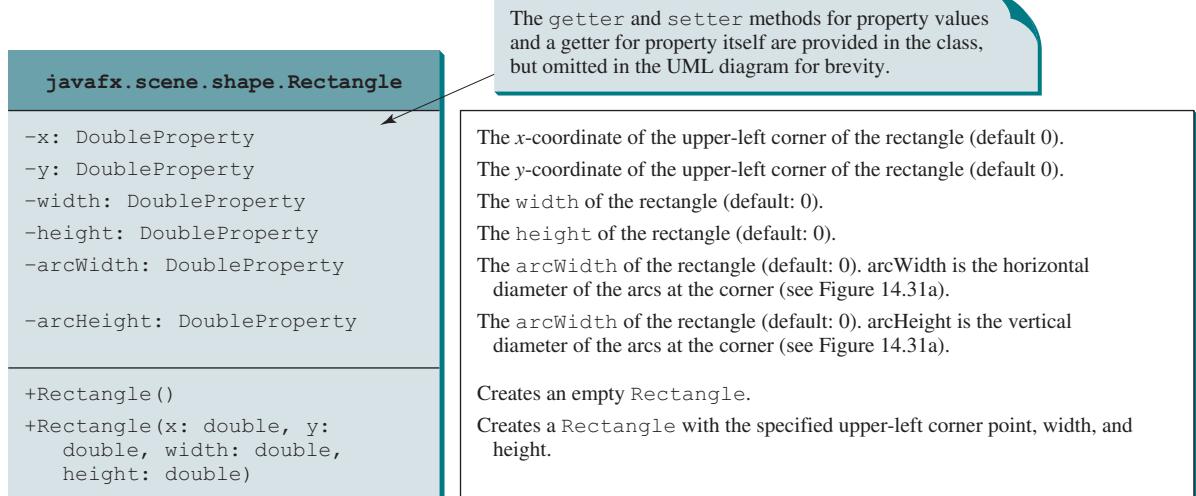


FIGURE 14.30 `Rectangle` defines a rectangle.

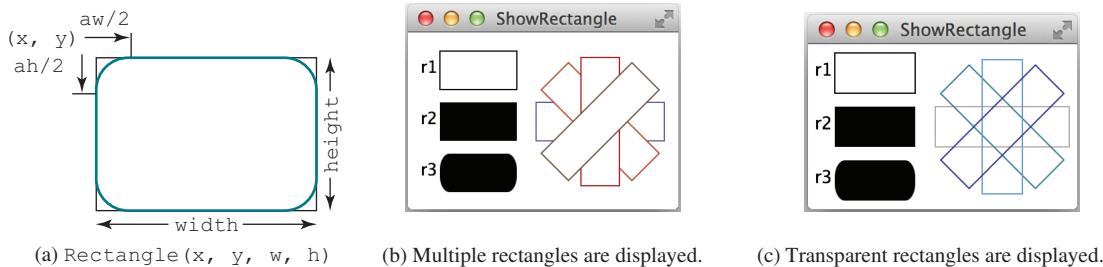


FIGURE 14.31 A `Rectangle` object is created to display a rectangle.

LISTING 14.16 ShowRectangle.java

```

1 import javafx.application.Application;
2 import javafx.scene.Group;
3 import javafx.scene.Scene;
4 import javafx.scene.layout.BorderPane;
5 import javafx.scene.paint.Color;
6 import javafx.stage.Stage;
7 import javafx.scene.text.Text;
8 import javafx.scene.shape.Rectangle;
9
10 public class ShowRectangle extends Application {
11     @Override // Override the start method in the Application class
12     public void start(Stage primaryStage) {
13         // Create rectangles
14         Rectangle r1 = new Rectangle(25, 10, 60, 30);
15         r1.setStroke(Color.BLACK);
16         r1.setFill(Color.WHITE);
17         Rectangle r2 = new Rectangle(25, 50, 60, 30);
18         Rectangle r3 = new Rectangle(25, 90, 60, 30);
19         r3.setArcWidth(15);
20         r3.setArcHeight(25);

```

create a rectangle r1
set r1's properties

create rectangle r2
create rectangle r3
set r3's arc width
set r3's arc height

```

22      // Create a group and add nodes to the group
23      Group group = new Group();
24      group.getChildren().addAll(new Text(10, 27, "r1"), r1,
25          new Text(10, 67, "r2"), r2, new Text(10, 107, "r3"), r3);
26
27      for (int i = 0; i < 4; i++) {
28          Rectangle r = new Rectangle(100, 50, 100, 30);
29          r.setRotate(i * 360 / 8);
30          r.setStroke(Color.color(Math.random(), Math.random(),
31              Math.random())));
32          r.setFill(Color.WHITE);
33          group.getChildren().add(r);
34      }
35
36      // Create a scene and place it in the stage
37      Scene scene = new Scene(new BorderPane(group), 250, 150);
38      primaryStage.setTitle("ShowRectangle"); // Set the stage title
39      primaryStage.setScene(scene); // Place the scene in the stage
40      primaryStage.show(); // Display the stage
41  }
42 }
```

create a group
add nodes to group
create a rectangle
rotate a rectangle
add rectangle to group
main method omitted

The program creates multiple rectangles. By default, the fill color is black. Thus, a rectangle is filled with black color. The stroke color is white by default. Line 15 sets stroke color of rectangle **r1** to black. The program creates rectangle **r3** (line 18) and sets its arc width and arc height (lines 19 and 20). Thus, **r3** is displayed as a rounded rectangle.

The program creates a **Group** to hold the nodes (lines 23–25). The program repeatedly creates a rectangle (line 28), rotates it (line 29), sets a random stroke color (lines 30 and 31), its fill color to white (line 32), and adds the rectangle to the group (line 33).

If line 32 is replaced by the following line:

```
r.setFill(null);
```

the rectangle is not filled with a color. Thus, they are displayed as shown in Figure 14.31c.

To center the nodes in the window, the program creates a **BorderPane** with the group in the center of the pane (line 37). If line 23 is replaced by the following?

```
Pane group = new Pane();
```

the rectangle will not be centered in the window. Therefore, using **Group** along with the **BorderPane** displays the contents of the group in the center of the window. Another advantage of using group is you can apply transformation to all nodes in the group. For example, if you add the following two lines in line 35:

```
group.setScaleX(2);
group.setScaleY(2);
```

the sizes of the nodes in the group are doubled.

14.11.4 Circle and Ellipse

You have used circles in several examples early in this chapter. A circle is defined by its parameters **centerX**, **centerY**, and **radius**. The **Circle** class defines a circle. The UML diagram for the **Circle** class is shown in Figure 14.32.

An ellipse is defined by its parameters **centerX**, **centerY**, **radiusX**, and **radiusY**, as shown in Figure 14.34a. The **Ellipse** class defines an ellipse. The UML diagram for the **Ellipse** class is shown in Figure 14.33. Listing 14.17 gives an example that demonstrates ellipses, as shown in Figure 14.34b.

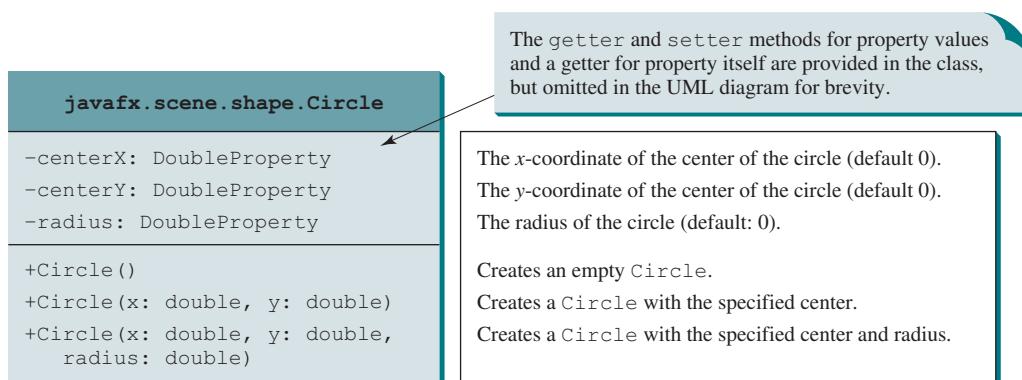


FIGURE 14.32 The `Circle` class defines circles.

