

ADVANCED JAVAFX AND FXML

Objectives

- To specify styles for UI nodes using JavaFX CSS (§31.2).
- To create quadratic curve, cubic curve, and path using the **QuadCurve**, **CubicCurve**, and **Path** classes (§31.3).
- To translate, rotate, and scale to perform coordinate transformations for nodes (§31.4).
- To define a shape's border using various types of strokes (§31.5).
- To create menus using the **Menu**, **MenuItem**, **CheckMenuItem**, and **RadioMenuItem** classes (§31.6).
- To create context menus using the **ContextMenu** class (§31.7).
- To use **SplitPane** to create adjustable horizontal and vertical panes (§31.8).
- To create tab panes using the **TabPane** control (§31.9).
- To create and display tables using the **TableView** and **TableColumn** classes (§31.10).
- To create JavaFX user interfaces using FMXL and the visual Scene Builder (§31.11).

CHAPTER 31





31.1 Introduction

JavaFX can be used to develop comprehensive rich Internet applications.

Chapters 14–16 introduced basics of JavaFX, event-driven programming, animations, and simple UI controls. This chapter introduces some advanced features for developing comprehensive GUI applications.



31.2 JavaFX CSS

JavaFX cascading style sheets can be used to specify styles for UI nodes.

JavaFX cascading style sheets are based on CSS with some extensions. CSS defines the style for webpages. It separates the contents of webpages from its style. JavaFX CSS can be used to define the style for the UI and separates the contents of the UI from the style. You can define the look and feel of the UI in a JavaFX CSS file and use the style sheet to set the color, font, margin, and border of the UI components. A JavaFX CSS file makes it easy to modify the style without modifying the Java source code.

A JavaFX style property is defined with a prefix **-fx-** to distinguish it from a property in CSS. All the available JavaFX properties are defined in <http://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html>. Listing 31.1 gives an example of a style sheet.

LISTING 31.1 mystyle.css

```
.plaincircle {
    -fx-fill: white;
    -fx-stroke: black;
}
.circleborder {
    -fx-stroke-width: 5;
    -fx-stroke-dash-array: 12 2 4 2;
}
.border {
    -fx-border-color: black;
    -fx-border-width: 5;
}
#redcircle {
    -fx-fill: red;
    -fx-stroke: red;
}
#greencircle {
    -fx-fill: green;
    -fx-stroke: green;
}
```

A style sheet uses the style class or style id to define styles. Multiple style classes can be applied to a single node, and a style id to a unique node. The syntax **.styleclass** defines a style class. Here, the style classes are named **plaincircle**, **circleborder**, and **circleborder**. The syntax **#styleid** defines a style id. Here, the style ids are named **red-circle** and **greencircle**.

Each node in JavaFX has a **styleClass** variable of the **List<String>** type, which can be obtained from invoking **getStyleClass()**. You can add multiple style classes to a node and only one id to a node. Each node in JavaFX has an **id** variable of the **String** type, which can be set using the **setID(String id)** method. You can set only one id to a node.

The **Scene** and **Parent** classes have the **stylesheets** property, which can be obtained from invoking the **getStylesheets()** method. This property is of the

`ObservableList<String>` type. You can add multiple style sheets into this property. You can load a style sheet into a `Scene` or a `Parent`. Note that `Parent` is the superclass for containers and UI control.

Listing 31.2 gives an example that uses the style sheet defined in Listing 31.1.

LISTING 31.2 `StyleSheetDemo.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.scene.shape.Circle;
6  import javafx.stage.Stage;
7
8  public class StyleSheetDemo extends Application {
9      @Override // Override the start method in the Application class
10     public void start(Stage primaryStage) {
11         HBox hBox = new HBox(5);
12         Scene scene = new Scene(hBox, 300, 250);
13         scene.getStylesheets().add("mystyle.css"); // Load the stylesheet
14
15         Pane pane1 = new Pane();
16         Circle circle1 = new Circle(50, 50, 30);
17         Circle circle2 = new Circle(150, 50, 30);
18         Circle circle3 = new Circle(100, 100, 30);
19         pane1.getChildren().addAll(circle1, circle2, circle3);
20         pane1.getStyleClass().add("border");
21
22         circle1.getStyleClass().add("plaincircle"); // Add a style class
23         circle2.getStyleClass().add("plaincircle"); // Add a style class
24         circle3.setId("redcircle"); // Add a style id
25
26         Pane pane2 = new Pane();
27         Circle circle4 = new Circle(100, 100, 30);
28         circle4.getStyleClass().addAll("circleborder", "plainCircle");
29         circle4.setId("greencircle"); // Add a style class
30         pane2.getChildren().add(circle4);
31         pane2.getStyleClass().add("border");
32
33         hBox.getChildren().addAll(pane1, pane2);
34
35         primaryStage.setTitle("StyleSheetDemo"); // Set the window title
36         primaryStage.setScene(scene); // Place the scene in the window
37         primaryStage.show(); // Display the window
38     }
39 }

```

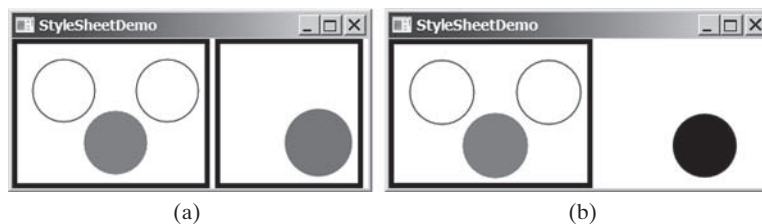


FIGURE 31.1 The style sheet is used to style the nodes in the scene.

The program loads the style sheet from the file `mystyle.css` by adding it to the `stylesheets` property (line 13). The file should be placed in the same directory with the source code for it to run correctly. After the style sheet is loaded, the program sets the style class `plaincircle` for `circle1` and `circle2` (lines 22 and 23) and sets the style id `redcircle` for `circle3` (line 24). The program sets style classes `circleborder` and `plaincircle` and an id `greencircle` for `circle4` (lines 28 and 29). The style class `border` is set for both `pane1` and `pane2` (lines 20 and 31).

The style sheet is set in the scene (line 13). All the nodes inside the scene can use this style sheet. What would happen if line 13 is deleted and the following line is inserted after line 15?

```
pane1.getStylesheets().add("mystyle.css");
```

In this case, only `pane1` and the nodes inside `pane1` can access the style sheet, but `pane2` and `circle4` cannot use this style sheet. Therefore, everything in `pane1` is displayed the same as before the change, and `pane2` and `circle4` are displayed without applying the style class and id, as shown in Figure 31.1b.

Note the style class `plaincircle` and id `greencircle` both are applied to `circle4` (lines 28 and 29). `plaincircle` sets `fill` to white and `greencircle` sets `fill` to green. The property settings in id take precedence over the ones in classes. Thus, `circle4` is displayed in green in this program.



- 31.2.1** How do you load a style sheet to a `Scene` or a `Parent`? Can you load multiple style sheets?
- 31.2.2** If a style sheet is loaded from a node, can the pane and all its containing nodes access the style sheet?
- 31.2.3** Can a node add multiple style classes? Can a node set multiple style ids?
- 31.2.4** If the same property is defined in both a style class and a style id and applied to a node, which one has the precedence?



31.3 QuadCurve, CubicCurve, and Path

JavaFX provides the `QuadCurve`, `CubicCurve`, and `Path` classes for creating advanced shapes.

Section 14.11 introduced drawing simple shapes using the `Line`, `Rectangle`, `Circle`, `Ellipse`, `Arc`, `Polygon`, and `Polyline` classes. This section introduces drawing advanced shapes using the `CubicCurve`, `QuadCurve`, and `Path` classes.

31.3.1 QuadCurve and CubicCurve

JavaFX provides the `QuadCurve` and `CubicCurve` classes for modeling quadratic curves and cubic curves. A quadratic curve is mathematically defined as a quadratic polynomial. To create a `QuadCurve`, use its no-arg constructor or the following constructor:

```
QuadCurve(double startX, double startY,  
          double controlX, double controlY, double endX, double endY)
```

where `(startX, startY)` and `(endX, endY)` specify two endpoints and `(controlX, controlY)` is a control point. The control point is usually not on the curve instead of defining the trend of the curve, as shown in Figure 31.2a. Figure 31.3 shows the UML diagram for the `QuadCurve` class.

31.3 QuadCurve, CubicCurve, and Path 31-5

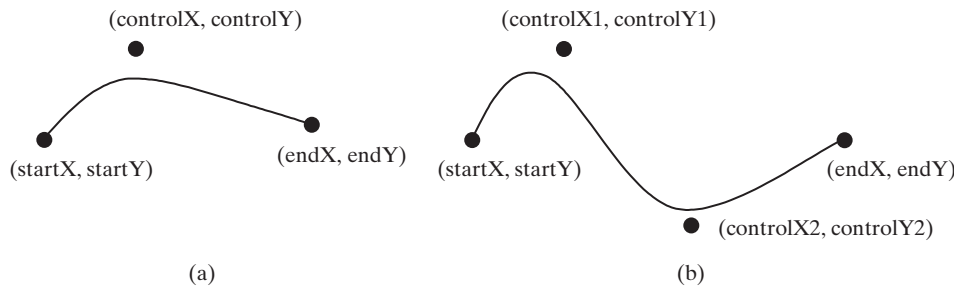


FIGURE 31.2 (a) A quadratic curve is specified using three points. (b) A cubic curve is specified using four points.

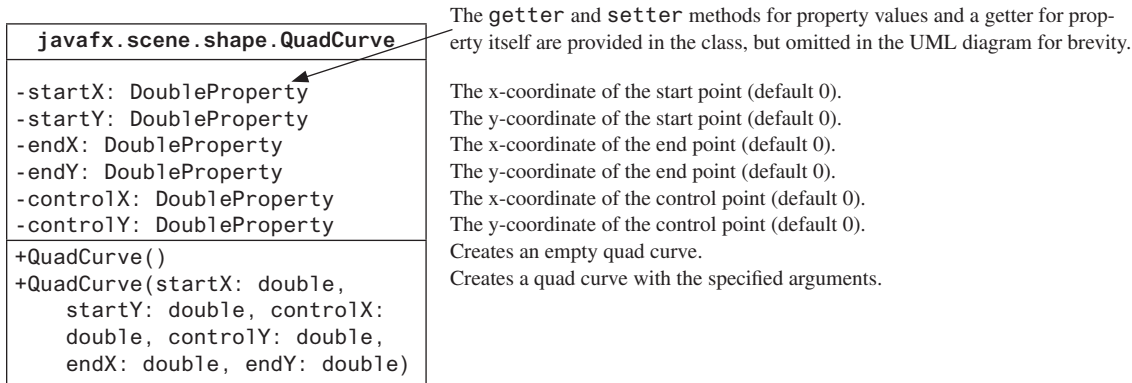


FIGURE 31.3 QuadCurve defines a quadratic curve.

A cubic curve is mathematically defined as a cubic polynomial. To create a **CubicCurve**, use its no-arg constructor or the following constructor:

```

CubicCurve(double startX, double startY, double controlX1,
           double controlY1, double controlX2, double controlY2,
           double endX, double endY)

```

where (startX, startY) and (endX, endY) specify two endpoints and (controlX1, controlY1) and (controlX2, controlY2) are two control points. The control points are usually not on the curve, instead define the trend of the curve, as shown in Figure 31.2b. Figure 31.4 shows the UML diagram for the **CubicCurve** class.

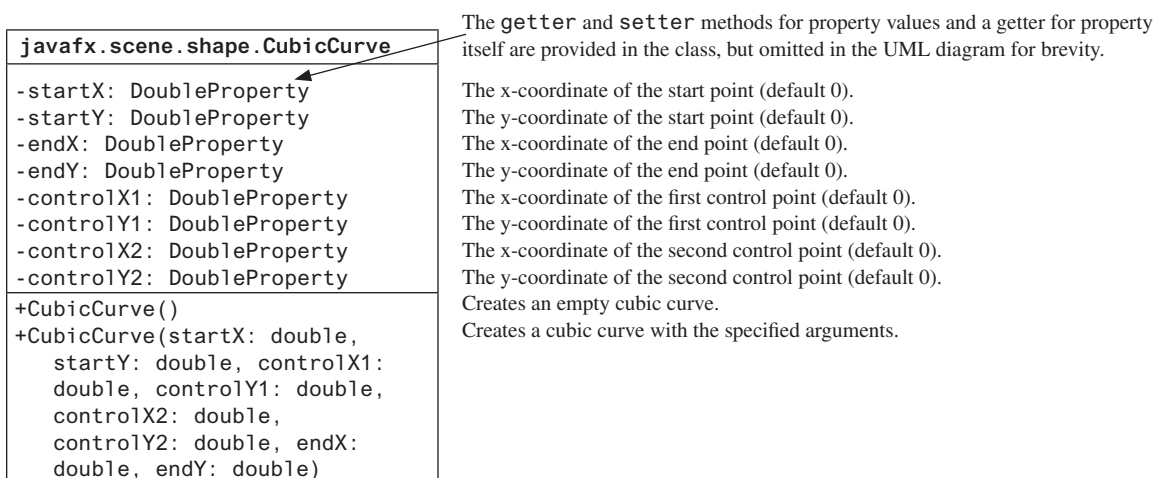


FIGURE 31.4 CubicCurve defines a quadratic curve.

Listing 31.3 gives a program that demonstrates how to draw quadratic and cubic curves. Figure 31.5a shows a sample run of the program.

LISTING 31.3 CurveDemo.java

```

1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.layout.Pane;
4  import javafx.scene.text.Text;
5  import javafx.scene.shape.Circle;
6  import javafx.scene.paint.Color;
7  import javafx.scene.shape.*;
8  import javafx.stage.Stage;
9
10 public class CurveDemo extends Application {
11     @Override // Override the start method in the Application class
12     public void start(Stage primaryStage) {
13         Pane pane = new Pane();
14
15         // Create a QuadCurve
16         QuadCurve quadCurve = new QuadCurve(10, 80, 40, 20, 150, 56);
17         quadCurve.setFill(Color.WHITE);
18         quadCurve.setStroke(Color.BLACK);
19
20         pane.getChildren().addAll(quadCurve, new Circle(40, 20, 6),
21             new Text(40 + 5, 20 - 5, "Control point"));
22
23         // Create a CubicCurve
24         CubicCurve cubicCurve = new CubicCurve
25             (200, 80, 240, 20, 350, 156, 450, 80);
26         cubicCurve.setFill(Color.WHITE);
27         cubicCurve.setStroke(Color.BLACK);
28
29         pane.getChildren().addAll(cubicCurve, new Circle(240, 20, 6),
30             new Text(240 + 5, 20 - 5, "Control point 1"),
31             new Circle(350, 156, 6),
32             new Text(350 + 5, 156 - 5, "Control point 2"));
33
34         Scene scene = new Scene(pane, 300, 250);
35         primaryStage.setTitle("CurveDemo"); // Set the window title
36         primaryStage.setScene(scene); // Place the scene in the window
37         primaryStage.show(); // Display the window
38     }
39 }

```

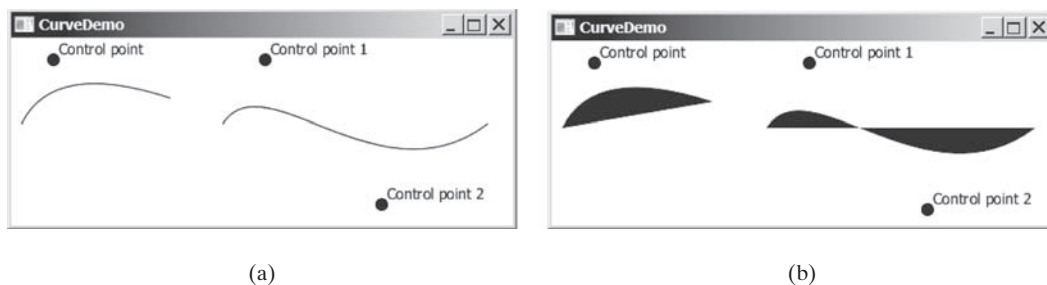


FIGURE 31.5 You can draw quadratic and cubic curves using `QuadCurve` and `CubicCurve`.

The program creates a `QuadCurve` with the specified start, control, and end points (line 16) and places the `QuadCurve` to the pane (line 20). To illustrate the control point, the program also displays the control point as a solid circle (line 21).

The program creates a **CubicCurve** with the specified start, first control, second control, and end points (lines 24 and 25) and places the **CubicCurve** to the pane (line 29). To illustrate the control points, the program also displays the control points in the pane (lines 29–32).

Note the curves are filled with color. The program sets the color to white and stroke to black in order to display the curves (lines 17 and 18, 26 and 27). If these code lines are removed from the program, the sample run would look like the one in Figure 31.5b.

31.3.2 Path

The **Path** class models an arbitrary geometric path. A path is constructed by adding path elements into the path. The **PathElement** is the root class for the path elements **MoveTo**, **HLineTo**, **VLineTo**, **LineTo**, **ArcTo**, **QuadCurveTo**, **CubicCurveTo**, and **ClosePath**.

You can create a **Path** using its no-arg constructor. The process of the path construction can be viewed as drawing with a pen. The path does not have a default initial position. You need to set an initial position by adding a **MoveTo(startX, startY)** path element to the path. Adding a **HLineTo(newX)** element draws a horizontal line from the current position to the new *x*-coordinate. Adding a **VLineTo(newY)** element draws a vertical line from the current position to the new *y*-coordinate. Adding a **LineTo(newX, newY)** element draws a line from the current position to the new position. Adding an **ArcTo(radiusX, radiusY, xAxisRotation, newX, newY, largeArcFlag, sweepArcFlag)** element draws an arc from the previous position to the new position with the specified radius. Adding a **QuadCurveTo(controlX, controlY, newX, newY)** element draws a quadratic curve from the previous position to the new position with the specified control point. Adding a **CubicCurveTo(controlX1, controlY1, controlX2, controlY2, newX, newY)** element draws a cubic curve from the previous position to the new position with the specified control points. Adding a **ClosePath()** element closes the path by drawing a line that connects the starting point with the end point of the path.

Listing 31.4 gives an example that creates a path. A sample run of the program is shown in Figure 31.6.

LISTING 31.4 PathDemo.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.*;
6  import javafx.stage.Stage;
7
8  public class PathDemo extends Application {
9      @Override // Override the start method in the Application class
10     public void start(Stage primaryStage) {
11         Pane pane = new Pane();
12
13         // Create a Path
14         Path path = new Path();
15         path.getElements().add(new MoveTo(50.0, 50.0));
16         path.getElements().add(new HLineTo(150.5));
17         path.getElements().add(new VLineTo(100.5));
18         path.getElements().add(new LineTo(200.5, 150.5));
19
20         ArcTo arcTo = new ArcTo(45, 45, 250, 100.5,
21             false, true);
22         path.getElements().add(arcTo);
23
24         path.getElements().add(new QuadCurveTo(50, 50, 350, 100));
25         path.getElements().add(
26             new CubicCurveTo(250, 100, 350, 250, 450, 10));

