will see that much of its functionality comes from its base classes. Furthermore, it supports a wide array of options. However, here we will use its default form. Buttons can contain text, graphics, or both. In this chapter, we will use text-based buttons. An example of a graphics-based button is shown in the next chapter.

Button defines three constructors. The one we will use is shown here:

Button(String *str*)

In this case, *str* is the message that is displayed in the button.

When a button is pressed, an **ActionEvent** is generated. **ActionEvent** is packaged in **javafx.event**. You can register a listener for this event by using **setOnAction( )**, which has this general form:

final void setOnAction(EventHandler<ActionEvent> *handler*)

Here, *handler* is the handler being registered. As mentioned, often you will use an anonymous inner class or lambda expression for the handler. The **setOnAction( )** method sets the property **onAction**, which stores a reference to the handler. As with all other Java event handling, your handler must respond to the event as fast as possible and then return. If your handler consumes too much time, it will noticeably slow down the application. For lengthy operations, you must use a separate thread of execution.

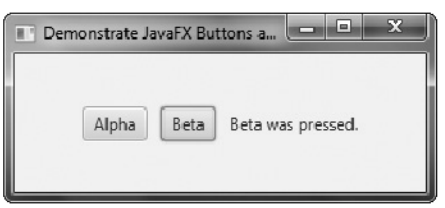
Demonstrating Event Handling and the Button

The following program demonstrates event handling. It uses two buttons and a label. Each time a button is pressed, the label is set to display which button was pressed.

```
// Demonstrate JavaFX events and buttons.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class JavaFXEventDemo extends Application {

  Label response;

  public static void main(String[] args) {

    // Start the JavaFX application by calling launch().
    launch(args);
  }

  // Override the start() method.
  public void start(Stage myStage) {

    // Give the stage a title.
    myStage.setTitle("Demonstrate JavaFX Buttons and Events.");
```

```
    // Use a FlowPane for the root node. In this case,
    // vertical and horizontal gaps of 10.
    FlowPane rootNode = new FlowPane(10, 10);

    // Center the controls in the scene.
    rootNode.setAlignment(Pos.CENTER);

    // Create a scene.
    Scene myScene = new Scene(rootNode, 300, 100);

    // Set the scene on the stage.
    myStage.setScene(myScene);

    // Create a label.
    response = new Label("Push a Button");

    // Create two push buttons.
    Button btnAlpha = new Button("Alpha");
    Button btnBeta = new Button("Beta");

    // Handle the action events for the Alpha button.
    btnAlpha.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        response.setText("Alpha was pressed.");
      }
    });

    // Handle the action events for the Beta button.
    btnBeta.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        response.setText("Beta was pressed.");
      }
    });

    // Add the label and buttons to the scene graph.
    rootNode.getChildren().addAll(btnAlpha, btnBeta, response);

    // Show the stage and its scene.
    myStage.show();
  }
}
```

Sample output from this program is shown here:



Let's examine a few key portions of this program. First, notice how buttons are created by these two lines:

```
Button btnAlpha = new Button("Alpha");
Button btnBeta = new Button("Beta");
```

This creates two text-based buttons. The first displays the string Alpha; the second displays Beta.

Next, an action event handler is set for each of these buttons. The sequence for the Alpha button is shown here:

```
// Handle the action events for the Alpha button.
btnAlpha.setOnAction(new EventHandler<ActionEvent>() {
  public void handle(ActionEvent ae) {
    response.setText("Alpha was pressed.");
  }
});
```

As explained, buttons respond to events of type **ActionEvent**. To register a handler for these events, the **setOnAction( )** method is called on the button. It uses an anonymous inner class to implement the **EventHandler** interface. (Recall that **EventHandler** defines only the **handle( )** method.) Inside **handle( )**, the text in the **response** label is set to reflect the fact that the Alpha button was pressed. Notice that this is done by calling the **setText( )** method on the label. Events are handled by the Beta button in the same way.

Note that **response** is declared as a field within **FXEventDemo**, rather than as a local variable. This is because it is accessed within the button event handlers, which are anonymous inner classes.

After the event handlers have been set, the **response** label and the buttons **btnAlpha** and **btnBeta** are added to the scene graph by using a call to **addAll( )**:

```
rootNode.getChildren().addAll(btnAlpha, btnBeta, response);
```

The **addAll( )** method adds a list of nodes to the invoking parent node. Of course, these nodes could have been added by three separate calls to **add( )**, but the **addAll( )** method is more convenient to use in this situation.

There are two other things of interest in this program that relate to the way the controls are displayed in the window. First, when the root node is created, this statement is used:

```
FlowPane rootNode = new FlowPane(10, 10);
```

Here, the **FlowPane** constructor is passed two values. These specify the horizontal and vertical gap that will be left around elements in the scene. If these gaps are not specified, then two elements (such as two buttons) would be positioned in such a way that no space is between them. Thus, the controls would run together, creating a very unappealing user interface. Specifying gaps prevents this.

The second point of interest is the following line, which sets the alignment of the elements in the **FlowPane**:

```
rootNode.setAlignment(Pos.CENTER);
```

Here, the alignment of the elements is centered. This is done by calling **setAlignment( )** on the **FlowPane**. The value **Pos.CENTER** specifies that both a vertical and horizontal center will be used. Other alignments are possible. **Pos** is an enumeration that specifies alignment constants. It is packaged in **javafx.geometry**.

Before moving on, one more point needs to be made. The preceding program used anonymous inner classes to handle button events. However, because the **EventHandler** interface defines only one abstract method, **handle( )**, a lambda expression could have passed to **setOnAction( )**, instead. In this case, the parameter type of **setOnAction( )** would supply the target context for the lambda expression. For example, here is the handler for the Alpha button, rewritten to use a lambda:

```
btnAlpha.setOnAction( (ae) ->
                        response.setText("Alpha was pressed.")
                      );
```

Notice that the lambda expression is more compact than the anonymous inner class. Because lambda expressions are a new feature just recently added to Java, but the anonymous inner class is a widely used construct, readily understood by nearly all Java programmers, the event handlers in subsequent examples will use anonymous inner classes. This will also allow the examples to be compiled by readers using JDK 7 (which does not support lambdas). However, on your own, you might want to experiment with converting them to lambda expressions. It is a good way to gain experience using lambdas in your own code.

Drawing Directly on a Canvas

As mentioned early on, JavaFX handles rendering tasks for you automatically, rather than you handling them manually. This is one of the most important ways that JavaFX improves on Swing. As you may know, in Swing or the AWT, you must call the **repaint( )** method to cause a window to be repainted. Furthermore, your application needs to store the window contents, redrawing them when painting is requested. JavaFX eliminates this tedious mechanism because it keeps track of what you display in a scene and redisplays that scene as needed. This is called *retained mode*. With this approach, there is no need to call a method like **repaint( )**. Rendering is automatic.