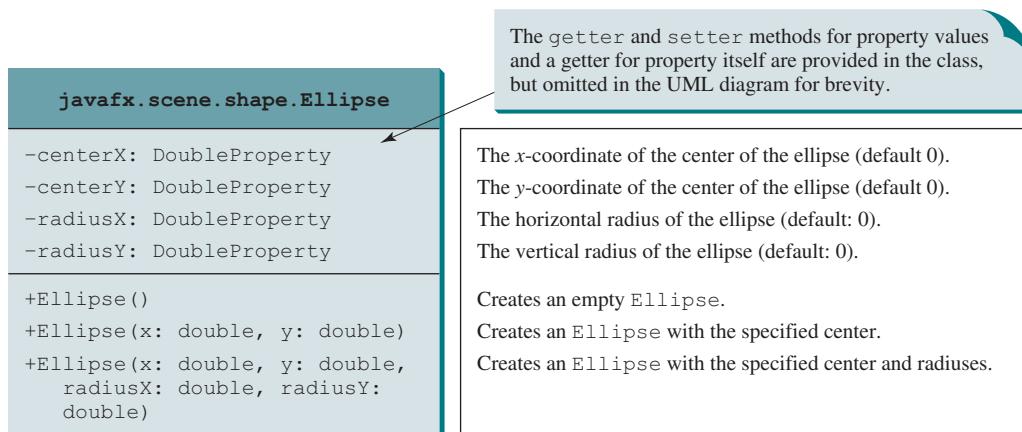


FIGURE 14.33 The `Ellipse` class defines ellipses.

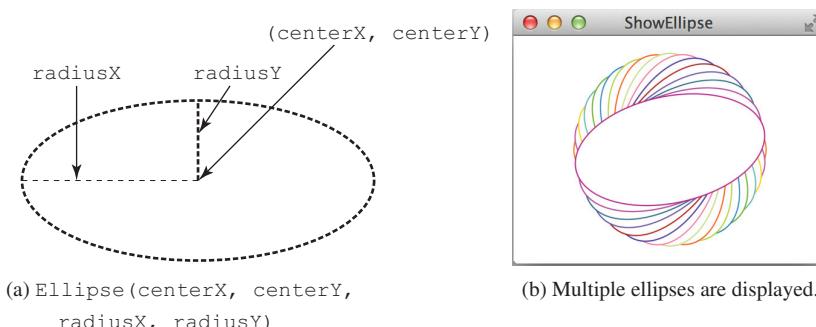


FIGURE 14.34 An `Ellipse` object is created to display an ellipse.

LISTING 14.17 ShowEllipse.java

```

1 import javafx.application.Application;
2 import javafx.scene.Scene;
3 import javafx.scene.layout.Pane;
4 import javafx.scene.paint.Color;
5 import javafx.stage.Stage;
6 import javafx.scene.shape.Ellipse;

```

```

7
8  public class ShowEllipse extends Application {
9      @Override // Override the start method in the Application class
10     public void start(Stage primaryStage) {
11         // Create a scene and place it in the stage
12         Scene scene = new Scene(new MyEllipse(), 300, 200);
13         primaryStage.setTitle("ShowEllipse"); // Set the stage title
14         primaryStage.setScene(scene); // Place the scene in the stage
15         primaryStage.show(); // Display the stage
16     }
17 }
18
19 class MyEllipse extends Pane {
20     private void paint() {
21         getChildren().clear();
22         for (int i = 0; i < 16; i++) {
23             // Create an ellipse and add it to pane
24             Ellipse e1 = new Ellipse(getWidth() / 2, getHeight() / 2,
25                                     getWidth() / 2 - 50, getHeight() / 2 - 50);
26             e1.setStroke(Color.color(Math.random(), Math.random(),
27                                     Math.random()));
28             e1.setFill(Color.WHITE);
29             e1.setRotate(i * 180 / 16);
30             getChildren().add(e1);
31         }
32     }
33
34     @Override
35     public void setWidth(double width) {
36         super.setWidth(width);
37         paint();
38     }
39
40     @Override
41     public void setHeight(double height) {
42         super.setHeight(height);
43         paint();
44     }
45 }

```

The program defines the `MyEllipse` class to draw the ellipses (lines 19–45) rather than creating ellipses directly in the `start` method (line 10) for two reasons. First, by defining the `MyEllipse` class for displaying the ellipses, you can easily reuse the code. Second, the `MyEllipse` class extends `Pane`. The contents in the pane can be resized when the stage is resized.

The `MyEllipse` class extends `Pane` and overrides the `setWidth` and `setHeight` methods (lines 34–44). A `MyEllipse` object's width and height are automatically set by invoking its `setWidth` and `setHeight` methods when it is displayed. When you resize the stage that contains a `MyEllipse`, the `MyEllipse`'s width and height are automatically resized by again invoking the `setWidth` and `setHeight` methods. The `setWidth` and `setHeight` methods invoke the `paint()` method for displaying the ellipses (lines 37 and 43). The `paint()` method first clears the contents in the pane (line 21), then repeatedly creates ellipses (lines 24 and 25), sets a random stroke color (lines 26 and 27), sets its fill color to white (line 28), rotates it (line 29), and adds the rectangle to the pane (line 30). Thus, when the stage that contains a `MyEllipse` object is resized, the contents in `MyEllipse` are redisplayed.

14.11.5 Arc

An arc is conceived as part of an ellipse, defined by the parameters `centerX`, `centerY`, `radiusX`, `radiusY`, `startAngle`, `length`, and an arc type (`ArcType.OPEN`, `ArcType.CHORD`, or `ArcType.ROUND`). The parameter `startAngle` is the starting angle, and `length` is the spanning angle (i.e., the angle covered by the arc). Angles are measured in degrees and follow the usual mathematical conventions (i.e., 0 degrees is in the easterly direction and positive angles indicate counterclockwise rotation from the easterly direction), as shown in Figure 14.36a.

The `Arc` class defines an arc. The UML diagram for the `Arc` class is shown in Figure 14.35. Listing 14.18 gives an example that demonstrates arcs, as shown in Figure 14.36b.

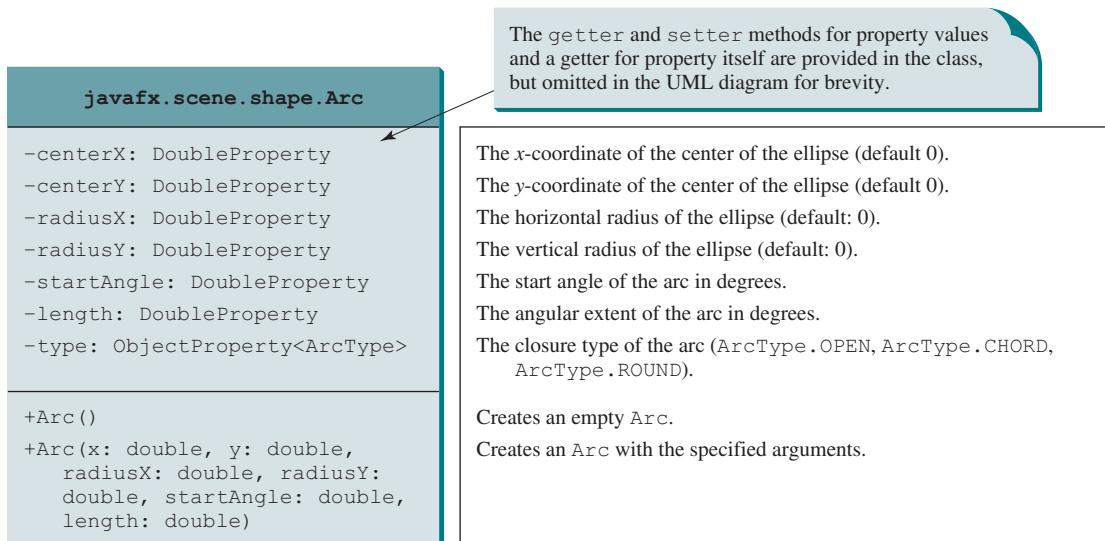


FIGURE 14.35 The `Arc` class defines an arc.

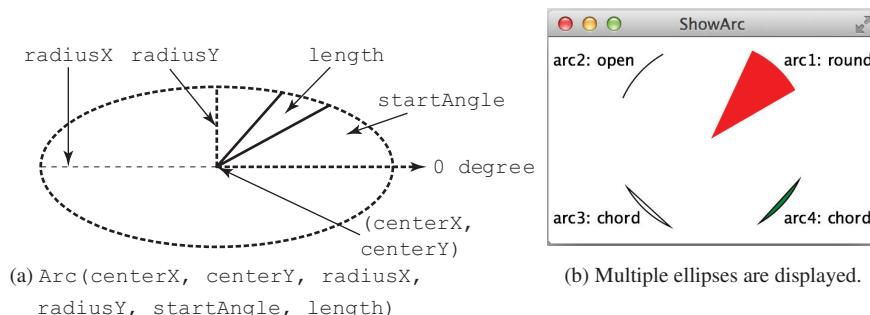


FIGURE 14.36 An `Arc` object is created to display an arc.