```

```

27
28     path.getElements().add(new ClosePath());
29
30     pane.getChildren().add(path);
31     path.setFill(null);
32     Scene scene = new Scene(pane, 300, 250);
33     primaryStage.setTitle("PathDemo"); // Set the window title
34     primaryStage.setScene(scene); // Place the scene in the window
35     primaryStage.show(); // Display the window
36 }
37 }

```

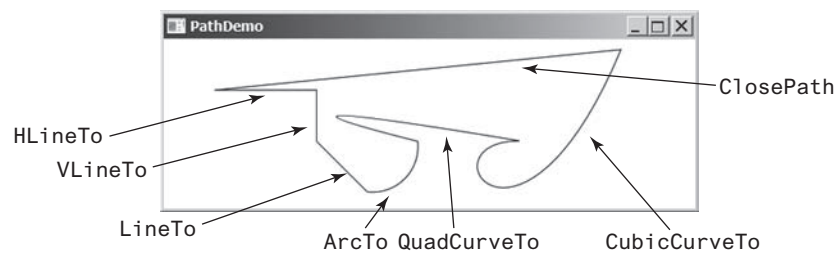


FIGURE 31.6 You can draw a path by adding path elements.

The program creates a **Path** (line 14), moves its position (line 15), and adds a horizontal line (line 16), a vertical line (line 17), and a line (line 18). The **getElements()** method returns an **ObservableList<PathElement>**.

The program creates an **ArcTo** object (lines 20 and 21). The **ArcTo** class contains the **largeArcFlag** and **sweepFlag** properties. By default, these property values are **false**. You may set these properties to **true** to display a large arc in the opposite direction.

The program adds a quadratic curve (line 24) and a cubic curve (lines 25 and 26) and closes the path (line 28).

By default, the path is not filled. You may change the **fill** property in the path to specify a color to fill the path.



31.3.1 Create a **QuadCurve** with starting point (100, 75.5), control point (40, 55.5), and end point (56, 80). Set its **fill** property to white and **stroke** to green.

31.3.2 Create **CubicCurve** object with starting point (100, 75.5), control point 1 (40, 55.5), control point 2 (78.5, 25.5), and end point (56, 80). Set its **fill** property to white and **stroke** to green.

31.3.3 Does a path have a default initial position? How do you set a position for a path?

31.3.4 How do you close a path?

31.3.5 How do you display a filled path?



31.4 Coordinate Transformations

JavaFX supports coordinate transformations using translation, rotation, and scaling.

You have used the **rotate** method to rotate a node. You can also perform translations and scaling.

31.4.1 Translations

You can use the **setTranslateX(double x)**, **setTranslateY(double y)**, and **setTranslateZ(double z)** methods in the **Node** class to translate the coordinates for a

node. For example, `setTranslateX(5)` moves the node 5 pixels to the right and `setTranslateY(-10)` 10 pixels up from the previous position. Figure 31.7 shows a rectangle displayed before and after applying translation. After invoking `rectangle.setTranslateX(-6)` and `rectangle.setTranslateY(4)`, the rectangle is moved 6 pixels to the left and 4 pixels down from the previous position. Note the coordinate transformation using translation, rotation, and scaling does not change the contents of the shape being transferred. For example, if a rectangle's x is 30 and width is 100, after applying transformations to the rectangle, its x is still 30 and width is still 100.

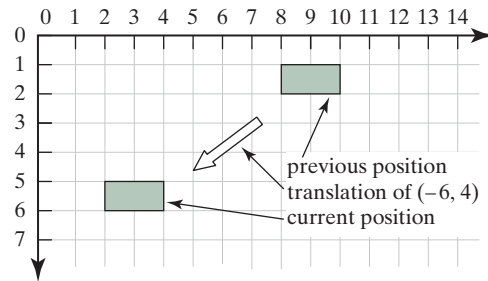


FIGURE 31.7 After applying translation of $(-6, 4)$, the rectangle is moved by the specified distance relative to the previous position.

LISTING 31.5 TranslationDemo.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.Rectangle;
6  import javafx.stage.Stage;
7
8  public class TranslationDemo extends Application {
9      @Override // Override the start method in the Application class
10     public void start(Stage primaryStage) {
11         Pane pane = new Pane();
12
13         double x = 10;
14         double y = 10;
15         java.util.Random random = new java.util.Random();
16         for (int i = 0; i < 10; i++) {
17             Rectangle rectangle = new Rectangle(10, 10, 50, 60);
18             rectangle.setFill(Color.WHITE);
19             rectangle.setStroke(Color.color(random.nextDouble(),
20                 random.nextDouble(), random.nextDouble()));
21             rectangle.setTranslateX(x += 20);
22             rectangle.setTranslateY(y += 5);
23             pane.getChildren().add(rectangle);
24         }
25
26         Scene scene = new Scene(pane, 300, 250);
27         primaryStage.setTitle("TranslationDemo"); // Set the window title
28         primaryStage.setScene(scene); // Place the scene in the window
29         primaryStage.show(); // Display the window
30     }
31 }
```

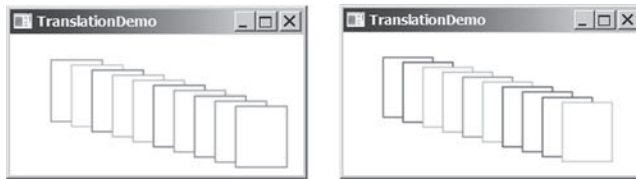


FIGURE 31.8 The rectangles are displayed successively in new locations.

The program repeatedly creates 10 rectangles (line 17). For each rectangle, it sets its `fill` property to white (line 18) and its `stroke` property to a random color (lines 19 and 20), and translates it to a new location (lines 21 and 22). The variables `x` and `y` are used to set the `translateX` and `translateY` properties. These two variable values are changed every time it is applied to a rectangle (see Figure 31.8).

31.4.2 Rotations

Rotation was introduced in Chapter 14. This section discusses it in more depth. You can use the `rotate(double theta)` method in the `Node` class to rotate a node by `theta` degrees from its pivot point clockwise, where `theta` is a double value in degrees. The pivot point is automatically computed based on the bounds of the node. For a circle, ellipse, and a rectangle, the pivot point is the center point of these nodes. For example, `rectangle.rotate(45)` rotates the rectangle 45 degrees clockwise along the eastern direction from the center, as shown in Figure 31.9.

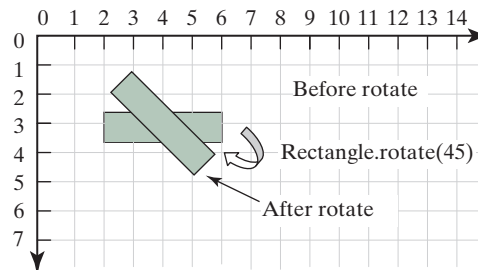


FIGURE 31.9 After performing `rectangle.rotate(45)`, the rectangle is rotated in 45 degrees from the center.

Listing 31.6 gives a program that demonstrates the effect of rotation of coordinates. Figure 31.10 shows a sample run of the program.

LISTING 31.6 RotateDemo.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.Rectangle;
6 import javafx.stage.Stage;
7
8 public class RotateDemo extends Application {
9     @Override // Override the start method in the Application class
10    public void start(Stage primaryStage) {
11        Pane pane = new Pane();
12        java.util.Random random = new java.util.Random();

```

```

13 // The radius of the circle for anchoring rectangles
14 double radius = 90;
15 double width = 20; // Width of the rectangle
16 double height = 40; // Height of the rectangle
17 for (int i = 0; i < 8; i++) {
18     // Center of a rectangle
19     double x = 150 + radius * Math.cos(i * 2 * Math.PI / 8);
20     double y = 150 + radius * Math.sin(i * 2 * Math.PI / 8);
21     Rectangle rectangle = new Rectangle(
22         x - width / 2, y - height / 2, width, height);
23     rectangle.setFill(Color.WHITE);
24     rectangle.setStroke(Color.color(random.nextDouble(),
25         random.nextDouble(), random.nextDouble()));
26     rectangle.setRotate(i * 360 / 8); // Rotate the rectangle
27     pane.getChildren().add(rectangle);
28 }
29
30 Scene scene = new Scene(pane, 300, 300);
31 primaryStage.setTitle("RotateDemo"); // Set the window title
32 primaryStage.setScene(scene); // Place the scene in the window
33 primaryStage.show(); // Display the window
34 }
35 }

```

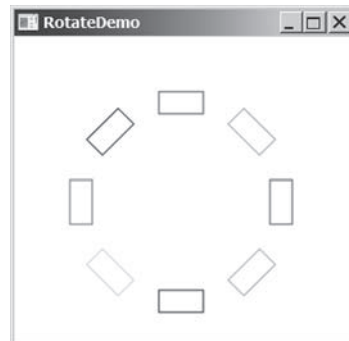


FIGURE 31.10 The `rotate` method rotates a node.

The program creates eight rectangles in a loop (lines 17–28). The center of each rectangle is located on the circle centered as (150, 150) (lines 19 and 20). A rectangle is created by specifying its upper left corner position with width and height (lines 21 and 22). The rectangle is rotated in line 26 and added to the pane in line 27.

31.4.3 Scaling

You can use the `setScaleX(double sx)`, `setScaleY(double sy)`, and `setScaleY(double sy)` methods in the `Node` class to specify a scaling factor. The node will appear larger or smaller depending on the scaling factor. Scaling alters the coordinate space of the node such that each unit of distance along the axis is multiplied by the scale factor. As with rotation transformations, scaling transformations are applied to enlarge or shrink the node around the pivot point. For a node of the rectangle shape, the pivot point is the center of the rectangle. For example, if you apply a scaling factor ($x = 2$, $y = 2$), the entire rectangle including the stroke will double in size, growing to the left, right, up, and down from the center, as shown in Figure 31.11.

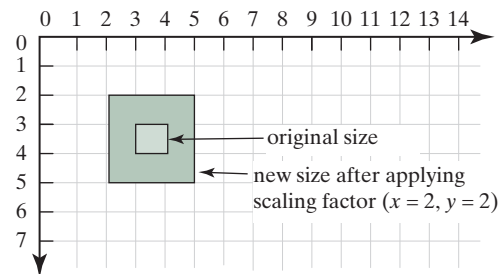


FIGURE 31.11 After applying scaling ($x = 2$, $y = 2$), the node is doubled in size.

Listing 31.7 gives a program that demonstrates the effect of using scaling. Figure 31.12 shows a sample run of the program.

LISTING 31.7 ScaleDemo.java

```

1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.layout.Pane;
4  import javafx.scene.shape.Line;
5  import javafx.scene.text.Text;
6  import javafx.scene.shape.Polyline;
7  import javafx.stage.Stage;
8
9  public class ScaleDemo extends Application {
10     @Override // Override the start method in the Application class
11     public void start(Stage primaryStage) {
12         // Create a polyline to draw a sine curve
13         Polyline polyline = new Polyline();
14         for (double angle = -360; angle <= 360; angle++) {
15             polyline.getPoints().addAll(
16                 angle, Math.sin(Math.toRadians(angle)));
17         }
18         polyline.setTranslateY(100);
19         polyline.setTranslateX(200);
20         polyline.setScaleX(0.5);
21         polyline.setScaleY(50);
22         polyline.setStrokeWidth(1.0 / 25);
23
24         // Draw x-axis
25         Line line1 = new Line(10, 100, 420, 100);
26         Line line2 = new Line(420, 100, 400, 90);
27         Line line3 = new Line(420, 100, 400, 110);
28
29         // Draw y-axis
30         Line line4 = new Line(200, 10, 200, 200);
31         Line line5 = new Line(200, 10, 190, 30);
32         Line line6 = new Line(200, 10, 210, 30);
33
34         // Draw x, y axis labels
35         Text text1 = new Text(380, 70, "X");
36         Text text2 = new Text(220, 20, "Y");
37
38         // Add nodes to a pane
39         Pane pane = new Pane();
40         pane.getChildren().addAll(polyline, line1, line2, line3, line4,
41             line5, line6, text1, text2);
42     }

```

```
43     Scene scene = new Scene(pane, 450, 200);
44     primaryStage.setTitle("ScaleDemo"); // Set the window title
45     primaryStage.setScene(scene); // Place the scene in the window
46     primaryStage.show(); // Display the window
47 }
48 }
```

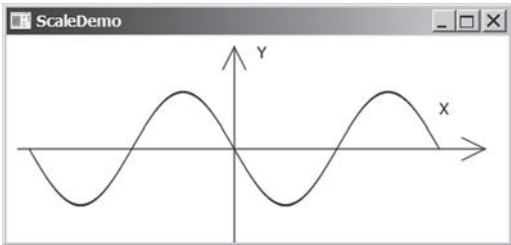


FIGURE 31.12 The `scale` method scales the coordinates in the node.

The program creates a polyline (line 13) and adds the points for a sine curve into the polyline (lines 14–17). Since $|\sin(x)| \leq 1$, the y -coordinates are too small. To see the sine curve, the program scales the y -coordinates up by 50 times (line 21) and shrinks the x -coordinates by half (line 20).

Note scaling also causes the stroke width to change. To compensate it, the stroke width is purposely set to $1.0 / 25$ (line 22).

- 31.4.1 Can you perform a coordinate transformation on any node? Does a coordinate transformation change the contents of a `Shape` object?
- 31.4.2 Does the method `setTranslateX(6)` move the node’s x -coordinate to 6? Does the method `setTranslateX(6)` move the node’s x -coordinate 6 pixel right from its current location?
- 31.4.3 Does the method `rotate(Math.PI / 2)` rotate a node 90 degrees? Does the method `rotate(90)` rotate a node 90 degrees?
- 31.4.4 How is the pivot point determined for performing a rotation?
- 31.4.5 What method do you use to scale a node two times on its x -axis?



31.5 Strokes

Stroke defines a shape’s border line style.

JavaFX allows you to specify the attributes of a shape’s boundary using the methods in Figure 31.13.



javafx.scene.shape.Shape	
+setStroke(Paint paint): void	Sets a paint for the stroke.
+setStrokeWidth(double width): void	Sets a width for the stroke (default 1).
+setStrokeType(StrokeType type): void	Sets a type for the stroke to indicate whether the stroke is placed inside, centered, or outside of the border (default: CENTERED).
+setStrokeLineCap(StrokeLineCap type): void	Specifies the end cap style for the stroke (default: BUTT).
+setStrokeLineJoin(StrokeLineJoin type): void	Specifies how two line segments are joined (default: MITER).
+getStrokeDashArray(): ObservableList<Double>	Returns a list that specifies a dashed pattern for line segments.
+setStrokeDashOffset(double distance): void	Specifies the offset to the first segment in the dashed pattern.