One place that JavaFX's approach to rendering is especially helpful is when displaying graphics objects, such as lines, circles, and rectangles. JavaFX's graphics methods are found in the **GraphicsContext** class, which is part of **java.scene.canvas**. These methods can be used to draw directly on the surface of a canvas, which is encapsulated by the **Canvas** class in **java.scene.canvas**. When you draw something, such as a line, on a canvas, JavaFX automatically renders it whenever it needs to be redisplayed.

Before you can draw on a canvas, you must perform two steps. First, you must create a **Canvas** instance. Second, you must obtain a **GraphicsContext** object that refers to that canvas. You can then use the **GraphicsContext** to draw output on the canvas.

The **Canvas** class is derived from **Node**; thus it can be used as a node in a scene graph. **Canvas** defines two constructors. One is the default constructor, and the other is the one shown here:

Canvas(double *width*, double *height*)

Here, *width* and *height* specify the dimensions of the canvas.

To obtain a **GraphicsContext** that refers to a canvas, call **getGraphicsContext2D( )**. Here is its general form:

GraphicsContext getGraphicsContext2D( )

The graphics context for the canvas is returned.

GraphicsContext defines a large number of methods that draw shapes, text, and images, and support effects and transforms. If sophisticated graphics programming is in your future, you will definitely want to study its capabilities closely. For our purposes, we will use only a few of its methods, but they will give you a sense of its power. They are described here.

You can draw a line using **strokeLine( )**, shown here:

void strokeLine(double *startX*, double *startY*, double *endX*, double *endY*)

It draws a line from *startX*,*startY* to *endX*,*endY*, using the current stroke, which can be a solid color or some more complex style.

To draw a rectangle, use either **strokeRect( )** or **fillRect( )**, shown here:

void strokeRect(double *topX*, double *topY*, double *width*, double *height*)

void fillRect(double *topX*, double *topY*, double *width*, double *height*)

The upper-left corner of the rectangle is at *topX*,*topY*. The *width* and *height* parameters specify its width and height. The **strokeRect( )** method draws the outline of a rectangle using the current stroke, and **fillRect( )** fills the rectangle with the current fill. The current fill can be as simple as a solid color or something more complex.

To draw an ellipse, use either **strokeOval( )** or **fillOval( )**, shown next:

void strokeOval(double *topX*, double *topY*, double *width*, double *height*)

void fillOval(double *topX*, double *topY*, double *width*, double *height*)

The upper-left corner of the rectangle that bounds the ellipse is at *topX*,*topY*. The *width* and *height* parameters specify its width and height. The **strokeOval( )** method draws the outline of an ellipse using the current stroke, and **fillOval( )** fills the oval with the current fill. To draw a circle, pass the same value for *width* and *height*.

You can draw text on a canvas by using the **strokeText( )** and **fillText( )** methods. We will use this version of **fillText( )**:

void fillText(String *str*, double *topX*, double *topY*)

It displays *str* starting at the location specified by *topX*,*topY*, filling the text with the current fill.

You can set the font and font size of the text being displayed by using **setFont( )**. You can obtain the font used by the canvas by calling **getFont( )**. By default, the system font is used. You can create a new font by constructing a **Font** object. **Font** is packaged in **javafx.scene.text**. For example, you can create a default font of a specified size by using this constructor:

Font(double *fontSize*)

Here, *fontSize* specifies the size of the font.

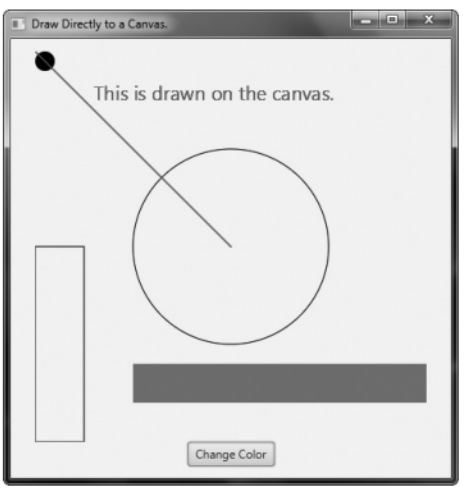
You can specify the fill and stroke using these two methods defined by **Canvas**:

void setFill(Paint *newFill*)

void setStroke(Paint *newStroke*)

Notice that the parameter of both methods is of type **Paint**. This is an abstract class packaged in **javafx.scene.paint**. Its subclasses define fills and strokes. The one we will use is **Color**, which simply describes a solid color. **Color** defines several static fields that specify a wide array of colors, such as **Color.BLUE**, **Color.RED**, **Color.GREEN**, and so on.

The following program uses the aforementioned methods to demonstrate drawing on a canvas. It first displays a few graphic objects on the canvas. Then, each time the Change Color button is pressed, the color of three of the objects changes color. If you run the program, you will see that the shapes whose color is not changed are unaffected by the change in color of the other objects. Furthermore, if you try covering and then uncovering the window, you will see that the canvas is automatically repainted, without any other actions on the part of your program. Sample output is shown here:




```
// Demonstrate drawing.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
import javafx.scene.shape.*;
import javafx.scene.canvas.*;
import javafx.scene.paint.*;
import javafx.scene.text.*;

public class DirectDrawDemo extends Application {

  GraphicsContext gc;
```

```
Color[] colors = { Color.RED, Color.BLUE, Color.GREEN, Color.BLACK };
int colorIdx = 0;

public static void main(String[] args) {

  // Start the JavaFX application by calling launch().
  launch(args);
}

// Override the start() method.
public void start(Stage myStage) {

  // Give the stage a title.
  myStage.setTitle("Draw Directly to a Canvas.");

  // Use a FlowPane for the root node.
  FlowPane rootNode = new FlowPane();

  // Center the nodes in the scene.
  rootNode.setAlignment(Pos.CENTER);

  // Create a scene.
  Scene myScene = new Scene(rootNode, 450, 450);

  // Set the scene on the stage.
  myStage.setScene(myScene);

  // Create a canvas.
  Canvas myCanvas = new Canvas(400, 400);

  // Get the graphics context for the canvas.
  gc = myCanvas.getGraphicsContext2D();

  // Create a push button.
  Button btnChangeColor = new Button("Change Color");

  // Handle the action events for the Change Color button.
  btnChangeColor.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {

      // Set the stroke and fill color.
      gc.setStroke(colors[colorIdx]);
      gc.setFill(colors[colorIdx]);

      // Redraw the line, text, and filled rectangle in the
      //  new color. This leaves the color of the other nodes
      // unchanged.
      gc.strokeLine(0, 0, 200, 200);
      gc.fillText("This is drawn on the canvas.", 60, 50);
      gc.fillRect(100, 320, 300, 40);
```

```
        // Change the color.
        colorIdx++;
        if(colorIdx == colors.length) colorIdx= 0;
      }
    });

    // Draw initial output on the canvas.
    gc.strokeLine(0, 0, 200, 200);
    gc.strokeOval(100, 100, 200, 200);
    gc.strokeRect(0, 200, 50, 200);
    gc.fillOval(0, 0, 20, 20);
    gc.fillRect(100, 320, 300, 40);

    // Set the font size to 20 and draw text.
    gc.setFont(new Font(20));
    gc.fillText("This is drawn on the canvas.", 60, 50);

    // Add the canvas and button to the scene graph.
    rootNode.getChildren().addAll(myCanvas, btnChangeColor);

    // Show the stage and its scene.
    myStage.show();
  }
}
```

It is important to emphasize that **GraphicsContext** supports many more operations than those demonstrated by the preceding program. For example, you can apply various transforms, rotations, and effects. Despite its power, its various features are easy to master and use. One other point: A canvas is transparent. Therefore, if you stack canvases, the contents of both will show. This may be useful in some situations.