LISTING 14.18 ShowArc.java

```

1 import javafx.application.Application;
2 import javafx.scene.Scene;
3 import javafx.scene.Group;
4 import javafx.scene.layout.BorderPane;
5 import javafx.scene.paint.Color;

```

```

6  import javafx.stage.Stage;
7  import javafx.scene.shape.Arc;
8  import javafx.scene.shape.ArcType;
9  import javafx.scene.text.Text;
10
11 public class ShowArc extends Application {
12     @Override // Override the start method in the Application class
13     public void start(Stage primaryStage) {
14         Arc arc1 = new Arc(150, 100, 80, 80, 30, 35); // Create an arc
15         arc1.setFill(Color.RED); // Set fill color
16         arc1.setType(ArcType.ROUND); // Set arc type
17
18         Arc arc2 = new Arc(150, 100, 80, 80, 30 + 90, 35);
19         arc2.setFill(Color.WHITE);
20         arc2.setType(ArcType.OPEN);
21         arc2.setStroke(Color.BLACK);
22
23         Arc arc3 = new Arc(150, 100, 80, 80, 30 + 180, 35);
24         arc3.setFill(Color.WHITE);
25         arc3.setType(ArcType.CHORD);
26         arc3.setStroke(Color.BLACK);
27
28         Arc arc4 = new Arc(150, 100, 80, 80, 30 + 270, 35);
29         arc4.setFill(Color.GREEN);
30         arc4.setType(ArcType.CHORD);
31         arc4.setStroke(Color.BLACK);
32
33         // Create a group and add nodes to the group
34         Group group = new Group();
35         group.getChildren().addAll(new Text(210, 40, "arc1: round"),
36             arc1, new Text(20, 40, "arc2: open"), arc2,
37             new Text(20, 170, "arc3: chord"), arc3,
38             new Text(210, 170, "arc4: chord"), arc4);
39
40         // Create a scene and place it in the stage
41         Scene scene = new Scene(new BorderPane(group), 300, 200);
42         primaryStage.setTitle("ShowArc"); // Set the stage title
43         primaryStage.setScene(scene); // Place the scene in the stage
44         primaryStage.show(); // Display the stage
45     }
46 }

```

main method omitted

The program creates an arc `arc1` centered at `(150, 100)` with `radiusX 80` and `radiusY 80`. The starting angle is `30` with length `35` (line 14). `arc1`'s arc type is set to `ArcType.ROUND` (line 16). Since `arc1`'s fill color is red, `arc1` is displayed filled with red round.

The program creates an arc `arc3` centered at `(150, 100)` with `radiusX 80` and `radiusY 80`. The starting angle is `30+180` with length `35` (line 23). `arc3`'s arc type is set to `ArcType.CHORD` (line 25). Since `arc3`'s fill color is white and stroke color is black, `arc3` is displayed with black outline as a chord.

negative degrees

Angles may be negative. A negative starting angle sweeps clockwise from the easterly direction, as shown in Figure 14.37. A negative spanning angle sweeps clockwise from the starting angle. The following two statements define the same arc:

```

new Arc(x, y, radiusX, radiusY, -30, -20);
new Arc(x, y, radiusX, radiusY, -50, 20);

```

The first statement uses negative starting angle `-30` and negative spanning angle `-20`, as shown in Figure 14.37a. The second statement uses negative starting angle `-50` and positive spanning angle `20`, as shown in Figure 14.37b.

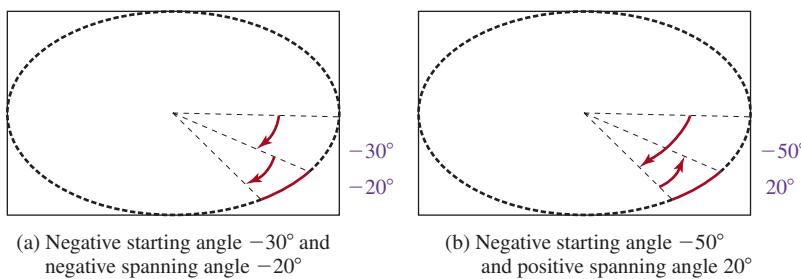


FIGURE 14.37 Angles may be negative.

Note the trigonometric methods in the **Math** class use the angles in radians, but the angles in the **Arc** class are in degrees.

14.11.6 Polygon and Polyline

The **Polygon** class defines a polygon that connects a sequence of points, as shown in Figure 14.38a. The **Polyline** class is similar to the **Polygon** class except that the **Polyline** class is not automatically closed, as shown in Figure 14.38b.

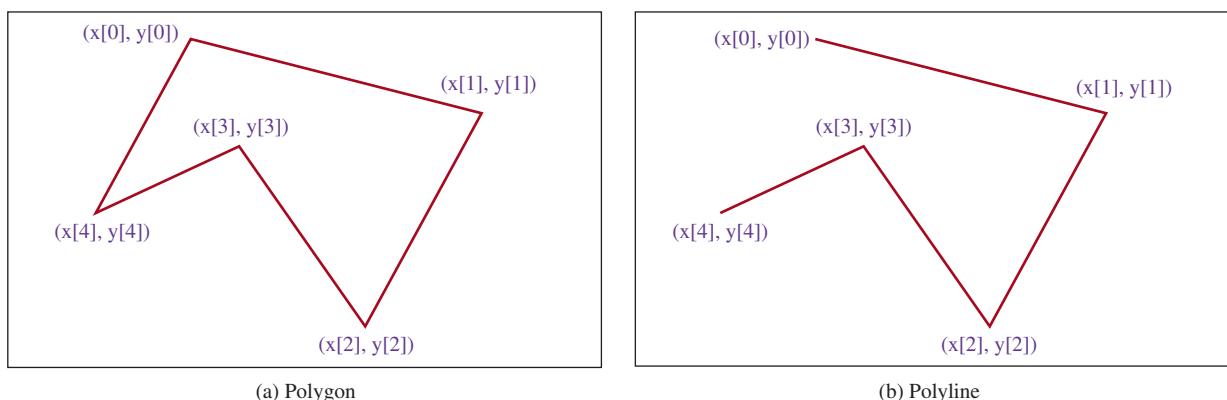


FIGURE 14.38 **Polygon** is closed and **Polyline** is not closed.

The UML diagram for the **Polygon** class is shown in Figure 14.39. Listing 14.19 gives an example that creates a hexagon, as shown in Figure 14.40.

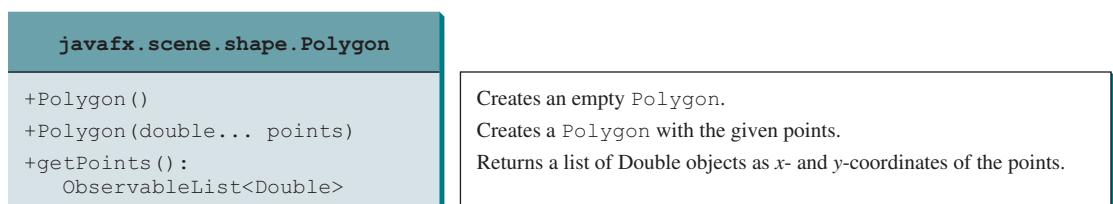
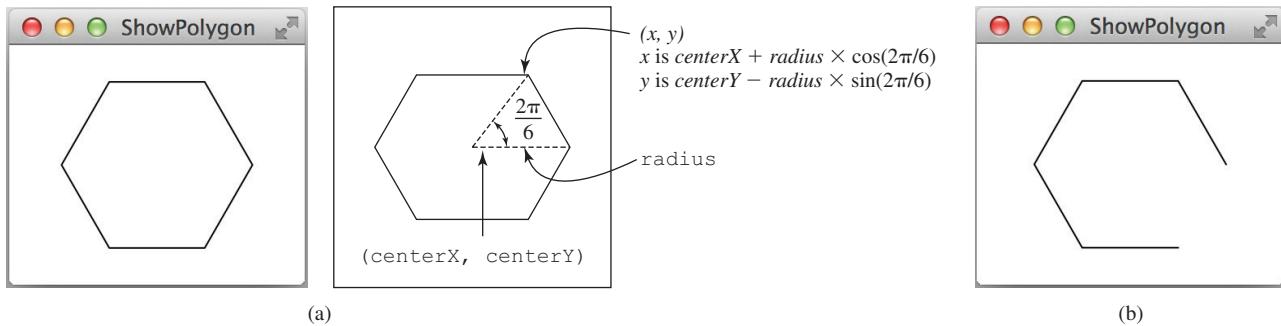


FIGURE 14.39 The **Polygon** class defines a polygon.