FIGURE 31.13 The `Shape` class contains the methods for setting stroke properties.

The `setStroke(Paint paint)` method sets a paint for the stroke. The width of the stroke can be specified using the `setStrokeWidth(double width)` method.

The `setStrokeType(type)` method sets a type for the stroke. The type defines whether the stroke is inside, outside, or in the center of the border using the constants `StrokeType.INSIDE`, `StrokeType.OUTSIDE`, or `StrokeType.CENTERED` (default), as shown in Figure 31.14.



FIGURE 31.14 (a) No stroke is used. (b) A stroke is placed inside the border. (c) A stroke is placed in the center of the border. (d) A stroke is placed outside of the border.

Note for the centered style, the stroke is applied by extending the boundary of the node by a distance of half of the `strokeWidth` on either side (inside and outside) of the boundary.

The `setStrokeLineCap(capType)` method sets an end cap style for the stroke. The styles are defined as `StrokeLineCap.BUTT` (default), `StrokeLineCap.ROUND`, and `StrokeLineCap.SQUARE`, as illustrated in Figure 31.15. The **BUTT** stroke ends an unclosed path with no added decoration. The **ROUND** stroke ends an unclosed side of a path with an added half circle whose radius is half of the stroke width. The **SQUARE** stroke ends an unclosed side of a path with an added square that extends half of the stroke width.



FIGURE 31.15 (a) No decoration for a BUTT line cap. (b) A half circle is added to an unclosed path. (c) A square with half of the stroke width is extended to an unclosed path.

The `setStrokeLineJoin` method defines the decoration applied where path segments meet. You can specify three types of line join using the constants `StrokeLineJoin.MITER` (default), `StrokeLineJoin.BEVEL`, and `StrokeLineJoin.ROUND`, as shown in Figure 31.16.



FIGURE 31.16 Path segments can be joined in three ways: (a) MITER, (b) BEVEL, and (c) ROUND.

The `Shape` class has a property named `strokeDashArray` of the `ObservableList<Double>` type. This property is used to define a dashed pattern for the stroke. Alternate numbers in the list specify the lengths of the opaque and transparent segments of the dashes. For example, the list `[10.0, 20.0, 30.0, 40.0]` specifies a pattern as shown in Figure 31.17.

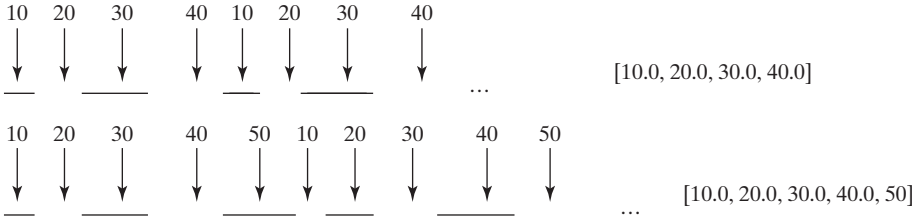


FIGURE 31.17 The numbers in the list specify the opaque and transparent segments of the stroke alternately.

The `setStrokeDashOffset(distance)` method defines the offset to the first segment in the dash pattern. Figure 31.18 illustrates the offset 5 for the dash list `[10.0, 20.0, 30.0, 40.0]`.

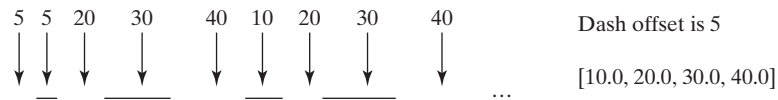


FIGURE 31.18 The dash offset specifies an offset for the first segment.

Listing 31.8 gives a program that demonstrates the methods to set attributes for a stroke. Figure 31.19 shows a sample run of the program.

LISTING 31.8 StrokeDemo.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.Rectangle;
7  import javafx.scene.shape.*;
8
9  public class StrokeDemo extends Application {
10     @Override // Override the start method in the Application class
11     public void start(Stage primaryStage) {
12         Rectangle rectangle1 = new Rectangle(20, 20, 70, 120);
13         rectangle1.setFill(Color.WHITE);
14         rectangle1.setStrokeWidth(15);
15         rectangle1.setStroke(Color.ORANGE);
16
17         Rectangle rectangle2 = new Rectangle(20, 20, 70, 120);
18         rectangle2.setFill(Color.WHITE);
19         rectangle2.setStrokeWidth(15);
20         rectangle2.setStroke(Color.ORANGE);
21         rectangle2.setTranslateX(100);
22         rectangle2.setStrokeLineJoin(StrokeLineJoin.BEVEL);
23
24         Rectangle rectangle3 = new Rectangle(20, 20, 70, 120);
25         rectangle3.setFill(Color.WHITE);
26         rectangle3.setStrokeWidth(15);
27         rectangle3.setStroke(Color.ORANGE);
28         rectangle3.setTranslateX(200);
29         rectangle3.setStrokeLineJoin(StrokeLineJoin.ROUND);
30
31         Line line1 = new Line(320, 20, 420, 20);
32         line1.setStrokeLineCap(StrokeLineCap.BUTT);
33         line1.setStrokeWidth(20);
34
35         Line line2 = new Line(320, 70, 420, 70);
36         line2.setStrokeLineCap(StrokeLineCap.ROUND);
37         line2.setStrokeWidth(20);
38
39         Line line3 = new Line(320, 120, 420, 120);
40         line3.setStrokeLineCap(StrokeLineCap.SQUARE);
41         line3.setStrokeWidth(20);
42
43         Line line4 = new Line(460, 20, 560, 120);
44         line4.getStrokeDashArray().addAll(10.0, 20.0, 30.0, 40.0);
45

```

31-16 Chapter 31 Advanced JavaFX and FXML

```
46     Pane pane = new Pane();
47     pane.getChildren().addAll(rectangle1, rectangle2, rectangle3,
48         line1, line2, line3, line4);
49
50     Scene scene = new Scene(pane, 610, 180);
51     primaryStage.setTitle("StrokeDemo"); // Set the window title
52     primaryStage.setScene(scene); // Place the scene in the window
53     primaryStage.show(); // Display the window
54 }
55
56 // Launch the program from command-line
57 public static void main(String[] args) {
58     launch(args);
59 }
60 }
```

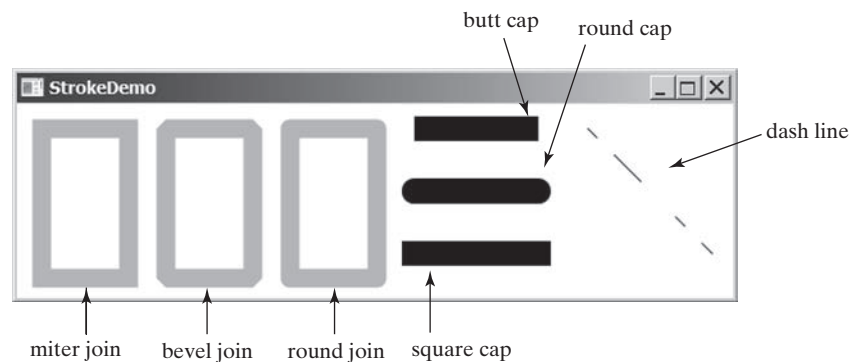


FIGURE 31.19 You can specify the attributes for strokes.

The program creates three rectangles (lines 12–29). Rectangle 1 uses default miter join, rectangle 2 uses bevel join (line 22), and rectangle 3 uses round join (line 29).

The program creates three lines with butt, round, and square end cap (lines 31–41).

The program creates a line and sets dash pattern for this line (line 44). Note the `strokeDashArray` property is of the `ObservableList<Double>` type. You have to add `Double` values to the list. Adding a number such as 10 would cause an error.



- 31.5.1** Are the methods for setting a stroke and its attributes defined in the `Node` or `Shape` class?
- 31.5.2** How do you set a stroke width to 3 pixels?
- 31.5.3** What are the stroke types? What is the default stroke type? How do you set a stroke type?
- 31.5.4** What are the stroke line join types? What is the default stroke line join type? How do you set a stroke line join type?
- 31.5.5** What are the stroke cap types? What is the default stroke cap type? How do you set a stroke cap type?
- 31.5.6** How do you specify a dashed pattern for strokes?



31.6 Menus

You can create menus in JavaFX.

Menus make selection easier and are widely used in window applications. JavaFX provides five classes that implement menus: `MenuBar`, `Menu`, `MenuItem`, `CheckMenuItem`, and `RadioButtonMenuItem`.

MenuBar is a top-level menu component used to hold the menus. A menu consists of menu items that the user can select (or toggle on or off). A menu item can be an instance of **MenuItem**, **CheckMenuItem**, or **RadioButtonMenuItem**. **Menu** items can be associated with nodes and keyboard accelerators.

31.6.1 Creating Menus

The sequence of implementing menus in JavaFX is as follows:

1. Create a menu bar and add it to a pane. For example, the following code creates a pane and a menu bar, and adds the menu bar to the pane:

```
MenuBar menuBar = new MenuBar();
Pane pane = new Pane();
pane.getChildren().add(menuBar);
```

2. Create menus and add them under the menu bar. For example, the following creates two menus and adds them to a menu bar, as shown in Figure 31.20a:

```
Menu menuFile = new Menu("File");
Menu menuHelp = new Menu("Help");
menuBar.getMenus().addAll(menuFile, menuHelp);
```

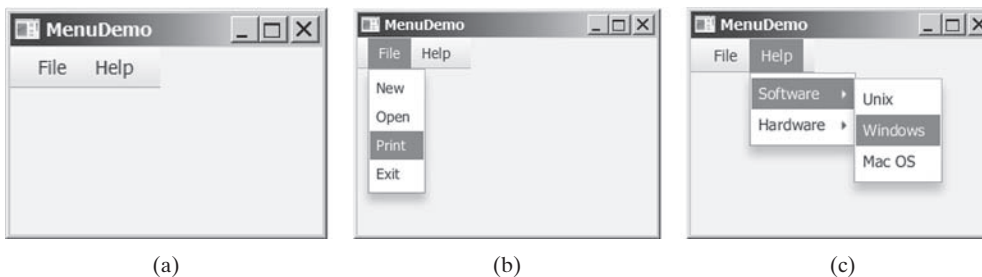


FIGURE 31.20 (a) The menus are placed under a menu bar. (b) Clicking a menu on the menu bar reveals the items under the menu. (c) Clicking a menu item reveals the submenu items under the menu item.

3. Create menu items and add them to the menus.

```
menuFile.getItems().addAll(new MenuItem("New"),
    new MenuItem("Open"), new MenuItem("Print"),
    new MenuItem("Exit"));
```

This code adds the menu items New, Open, Print, and Exit, in this order, to the File menu, as shown in Figure 31.20b.

3.1. Creating submenu items.

You can also embed menus inside menus so the embedded menus become submenus. Here is an example:

```
Menu softwareHelpSubMenu = new Menu("Software");
Menu hardwareHelpSubMenu = new Menu("Hardware");
menuHelp.getItems().add(softwareHelpSubMenu);
menuHelp.getItems().add(hardwareHelpSubMenu);
softwareHelpSubMenu.getItems().add(new MenuItem("Unix"));
softwareHelpSubMenu.getItems().add(new MenuItem("Windows"));
softwareHelpSubMenu.getItems().add(new MenuItem("Mac OS"));
```

This code adds two submenus, `softwareHelpSubMenu` and `hardwareHelpSubMenu`, in `MenuHelp`. The menu items `Unix`, `NT`, and `Win95` are added to `softwareHelpSubMenu` (see Figure 31.20c).

3.2. Creating check-box menu items.

You can also add a `CheckMenuItem` to a `Menu`. `CheckMenuItem` is a subclass of `MenuItem` that adds a Boolean state to the `MenuItem` and displays a check when its state is true. You can click a menu item to turn it on or off. For example, the following statement adds the check-box menu item `Check it` (see Figure 31.21a).

```
menuHelp.getItems().add(new CheckMenuItem("Check it"));
```

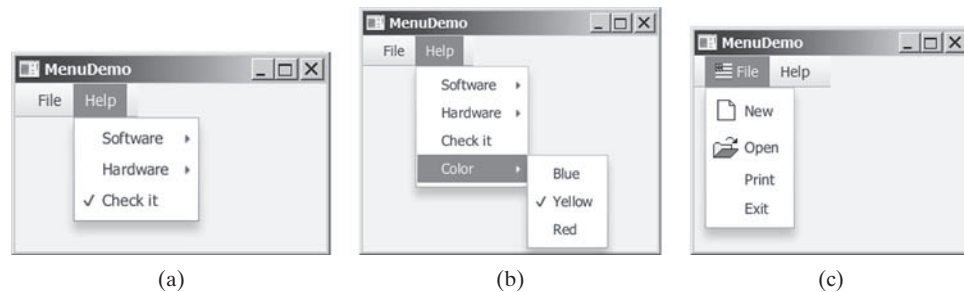


FIGURE 31.21 (a) A check box menu item lets you check or uncheck a menu item just like a check box. (b) You can use `RadioMenuItem` to choose among mutually exclusive menu choices. (c) You can set image icons and keyboard accelerators in menus.

3.3. Creating radio menu items.

You can also add radio menu items to a menu, using the `RadioMenuItem` class. This is often useful when you have a group of mutually exclusive choices in the menu. For example, the following statements add a submenu named `Color` and a set of radio buttons for choosing a color (see Figure 31.21b):

```
RadioMenuItem rmiBlue, rmiYellow, rmiRed;
colorHelpSubMenu.getItems().add(rmiBlue =
    new RadioMenuItem("Blue"));
colorHelpSubMenu.getItems().add(rmiYellow =
    new RadioMenuItem("Yellow"));
colorHelpSubMenu.getItems().add(rmiRed =
    new RadioMenuItem("Red"));

ToggleGroup group = new ToggleGroup();
rmiBlue.setToggleGroup(group);
rmiYellow.setToggleGroup(group);
rmiRed.setToggleGroup(group);
```

4. The menu items generate `ActionEvent`. To handle `ActionEvent`, implement the `setOnAction` method.

5. Image Icons and Keyboard Accelerators

The `Menu`, `CheckMenuItem`, and `RadioMenuItem` are the subclasses of `MenuItem`. The `MenuItem` has a `graphic` property for specifying a node to be displayed in the menu item. Usually, the graphic is an image view. The classes `Menu`, `MenuItem`, `CheckMenuItem`, and `RadioMenuItem` have another constructor that you can use to specify a graphic. For example, the following code adds an image to the menu, menu item, check menu item, and radio menu item (see Figure 31.21c).