NOTE **javafx.scene.shape** contains several classes that can also be used to draw various types of graphical shapes, such as circles, arcs, and lines. These are represented by nodes and can, therefore, be directly part of the scene graph. You will want to explore these on your own.

This page has been intentionally left blank

CHAPTER

35 Exploring JavaFX Controls

The previous chapter described several of the core concepts relating to the JavaFX GUI framework. In the process, it introduced two controls: the label and the button. This chapter continues the discussion of JavaFX controls. It begins by describing how to include images in a label and button. It then presents an overview of several more JavaFX controls, including check boxes, lists, and trees. Keep in mind that JavaFX is a rich and powerful framework. The purpose of this chapter is to introduce a representative sampling of the JavaFX controls and to describe several common techniques. Once you understand the basics, you will be able to easily learn the other controls.

The JavaFX control classes discussed in this chapter are shown here:

Button	ListView	TextField
CheckBox	RadioButton	ToggleButton
Label	ScrollPane	TreeView

These and the other JavaFX controls are packaged in **javafx.scene.control**.

Also discussed are the **Image** and **ImageView** classes, which provide support for images in controls; **Tooltip**, which is used to add tooltips to a control; as well as various effects and transforms.

Using Image and ImageView

Several of JavaFX's controls let you include an image. For example, in addition to text, you can specify an image in a label or a button. Furthermore, you can embed stand-alone images in a scene directly. At the foundation for JavaFX's support for images are two classes: **Image** and **ImageView**. **Image** encapsulates the image, itself, and **ImageView** manages the display of an image. Both classes are packaged in **javafx.scene.image**.

The **Image** class loads an image from either an **InputStream**, a URL, or a path to the image file. **Image** defines several constructors; this is the one we will use:

Image(String *url*)

Here, *url* specifies a URL or a path to a file that supplies the image. The argument is assumed to refer to a path if it does not constitute a properly formed URL. Otherwise, the image is loaded from the URL. The examples that follow will load images from files on the local file system. Other constructors let you specify various options, such as the image's width and height. One other point: **Image** is not derived from **Node**. Thus, it cannot, itself, be part of a scene graph.

Once you have an **Image**, you will use **ImageView** to display it. **ImageView** is derived from **Node**, which means that it can be part of a scene graph. **ImageView** defines three constructors. The first one we will use is shown here:

ImageView(Image *image*)

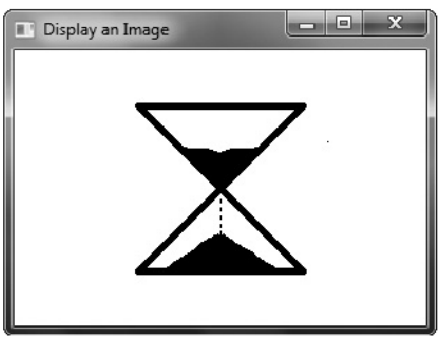
This constructor creates an **ImageView** that uses *image* for its image.

Putting the preceding discussion into action, here is a program that loads an image of an hourglass and displays it via **ImageView**. The hourglass image is contained in a file called **hourglass.png**, which is assumed to be in the local directory.

```
// Load and display an image.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.geometry.*;
import javafx.scene.image.*;


public class ImageDemo extends Application {

  public static void main(String[] args) {

    // Start the JavaFX application by calling launch().
    launch(args);
  }

  // Override the start() method.
  public void start(Stage myStage) {

    // Give the stage a title.
    myStage.setTitle("Display an Image");

    // Use a FlowPane for the root node.
    FlowPane rootNode = new FlowPane();

    // Use center alignment.
    rootNode.setAlignment(Pos.CENTER);

    // Create a scene.
    Scene myScene = new Scene(rootNode, 300, 200);
```

```
    // Set the scene on the stage.
    myStage.setScene(myScene);

    // Create an image.
    Image hourglass = new Image("hourglass.png");

    // Create an image view that uses the image.
    ImageView hourglassIV = new ImageView(hourglass);

    // Add the image to the scene graph.
    rootNode.getChildren().add(hourglassIV);

    // Show the stage and its scene.
    myStage.show();
  }
}
```

Sample output from the program is shown here:



In the program, pay close attention to the following sequence that loads the image and then creates an **ImageView** that uses that image:

```
// Create an image.
Image hourglass = new Image("HourGlass.png");

// Create an image view that uses the image.
ImageView hourglassIV = new ImageView(hourglass);
```

As explained, an image by itself cannot be added to the scene graph. It must first be embedded in an **ImageView**.

In cases in which you won't make further use of the image, you can specify a URL or filename when creating an **ImageView**. In this case, there is no need to explicitly create an **Image**. Instead, an **Image** instance containing the specified image is constructed automatically and embedded in the **ImageView**. Here is the **ImageView** constructor that does this:

```
ImageView(String url)
```

Here, *url* specifies the URL or the path to a file that contains the image.

Adding an Image to a Label

As explained in the previous chapter, the **Label** class encapsulates a label. It can display a text message, a graphic, or both. So far, we have used it to display only text, but it is easy to add an image. To do so, use this form of **Label**'s constructor:

Label(String *str*, Node *image*)

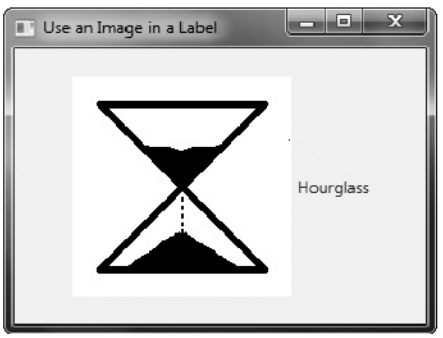
Here, *str* specifies the text message and *image* specifies the image. Notice that the image is of type **Node**. This allows great flexibility in the type of image added to the label, but for our purposes, the image type will be **ImageView**.