FIGURE 14.40 (a) A **Polygon** is displayed. (b) A **Polyline** is displayed.**LISTING 14.19 ShowPolygon.java**

```

1  import javafx.application.Application;
2  import javafx.collections.ObservableList;
3  import javafx.scene.Scene;
4  import javafx.scene.layout.Pane;
5  import javafx.scene.paint.Color;
6  import javafx.stage.Stage;
7  import javafx.scene.shape.Polygon;
8
9  public class ShowPolygon extends Application {
10    @Override // Override the start method in the Application class
11    public void start(Stage primaryStage) {
12      // Create a scene and place it in the stage
13      Scene scene = new Scene(new MyPolygon(), 400, 400);
14      primaryStage.setTitle("ShowPolygon"); // Set the stage title
15      primaryStage.setScene(scene); // Place the scene in the stage
16      primaryStage.show(); // Display the stage
17    }
18  }
19
20  class MyPolygon extends Pane {
21    private void paint() {
22      // Create a polygon and place polygon to pane
23      Polygon polygon = new Polygon();
24      polygon.setFill(Color.WHITE);
25      polygon.setStroke(Color.BLACK);
26      ObservableList<Double> list = polygon.getPoints();
27
28      double centerX = getWidth() / 2, centerY = getHeight() / 2;
29      double radius = Math.min(getWidth(), getHeight()) * 0.4;
30
31      // Add points to the polygon list
32      for (int i = 0; i < 6; i++) {
33        list.add(centerX + radius * Math.cos(2 * i * Math.PI / 6));
34        list.add(centerY - radius * Math.sin(2 * i * Math.PI / 6));
35      }
36
37      getChildren().clear();
38      getChildren().add(polygon);
39    }
40
41  @Override

```

add pane to scene

main method omitted

extends Pane

create a polygon

get a list of points

add x-coordinate of a point

add y-coordinate of a point

```

42     public void setWidth(double width) {
43         super.setWidth(width);
44         paint();
45     }
46
47     @Override
48     public void setHeight(double height) {
49         super.setHeight(height);
50         paint();
51     }
52 }
```

The program defines the **MyPolygon** class that extends **Pane** (lines 20–52). The **setWidth** and **setHeight** methods in the **Pane** class are overridden in **MyPolygon** to invoke the **paint()** method.

override **setWidth** and
setHeight

The **paint()** method creates a polygon (line 23) and adds it to a pane (line 38). The **polygon.getPoints()** method returns an **ObservableList<Double>** (line 26), which contains the **add** method for adding an element to the list (lines 33 and 34). Note the value passed to **add(value)** must be a **double** value. If an **int** value is passed, the **int** value would be automatically boxed into an **Integer**. This would cause an error, because the **ObservableList<Double>** consists of **Double** elements.

The **centerX**, **centerY**, and **radius** are obtained in proportion to the width and height of the pane (lines 28 and 29). The loop adds six points to the polygon (lines 32–35). Each point is represented by its *x*- and *y*-coordinates, computed using **centerX**, **centerY**, and **radius**. For each point, its *x*-coordinate is added to the polygon's list (line 33) then its *y*-coordinate is added to the list (line 34). The formula for computing the *x*- and *y*-coordinates for a point in the hexagon is illustrated in Figure 14.40a.

If you replace **Polygon** by **Polyline** (line 23), the program displays a polyline as shown in Figure 14.40b. The **Polyline** class is used in the same way as **Polygon**, except that the starting and ending points are not connected in **Polyline**.

14.11.1 How do you display a text, line, rectangle, circle, ellipse, arc, polygon, and polyline?



14.11.2 Write code fragments to display a string rotated 45 degrees in the center of the pane.

14.11.3 Write code fragments to display a thick line of **10** pixels from **(10, 10)** to **(70, 30)**.

14.11.4 Write code fragments to fill red color in a rectangle of width **100** and height **50** with the upper-left corner at **(10, 10)**.

14.11.5 Write code fragments to display a round-cornered rectangle with width **100**, height **200** with the upper-left corner at **(10, 10)**, corner horizontal diameter **40**, and corner vertical diameter **20**.

14.11.6 Write code fragments to display an ellipse with horizontal radius **50** and vertical radius **100**.

14.11.7 Write code fragments to display the outline of the upper half of a circle with radius **50**.

14.11.8 Write code fragments to display the lower half of a circle with radius **50** filled with the red color.

14.11.9 Write code fragments to display a polygon connecting the following points: **(20, 40)**, **(30, 50)**, **(40, 90)**, **(90, 10)**, and **(10, 30)**, and fill the polygon with green color.

14.11.10 Write code fragments to display a polyline connecting the following points: **(20, 40)**, **(30, 50)**, **(40, 90)**, **(90, 10)**, and **(10, 30)**.

14.11.11 What is wrong in the following code?

```
public void start(Stage primaryStage) {
    // Create a polygon and place it in the scene
    Scene scene = new Scene(new Polygon(), 400, 400);
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
}
```

14.12 Case Study: The **ClockPane** Class



This case study develops a class that displays a clock on a pane.

The contract of the **ClockPane** class is shown in Figure 14.41.

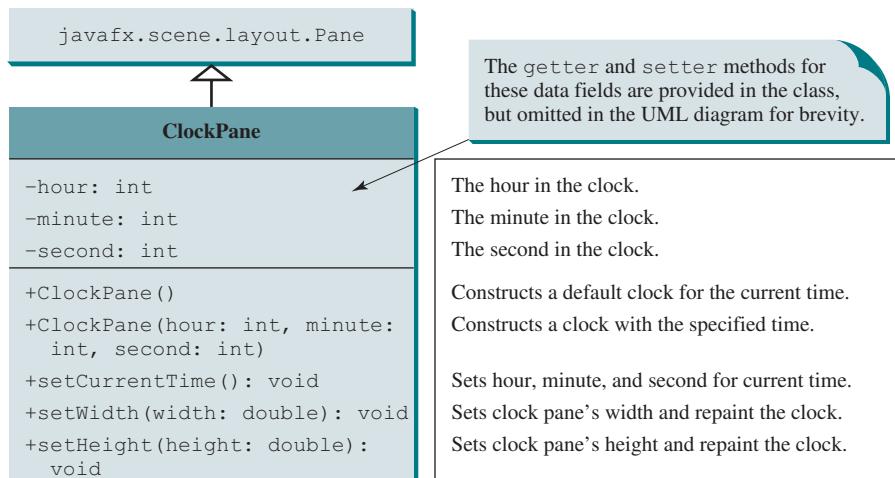


FIGURE 14.41 **ClockPane** displays an analog clock.

Assume **ClockPane** is available; we write a test program in Listing 14.20 to display an analog clock and use a label to display the hour, minute, and second, as shown in Figure 14.42a.

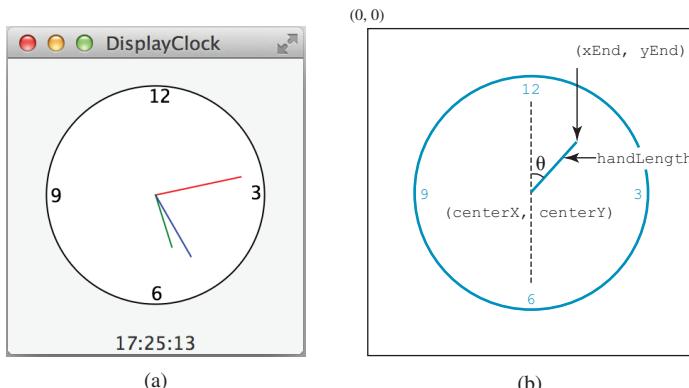


FIGURE 14.42 (a) The **DisplayClock** program displays a clock that shows the current time. (b) The endpoint of a clock hand can be determined, given the spanning angle, the hand length, and the center point.