```

Menu menuFile = new Menu("File",
    new ImageView("image/usIcon.gif"));
MenuItem menuItemOpen = new MenuItem("New",
    new ImageView("image/new.gif"));
CheckMenuItem checkMenuItem = new CheckMenuItem("Check it",
    new ImageView("image/us.gif"));
RadioMenuItem rmiBlue = new RadioMenuItem("Blue",
    new ImageView("image/us.gif"));

```

6. A key accelerator lets you select a menu item directly by pressing the CTRL and the accelerator key. For example, by using the following code, you can attach the accelerator key CTRL+N to the Open menu item:

```

menuItemOpen.setAccelerator(
    KeyCombination.keyCombination("Ctrl+N"));

```

31.6.2 Example: Using Menus

This section gives an example that creates a user interface to perform arithmetic. The interface contains labels and text fields for Number 1, Number 2, and Result. The Result text field displays the result of the arithmetic operation between Number 1 and Number 2. Figure 31.22 contains a sample run of the program.

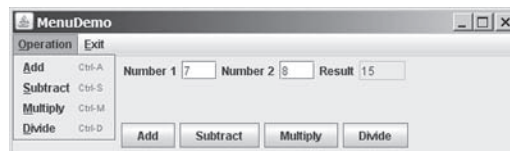


FIGURE 31.22 Arithmetic operations can be performed by clicking buttons or by choosing menu items from the Operation menu.

Here are the major steps in the program (Listing 31.9):

1. Create a menu bar and add it into a **VBox**. Create the menus Operation and Exit, and add them to the menu bar. Add the menu items Add, Subtract, Multiply, and Divide under the Operation menu and add the menu item Close under the Exit menu.
2. Create an **HBox** to hold labels and text fields and place it into the **VBox**.
3. Create an **HBox** to hold the four buttons labeled Add, Subtract, Multiply, and Divide and place it into the **VBox**.
4. Implement the handlers to process the events from the menu items and the buttons.

LISTING 31.9 MenuDemo.java

```

1 import javafx.application.Application;
2 import javafx.geometry.Pos;
3 import javafx.scene.Scene;
4 import javafx.scene.control.Button;
5 import javafx.scene.control.Label;
6 import javafx.scene.control.Menu;
7 import javafx.scene.control.MenuBar;
8 import javafx.scene.control.MenuItem;
9 import javafx.scene.control.TextField;
10 import javafx.scene.input.KeyCombination;

```

```

11 import javafx.scene.layout.HBox;
12 import javafx.scene.layout.VBox;
13 import javafx.stage.Stage;
14
15 public class MenuDemo extends Application {
16     private TextField tfNumber1 = new TextField();
17     private TextField tfNumber2 = new TextField();
18     private TextField tfResult = new TextField();
19
20     @Override // Override the start method in the Application class
21     public void start(Stage primaryStage) {
22         MenuBar menuBar = new MenuBar();
23
24         Menu menuOperation = new Menu("Operation");
25         Menu menuExit = new Menu("Exit");
26         menuBar.getMenus().addAll(menuOperation, menuExit);
27
28         MenuItem menuItemAdd = new MenuItem("Add");
29         MenuItem menuItemSubtract = new MenuItem("Subtract");
30         MenuItem menuItemMultiply = new MenuItem("Multiply");
31         MenuItem menuItemDivide = new MenuItem("Divide");
32         menuOperation.getItems().addAll(menuItemAdd, menuItemSubtract,
33             menuItemMultiply, menuItemDivide);
34
35         MenuItem menuItemClose = new MenuItem("Close");
36         menuExit.getItems().add(menuItemClose);
37
38         menuItemAdd.setAccelerator(
39             KeyCombination.keyCombination("Ctrl+A"));
40         menuItemSubtract.setAccelerator(
41             KeyCombination.keyCombination("Ctrl+S"));
42         menuItemMultiply.setAccelerator(
43             KeyCombination.keyCombination("Ctrl+M"));
44         menuItemDivide.setAccelerator(
45             KeyCombination.keyCombination("Ctrl+D"));
46
47         HBox hBox1 = new HBox(5);
48         tfNumber1.setPrefColumnCount(2);
49         tfNumber2.setPrefColumnCount(2);
50         tfResult.setPrefColumnCount(2);
51         hBox1.getChildren().addAll(new Label("Number 1:"), tfNumber1,
52             new Label("Number 2:"), tfNumber2, new Label("Result:"),
53             tfResult);
54         hBox1.setAlignment(Pos.CENTER);
55
56         HBox hBox2 = new HBox(5);
57         Button btAdd = new Button("Add");
58         Button btSubtract = new Button("Subtract");
59         Button btMultiply = new Button("Multiply");
60         Button btDivide = new Button("Divide");
61         hBox2.getChildren().addAll(btAdd, btSubtract, btMultiply, btDivide);
62         hBox2.setAlignment(Pos.CENTER);
63
64         VBox vBox = new VBox(10);
65         vBox.getChildren().addAll(menuBar, hBox1, hBox2);
66         Scene scene = new Scene(vBox, 300, 250);
67         primaryStage.setTitle("MenuDemo"); // Set the window title
68         primaryStage.setScene(scene); // Place the scene in the window
69         primaryStage.show(); // Display the window
70

```

```

71     // Handle menu actions
72     menuItemAdd.setOnAction(e -> perform('+'));
73     menuItemSubtract.setOnAction(e -> perform('-'));
74     menuItemMultiply.setOnAction(e -> perform('*'));
75     menuItemDivide.setOnAction(e -> perform('/'));
76     menuItemClose.setOnAction(e -> System.exit(0));
77
78     // Handle button actions
79     btAdd.setOnAction(e -> perform('+'));
80     btSubtract.setOnAction(e -> perform('-'));
81     btMultiply.setOnAction(e -> perform('*'));
82     btDivide.setOnAction(e -> perform('/'));
83 }
84
85 private void perform(char operator) {
86     double number1 = Double.parseDouble(tfNumber1.getText());
87     double number2 = Double.parseDouble(tfNumber2.getText());
88
89     double result = 0;
90     switch (operator) {
91         case '+': result = number1 + number2; break;
92         case '-': result = number1 - number2; break;
93         case '*': result = number1 * number2; break;
94         case '/': result = number1 / number2; break;
95     }
96
97     tfResult.setText(result + "");
98 }
100 }

```

The program creates a menu bar (line 22), which holds two menus: **menuOperation** and **menuExit** (lines 24–36). The **menuOperation** contains four menu items for doing arithmetic: Add, Subtract, Multiply, and Divide. The **menuExit** contains the menu item Close for exiting the program. The menu items in the Operation menu are created with keyboard accelerators (lines 38–45).

The labels and text fields are placed in an **HBox** (lines 47–54) and four buttons are placed in another **HBox** (lines 56–62). The menu bar and these two **HBoxes** are added to a **VBox** (line 65), which is placed in the scene (line 66).

The user enters two numbers in the number fields. When an operation is chosen from the menu, its result, involving two numbers, is displayed in the Result field. The user can also click the buttons to perform the same operation.

The program sets actions for the menu items and buttons in lines 72–82. The private method **perform(char operator)** (lines 85–98) retrieves operands from the text fields in Number 1 and Number 2, applies the binary operator on the operands, and sets the result in the Result text field.

- 31.6.1** How do you create a menu bar, menu, menu item, check menu item, and radio menu item?
- 31.6.2** How do you place a menu into a menu bar? How do you place a menu item, check menu item, and radio menu item into a menu?
- 31.6.3** Can you place a menu item into another menu item or a check menu or a radio menu item into a menu item?
- 31.6.4** How do you associate an image with a menu, menu item, check menu item, and radio menu item?
- 31.6.5** How do you associate an accelerator CTRL+O with a menu item, check menu item, and radio menu item?





31.7 Context Menus

You can create context menus in JavaFX.

A *context menu*, also known as a *popup menu*, is like a regular menu, but does not have a menu bar and can float anywhere on the screen. Creating a context menu is similar to creating a regular menu. First, you create an instance of `ContextMenu`, and then you can add `MenuItem`, `CheckMenuItem`, and `RadioMenuItem` to the context menu. For example, the following code creates a `ContextMenu`, then adds `MenuItems` into it:

```
ContextMenu contextMenu = new ContextMenu();
ContextMenu.getItems().add(new MenuItem("New"));
ContextMenu.getItems().add(new MenuItem("Open"));
```

A regular menu is always added to a menu bar, but a context menu is associated with a parent node and is displayed using the `show` method in the `ContextMenu` class. You specify the parent node and the location of the context menu, using the coordinate system of the parent like this:

```
contextMenu.show(node, x, y);
```

Customarily, you display a context menu by pointing to a GUI component and clicking a certain mouse button, the so-called popup trigger. Popup triggers are system dependent. In Windows, the context menu is displayed when the right mouse button is released. In Motif, the context menu is displayed when the third mouse button is pressed and held down.

Listing 31.10 gives an example that creates a pane. When the mouse points to the pane, clicking a mouse button displays a context menu, as shown in Figure 31.23.

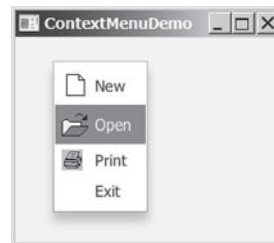


FIGURE 31.23 A context menu is displayed when the popup trigger is issued on the pane.

Here are the major steps in the program (Listing 31.10):

1. Create a context menu using `ContextMenu`. Create menu items for New, Open, Print, and Exit using `MenuItem`.
2. Add the menu items into the context menu.
3. Create a pane and place it in the scene.
4. Implement the handler to process the events from the menu items.
5. Implement the `mousePressed` handler to display the context menu.

LISTING 31.10 ContextMenuDemo.java

```
1 import javafx.application.Application;
2 import javafx.scene.Scene;
3 import javafx.scene.control.ContextMenu;
4 import javafx.scene.control.MenuItem;
5 import javafx.scene.image.ImageView;
6 import javafx.scene.layout.Pane;
7 import javafx.stage.Stage;
8
```

```

9  public class ContextMenuDemo extends Application {
10     @Override // Override the start method in the Application class
11     public void start(Stage primaryStage) {
12         ContextMenu contextMenu = new ContextMenu();
13         MenuItem menuItemNew = new MenuItem("New",
14             new ImageView("image/new.gif"));
15         MenuItem menuItemOpen = new MenuItem("Open",
16             new ImageView("image/open.gif"));
17         MenuItem menuItemPrint = new MenuItem("Print",
18             new ImageView("image/print.gif"));
19         MenuItem menuItemExit = new MenuItem("Exit");
20         contextMenu.getItems().addAll(menuItemNew, menuItemOpen,
21             menuItemPrint, menuItemExit);
22
23         Pane pane = new Pane();
24         Scene scene = new Scene(pane, 300, 250);
25         primaryStage.setTitle("ContextMenuDemo"); // Set the window title
26         primaryStage.setScene(scene); // Place the scene in the window
27         primaryStage.show(); // Display the window
28
29         pane.setOnMousePressed(
30             e -> contextMenu.show(pane, e.getScreenX(), e.getScreenY()));
31
32         menuItemNew.setOnAction(e -> System.out.println("New"));
33         menuItemOpen.setOnAction(e -> System.out.println("Open"));
34         menuItemPrint.setOnAction(e -> System.out.println("Print"));
35         menuItemExit.setOnAction(e -> System.exit(0));
36     }
37 }

```

The process of creating context menus is similar to the process for creating regular menus. To create a context menu, create a **ContextMenu** as the basis (line 12) and add **MenuItems** to it (lines 13–21).

To show a context menu, use the show method by specifying the parent node and the location for the context menu (lines 29 and 30). The show method is invoked when the context menu is triggered by a mouse click on the pane (line 30).

31.7.1 How do you create a context menu? How do you add menu items, check menu items, and radio menu items into a context menu?



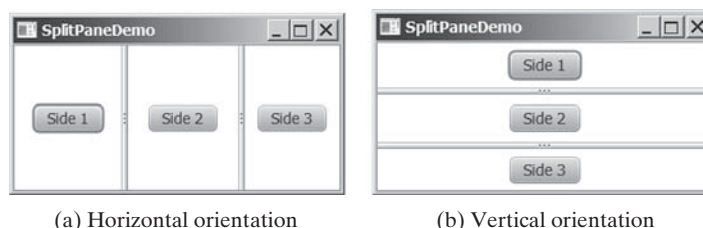
31.7.2 How do you show a context menu?

31.8 SplitPane

The **SplitPane** class can be used to display multiple panes and allow the user to adjust the size of the panes.



The **SplitPane** is a control that contains two components with a separate bar known as a divider, as shown in Figure 31.24.



(a) Horizontal orientation

(b) Vertical orientation

FIGURE 31.24 **SplitPane** divides a container into two parts.

31-24 Chapter 31 Advanced JavaFX and FXML

The two sides separated by the divider can appear in horizontal or vertical orientation. The divider separating two sides can be dragged to change the amount of space occupied by each side. Figure 31.25 shows the frequently used properties, constructors, and methods in `SplitPane`.

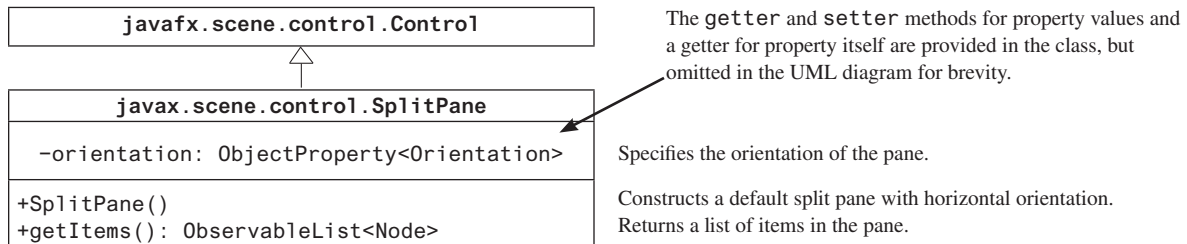


FIGURE 31.25 `SplitPane` provides methods to specify the properties of a split pane and for manipulating the components in a split pane.

Listing 31.11 gives an example that uses radio buttons to let the user select a country and displays the country's flag and description in separate sides, as shown in Figure 31.26. The description of the currently selected layout manager is displayed in a text area. The radio buttons, buttons, and text area are placed in two split panes.

LISTING 31.11 `SplitPaneDemo.java`

```
1 import javafx.application.Application;
2 import javafx.geometry.Orientation;
3 import javafx.scene.Scene;
4 import javafx.scene.control.RadioButton;
5 import javafx.scene.control.ScrollPane;
6 import javafx.scene.control.SplitPane;
7 import javafx.scene.control.TextArea;
8 import javafx.scene.control.ToggleGroup;
9 import javafx.scene.image.Image;
10 import javafx.scene.image.ImageView;
11 import javafx.scene.layout.StackPane;
12 import javafx.scene.layout.VBox;
13 import javafx.stage.Stage;
14
15 public class SplitPaneDemo extends Application {
16     private Image usImage = new Image(
17         "http://www.cs.armstrong.edu/liang/common/image/us.gif");
18     private Image ukImage = new Image(
19         "http://www.cs.armstrong.edu/liang/common/image/uk.gif");
20     private Image caImage = new Image(
21         "http://www.cs.armstrong.edu/liang/common/image/ca.gif");
22     private String usDescription = "Description for US ...";
23     private String ukDescription = "Description for UK ...";
24     private String caDescription = "Description for CA ...";
25
26     @Override // Override the start method in the Application class
27     public void start(Stage primaryStage) {
28         VBox vbox = new VBox(10);
29         RadioButton rbUS = new RadioButton("US");
```

```

30  RadioButton rbUK = new RadioButton("UK");
31  RadioButton rbCA = new RadioButton("CA");
32  vbox.getChildren().addAll(rbUS, rbUK, rbCA);
33
34  SplitPane content = new SplitPane();
35  content.setOrientation(Orientation.VERTICAL);
36  ImageView imageView = new ImageView(usImage);
37  StackPane imagePane = new StackPane();
38  imagePane.getChildren().add(imageView);
39  TextArea taDescription = new TextArea();
40  taDescription.setText(usDescription);
41  content.getItems().addAll(
42      imagePane, new ScrollPane(taDescription));
43
44  SplitPane sp = new SplitPane();
45  sp.getItems().addAll(vBox, content);
46
47  Scene scene = new Scene(sp, 300, 250);
48  primaryStage.setTitle("SplitPaneDemo"); // Set the window title
49  primaryStage.setScene(scene); // Place the scene in the window
50  primaryStage.show(); // Display the window
51
52  // Group radio buttons
53  ToggleGroup group = new ToggleGroup();
54  rbUS.setToggleGroup(group);
55  rbUK.setToggleGroup(group);
56  rbCA.setToggleGroup(group);
57
58  rbUS.setSelected(true);
59  rbUS.setOnAction(e -> {
60      imageView.setImage(usImage);
61      taDescription.setText(usDescription);
62  });
63
64  rbUK.setOnAction(e -> {
65      imageView.setImage(ukImage);
66      taDescription.setText(ukDescription);
67  });
68
69  rbCA.setOnAction(e -> {
70      imageView.setImage(caImage);
71      taDescription.setText(caDescription);
72  });
73  }
74  }

```

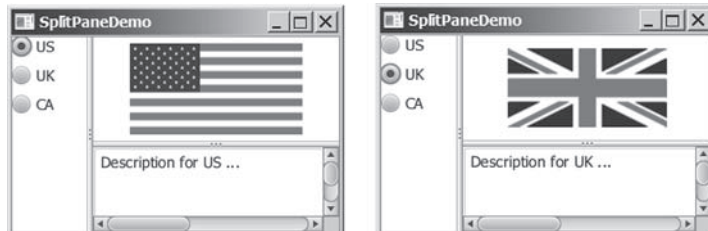


FIGURE 31.26 You can adjust the component size in the split panes.

The program places three radio buttons in a **VBox** (lines 28–32) and creates a vertical split pane for holding an image view and a text area (lines 34–42). Split panes can be embedded. The program creates a horizontal split pane and places the **VBox** and the vertical split pane into it (lines 44 and 45).

Adding a split pane to an existing split pane results in three split panes. The program creates two split panes (lines 34, 42) to hold a panel for radio buttons, a panel for buttons, and a scroll pane.

The program groups radio buttons (lines 53–56) and processes the action for radio buttons (lines 59–72).



- 31.8.1** How do you create a horizontal **SplitPane**? How do you create a vertical **SplitPane**?
- 31.8.2** How do you add items into a **SplitPane**? Can you add a **SplitPane** to another **SplitPane**?



31.9 TabPane

*The **TabPane** class can be used to display multiple panes with tabs.*

TabPane is a useful control that provides a set of mutually exclusive tabs, as shown in Figure 31.27. You can switch between a group of tabs. Only one tab is visible at a time. A **Tab** can be added to a **TabPane**. Tabs in a **TabPane** can be placed in the position top, left, bottom, or right.

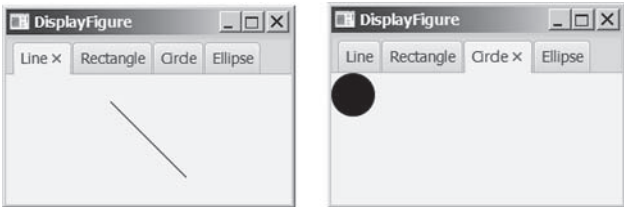


FIGURE 31.27 **TabPane** holds a group of tabs.

Each tab represents a single page. Tabs are defined in the **Tab** class. Tabs can contain any **Node** such as a pane, a shape, or a control. A tab can contain another pane. Therefore, you can create a multilayered tab pane. Figures 31.28 and 31.29 show the frequently used properties, constructors, and methods in **TabPane** and **Tab**.

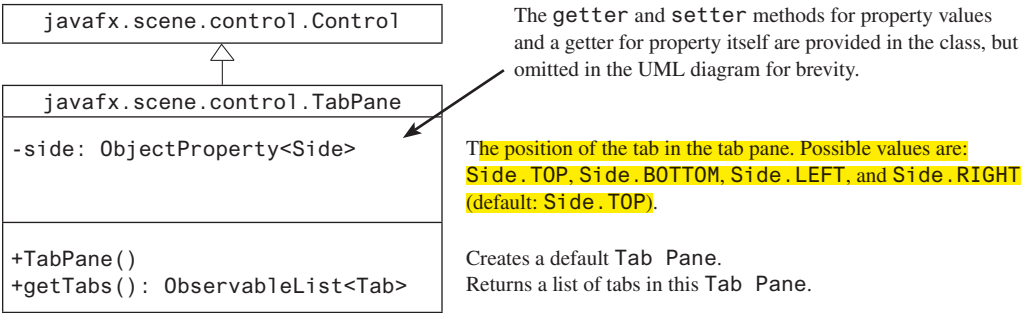


FIGURE 31.28 **TabPane** displays and manages the tabs.

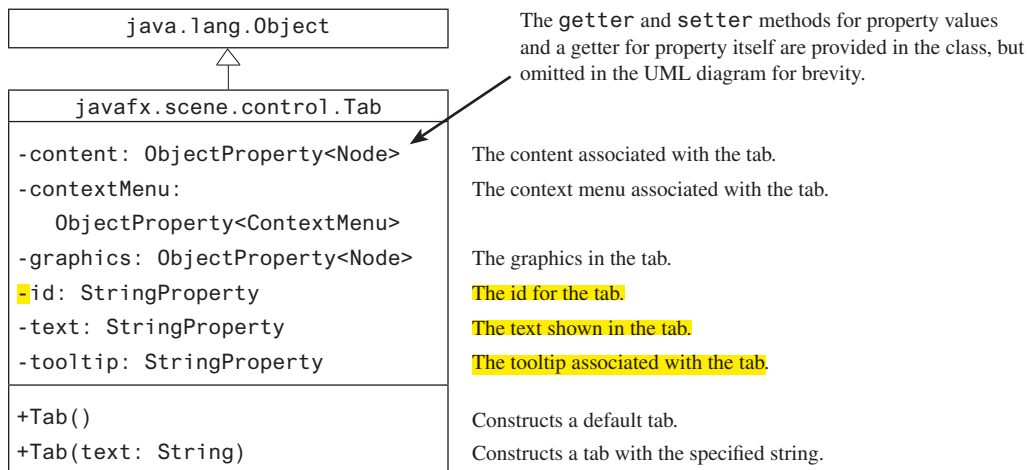


FIGURE 31.29 Tab contains a node.

Listing 31.12 gives an example that uses a tab pane with four tabs to display four types of figures: line, rectangle, rounded rectangle, and oval. You can select a figure to display by clicking the corresponding tab, as shown in Figure 31.27.

LISTING 31.12 TabPaneDemo.java