Here is a program that demonstrates a label that includes a graphic. It creates a label that displays the string "Hourglass" and shows the image of an hourglass that is loaded from the **hourglass.png** file.

```
// Demonstrate an image in a label.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.geometry.*;
import javafx.scene.image.*;

public class LabelImageDemo extends Application {

  public static void main(String[] args) {

    // Start the JavaFX application by calling launch().
    launch(args);
  }

  // Override the start() method.
  public void start(Stage myStage) {

    // Give the stage a title.
    myStage.setTitle("Use an Image in a Label");

    // Use a FlowPane for the root node.
    FlowPane rootNode = new FlowPane();

    // Use center alignment.
    rootNode.setAlignment(Pos.CENTER);

    // Create a scene.
    Scene myScene = new Scene(rootNode, 300, 200);

    // Set the scene on the stage.
    myStage.setScene(myScene);
```

```
    // Create an ImageView that contains the specified image.
    ImageView hourglassIV = new ImageView("hourglass.png");

    // Create a label that contains both an image and text.
    Label hourglassLabel = new Label("Hourglass", hourglassIV);

    // Add the label to the scene graph.
    rootNode.getChildren().add(hourglassLabel);

    // Show the stage and its scene.
    myStage.show();
  }
}
```

Here is the window produced by the program:



As you can see, both the image and the text are displayed. Notice that the text is to the right of the image. This is the default. You can change the relative positions of the image and text by calling **setContentDisplay( )** on the label. It is shown here:

final void setContentDisplay(ContentDisplay *position*)

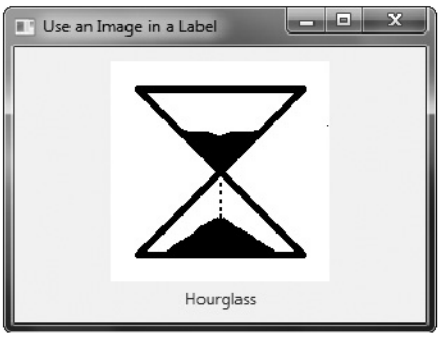
The value passed to *position* determines how the text and image is displayed. It must be one of these values, which are defined by the **ContentDisplay** enumeration:

BOTTOM	RIGHT
CENTER	TEXT_ONLY
GRAPHIC_ONLY	TOP
LEFT	

With the exception of **TEXT_ONLY** and **GRAPHIC_ONLY**, the values specify the location of the image. For example, if you add this line to the preceding program:

```
hourglassLabel.setContentDisplay(ContentDisplay.TOP);
```

the image of the hourglass will be above the text, as shown here:



The other two values let you display either just the text or just the image. This might be useful if your application wants to use an image at some times, and not at others, for example. (If you want only an image, you can simply display it without using a label, as described in the previous section.)

You can also add an image to a label after it has been constructed by using the **setGraphic( )** method. It is shown here:

final void setGraphic(Node *image*)

Here, *image* specifies the image to add.

Using an Image with a Button

Button is JavaFX's class for push buttons. The preceding chapter introduced the **Button** class. There, you saw an example of a button that contained text. Although such buttons are common, you are not limited to this approach because you can include an image. You can also use only the image if you choose. The procedure for adding an image to a button is similar to that used to add an image to a label. First obtain an **ImageView** of the image. Then add it to the button. One way to add the image is to use this constructor:

Button(String *str*, Node *image*)

Here, *str* specifies the text that is displayed within the button and *image* specifies the image. You can specify the position of the image relative to the text by using **setContentDisplay( )** in the same way as just described for **Label**.

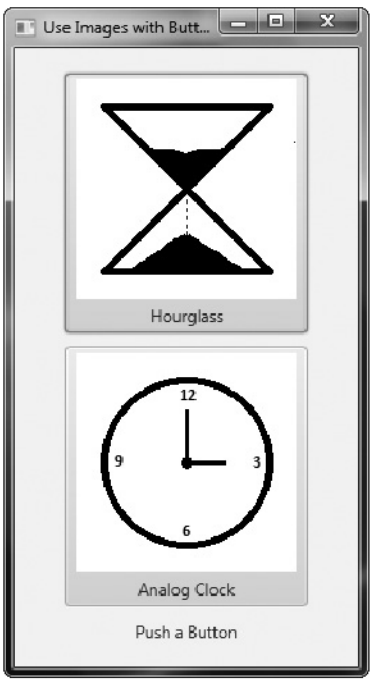
Here is an example that displays two buttons that contain images. The first shows an hourglass. The second shows an analog clock. When a button is pressed, the selected timepiece is reported. Notice that the text is displayed beneath the image.

```
// Use an image with a button.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
```

```
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
import javafx.scene.image.*;

public class ButtonImageDemo extends Application {

  Label response;

  public static void main(String[] args) {

    // Start the JavaFX application by calling launch().
    launch(args);
  }

  // Override the start() method.
  public void start(Stage myStage) {

    // Give the stage a title.
    myStage.setTitle("Use Images with Buttons");

    // Use a FlowPane for the root node. In this case,
    // vertical and horizontal gaps of 10.
    FlowPane rootNode = new FlowPane(10, 10);

    // Center the controls in the scene.
    rootNode.setAlignment(Pos.CENTER);

    // Create a scene.
    Scene myScene = new Scene(rootNode, 250, 450);

    // Set the scene on the stage.
    myStage.setScene(myScene);

    // Create a label.
    response = new Label("Push a Button");

    // Create two image-based buttons.
    Button btnHourglass = new Button("Hourglass",
                                     new ImageView("hourglass.png"));
    Button btnAnalogClock = new    Button("Analog Clock",
                                     new ImageView("analog.png"));

    // Position the text under the image.
    btnHourglass.setContentDisplay(ContentDisplay.TOP);
    btnAnalogClock.setContentDisplay(ContentDisplay.TOP);

    // Handle the action events for the hourglass button.
    btnHourglass.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        response.setText("Hourglass Pressed");
      }
    });
```

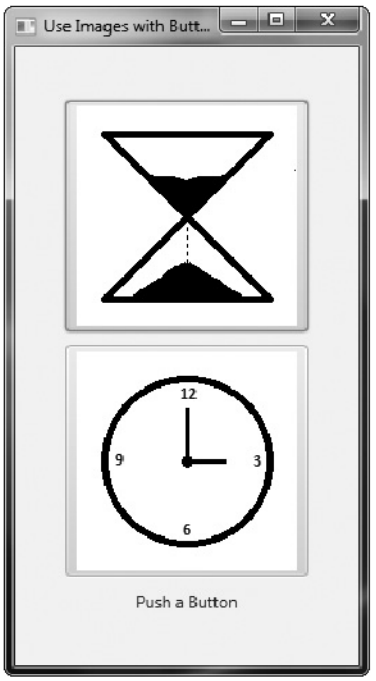
```
    // Handle the action events for the analog clock button.
    btnAnalogClock.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        response.setText("Analog Clock Pressed");
      }
    });

    // Add the label and buttons to the scene graph.
    rootNode.getChildren().addAll(btnHourglass, btnAnalogClock, response);

    // Show the stage and its scene.
    myStage.show();
  }
}
```