LISTING 14.20 DisplayClock.java

```

1 import javafx.application.Application;
2 import javafx.geometry.Pos;
3 import javafx.stage.Stage;
4 import javafx.scene.Scene;
5 import javafx.scene.control.Label;
6 import javafx.scene.layout.BorderPane;
7
8 public class DisplayClock extends Application {
9     @Override // Override the start method in the Application class
10    public void start(Stage primaryStage) {
11        // Create a clock and a label
12        ClockPane clock = new ClockPane();
13        String timeString = clock.getHour() + ":" + clock.getMinute()
14            + ":" + clock.getSecond();
15        Label lblCurrentTime = new Label(timeString);           create a label
16
17        // Place clock and label in border pane
18        BorderPane pane = new BorderPane();                      add a clock
19        pane.setCenter(clock);                                  add a label
20        pane.setBottom(lblCurrentTime);
21        BorderPane.setAlignment(lblCurrentTime, Pos.TOP_CENTER); 
22
23        // Create a scene and place it in the stage
24        Scene scene = new Scene(pane, 250, 250);
25        primaryStage.setTitle("DisplayClock"); // Set the stage title
26        primaryStage.setScene(scene); // Place the scene in the stage
27        primaryStage.show(); // Display the stage
28    }
29}

```

main method omitted

The rest of this section explains how to implement the **ClockPane** class. Since you can use the class without knowing how it is implemented, you may skip the implementation if you wish.

skip implementation?
implementation

To draw a clock, you need to draw a circle and three hands for the second, minute, and hour. To draw a hand, you need to specify the two ends of the line. As shown in Figure 14.42b, one end is the center of the clock at (**centerX**, **centerY**); the other end, at (**endX**, **endY**), is determined by the following formula:

$$\begin{aligned} \text{endX} &= \text{centerX} + \text{handLength} \times \sin(\theta) \\ \text{endY} &= \text{centerY} - \text{handLength} \times \cos(\theta) \end{aligned}$$

Since there are 60 seconds in one minute, the angle for the second hand is

$$\text{second} \times (2\pi/60)$$

The position of the minute hand is determined by the minute and second. The exact minute value combined with seconds is **minute + second/60**. For example, if the time is 3 minutes and 30 seconds, the total minutes are 3.5. Since there are 60 minutes in one hour, the angle for the minute hand is

$$(\text{minute} + \text{second}/60) \times (2\pi/60)$$

Since one circle is divided into 12 hours, the angle for the hour hand is

$$(\text{hour} + \text{minute}/60 + \text{second}/(60 \times 60)) \times (2\pi/12)$$

For simplicity in computing the angles of the minute and hour hands, you can omit the seconds because they are negligibly small. Therefore, the endpoints for the second, minute, and hour hands can be computed as follows:

```
secondX = centerX + secondHandLength * sin(second * (2π/60))
secondY = centerY - secondHandLength * cos(second * (2π/60))
minuteX = centerX + minuteHandLength * sin(minute * (2π/60))
minuteY = centerY - minuteHandLength * cos(minute * (2π/60))
hourX = centerX + hourHandLength * sin((hour + minute/60) * (2π/12))
hourY = centerY - hourHandLength * cos((hour + minute/60) * (2π/12))
```

The `ClockPane` class is implemented in Listing 14.21.

LISTING 14.21 ClockPane.java

```
1 import java.util.Calendar;
2 import java.util.GregorianCalendar;
3 import javafx.scene.layout.Pane;
4 import javafx.scene.paint.Color;
5 import javafx.scene.shape.Circle;
6 import javafx.scene.shape.Line;
7 import javafx.scene.text.Text;
8
9 public class ClockPane extends Pane {
10     private int hour;
11     private int minute;
12     private int second;
13
14     /** Construct a default clock with the current time */
15     public ClockPane() {
16         setCurrentTime();
17     }
18
19     /** Construct a clock with specified hour, minute, and second */
20     public ClockPane(int hour,int minute,int second) {
21         this.hour = hour;
22         this.minute = minute;
23         this.second = second;
24     }
25
26     /** Return hour */
27     public int getHour() {
28         return hour;
29     }
30
31     /** Set a new hour */
32     public void setHour(int hour) {
33         this.hour = hour;
34         paintClock();
35     }
36
37     /** Return minute */
38     public int getMinute() {
39         return minute;
40     }
41
42     /** Set a new minute */
43     public void setMinute(int minute) {
44         this.minute = minute;
```

clock properties

no-arg constructor

constructor

set a new hour

paint clock

set a new minute

```

45     paintClock();                                paint clock
46 }
47
48 /** Return second */
49 public int getSecond() {
50     return second;
51 }
52
53 /** Set a new second */
54 public void setSecond(int second) {             set a new second
55     this.second = second;
56     paintClock();                               paint clock
57 }
58
59 /* Set the current time for the clock */
60 public void setCurrentTime() {                 set current time
61     // Construct a calendar for the current date and time
62     Calendar calendar = new GregorianCalendar();
63
64     // Set current hour, minute and second
65     this.hour = calendar.get(Calendar.HOUR_OF_DAY);
66     this.minute = calendar.get(Calendar.MINUTE);
67     this.second = calendar.get(Calendar.SECOND);
68
69     paintClock(); // Repaint the clock           paint clock
70 }
71
72 /** Paint the clock */
73 private void paintClock() {                     paint clock
74     // Initialize clock parameters
75     double clockRadius =                      get radius
76         Math.min(getWidth(), getHeight()) * 0.8 * 0.5;
77     double centerX = getWidth() / 2;            get center
78     double centerY = getHeight() / 2;
79
80     // Draw circle
81     Circle circle = new Circle(centerX, centerY, clockRadius);    create a circle
82     circle.setFill(Color.WHITE);
83     circle.setStroke(Color.BLACK);
84     Text t1 = new Text(centerX - 5, centerY - clockRadius + 12, "12");   create texts
85     Text t2 = new Text(centerX - clockRadius + 3, centerY + 5, "9");
86     Text t3 = new Text(centerX + clockRadius - 10, centerY + 3, "3");
87     Text t4 = new Text(centerX - 3, centerY + clockRadius - 3, "6");
88
89     // Draw second hand
90     double sLength = clockRadius * 0.8;
91     double secondX = centerX + sLength *          create second hand
92         Math.sin(second * (2 * Math.PI / 60));
93     double secondY = centerY - sLength *
94         Math.cos(second * (2 * Math.PI / 60));
95     Line sLine = new Line(centerX, centerY, secondX, secondY);
96     sLine.setStroke(Color.RED);
97
98     // Draw minute hand
99     double mLength = clockRadius * 0.65;
100    double xMinute = centerX + mLength *        create minute hand
101        Math.sin(minute * (2 * Math.PI / 60));
102    double minuteY = centerY - mLength *
103        Math.cos(minute * (2 * Math.PI / 60));
104    Line mLine = new Line(centerX, centerY, xMinute, minuteY);

```

```

105     mLine.setStroke(Color.BLUE);
106
107     // Draw hour hand
108     double hLength = clockRadius * 0.5;
109     double hourX = centerX + hLength *
110         Math.sin((hour % 12 + minute / 60.0) * (2 * Math.PI / 12));
111     double hourY = centerY - hLength *
112         Math.cos((hour % 12 + minute / 60.0) * (2 * Math.PI / 12));
113     Line hLine = new Line(centerX, centerY, hourX, hourY);
114     hLine.setStroke(Color.GREEN);
115
116     getChildren().clear();
117     getChildren().addAll(circle, t1, t2, t3, t4, sLine, mLine, hLine);
118 }
119
120 @Override
121 public void setWidth(double width) {
122     super.setWidth(width);
123     paintClock();
124 }
125
126 @Override
127 public void setHeight(double height) {
128     super.setHeight(height);
129     paintClock();
130 }
131 }
```

create hour hand

clear pane
add nodes to pane

set a new width

paint clock

set a new height

paint clock

override setWidth and
setHeight

The program displays a clock for the current time using the no-arg constructor (lines 15–17) and displays a clock for the specified hour, minute, and second using the other constructor (lines 20–24).

The class defines the properties `hour`, `minute`, and `second` to store the time represented in the clock (lines 10–12). The current hour, minute, and second are obtained by using the `GregorianCalendar` class (lines 62–67). The `GregorianCalendar` class in the Java API enables you to create a `Calendar` instance for the current time using its no-arg constructor. You can then use its methods `get(Calendar.HOUR)`, `get(Calendar.MINUTE)`, and `get(Calendar.SECOND)` to return the hour, minute, and second from a `Calendar` object.

The `paintClock()` method paints the clock (lines 73–118). The clock radius is proportional to the width and height of the pane (lines 75–78). A circle for the clock is created at the center of the pane (line 81). The text for showing the hours 12, 3, 6, and 9 are created in lines 84–87. The second, minute, and hour hands are the lines created in lines 90–114. The `paintClock()` method places all these shapes in the pane using the `addAll` method (line 117). Before adding new contents into the pane, the old contents are cleared from the pane (line 116).