```

1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Tab;
4  import javafx.scene.control.TabPane;
5  import javafx.scene.layout.StackPane;
6  import javafx.scene.shape.Circle;
7  import javafx.scene.shape.Ellipse;
8  import javafx.scene.shape.Line;
9  import javafx.scene.shape.Rectangle;
10 import javafx.stage.Stage;
11
12 public class TabPaneDemo extends Application {
13     @Override // Override the start method in the Application class
14     public void start(Stage primaryStage) {
15         TabPane tabPane = new TabPane();
16         Tab tab1 = new Tab("Line");
17         StackPane pane1 = new StackPane();
18         pane1.getChildren().add(new Line(10, 10, 80, 80));
19         tab1.setContent(pane1);
20         Tab tab2 = new Tab("Rectangle");
21         tab2.setContent(new Rectangle(10, 10, 200, 200));
22         Tab tab3 = new Tab("Circle");
23         tab3.setContent(new Circle(50, 50, 20));
24         Tab tab4 = new Tab("Ellipse");
25         tab4.setContent(new Ellipse(10, 10, 100, 80));
26         tabPane.getTabs().addAll(tab1, tab2, tab3, tab4);
27
28         Scene scene = new Scene(tabPane, 300, 250);
29         primaryStage.setTitle("DisplayFigure"); // Set the window title
30         primaryStage.setScene(scene); // Place the scene in the window
31         primaryStage.show(); // Display the window
32     }
33 }
  
```

The program creates a tab pane (line 15) and four tabs (lines 16, 20, 22, and 24). A stack pane is created to hold a line (line 18) and placed into `tab1` (line 19). A rectangle, circle, and oval are created and placed into `tab2`, `tab3`, and `tab4`. Note the line is centered in `tab1` because it is placed in a stack pane. The other shapes are directly placed into the tab. They are displayed at the upper left corner of the tab.

By default, the tabs are placed at the top of the tab pane. You can use the `setSide` method to change its location.



- 31.9.1** How do you create a tab pane? How do you create a tab? How do you add a tab to a tab pane?
- 31.9.2** How do you place the tabs on the left of the tab pane?
- 31.9.3** Can a tab have a text as well as an image? Write the code to set an image for `tab1` in Listing 31.12.



31.10 TableView

You can display tables using the `TableView` class.

`TableView` is a control that displays data in rows and columns in a two-dimensional grid, as shown in Figure 31.30.

TableViewDemo			
Country	Capital	Population (million)	Is Democratic?
USA	Washington DC	280.0	true
Canada	Ottawa	32.0	true
United Kingdom	London	60.0	true
Germany	Berlin	83.0	true
France	Paris	60.0	true

FIGURE 31.30 `TableView` displays data in a table.

`TableView`, `TableColumn`, and `TableCell` are used to display and manipulate a table. `TableView` displays a table. `TableColumn` defines the columns in a table. `TableCell` represents a cell in the table. Creating a `TableView` is a multistep process. First, you need to create an instance of `TableView` and associate data with the `TableView`. Second, you need to create columns using the `TableColumn` class and set a column cell value factory to specify how to populate all cells within a single `TableColumn`.

Listing 31.13 gives a simple example to demonstrate using `TableView` and `TableColumn`. A sample run of the program is shown in Figure 31.31.

LISTING 31.13 `TableViewDemo.java`

```
1 import javafx.application.Application;
2 import javafx.beans.property.SimpleBooleanProperty;
3 import javafx.beans.property.SimpleDoubleProperty;
4 import javafx.beans.property.SimpleStringProperty;
5 import javafx.collections.FXCollections;
6 import javafx.collections.ObservableList;
7 import javafx.scene.Scene;
8 import javafx.scene.control.TableColumn;
9 import javafx.scene.control.TableView;
10 import javafx.scene.control.cell.PropertyValueFactory;
11 import javafx.scene.layout.Pane;
12 import javafx.stage.Stage;
```

```

13
14 public class TableViewDemo extends Application {
15     @Override // Override the start method in the Application class
16     public void start(Stage primaryStage) {
17         TableView<Country> tableView = new TableView<>();
18         ObservableList<Country> data =
19             FXCollections.observableArrayList(
20                 new Country("USA", "Washington DC", 280, true),
21                 new Country("Canada", "Ottawa", 32, true),
22                 new Country("United Kingdom", "London", 60, true),
23                 new Country("Germany", "Berlin", 83, true),
24                 new Country("France", "Paris", 60, true));
25         tableView.setItems(data);
26
27         TableColumn countryColumn = new TableColumn("Country");
28         countryColumn.setMinWidth(100);
29         countryColumn.setCellValueFactory(
30             new PropertyValueFactory<Country, String>("country"));
31
32         TableColumn capitalColumn = new TableColumn("Capital");
33         capitalColumn.setMinWidth(100);
34         capitalColumn.setCellValueFactory(
35             new PropertyValueFactory<Country, String>("capital"));
36
37         TableColumn populationColumn =
38             new TableColumn("Population (million)");
39         populationColumn.setMinWidth(200);
40         populationColumn.setCellValueFactory(
41             new PropertyValueFactory<Country, Double>("population"));
42
43         TableColumn democraticColumn =
44             new TableColumn("Is Democratic?");
45         democraticColumn.setMinWidth(200);
46         democraticColumn.setCellValueFactory(
47             new PropertyValueFactory<Country, Boolean>("democratic"));
48
49         tableView.getColumns().addAll(countryColumn, capitalColumn,
50             populationColumn, democraticColumn);
51
52         Pane pane = new Pane();
53         pane.getChildren().add(tableView);
54         Scene scene = new Scene(pane, 300, 250);
55         primaryStage.setTitle("TableViewDemo"); // Set the window title
56         primaryStage.setScene(scene); // Place the scene in the window
57         primaryStage.show(); // Display the window
58     }
59
60     public static class Country {
61         private final SimpleStringProperty country;
62         private final SimpleStringProperty capital;
63         private final SimpleDoubleProperty population;
64         private final SimpleBooleanProperty democratic;
65
66         private Country(String country, String capital,
67             double population, boolean democratic) {
68             this.country = new SimpleStringProperty(country);
69             this.capital = new SimpleStringProperty(capital);
70             this.population = new SimpleDoubleProperty(population);
71             this.democratic = new SimpleBooleanProperty(democratic);
72         }

```

```

73
74     public String getCountry() {
75         return country.get();
76     }
77
78     public void setCountry(String country) {
79         this.country.set(country);
80     }
81
82     public String getCapital() {
83         return capital.get();
84     }
85
86     public void setCapital(String capital) {
87         this.capital.set(capital);
88     }
89
90     public double getPopulation() {
91         return population.get();
92     }
93
94     public void setPopulation(double population) {
95         this.population.set(population);
96     }
97
98     public boolean isDemocratic() {
99         return democratic.get();
100    }
101
102    public void setDemocratic(boolean democratic) {
103        this.democratic.set(democratic);
104    }
105 }
106 }

```

The program creates a **TableView** (line 17). The **TableView** class is a generic class whose concrete type is **Country**. Therefore, this **TableView** is for displaying **Country**. The table data is an **ObservableList<Country>**. The program creates the list (lines 18–24) and associates the list with the **TableView** (line 25).

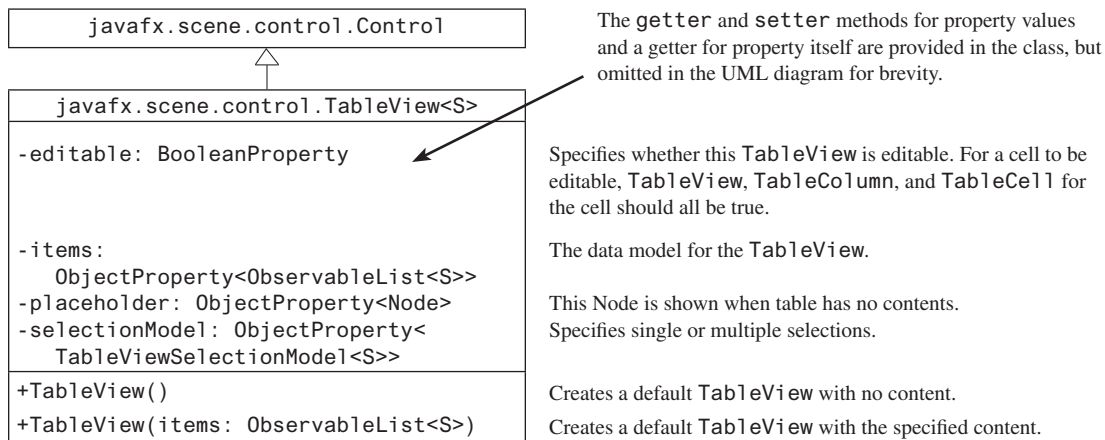
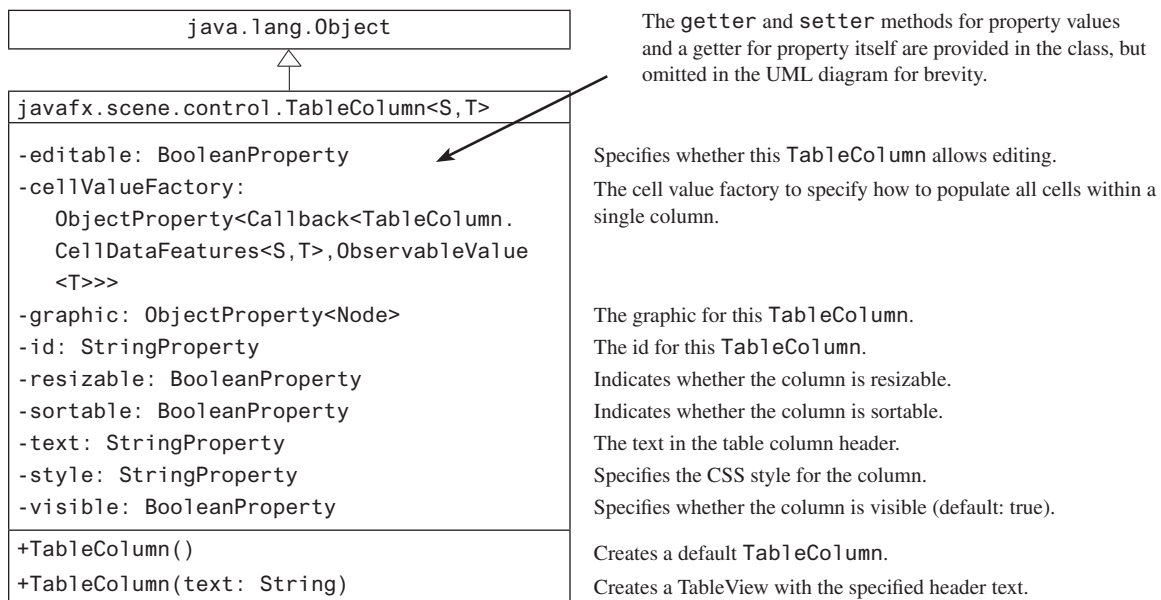
The program creates a **TableColumn** for each column in the table (lines 27–47). A **PropertyValueFactory** object is created and set for each column (line 30). This object is used to populate the data in the column. The **PropertyValueFactory<S, T>** class is a generic class. **S** is for the class displayed in the **TableView** and **T** is the class for the values in the column. The **PropertyValueFactory** object associates a property in class **S** with a column.

When you create a table in a JavaFX application, it is a best practice to define the data model in a class. The **Country** class defines the data for **TableView**. Each property in the class defines a column in the table. This property should be defined as binding property with the getter and setter methods for the value.

The program adds the columns into the **TableView** (lines 49 and 50), adds the **TableView** in a pane (line 53), and places the pane in the scene (line 54). Note line 31 can be simplified using the following code:

```
new PropertyValueFactory<>("country");
```

From this example, you see how to display data in a table using the **TableView** and **TableColumn** classes. The frequently used properties and methods for the **TableView** and **TableColumn** classes are given in Figures 31.32 and 31.33.

FIGURE 31.31 **TableView** displays a table.FIGURE 31.32 **TableColumn** defines a column in the **TableView**.

You can create nested columns. For example, the following code creates two subcolumns under **Location**, as shown in Figures 31.33.