The output produced by this program is shown here:



If you want a button that contains only the image, pass a null string for the text when constructing the button and then call **setContentDisplay( )**, passing in the parameter

ContentDisplay.GRAPHIC_ONLY. For example, if you make these modifications to the previous program, the output will look like this:



ToggleButton

A useful variation on the push button is called the *toggle button*. A toggle button looks just like a push button, but it acts differently because it has two states: pushed and released. That is, when you press a toggle button, it stays pressed rather than popping back up as a regular push button does. When you press the toggle button a second time, it releases (pops up). Therefore, each time a toggle button is pushed, it toggles between these two states. In JavaFX, a toggle button is encapsulated in the **ToggleButton** class. Like **Button**, **ToggleButton** is also derived from **ButtonBase**. It implements the **Toggle** interface, which defines functionality common to all types of two-state buttons.

ToggleButton defines three constructors. This is the one we will use:

ToggleButton(String *str*)

Here, *str* is the text displayed in the button. Another constructor allows you to include an image. Like other buttons, a **ToggleButton** generates an action event when it is pressed.

Because **ToggleButton** defines a two-state control, it is commonly used to let the user select an option. When the button is pressed, the option is selected. When the button is

released, the option is deselected. For this reason, a program usually needs to determine the toggle button's state. To do this, use the **isSelected( )** method, shown here:

```
final boolean isSelected( )
```

It returns **true** if the button is pressed and **false** otherwise.

Here is a short program that demonstrates **ToggleButton**:

```
// Demonstrate a toggle button.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class ToggleButtonDemo extends Application {

  ToggleButton tbOnOff;
  Label response;

  public static void main(String[] args) {

    // Start the JavaFX application by calling launch().
    launch(args);
  }

  // Override the start() method.
  public void start(Stage myStage) {

    // Give the stage a title.
    myStage.setTitle("Demonstrate a Toggle Button");

    // Use a FlowPane for the root node. In this case,
    // vertical and horizontal gaps of 10.
    FlowPane rootNode = new FlowPane(10, 10);

    // Center the controls in the scene.
    rootNode.setAlignment(Pos.CENTER);

    // Create a scene.
    Scene myScene = new Scene(rootNode, 220, 120);

    // Set the scene on the stage.
    myStage.setScene(myScene);

    // Create a label.
    response = new Label("Push the Button.");

    // Create the toggle button.
    tbOnOff = new ToggleButton("On/Off");
```

```
    // Handle action events for the toggle button.
    tbOnOff.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        if(tbOnOff.isSelected()) response.setText("Button is on.");
        else response.setText("Button is off.");
      }
    });

    // Add the label and buttons to the scene graph.
    rootNode.getChildren().addAll(tbOnOff, response);

    // Show the stage and its scene.
    myStage.show();
  }
}
```

Sample output produced by the program is shown here, with the button pressed:

In the program, notice how the pressed/released state of the toggle button is determined by the following lines of code inside the button's action event handler.

```
if(tbOnOff.isSelected()) response.setText("Button is on.");
else response.setText("Button is off.");
```

When the button is pressed, **isSelected( )** returns **true**. When the button is released, **isSelected( )** returns **false**.

One other point: It is possible to use two or more toggle buttons in a group. In this case, only one button can be in its pressed state at any one time. The process of creating and using a group of toggle buttons is similar to that required to use radio buttons. It is described in the following section.

RadioButton

Another type of button provided by JavaFX is the *radio button*. Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any one time. They are supported by the **RadioButton** class, which extends both **ButtonBase** and **ToggleButton**. It also implements the **Toggle** interface. Thus, a radio button is a specialized form of a toggle button. You have almost certainly seen radio buttons in action because they are the primary control employed when the user must select only one option among several alternatives.

To create a radio button, we will use the following constructor:

RadioButton(String *str*)

Here, *str* is the label for the button. Like other buttons, when a **RadioButton** is used, an action event is generated.

For their mutually exclusive nature to be activated, radio buttons must be configured into a group. Only one of the buttons in the group can be selected at any time. For example,

if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected. A button group is created by the **ToggleGroup** class, which is packaged in **javafx.scene.control**. **ToggleGroup** provides only a default constructor.

Radio buttons are added to the toggle group by calling the **setToggleGroup( )** method, defined by **ToggleButton**, on the button. It is shown here:

final void setToggleGroup(ToggleGroup *tg*)

Here, *tg* is a reference to the toggle button group to which the button is added. After all radio buttons have been added to the same group, their mutually exclusive behavior will be enabled.

In general, when radio buttons are used in a group, one of the buttons is selected when the group is first displayed in the GUI. Here are two ways to do this.

First, you can call **setSelected( )** on the button that you want to select. It is defined by **ToggleButton** (which is a superclass of **RadioButton**). It is shown here:

final void setSelected(boolean *state*)

If *state* is **true**, the button is selected. Otherwise, it is deselected. Although the button is selected, no action event is generated.

A second way to initially select a radio button is to call **fire( )** on the button. It is shown here:

void fire( )

This method results in an action event being generated for the button if the button was previously not selected.

There are a number of different ways to use radio buttons. Perhaps the simplest is to simply respond to the action event that is generated when one is selected. The following program shows an example of this approach. It uses radio buttons to allow the user to select a type of transportation.

```
// A simple demonstration of Radio Buttons.
//
// This program responds to the action events generated
// by a radio button selection. It also shows how to
// fire the button under program control.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class RadioButtonDemo extends Application {

  Label response;

  public static void main(String[] args) {
```

```
    // Start the JavaFX application by calling launch().
    launch(args);
  }

  // Override the start() method.
  public void start(Stage myStage) {

    // Give the stage a title.
    myStage.setTitle("Demonstrate Radio Buttons");

    // Use a FlowPane for the root node. In this case,
    // vertical and horizontal gaps of 10.
    FlowPane rootNode = new FlowPane(10, 10);

    // Center the controls in the scene.
    rootNode.setAlignment(Pos.CENTER);

    // Create a scene.
    Scene myScene = new Scene(rootNode, 220, 120);

    // Set the scene on the stage.
    myStage.setScene(myScene);

    // Create a label that will report the selection.
    response = new Label("");

    // Create the radio buttons.
    RadioButton rbTrain = new RadioButton("Train");
    RadioButton rbCar = new RadioButton("Car");
    RadioButton rbPlane = new RadioButton("Airplane");

    // Create a toggle group.
    ToggleGroup tg = new ToggleGroup();

    // Add each button to a toggle group.
    rbTrain.setToggleGroup(tg);
    rbCar.setToggleGroup(tg);
    rbPlane.setToggleGroup(tg);

    // Handle action events for the radio buttons.
    rbTrain.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        response.setText("Transport selected is train.");
      }
    });

    rbCar.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        response.setText("Transport selected is car.");
      }
    });

    rbPlane.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        response.setText("Transport selected is airplane.");
```

```
      }
    });

    // Fire the event for the first selection. This causes
    // that radio button to be selected and an action event
    // for that button to occur.
    rbTrain.fire();

    // Add the label and buttons to the scene graph.
    rootNode.getChildren().addAll(rbTrain, rbCar, rbPlane, response);

    // Show the stage and its scene.
    myStage.show();
  }
}
```