The `setWidth` and `setHeight` methods defined in the `Pane` class are overridden in the `ClockPane` class to repaint the clock after the width or height is changed in the clock pane (lines 120–130). The `paintClock()` method is invoked whenever a new property (`hour`, `minute`, `second`, `width`, and `height`) is set (lines 34, 45, 56, 69, 123, and 129).

In Listing 14.20, the clock is placed inside a border pane, the border pane is placed in the scene, and the scene is placed in the stage. When a stage is displayed or resized, all these components inside the stage are automatically resized by invoking their respective `setWidth` and `setHeight` methods. Since the `setWidth` and `setHeight` methods are overridden to invoke the `paintClock()` method, the clock is automatically resized in response to the change of the stage size.

- 14.12.1** What will happen if lines 120–130 are removed in Listing 14.21? Run the `DisplayClock` class in Listing 14.20 to test it.



KEY TERMS

AWT 564	primary stage 565
bidirectional binding 573	property getter method 571
bindable object 570	shape 567
binding object 570	Swing 564
binding property 570	UI control 567
JavaFX 564	unidirectional binding 573
node 567	value getter method 571
observable object 570	value setter method 571
pane 567	

CHAPTER SUMMARY

- JavaFX is the new framework for developing rich GUI applications. JavaFX completely replaces Swing and AWT.
- A main JavaFX class must extend `javafx.application.Application` and implement the `start` method. The primary stage is automatically created by the JVM and passed to the `start` method.
- A stage is a window for displaying a scene. You can add nodes to a scene. Panes, groups, controls, and shapes are nodes. Panes can be used as the containers for nodes.
- A binding property can be bound to an observable source object. A change in the source object will be automatically reflected in the binding property. A binding property has a value getter method, value setter method, and property getter method.
- The `Node` class defines many properties that are common to all nodes. You can apply these properties to panes, groups, controls, and shapes.
- You can create a `Color` object with the specified red, green, blue components, and opacity value.
- You can create a `Font` object and set its name, size, weight, and posture.
- The `javafx.scene.image.Image` class can be used to load an image, and this image can be displayed in an `ImageView` object.
- JavaFX provides many types of panes for automatically laying out nodes in a desired location and size. The `Pane` is the base class for all panes. It contains the `getChildren()` method to return an `ObservableList`. You can use `ObservableList`'s `add(node)` and `addAll(node1, node2, ...)` methods for adding nodes into a pane.
- A `FlowPane` arranges the nodes in the pane horizontally from left to right or vertically from top to bottom, in the order in which they were added. A `GridPane` arranges nodes

in a grid (matrix) formation. The nodes are placed in the specified column and row indices. A **BorderPane** can place nodes in five regions: top, bottom, left, right, and center. An **HBox** lays out its children in a single horizontal row. A **VBox** lays out its children in a single vertical column.

- 11.** JavaFX provides many shape classes for drawing texts, lines, circles, rectangles, ellipses, arcs, polygons, and polylines.



Quiz

Answer the quiz for this chapter online at the book Companion Website.

MyProgrammingLab™

download image files



Note

The image files used in the exercises can be obtained from www.pearsonglobaleditions.com under the image folder.

Sections 14.2–14.9

- 14.1** (*Display images*) Write a program that displays four images in a grid pane, as shown in Figure 14.43a.

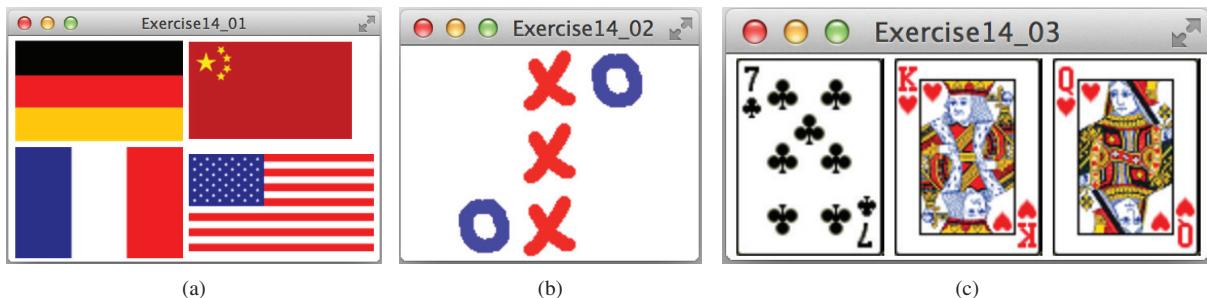


FIGURE 14.43 (a) Exercise 14.1 displays four images. *Source:* booka/Fotolia. Figure 14.43a4: United States Government. (b) Exercise 14.2 displays a tic-tac-toe board with images. (c) Three cards are randomly selected. *Source:* pandawild/Fotolia.



VideoNote

Display a tic-tac-toe board

- *14.2** (*Tic-tac-toe board*) Write a program that displays a tic-tac-toe board, as shown in Figure 14.43b. A cell may be X, O, or empty. What to display at each cell is randomly decided. The X and O are the image files **x.gif** and **o.gif**.

- *14.3** (*Display three cards*) Write a program that displays three cards randomly selected from a deck of 52, as shown in Figure 14.43c. The card image files are named **1.png**, **2.png**, . . . , **52.png** and stored in the **image/card** directory. All three cards are distinct and selected randomly. (*Hint:* You can select random cards by storing the numbers 1–52 to an array list, perform a random shuffle introduced in Section 11.12, and use the first three numbers in the array list as the file names for the image.)

- 14.4** (*Color and font*) Write a program that displays five texts vertically, as shown in Figure 14.44a. Set a random color and opacity for each text and set the font of each text to Times Roman, bold, italic, and 22 pixels.

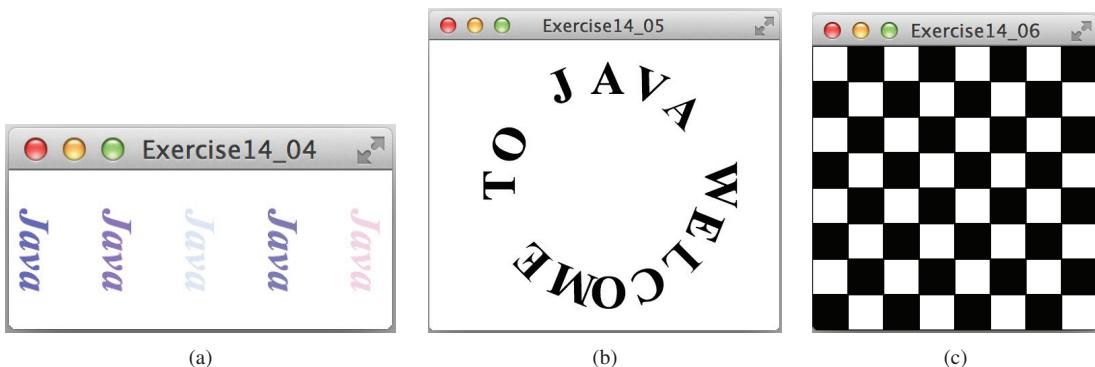


FIGURE 14.44 (a) Five texts are displayed with a random color and a specified font. (b) A string is displayed around the circle. (c) A checkerboard is displayed using rectangles.

14.5 (*Characters around circle*) Write a program that displays a string “Welcome to Java” around the circle, as shown in Figure 14.44b. (*Hint:* You need to display each character in the right location with appropriate rotation using a loop.)

***14.6** (*Game: display a checkerboard*) Write a program that displays a checkerboard in which each white and black cell is a **Rectangle** with a fill color black or white, as shown in Figure 14.44c.

Sections 14.10 and 14.11

***14.7** (*Display random 0 or 1*) Write a program that displays a 10-by-10 square matrix, as shown in Figure 14.45a. Each element in the matrix is **0** or **1**, randomly generated. Display each number centered in a text field. Use **TextField**’s **setText** method to set value **0** or **1** as a string.

Display a random matrix

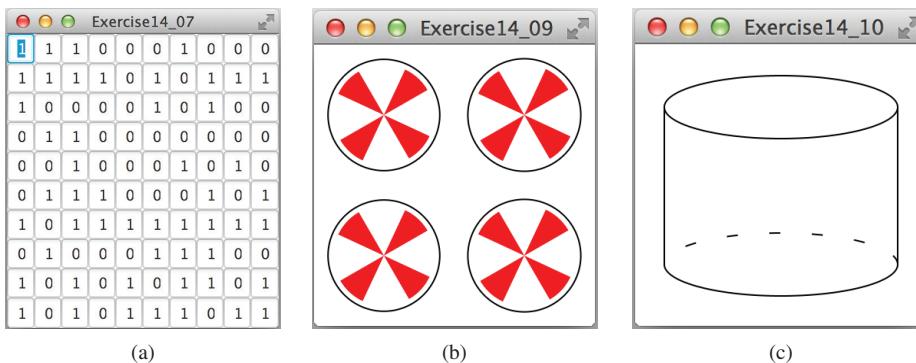


FIGURE 14.45 (a) The program randomly generates 0s and 1s. (b) Exercise 14.9 draws four fans. (c) Exercise 14.10 draws a cylinder.