```

TableColumn locationColumn = new TableColumn("Location");
locationColumn.getColumns().addAll(new TableColumn("latitude"),
    new TableColumn("longitude"));
  
```

TableViewDemo					
Country	Capital	Population...	Is Democr...	Location	
				latitude	longitude
USA	Washington DC	280.0	true		
Canada	Ottawa	32.0	true		
United Kingdom	London	60.0	true		
Germany	Berlin	83.0	true		
France	Paris	60.0	true		

FIGURE 31.33 You can add subcolumns in a column.

The `TableView` data model is an observable list. When data is changed, the change is automatically shown in the table. Listing 31.14 gives an example that lets the user add new rows to the table.

LISTING 31.14 AddNewRowDemo.java

```

1  import javafx.application.Application;
2  import javafx.beans.property.SimpleBooleanProperty;
3  import javafx.beans.property.SimpleDoubleProperty;
4  import javafx.beans.property.SimpleStringProperty;
5  import javafx.collections.FXCollections;
6  import javafx.collections.ObservableList;
7  import javafx.scene.Scene;
8  import javafx.scene.control.Button;
9  import javafx.scene.control.CheckBox;
10 import javafx.scene.control.Label;
11 import javafx.scene.control.TableColumn;
12 import javafx.scene.control.TableView;
13 import javafx.scene.control.TextField;
14 import javafx.scene.control.cell.PropertyValueFactory;
15 import javafx.scene.layout.BorderPane;
16 import javafx.scene.layout.FlowPane;
17 import javafx.stage.Stage;
18
19 public class AddNewRowDemo extends Application {
20     @Override // Override the start method in the Application class
21     public void start(Stage primaryStage) {
22         TableView<Country> tableView = new TableView<>();
23         ObservableList<Country> data =
24             FXCollections.observableArrayList(
25                 new Country("USA", "Washington DC", 280, true),
26                 new Country("Canada", "Ottawa", 32, true),
27                 new Country("United Kingdom", "London", 60, true),
28                 new Country("Germany", "Berlin", 83, true),
29                 new Country("France", "Paris", 60, true));
30         tableView.setItems(data);
31
32         TableColumn countryColumn = new TableColumn("Country");
33         countryColumn.setMinWidth(100);
34         countryColumn.setCellValueFactory(
35             new PropertyValueFactory<Country, String>("country"));
36
37         TableColumn capitalColumn = new TableColumn("Capital");
38         capitalColumn.setMinWidth(100);
39         capitalColumn.setCellValueFactory(
40             new PropertyValueFactory<Country, String>("capital"));
41
42         TableColumn populationColumn =
43             new TableColumn("Population (million)");
44         populationColumn.setMinWidth(100);
45         populationColumn.setCellValueFactory(
46             new PropertyValueFactory<Country, Double>("population"));
47
48         TableColumn democraticColumn =
49             new TableColumn("Is Democratic?");
50         democraticColumn.setMinWidth(100);
51         democraticColumn.setCellValueFactory(
52             new PropertyValueFactory<Country, Boolean>("democratic"));
53
54         tableView.getColumns().addAll(countryColumn, capitalColumn,

```

```

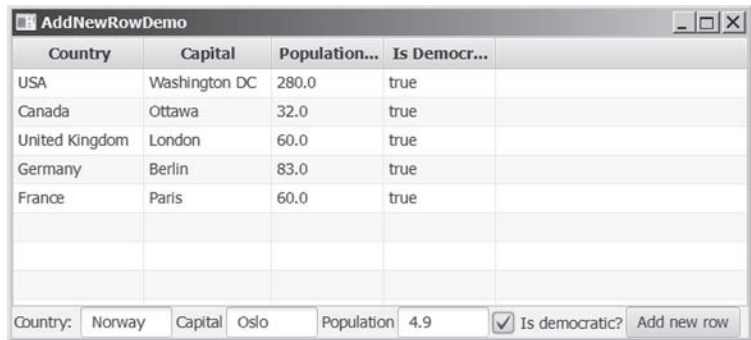
55     populationColumn, democraticColumn);
56
57     FlowPane flowPane = new FlowPane(3, 3);
58     TextField tfCountry = new TextField();
59     TextField tfCapital = new TextField();
60     TextField tfPopulation = new TextField();
61     CheckBox chkDemocratic = new CheckBox("Is democratic?");
62     Button btAddRow = new Button("Add new row");
63     tfCountry.setPrefColumnCount(5);
64     tfCapital.setPrefColumnCount(5);
65     tfPopulation.setPrefColumnCount(5);
66     flowPane.getChildren().addAll(new Label("Country: "),
67     tfCountry, new Label("Capital"), tfCapital,
68     new Label("Population"), tfPopulation, chkDemocratic,
69     btAddRow);
70
71     btAddRow.setOnAction(e -> {
72         data.add(new Country(tfCountry.getText(), tfCapital.getText(),
73             Double.parseDouble(tfPopulation.getText()),
74             chkDemocratic.isSelected()));
75         tfCountry.clear();
76         tfCapital.clear();
77         tfPopulation.clear();
78     });
79
80     BorderPane pane = new BorderPane();
81     pane.setCenter(tableView);
82     pane.setBottom(flowPane);
83
84     Scene scene = new Scene(pane, 500, 250);
85     primaryStage.setTitle("AddNewRowDemo"); // Set the window title
86     primaryStage.setScene(scene); // Place the scene in the window
87     primaryStage.show(); // Display the window
88 }
89
90 public static class Country {
91     private final SimpleStringProperty country;
92     private final SimpleStringProperty capital;
93     private final SimpleDoubleProperty population;
94     private final SimpleBooleanProperty democratic;
95
96     private Country(String country, String capital,
97         double population, boolean democratic) {
98         this.country = new SimpleStringProperty(country);
99         this.capital = new SimpleStringProperty(capital);
100        this.population = new SimpleDoubleProperty(population);
101        this.democratic = new SimpleBooleanProperty(democratic);
102    }
103
104    public String getCountry() {
105        return country.get();
106    }
107
108    public void setCountry(String country) {
109        this.country.set(country);
110    }
111
112    public String getCapital() {
113        return capital.get();
114    }

```

```
115
116     public void setCapital(String capital) {
117         this.capital.set(capital);
118     }
119
120     public double getPopulation() {
121         return population.get();
122     }
123
124     public void setPopulation(double population) {
125         this.population.set(population);
126     }
127
128     public boolean isDemocratic() {
129         return democratic.get();
130     }
131
132     public void setDemocratic(boolean democratic) {
133         this.democratic.set(democratic);
134     }
135 }
136 }
```

The program is the same in Listing 31.13 except that a new code is added to let the user enter a new row (lines 57–82). The user enters the new row from the text fields and a check box and presses the *Add New Row* button to add a new row to the data. Since data is an observable list, the change in data is automatically updated in the table.

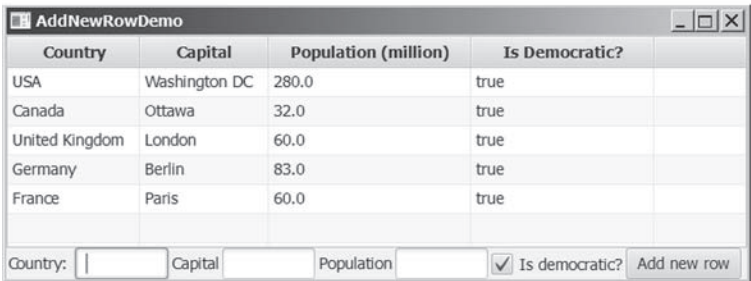
As shown in Figure 31.34a, a new country information is entered in the text fields. After clicking the *Add New Row* button, the new country is displayed in the table view.



Country	Capital	Population...	Is Democr...
USA	Washington DC	280.0	true
Canada	Ottawa	32.0	true
United Kingdom	London	60.0	true
Germany	Berlin	83.0	true
France	Paris	60.0	true

Country: Norway Capital: Oslo Population: 4.9 ☒ Is democratic? Add new row

(a)



Country	Capital	Population (million)	Is Democratic?
USA	Washington DC	280.0	true
Canada	Ottawa	32.0	true
United Kingdom	London	60.0	true
Germany	Berlin	83.0	true
France	Paris	60.0	true

Country: Capital: Population: ☒ Is democratic? Add new row

(b)

FIGURE 31.34 Change in the table data model is automatically displayed in the table view.

TableView not only displays data, but also allows data to be edited. To enable data editing in the table, write the code as follows:

1. Set the **TableView**'s **editable** to true.
2. Set the column's cell factory to a text field table cell.
3. Implement the column's **setOnEditCommit** method to assign the edited value to the data model.

Here is the example of enabling editing for the **countryColumn**.

```
tableView.setEditable(true);
countryColumn.setCellFactory(TextFieldTableCell.forTableColumn());
countryColumn.setOnEditCommit(
    new EventHandler<CellEditEvent<Country, String>>() {
        @Override
        public void handle(CellEditEvent<Country, String> t) {
            t.getTableView().getItems().get(
                t.getTablePosition().getRow())
                .setCountry(t.getNewValue());
        }
    }
);
```

31.10.1 How do you create a table view? How do you create a table column? How do you add a table column to a table view?



31.10.2 What is the data type for a **TableView**'s data model? How do you associate a data model with a **TableView**?

31.10.3 How do you set a cell value factory for a **TableColumn**?

31.10.4 How do you set an image in a table column header?

31.11 Developing JavaFX Programs Using FXML

You can create JavaFX user interfaces using FXML scripts.

There are two ways to develop JavaFX applications. One way is to write everything in Java source code as you have done so far. The other way is to use FXML. FXML is an XML-based script language for describing the user interface. Using FXML enables you to separate user interface from the logic of the Java code. JavaFX Scene Builder is a visual design tool for creating the user interface without manually writing the FXML script. You drag and drop the UI components to the content pane and set properties for the components in the Inspector. The Scene Builder automatically generates the FXML scripts for the user interface. This section demonstrates how to use the Scene Builder to create JavaFX applications.



NOTE

It is important that you first learn how to write the JavaFX code without using FXML to grasp the fundamentals of JavaFX programming before learning FXML. Once you understand the basics of JavaFX, it is easy to create JavaFX programs using FXML. For this reason, FXML is introduced after you have learned the basics of JavaFX programming.

31.11.1 Installing JavaFX Scene Builder

You can use the JavaFX Scene Builder standalone or with an IDE such as NetBeans or Eclipse. This section demonstrates using the JavaFX Scene Builder with NetBeans. You can download the latest version of Scene Builder from <http://gluonhq.com/open-source/scene-builder/>.

31.11.2 Creating a JavaFX FXML Project

To use JavaFX FXML, you need to create a JavaFX FXML in NetBeans. Here are the steps of creating a JavaFX FXML project:

1. Choose *File, New Project* to display the New Project dialog box, as shown in Figure 31.35.
2. Choose *JavaFX* in the Categories and *JavaFX FXML Application* in the Projects. Click *Next* to display the New JavaFX Application dialog box, as shown in Figure 31.36.
3. Enter *Calculator* as the project name and click *Finish* to create the project. You will see the project created as shown in Figure 31.37.

Three files, `Calculator.java`, `FXMLDocument.fxml`, and `FXMLDocumentController.java`, are created in the project. Their source codes are shown in Listings 31.15, 31.16, and 31.17. From the perspective of the MVC architecture, these three files correspond to model, view, and controller. You can define data model in the `Calculator.java` class. The `.fxml` file describes the user interface. The controller file defines the actions for processing the events for the user interface.

LISTING 31.15 Calculator.java

```

1  package calculator;
2
3  import javafx.application.Application;
4  import javafx.fxml.FXMLLoader;
5  import javafx.scene.Parent;
6  import javafx.scene.Scene;
7  import javafx.stage.Stage;
8
9  public class Calculator extends Application {
10     @Override
11     public void start(Stage stage) throws Exception {
12         Parent root =
13             FXMLLoader.load(getClass().getResource("FXMLDocument.fxml"));
14         Scene scene = new Scene(root);
15         stage.setScene(scene);
16         stage.show();
17     }
18
19     /**
20      * @param args the command line arguments
21      */
22     public static void main(String[] args) {
23         launch(args);
24     }
25 }

```

LISTING 31.16 FXMLDocument.fxml

```

<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<AnchorPane id="AnchorPane" prefHeight="200" prefWidth="320"
    xmlns:fx="http://javafx.com/fxml/1"

```

```

        fx:controller="calculator.FXMLDocumentController">
<children>
    <Button layoutX="126" layoutY="90" text="Click Me!"
        onAction="#handleButtonAction" fx:id="button" />
    <Label layoutX="126" layoutY="120" minHeight="16" minWidth="69"
        fx:id="label" />
</children>
</AnchorPane>

```

LISTING 31.17 FXMLDocumentController.java

```

package calculator;

import java.net.URL;
import java.util.ResourceBundle;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Label;

public class FXMLDocumentController implements Initializable {
    @FXML
    private Label label;

    @FXML
    private void handleButtonAction(ActionEvent event) {
        System.out.println("You clicked me!");
        label.setText("Hello World!");
    }