Sample output is shown here:

In the program, pay special attention to how the radio buttons and the toggle group are created. First, the buttons are created using this sequence:

```
RadioButton rbTrain = new RadioButton("Train");
RadioButton rbCar = new RadioButton("Car");
RadioButton rbPlane = new RadioButton("Airplane");
```

Next, a **ToggleGroup** is constructed:

```
ToggleGroup tg = new ToggleGroup();
```

Finally, each radio button is added to the toggle group:

```
rbTrain.setToggleGroup(tg);
rbCar.setToggleGroup(tg);
rbPlane.setToggleGroup(tg);
```

As explained, radio buttons must be part of a toggle group in order for their mutually exclusive behavior to be activated.

After the event handlers for each radio button have been defined, the **rbTrain** button is selected by calling **fire( )** on it. This causes that button to be selected and an action event to be generated for it. This causes the button to be initialized with the default selection.

Handling Change Events in a Toggle Group

Although there is nothing wrong, per se, with managing radio buttons by handling action events, as just shown, sometimes it is more appropriate (and easier) to listen to the entire toggle group for changes. When a change takes place, the event handler can easily determine which radio button has been selected and take action accordingly. To use this approach, you must register a **ChangeListener** on the toggle group. When a change event occurs, you

can then determine which button was selected. To try this approach, remove the action event handlers and the call to **fire( )** from the preceding program and substitute the following:

```
// Use a change listener to respond to a change of selection within
// the group of radio buttons.
tg.selectedToggleProperty().addListener(new ChangeListener<Toggle>() {
  public void changed(ObservableValue<? extends Toggle> changed,
                      Toggle oldVal, Toggle newVal) {

    // Cast new to RadioButton.
    RadioButton rb = (RadioButton) newVal;

    // Display the selection.
    response.setText("Transport selected is " + rb.getText());
  }
});

// Select the first button. This will cause a change event
// on the toggle group.
rbTrain.setSelected(true);
```

You will also need to add this **import** statement:

```
import javafx.beans.value.*;
```

It supports the **ChangeListener** interface.

The output from this program is the same as before; each time a selection is made, the **response** label is updated. However, in this case, only one event handler is needed for the enter group, rather than three (one for each button). Let's now look at this code more closely.

First, a change event listener is registered for the toggle group. To listen for change events, you must implement the **ChangeListener** interface. This is done by calling **addListener( )** on the object returned by **selectedToggleProperty( )**. The **ChangeListener** interface defines only one method, called **changed( )**. It is shown here:

void changed(ObservableValue<? extends T> *changed*, T *oldVal*, T *newVal*)

In this case, *changed* is the instance of **ObservableValue<T>**, which encapsulates an object that can be watched for changes. The *oldVal* and *newVal* parameters pass the previous value and the new value, respectively. Thus, in this case, *newVal* holds a reference to the radio button that has just been selected.

In this example, the **setSelected( )** method, rather than **fire( )**, is called to set the initial selection. Because setting the initial selection causes a change to the toggle group, it results in a change event being generated when the program first begins. You can also use **fire( )**, but **setSelected( )** was used to demonstrate that any change to the toggle group generates a change event.

An Alternative Way to Handle Radio Buttons

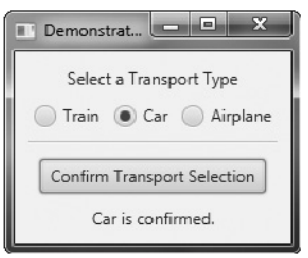
Although handling events generated by radio buttons is often useful, sometimes it is more appropriate to ignore those events and simply obtain the currently selected button when that information is needed. This approach is demonstrated by the following program. It

adds a button called Confirm Transport Selection. When this button is pressed, the currently selected radio button is obtained and then the selected transport is displayed in a label. When you try the program, notice that changing the selected radio button does not cause the confirmed transport to change until you press the Confirm Transport Selection button.

```
// This radio button example demonstrates how the
// currently selected button in a group can be obtained
// under program control, when it is needed, rather
// than responding to action or change events.
//
// In this example, no events related to the radio
// buttons are handled. Instead, the current selection
// is simply obtained when the Confirm Transport Selection push
// button is pressed.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class RadioButtonDemo2 extends Application {

  Label response;
  ToggleGroup tg;

  public static void main(String[] args) {

    // Start the JavaFX application by calling launch().
    launch(args);
  }

  // Override the start() method.
  public void start(Stage myStage) {

    // Give the stage a title.
    myStage.setTitle("Demonstrate Radio Buttons");

    // Use a FlowPane for the root node. In this case,
    // vertical and horizontal gaps of 10.
    FlowPane rootNode = new FlowPane(10, 10);

    // Center the controls in the scene.
    rootNode.setAlignment(Pos.CENTER);

    // Create a scene.
    Scene myScene = new Scene(rootNode, 200, 140);

    // Set the scene on the stage.
    myStage.setScene(myScene);
```

```
    // Create two labels.
    Label choose = new Label("    Select a Transport Type    ");
    response = new Label("No transport confirmed");

    // Create push button used to confirm the selection.
    Button btnConfirm = new Button("Confirm Transport Selection");

    // Create the radio buttons.
    RadioButton rbTrain = new RadioButton("Train");
    RadioButton rbCar = new RadioButton("Car");
    RadioButton rbPlane = new RadioButton("Airplane");

    // Create a toggle group.
    tg = new ToggleGroup();

    // Add each button to a toggle group.
    rbTrain.setToggleGroup(tg);
    rbCar.setToggleGroup(tg);
    rbPlane.setToggleGroup(tg);

    // Initially select one of the radio buttons.
    rbTrain.setSelected(true);

    // Handle action events for the confirm button.
    btnConfirm.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        // Get the radio button that is currently selected.
        RadioButton rb = (RadioButton) tg.getSelectedToggle();

        // Display the selection.
        response.setText(rb.getText() + " is confirmed.");
      }
    });

    // Use a separator to better organize the layout.
    Separator separator = new Separator();
    separator.setPrefWidth(180);

    // Add the label and buttons to the scene graph.
    rootNode.getChildren().addAll(choose, rbTrain, rbCar, rbPlane,
                                  separator, btnConfirm, response);

    // Show the stage and its scene.
    myStage.show();
  }
}
```

The output from the program is shown here:



Most of the program is easy to understand, but two key points are of special interest. First, inside the action event handler for the **btnConfirm** button, notice that the selected radio button is obtained by the following line:

```
RadioButton rb = (RadioButton) tg.getSelectedToggle();
```

Here, the **getSelectedToggle( )** method (defined by **ToggleGroup**) obtains the current selection for the toggle group (which, in this case, is a group of radio buttons). It is shown here:

final Toggle getSelectedToggle( )

It returns a reference to the **Toggle** that is selected. In this case, the return value is cast to **RadioButton** because this is the type of button in the group.