14.8 (*Display 54 cards*) Expand Exercise 14.3 to display all 54 cards (including two jokers), nine per row. The image files are jokers and are named 53.png and 54.png.

***14.9** (*Create four fans*) Write a program that places four fans in a **GridPane** with two rows and two columns, as shown in Figure 14.45b.

***14.10** (*Display a cylinder*) Write a program that draws a cylinder, as shown in Figure 14.45c. You can use the following method to set the dashed stroke for an arc:

```
arc.getStrokeDashArray().addAll(6.0, 21.0);
```

The solution posted on the website enables the cylinder to resize horizontally. Can you revise it to resize vertically as well?

- *14.11 (*Paint a smiley face*) Write a program that paints a smiley face, as shown in Figure 14.46a.

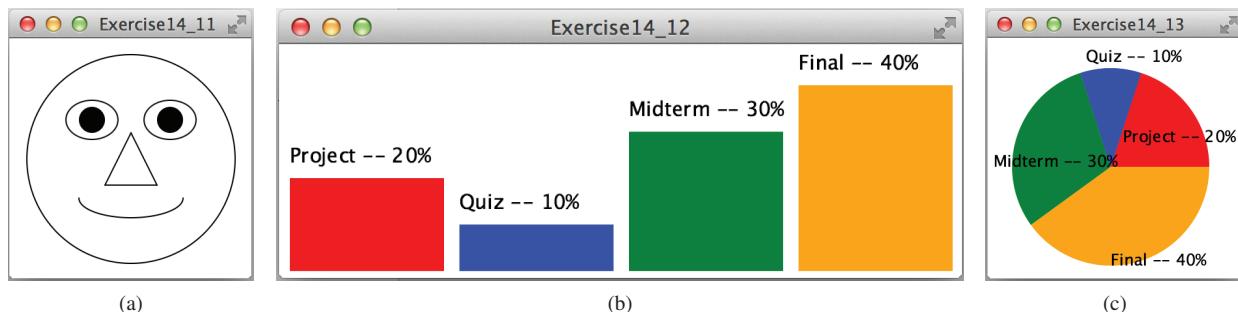


FIGURE 14.46 (a) Exercise 14.11 paints a smiley face. (b) Exercise 14.12 paints a bar chart. (c) Exercise 14.13 paints a pie chart.



VideoNote

Display a bar chart

- **14.12 (*Display a bar chart*) Write a program that uses a bar chart to display the percentages of the overall grade represented by projects, quizzes, midterm exams, and the final exam, as shown in Figure 14.46b. Suppose projects take **20%** and are displayed in red, quizzes take **10%** and are displayed in blue, midterm exams take **30%** and are displayed in green, and the final exam takes **40%** and is displayed in orange. Use the **Rectangle** class to display the bars. Interested readers may explore the JavaFX **BarChart** class for further study.

- **14.13 (*Display a pie chart*) Write a program that uses a pie chart to display the percentages of the overall grade represented by projects, quizzes, midterm exams, and the final exam, as shown in Figure 14.46c. Suppose projects take **20%** and are displayed in red, quizzes take **10%** and are displayed in blue, midterm exams take **30%** and are displayed in green, and the final exam takes **40%** and is displayed in orange. Use the **Arc** class to display the pies. Interested readers may explore the JavaFX **PieChart** class for further study.

- 14.14 (*Display a rectanguloid*) Write a program that displays a rectanguloid, as shown in Figure 14.47a. The cube should grow and shrink as the window grows or shrinks.

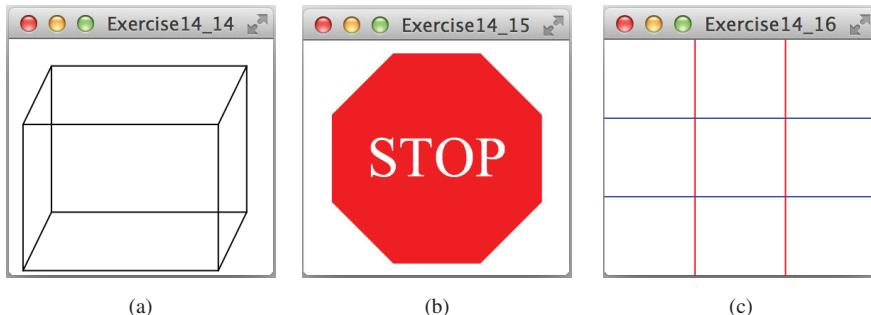


FIGURE 14.47 (a) Exercise 14.14 paints a rectanguloid. (b) Exercise 14.15 paints a STOP sign. (c) Exercise 14.16 paints a grid.

***14.15** (*Display a STOP sign*) Write a program that displays a STOP sign, as shown in Figure 14.47b. The octagon is in red and the sign is in white. (*Hint:* Place an octagon and a text in a stack pane.)

***14.16** (*Display a 3×3 grid*) Write a program that displays a 3×3 grid, as shown in Figure 14.47c. Use red color for vertical lines and blue for horizontals. The lines are automatically resized when the window is resized.

14.17 (*Game: hangman*) Write a program that displays a drawing for the popular hangman game, as shown in Figure 14.48a.

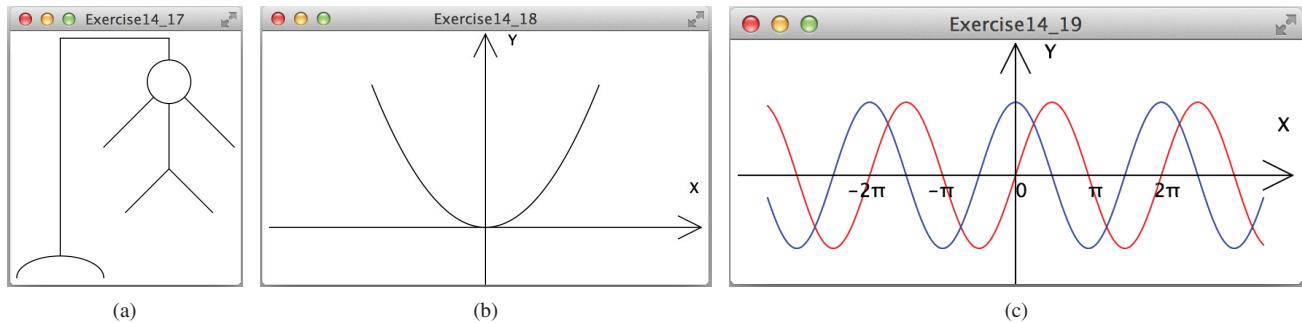


FIGURE 14.48 (a) Exercise 14.17 draws a sketch for the hangman game. (b) Exercise 14.18 plots the quadratic function. (c) Exercise 14.19 plots the sine/cosine functions.

***14.18** (*Plot the square function*) Write a program that draws a diagram for the function $f(x) = x^2$ (see Figure 14.48b).

Hint: Add points to a polyline using the following code:

```
Polyline polyline = new Polyline();
ObservableList<Double> list = polyline.getPoints();
double scaleFactor = 0.0125;
for (int x = -100; x <= 100; x++) {
    list.add(x + 200.0);
    list.add(scaleFactor * x * x);
}
```

****14.19** (*Plot the sine and cosine functions*) Write a program that plots the sine function in red and cosine in blue, as shown in Figure 14.48c.