    @Override
    public void initialize(URL url, ResourceBundle rb) {
        // TODO
    }
}

```

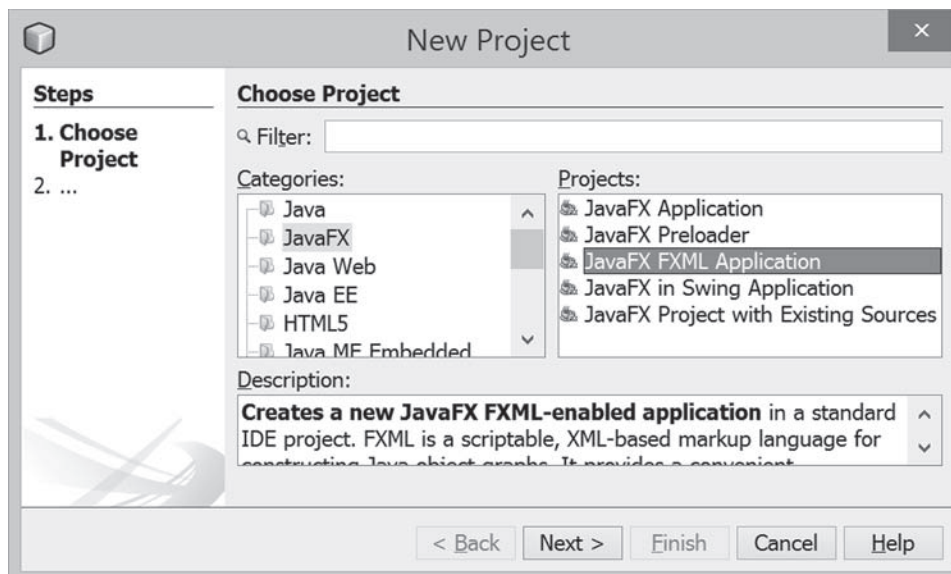


FIGURE 31.35 You can choose JavaFX in the Categories and JavaFX FXML Application in the Project to create a FXML project.

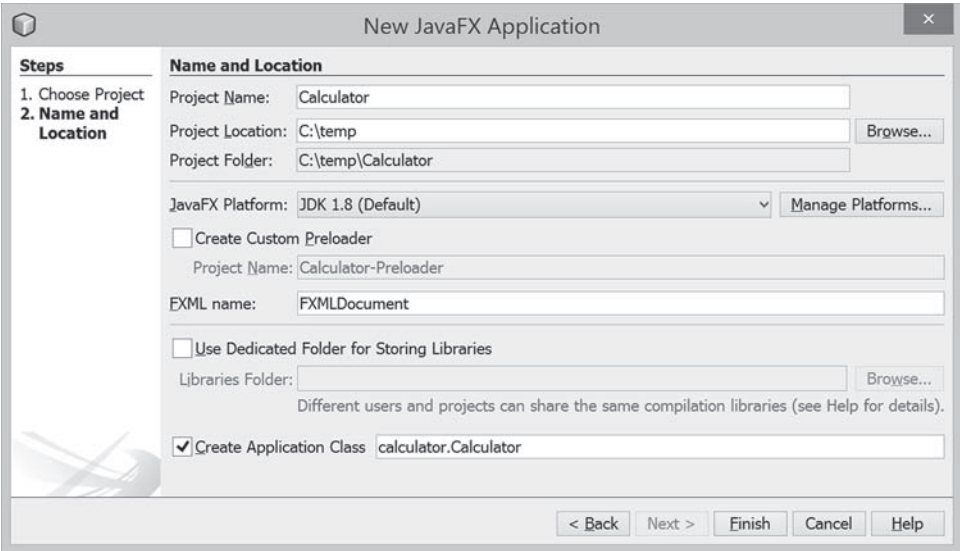


FIGURE 31.36 You can enter project information in the New JavaFX Application dialog.

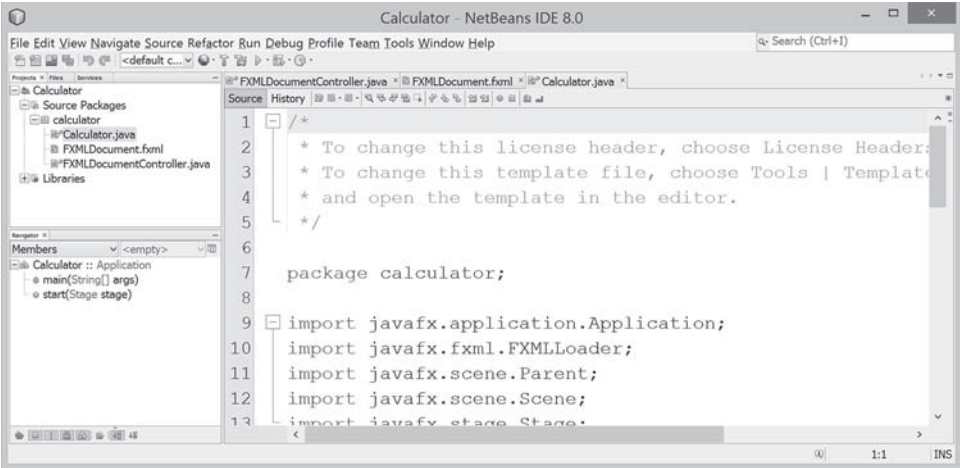


FIGURE 31.37 A FXML project is created.

31.11.3 Creating User Interfaces

We now turn our attention to developing a simple calculator as shown in Figure 31.38. The Calculator program enables the user to enter numbers and perform addition, subtraction, multiplication, and division.



FIGURE 31.38 The application performs arithmetic operations.

When you create a JavaFX FXML project, NetBeans creates a default .fxml file that contains the contents for a simple sample user interface. To view the user interface, double-click the .fxml file to open the Scene Builder, as shown in Figure 31.39. Note NetBeans can automatically detect the Scene Builder after it is installed on your machine.

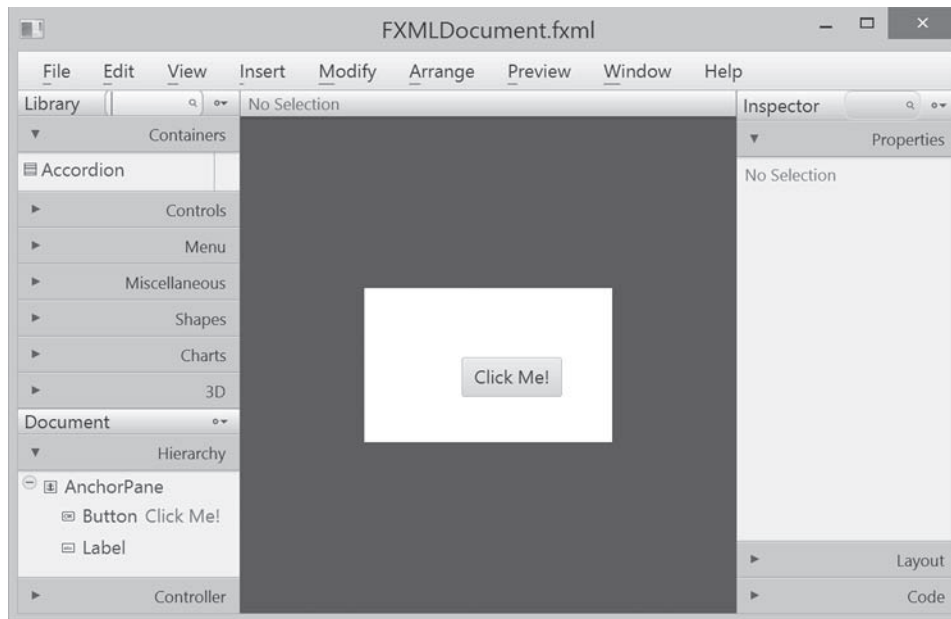


FIGURE 31.39 Double-click the .fxml file to open the Scene Builder.

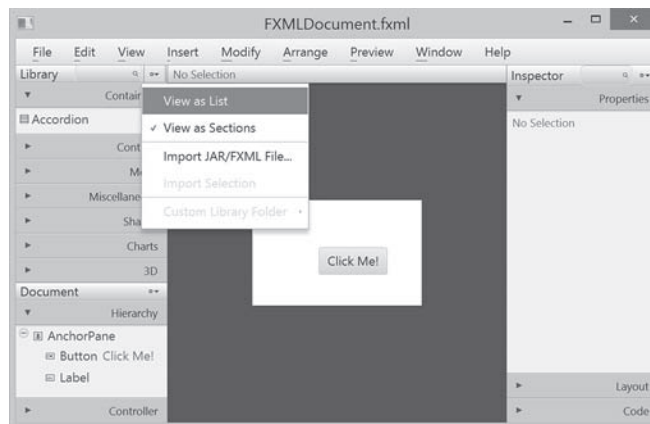


FIGURE 31.40 You can open the Library pane by clicking the Library icon and choose View as List.

To start a new user interface, delete the default user interface in the .fxml file from the content pane, as shown in Figure 31.41. Here are the steps to create a new user interface:

1. (Optional) On some systems, the components in the Library pane are not visible by sections. Click the Library icon to open the context menu as shown in Figure 31.40 and choose *View as List*.
2. Drag a **BorderPane** into the user interface and drag an **HBox** to the center of the **BorderPane** and another **HBox** to the bottom of the **BorderPane**. Set the alignment of both **HBox** to **CENTER** as shown in Figure 31.42. Set the **Spacing** property in the Layout section of the Inspector to 5. When you select a component in the visual layout, the properties of the component are displayed in the Inspector pane, where you can set the properties.

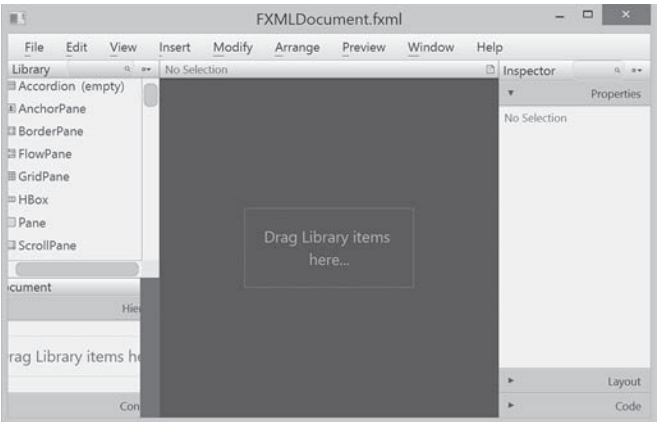


FIGURE 31.41 The UI is empty after deleting the default button in the pane.

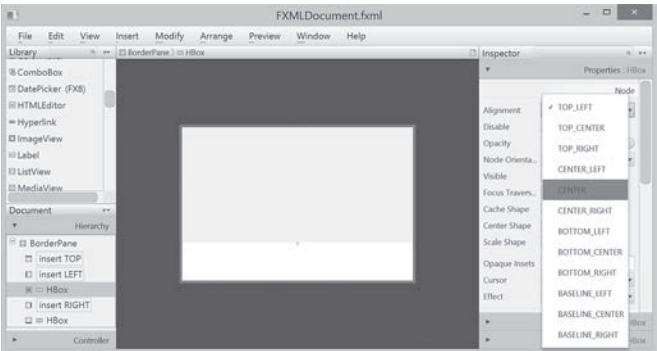


FIGURE 31.42 A BorderPane is dropped to the UI and an HBox is placed at the bottom of the BorderPane.

3. Drag and drop a **Label**, a **TextField**, a **Label**, a **TextField**, a **Label**, and a **TextField** and change the label's text to Number 1, Number 2, and Result, as shown in Figure 31.43. Set the Pref Column Count property for each text field to 2 in the Layout section of the Inspector. In the Code section of the Inspector, set the id for the text fields to **tfNumber1**, **tfNumber2**, and **tfResult**, as shown in Figure 31.44. These ids are useful to reference the text fields and obtain their values in the controller.
4. Drag and drop four **Buttons** to the second **HBox** and set their text property to Add, Subtract, Multiply, and Divide, as shown in Figure 31.45.

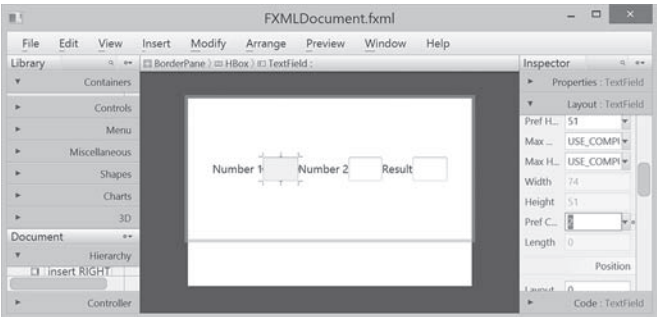


FIGURE 31.43 The labels and text fields are dropped to the UI.

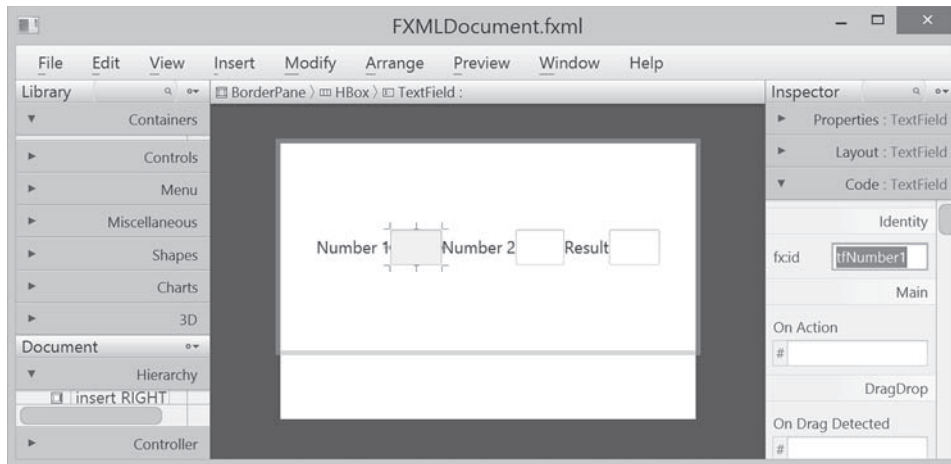


FIGURE 31.44 Set the appropriate id for the text fields.

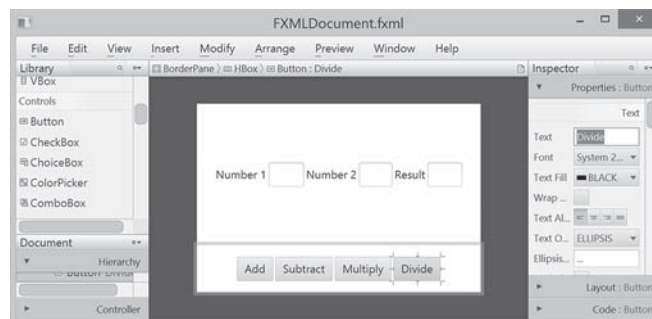


FIGURE 31.45 The buttons are dropped to the HBox.

After you create and make changes to the user interface in the content pane, you need to save the changes by choosing *File, Save* from the menu bar in the Scene Builder. The .fxml file is updated and synchronized with the changes in the content pane. You can view the contents in the .fxml file from NetBeans, as shown in Figure 31.46.

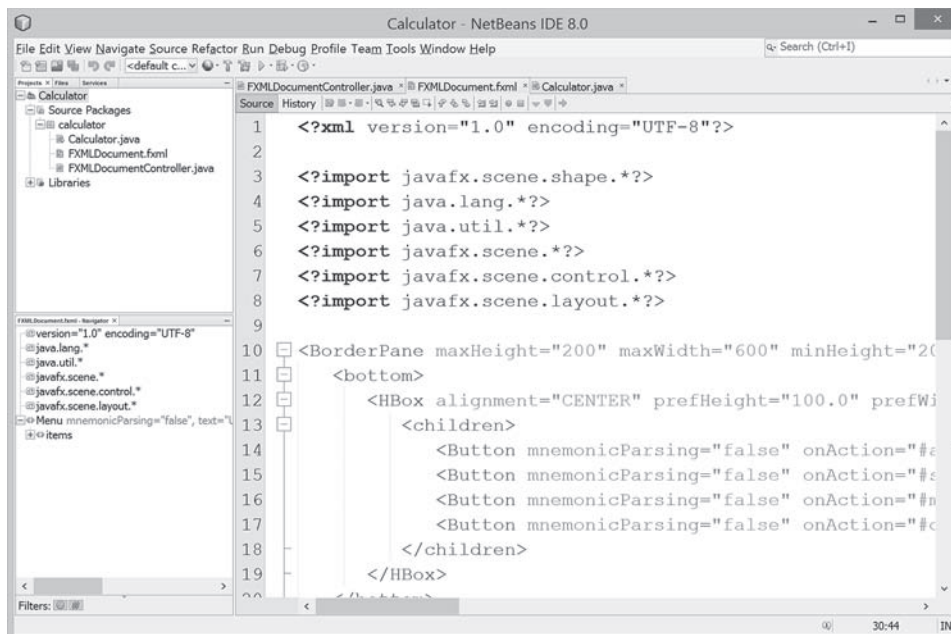


FIGURE 31.46 You can view the contents of the FXML file.

31.11.4 Handling Events in the Controller

The .fxml file describes the user interface. You write the code to implement the logic in the controller file, as shown in Listing 31.18.

LISTING 31.18 FXMLDocumentController.java

```

1  package calculator;
2
3  import javafx.event.ActionEvent;
4  import javafx.fxml.FXML;
5  import javafx.scene.control.TextField;
6
7  public class FXMLDocumentController {
8      @FXML
9      private TextField tfNumber1, tfNumber2, tfResult;
10
11     @FXML
12     private void addButtonAction(ActionEvent event) {
13         tfResult.setText(getResult('+') + "");
14     }
15
16     @FXML
17     private void subtractButtonAction(ActionEvent event) {
18         tfResult.setText(getResult('-') + "");
19     }
20
21     @FXML
22     private void multiplyButtonAction(ActionEvent event) {
23         tfResult.setText(getResult('*') + "");
24     }
25
26     @FXML
27     private void divideButtonAction(ActionEvent event) {
28         tfResult.setText(getResult('/') + "");
29     }
30
31     private double getResult(char op) {
32         double number1 = Double.parseDouble(tfNumber1.getText());
33         double number2 = Double.parseDouble(tfNumber2.getText());
34         switch (op) {
35             case '+': return number1 + number2;
36             case '-': return number1 - number2;
37             case '*': return number1 * number2;
38             case '/': return number1 / number2;
39         }
40         return Double.NaN;
41     }
42 }

```

The controller class declares three `TextFields`, `tfNumber1`, `tfNumber2`, and `tfResult` (line 9). The `@FXML` annotation denotes that these data fields are linked to the text fields in the user interface. Recall in the user interface, we set the id for the three text fields as `tfNumber1`, `tfNumber2`, and `tfResult`.

The codes for handling the events from the buttons are defined in the methods `addButtonAction`, `subtractButtonAction`, `multiplyButtonAction`, and `divideButtonAction` (lines 11–29). The `@FXML` annotation is used to denote that these methods will be tied to the button actions in the view.

Through the `@FXML` annotation, the data fields and methods in the controller are linked to the components and actions defined in the .fxml file.

31.11.5 Linking View with Controller

You can now link the actions from the components in the view with the processing methods in the controller. Here are the steps to accomplish it:

1. Add the following attribute in the `<BorderPane>` tag for using a controller with the view.

```
fx:controller="calculator.FXMLDocumentController"
```

2. Double-click the `.fxml` file in the project to display the visual layout window. In the Inspector for the *Add* button, choose **addButtonAction** from a list of action processing methods, as shown in Figure 31.47. The complete code for the `.fxml` file is shown in Listing 31.19.

LISTING 31.19 FXMLDocument.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<BorderPane maxHeight="200" maxWidth="600" minHeight="200"
    minWidth="600" prefHeight="400.0" prefWidth="600.0"
    xmlns="http://javafx.com/javafx/8"
    xmlns:fx="http://javafx.com/fxml/1"
    fx:controller="calculator.FXMLDocumentController">
    <bottom>
        <HBox alignment="CENTER" prefHeight="100.0" prefWidth="200.0"
            spacing="5.0" BorderPane.alignment="CENTER">
            <children>
                <Button mnemonicParsing="false"
                    onAction="#addButtonAction" text="Add" />
                <Button mnemonicParsing="false"
                    onAction="#subtractButtonAction" text="Subtract" />
                <Button mnemonicParsing="false"
                    onAction="#multiplyButtonAction" text="Multiply" />
                <Button mnemonicParsing="false"
                    onAction="#divideButtonAction" text="Divide" />
            </children>
        </HBox>
    </bottom>
    <center>
        <HBox alignment="CENTER" prefHeight="232.0" prefWidth="572.0"
            spacing="5.0" BorderPane.alignment="CENTER">
            <children>
                <Label text="Number 1" />
                <TextField fx:id="tfNumber1" prefColumnCount="2"
                    prefHeight="51.0" prefWidth="74.0" />
                <Label text="Number 2" />
                <TextField fx:id="tfNumber2" prefColumnCount="2"
                    prefHeight="51.0" prefWidth="70.0" />
                <Label text="Result" />
                <TextField fx:id="tfResult" prefColumnCount="2" />
            </children>
        </HBox>
    </center>
</BorderPane>
```

31.11.6 Running the Project

The code in the model is automatically generated as shown in Listing 31.15. This is the main program that loads the FXML to create the user interface in a Parent object (lines 12 and 13). The parent object is then added to the scene (line 14). The scene is set to the stage (line 15). The stage is displayed in line 16.