The second thing to notice is the use of a visual separator, which is created by this sequence:

```
Separator separator = new Separator();
separator.setPrefWidth(180);
```

The **Separator** class creates a line, which can be either vertical or horizontal. By default, it creates a horizontal line. (A second constructor lets you choose a vertical separator.) **Separator** helps visually organize the layout of controls. It is packaged in **javafx.scene.control**. Next, the width of the separator line is set by calling **setPrefWidth( )**, passing in the width.

CheckBox

The **CheckBox** class encapsulates the functionality of a check box. Its immediate superclass is **ButtonBase**. Although you are no doubt familiar with check boxes because they are widely used controls, the JavaFX check box is a bit more sophisticated than you may at first think. This is because **CheckBox** supports three states. The first two are checked or unchecked, as you would expect, and this is the default behavior. The third state is *indeterminate* (also called *undefined*). It is typically used to indicate that the state of the check box has not been set or that it is not relevant to a specific situation. If you need the indeterminate state, you will need to explicitly enable it.

CheckBox defines two constructors. The first is the default constructor. The second lets you specify a string that identifies the box. It is shown here:

CheckBox(String *str*)

It creates a check box that has the text specified by *str* as a label. As with other buttons, a **CheckBox** generates an action event when it is selected.

Here is a program that demonstrates check boxes. It displays check boxes that let the user select various deployment options, which are Web, Desktop, and Mobile. Each time a check box state changes, an action event is generated and handled by displaying the new state (selected or cleared) and by displaying a list of all selected boxes.

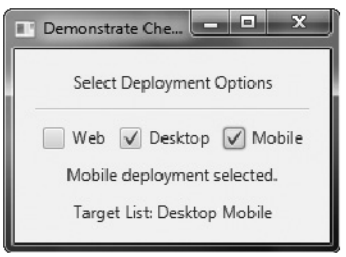
```
// Demonstrate Check Boxes.

import javafx.application.*;
import javafx.scene.*;
```

```
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class CheckboxDemo extends Application {

  CheckBox cbWeb;
  CheckBox cbDesktop;
  CheckBox cbMobile;

  Label response;
  Label allTargets;

  String targets = "";

  public static void main(String[] args) {

    // Start the JavaFX application by calling launch().
    launch(args);
  }

  // Override the start() method.
  public void start(Stage myStage) {

    // Give the stage a title.
    myStage.setTitle("Demonstrate Checkboxes");

    // Use a FlowPane for the root node. In this case,
    // vertical and horizontal gaps of 10.
    FlowPane rootNode = new FlowPane(10, 10);

    // Center the controls in the scene.
    rootNode.setAlignment(Pos.CENTER);

    // Create a scene.
    Scene myScene = new Scene(rootNode, 230, 140);

    // Set the scene on the stage.
    myStage.setScene(myScene);

    Label heading = new Label("Select Deployment Options");

    // Create a label that will report the state of the
    // selected check box.
    response = new Label("No Deployment Selected");

    // Create a label that will report all targets selected.
    allTargets = new Label("Target List: <none>");

    // Create the check boxes.
    cbWeb = new CheckBox("Web");
```

```
    cbDesktop = new CheckBox("Desktop");
    cbMobile = new CheckBox("Mobile");

    // Handle action events for the check boxes.
    cbWeb.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        if(cbWeb.isSelected())
          response.setText("Web deployment selected.");
        else
          response.setText("Web deployment cleared.");

        showAll();
      }
    });

    cbDesktop.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        if(cbDesktop.isSelected())
          response.setText("Desktop deployment selected.");
        else
          response.setText("Desktop deployment cleared.");

        showAll();
      }
    });

    cbMobile.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        if(cbMobile.isSelected())
          response.setText("Mobile deployment selected.");
        else
          response.setText("Mobile deployment cleared.");

        showAll();
      }
    });

    // Use a separator to better organize the layout.
    Separator separator = new Separator();
    separator.setPrefWidth(200);

    // Add controls to the scene graph.
    rootNode.getChildren().addAll(heading, separator, cbWeb, cbDesktop,
                                  cbMobile, response, allTargets);

    // Show the stage and its scene.
    myStage.show();
  }

  // Update and show the targets list.
  void showAll() {
    targets = "";
    if(cbWeb.isSelected()) targets = "Web ";
```

```
    if(cbDesktop.isSelected()) targets += "Desktop ";
    if(cbMobile.isSelected()) targets += "Mobile";

    if(targets.equals("")) targets = "<none>";

    allTargets.setText("Target List: " + targets);
  }
}
```

Sample output is shown here:



The operation of this program is straightforward. Each time a check box is changed, an action command is generated. To determine if the box is checked or unchecked, the **isSelected( )** method is called.

As mentioned, by default, **CheckBox** implements two states: checked and unchecked. If you want to add the indeterminate state, it must be explicitly enabled. To do this, call **setAllowIndeterminate( )**, shown here:

final void setAllowIndeterminate(boolean *enable*)

In this case, if *enable* is **true**, the indeterminate state is enabled. Otherwise, it is disabled. When the indeterminate state is enabled, the user can select between checked, unchecked, and indeterminate.

You can determine if a check box is in the indeterminate state by calling **isIndeterminate( )**, shown here:

final boolean isIndeterminate( )

It returns **true** if the checkbox state is indeterminate and **false** otherwise.

You can see the effect of a three-state check box by modifying the preceding program. First, enable the indeterminate state on the check boxes by calling **setAllowIndeterminate( )** on each check box, as shown here:

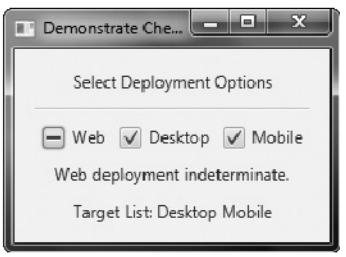
```
cbWeb.setAllowIndeterminate(true);
cbDesktop.setAllowIndeterminate(true);
cbMobile.setAllowIndeterminate(true);
```

Next, handle the indeterminate state inside the action event handlers. For example, here is the modified handler for **cbWeb**:

```
cbWeb.setOnAction(new EventHandler<ActionEvent>() {
  public void handle(ActionEvent ae) {
    if(cbWeb.isIndeterminate())
```

```
      response.setText("Web deployment indeterminate.");
    else if(cbWeb.isSelected())
      response.setText("Web deployment selected.");
    else
      response.setText("Web deployment cleared.");

    showAll();
  }
});
```

Now, all three states are tested. Update the other two handlers in the same way. After making these changes, the indeterminate state can be selected, as this sample output shows:



Here, the Web check box is indeterminate.