Hint: The Unicode for π is \u03c0. To display -2π , use `Text(x, y, "-2\u03c0")`. For a trigonometric function like `sin(x)`, `x` is in radians. Use the following loop to add the points to a polyline:

```
Polyline polyline = new Polyline();
ObservableList<Double> list = polyline.getPoints();
double scaleFactor = 50;
for (int x = -170; x <= 170; x++) {
    list.add(x + 200.0);
    list.add(100 - scaleFactor * Math.sin((x / 100.0) * 2 * Math.PI));
```

Note that `x` in the loop is a point in the X-Axis and `x` does not correspond to angles in degrees. The entire expression `(x / 100.0) * 2 * Math.PI` represents an angle in radians.

```

When x is -100, Math.sin((x / 100.0) * 2 * Math.PI) is 0
When x is -75, Math.sin((x / 100.0) * 2 * Math.PI) is 1
When x is -50, Math.sin((x / 100.0) * 2 * Math.PI) is 0
When x is -25, Math.sin((x / 100.0) * 2 * Math.PI) is -1
When x is 0, Math.sin((x / 100.0) * 2 * Math.PI) is 0
When x is 25, Math.sin((x / 100.0) * 2 * Math.PI) is 1
When x is 50, Math.sin((x / 100.0) * 2 * Math.PI) is 0
When x is 75, Math.sin((x / 100.0) * 2 * Math.PI) is -1
When x is 100, Math.sin((x / 100.0) * 2 * Math.PI) is 0

```

- **14.20** (*Draw an arrow line*) Write a static method that draws an arrow line from a starting point to an ending point in a pane using the following method header:

```
public static void drawArrowLine(double startX, double startY,
                                 double endX, double endY, Pane pane)
```

Write a test program that randomly draws an arrow line, as shown in Figure 14.49a.

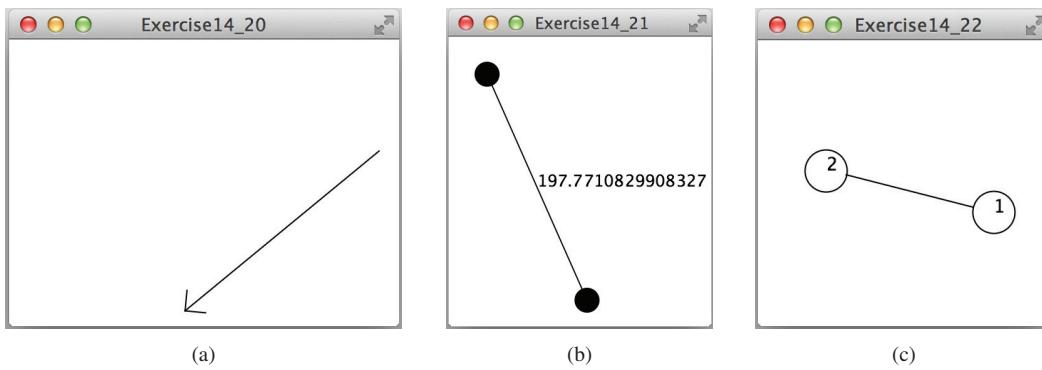


FIGURE 14.49 (a) The program displays an arrow line. (b) Exercise 14.21 connects the centers of two filled circles. (c) Exercise 14.22 connects two circles from their perimeter.

- *14.21** (*Two circles and their distance*) Write a program that draws two circles with radius **15** pixels, centered at random locations, with a line connecting the two circles. The distance between the two centers is displayed on the line, as shown in Figure 14.49b.

- *14.22** (*Connect two circles*) Write a program that draws two filled circles with radius **15** pixels, centered at random locations, with a line connecting the two circles. The line should not cross inside the circles, as shown in Figure 14.49c.

- *14.23** (*Geometry: two rectangles*) Write a program that prompts the user to enter the center coordinates, width, and height of two rectangles from the command line. The program displays the rectangles and a text indicating whether the two are overlapping, whether one is contained in the other, or whether they don't overlap, as shown in Figure 14.50. See Programming Exercise 10.13 for checking the relationship between two rectangles.

- *14.24** (*Geometry: Inside a polygon?*) Write a program that prompts the user to enter the coordinates of five points from the command line. The first four points form a polygon, and the program displays the polygon and a text that indicates whether the fifth point is inside the polygon, as shown in Figure 14.51a. (*Hint:* Use the **Node's contains** method to test whether a point is inside a node.)

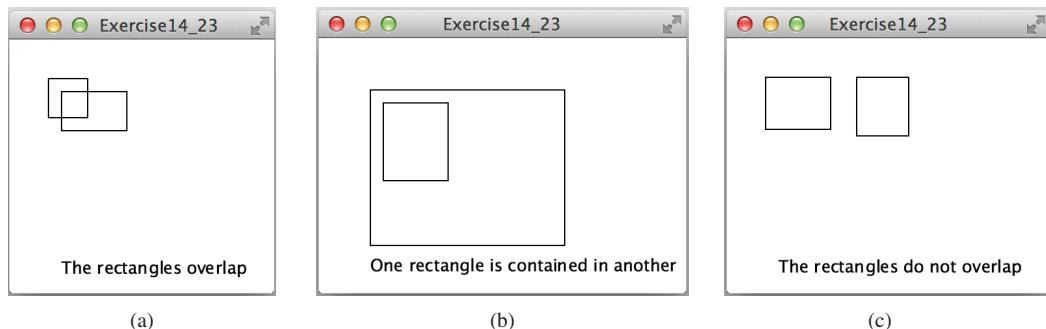


FIGURE 14.50 Two rectangles are displayed.

- *14.25** (*Random points on a circle*) Modify Programming Exercise 4.6 to create five random points on a circle, form a polygon by connecting the points clockwise, and display the circle and the polygon, as shown in Figure 14.51b.

Section 14.12

- 14.26** (*Use the **ClockPane** class*) Write a program that displays two clocks. The hour, minute, and second values are **4, 20, 45** for the first clock, and **22, 46, 15** for the second clock, as shown in Figure 14.51c.
- *14.27** (*Draw a detailed clock*) Modify the **ClockPane** class in Section 14.12 to draw the clock with more details on the hours and minutes, as shown in Figure 14.52a.

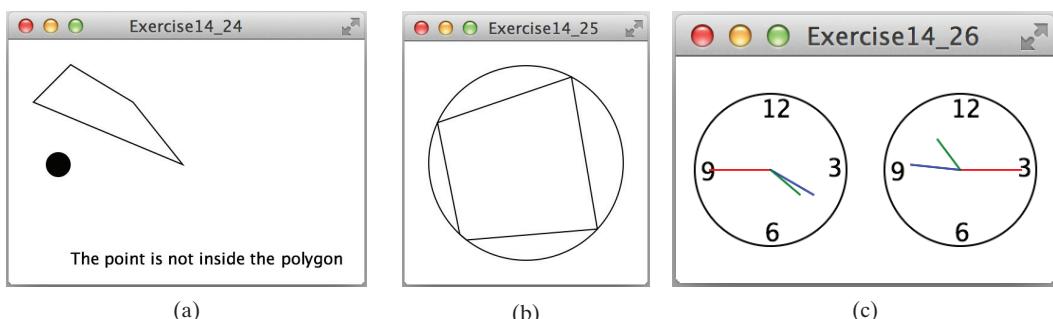


FIGURE 14.51 (a) The polygon and a point are displayed. (b) Exercise 14.25 connects five random points on a circle. (c) Exercise 14.26 displays two clocks.

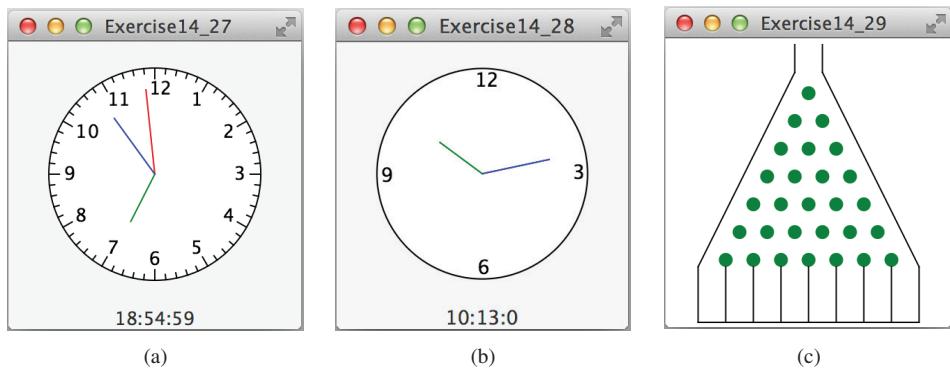


FIGURE 14.52 (a) Exercise 14.27 displays a detailed clock. (b) Exercise 14.28 displays a clock with random hour and minute values. (c) Exercise 14.29 displays a bean machine.

***14.28** (*Random time*) Modify the `ClockPane` class with three new Boolean properties—`hourHandVisible`, `minuteHandVisible`, and `secondHandVisible`—and their associated accessor and mutator methods. You can use the `set` methods to make a hand visible or invisible. Write a test program that displays only the hour and minute hands. The hour and minute values are randomly generated. The hour is between `0` and `11`, and the minute is either `0` or `30`, as shown in Figure 14.52b.

****14.29** (*Game: bean machine*) Write a program that displays a bean machine introduced in Programming Exercise 7.37, as shown in Figure 14.52c.