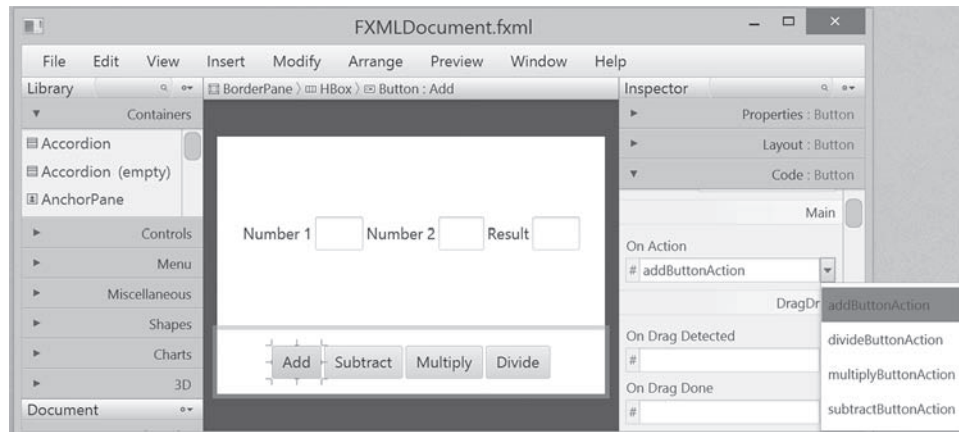


FIGURE 31.47 Choosing `addButtonAction` to generate the code for handling action for the Add button.

CHAPTER SUMMARY

1. JavaFX provides the cascading style sheets based on CSS. You can use the `getStylesheets` method to load a style sheet and use the `setStyle`, `setStyleClass`, and `setId` methods to set JavaFX CSS for nodes.
2. JavaFX provides the `QuadCurve`, `CubicCurve`, and `Path` classes for creating advanced shapes.
3. JavaFX supports coordinate transformations using translation, rotation, and scaling.
4. You can specify the pattern for a stroke, how the lines are joined in a stroke, the width of a stroke, and the type of a stroke.
5. You can create menus using the `Menu`, `MenuItem`, `CheckMenuItem`, and `RadioMenuItem` classes.
6. You can create context menus using the `ContextMenu` class.
7. The `SplitPane` can be used to display multiple panes horizontally or vertically and allows the user to adjust the sizes of the panes.
8. The `TabPane` can be used to display multiple panes with tabs for selecting panes.
9. You can create and display tables using the `TableView` and `TableColumn` classes.
10. You can create JavaFX user interfaces using FXML. FXML is XML-based script language for describing the user interface. Using FXML enables you to separate user interface from the logic of Java code.
11. JavaFX Scene Builder is a visual tool for creating the user interface without manually writing the FXML scripts.

Quiz

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



PROGRAMMING EXERCISES

MyProgrammingLab™

Sections 31.2

31.1 (Use JavaFX CSS) Create a CSS style sheet that defines a class for white fill and black stroke color and an id for red stroke and green color. Write a program that displays four circles and uses the style class and id. The sample run of the program is shown in Figure 31.48a.

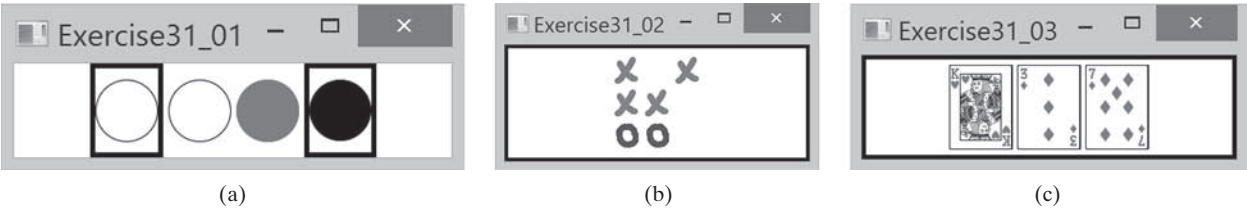


FIGURE 31.48 (a) The border and the color style for the shapes are defined in a style class. (b) Exercise 31.2 displays a tic-tac-toe board with images using style sheet for border. (c) Three cards are randomly selected.

- *31.2** (Tic-tac-toe board) Write a program that displays a tic-tac-toe board, as shown in Figure 31.48b. A cell may be X, O, or empty. What to display at each cell is randomly decided. The X and O are images in the files `x.gif` and `o.gif`. Use the style sheet for border.
- *31.3** (Display three cards) Write a program that displays three cards randomly selected from a deck of 52, as shown in Figure 31.48c. The card image files are named `1.png`, `2.png`, . . . , `52.png` and stored in the `image/card` directory. All the three cards are distinct and selected randomly. Hint: You can select random cards by storing the numbers 1–52 to an array, perform a random shuffle using Section 7.2.6, and use the first three numbers in the array as the file names for the image. Use the style sheet for border.

Sections 31.3

31.4 (Color and font) Write a program that displays five texts vertically, as shown in Figure 31.49a. Set a random color and opacity for each text and set the font of each text to Times Roman, bold, italic, and 22 pixels.

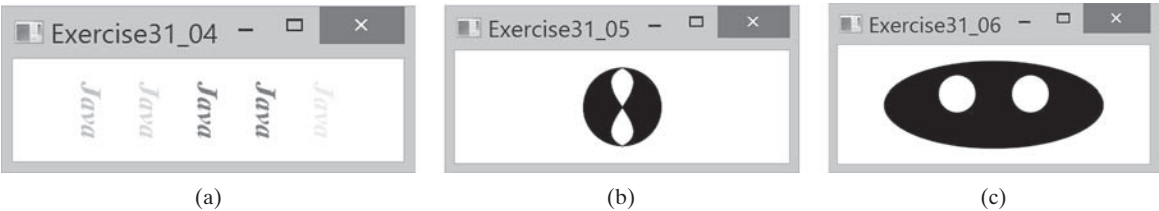


FIGURE 31.49 (a) Five texts are displayed with a random color and a specified font. (b) A path is displayed inside the circle. (c) Two circles are displayed in an oval.

***31.5** (*Cubic curve*) Write a program that creates two shapes: a circle and a path consisting of two cubic curves, as shown in Figure 31.49b.

***31.6** (*Eyes*) Write a program that displays two eyes in an oval, as shown in Figure 31.49c.

Sections 31.4

***31.7** (*Translation*) Write a program that displays a rectangle with upper-left corner point at (40, 40), width 50, and height 40. Enter the values in the text fields x and y and press the *Translate* button to translate the rectangle to a new location, as shown in Figure 31.50a.

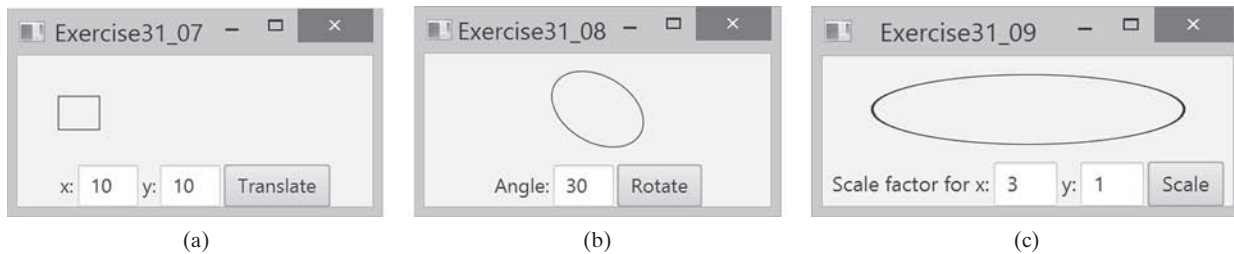


FIGURE 31.50 (a) Exercise 31.7 translates coordinates. (b) Exercise 31.8 rotates coordinates. (c) Exercise 31.9 scales coordinates.

***31.8** (*Rotation*) Write a program that displays an ellipse. The ellipse is centered in the pane with width 60 and height 40. Enter the value in the text field Angle and press the *Rotate* button to rotate the ellipse, as shown in Figure 31.50b.

***31.9** (*Scale graphics*) Write a program that displays an ellipse. The ellipse is centered in the pane with width 60 and height 40. Enter the scaling factors in the text fields and press the *Scale* button to scale the ellipse, as shown in Figure 31.50c.

***31.10** (*Plot the sine function*) Write a program that plots the sine function, as shown in Figure 31.51a.

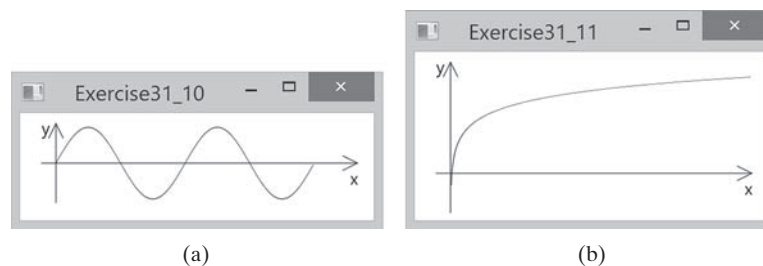


FIGURE 31.51 (a) Exercise 31.10 displays a sine function. (b) Exercise 31.11 displays the log function.

***31.11** (*Plot the log function*) Write a program that plots the log function, as shown in Figure 31.51a.

***31.12** (*Plot the n^2 function*) Write a program that plots the n^2 function, as shown in Figure 31.51b 2a.

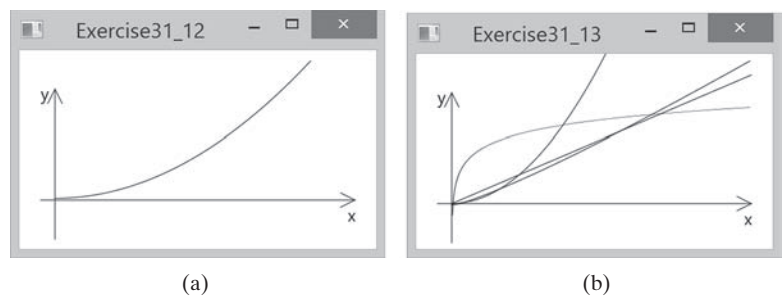


FIGURE 31.52 (a) Exercise 31.13 displays the n^2 function. (b) Exercise 31.13 displays several functions.

- *31.13** (*Plot the log, n , $n \log n$, and n^2 functions*) Write a program that plots the log, n , $n \log n$, and n^2 functions, as shown in Figure 31.52b.
- *31.14** (*Scale and rotate graphics*) Write a program that enables the user to scale and rotate the STOP sign, as shown in Figure 31.53. The user can press the UP/DOWN arrow key to increase/decrease the size and press the RIGHT/LEFT arrow key to rotate left or right.



FIGURE 31.53 The program can rotate and scale the painting.

Sections 31.5

- *31.15** (*Sunshine*) Write a program that displays a circle filled with a gradient color to animate a sun and display light rays coming out from the sun using dashed lines, as shown in Figure 31.54a.

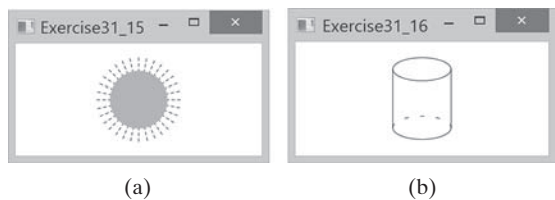


FIGURE 31.54 (a) Exercise 31.15 displays the sunshine. (b) Exercise 31.16 displays a cylinder.

- *31.16** (*Display a cylinder*) Write a program that displays a cylinder, as shown in Figure 31.54b. Use dashed strokes to draw the dashed arc.

Sections 31.6

***31.17** (*Create an investment value calculator*) Write a program that calculates the future value of an investment at a given interest rate for a specified number of years. The formula for the calculation is as follows:

$$\text{futureValue} = \text{investmentAmount} \times (1 + \text{monthlyInterestRate})^{\text{years} \times 12}$$

Use text fields for interest rate, investment amount, and years. Display the future amount in a text field when the user clicks the *Calculate* button or chooses *Calculate* from the *Operation* menu (see Figure 31.55). Click the *Exit* menu to exit the program.

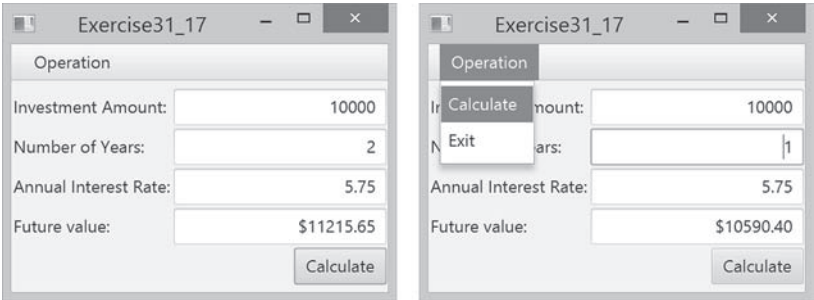


FIGURE 31.55 The user enters the investment amount, years, and interest rate to compute future value.

Sections 31.8

***31.18** (*Use popup menus*) Modify **Listing 31.9**, *MenuDemo.java*, to create a popup menu that contains the menus *Operations* and *Exit*, as shown in Figure 31.56. The popup is displayed when you click the right mouse button on the panel that contains the labels and the text fields.

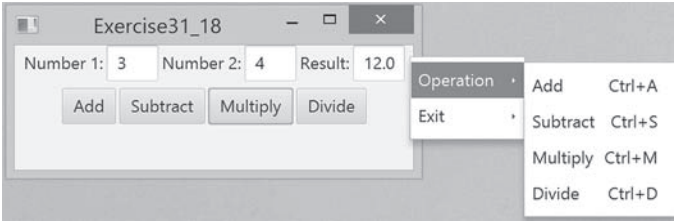


FIGURE 31.56 The popup menu contains the commands to perform arithmetic operations.

***31.19** (*Use `SplitPane`*) Create a program that displays four shapes in split panes, as shown in Figure 31.57a.

Sections 31.9

***31.20** (*Use tab panes*) Modify **Listing 31.12**, *TabPaneDemo.java*, to add a pane of radio buttons for specifying the tab placement of the tab pane, as shown in Figure 31.57b and c.

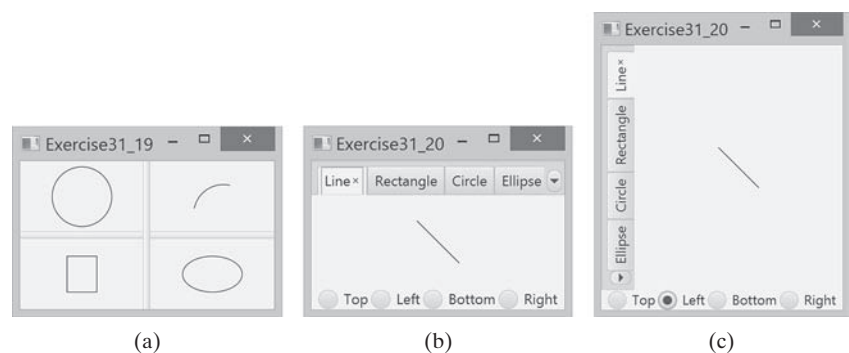


FIGURE 31.57 (a) Four shapes are displayed in split panes. (b and c) The radio buttons let you choose the tab placement of the tabbed pane.

***31.21** (*Use tab panes*) Write a program using tab panes for performing integer and rational number arithmetic as shown in Figure 31.58.

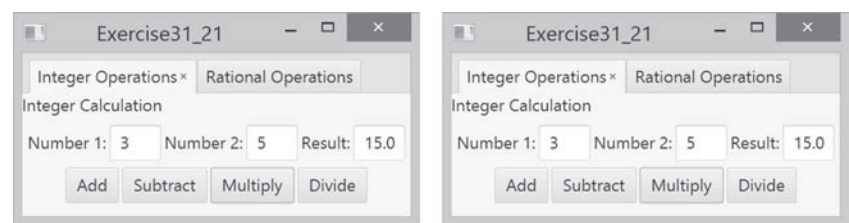


FIGURE 31.58 A tab pane is used to select panes that perform integer operations and rational number operations.

Sections 31.10

***31.22** (*Use table view*) Revise Listing 31.14 to add a button to delete the selected row from the table, as shown in Figure 31.59.

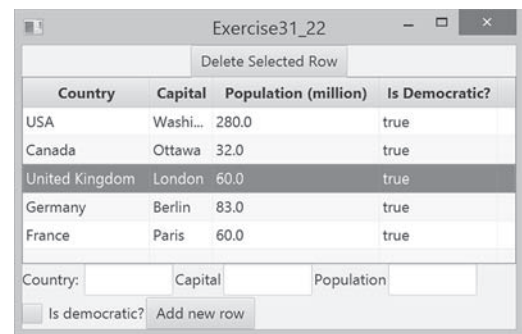


FIGURE 31.59 Clicking the *Delete Selected Row* button removes the selected row from the table.