ListView

Another commonly used control is the list view, which in JavaFX is encapsulated by **ListView**. List views are controls that display a list of entries from which you can select one or more. Because of their ability to make efficient use of limited screen space, list views are popular alternatives to other types of selection controls.

ListView is a generic class that is declared like this:

```
class ListView<T>
```

Here, **T** specifies the type of entries stored in the list view. Often, these are entries of type **String**, but other types are also allowed.

ListView defines two constructors. The first is the default constructor, which creates an empty **ListView**. The second lets you specify the list of entries in the list. It is shown here:

```
ListView(ObservableList<T> list)
```

Here, *list* specifies a list of the items that will be displayed. It is an object of type **ObservableList**, which defines a list of observable objects. It inherits **java.util.List**. Thus, it supports the standard collection methods. **ObservableList** is packaged in **javafx.collections**.

Probably the easiest way to create an **ObservableList** for use in a **ListView** is to use the factory method **observableArrayList( )**, which is a static method defined by the **FXCollections** class (which is also packaged in **javafx.collections**). The version we will use is shown here:

```
static <E> ObservableList<E> observableArrayList( E ... elements)
```

In this case, **E** specifies the type of elements, which are passed via *elements*.

By default, a **ListView** allows only one item in the list to be selected at any one time. However, you can allow multiple selections by changing the selection mode. For now, we will use the default, single-selection model.

Although **ListView** provides a default size, sometimes you will want to set the preferred height and/or width to best match your needs. One way to do this is to call the **setPrefHeight( )** and **setPrefWidth( )** methods, shown here:

final void setPrefHeight(double *height*)

final void setPrefWidth(double *width*)

Alternatively, you can use a single call to set both dimensions at the same time by use of **setPrefSize( )**, shown here:

void setPrefSize(double *width*, double *height*)

There are two basic ways in which you can use a **ListView**. First, you can ignore events generated by the list and simply obtain the selection in the list when your program needs it. Second, you can monitor the list for changes by registering a change listener. This lets you respond each time the user changes a selection in the list. This is the approach used here.

To listen for change events, you must first obtain the selection model used by the **ListView**. This is done by calling **getSelectionModel( )** on the list. It is shown here:

final MultipleSelectionModel<T> getSelectionModel( )

It returns a reference to the model. **MultipleSelectionModel** is a class that defines the model used for multiple selections, and it inherits **SelectionModel**. However, multiple selections are allowed in a **ListView** only if multiple-selection mode is turned on.

Using the model returned by **getSelectionModel( )**, you will obtain a reference to the selected item property that defines what takes place when an element in the list is selected. This is done by calling **selectedItemProperty( )**, shown next:

final ReadOnlyObjectProperty<T> selectedItemProperty( )

You will add the change listener to this property.

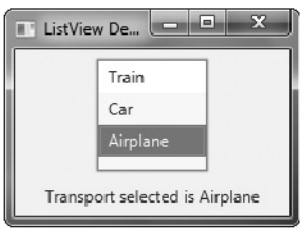
The following example puts the preceding discussion into action. It creates a list view that displays various types of transportation, allowing the user to select one. When one is chosen, the selection is displayed.

```
// Demonstrate a list view.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.geometry.*;
import javafx.beans.value.*;
import javafx.collections.*;

public class ListViewDemo extends Application {

  Label response;
```

```
public static void main(String[] args) {

  // Start the JavaFX application by calling launch().
  launch(args);
}

// Override the start() method.
public void start(Stage myStage) {

  // Give the stage a title.
  myStage.setTitle("ListView Demo");

  // Use a FlowPane for the root node. In this case,
  // vertical and horizontal gaps of 10.
  FlowPane rootNode = new FlowPane(10, 10);

  // Center the controls in the scene.
  rootNode.setAlignment(Pos.CENTER);

  // Create a scene.
  Scene myScene = new Scene(rootNode, 200, 120);

  // Set the scene on the stage.
  myStage.setScene(myScene);

  // Create a label.
  response = new Label("Select Transport Type");

  // Create an ObservableList of entries for the list view.
  ObservableList<String> transportTypes =
    FXCollections.observableArrayList( "Train", "Car", "Airplane" );

  // Create the list view.
  ListView<String> lvTransport = new ListView<String>(transportTypes);

  // Set the preferred height and width.
  lvTransport.setPrefSize(80, 80);

  // Get the list view selection model.
  MultipleSelectionModel<String> lvSelModel =
                                    lvTransport.getSelectionModel();

  // Use a change listener to respond to a change of selection within
  // a list view.
  lvSelModel.selectedItemProperty().addListener(
                                     new ChangeListener<String>() {
    public void changed(ObservableValue<? extends String> changed,
                        String oldVal, String newVal) {

      // Display the selection.
      response.setText("Transport selected is " + newVal);
    }
   });
```

```
    // Add the label and list view to the scene graph.
    rootNode.getChildren().addAll(lvTransport, response);

    // Show the stage and its scene.
    myStage.show();
  }
}
```

Sample output is shown here:



In the program, pay special attention to how the **ListView** is constructed. First, an **ObservableList** is created by this line:

```
ObservableList<String> transportTypes =
  FXCollections.observableArrayList( "Train", "Car", "Airplane" );
```

It uses the **observableArrayList( )** method to create a list of strings. Then, the **ObservableList** is used to initialize a **ListView**, as shown here:

```
ListView<String> lvTransport = new ListView<String>(transportTypes);
```

The program then sets the preferred width and height of the control.

Now, notice how the selection model is obtained for **lvTransport**:

```
MultipleSelectionModel<String> lvSelModel =
                                  lvTransport.getSelectionModel();
```

As explained, **ListView** uses **MultipleSelectionModel**, even when only a single selection is allowed. The **selectedItemProperty( )** method is then called on the model and a change listener is registered to the returned item.

ListView Scrollbars

One very useful feature of **ListView** is that when the number of items in the list exceeds the number that can be displayed within its dimensions, scrollbars are automatically added. For example, if you change the declaration of **transportTypes** so that it includes "Bicycle" and "Walking", as shown here:

```
ObservableList<String> transportTypes =
  FXCollections.observableArrayList( "Train", "Car", "Airplane",
                                     "Bicycle", "Walking" );
```

the **lvTransport** control now looks like the one shown here:

Enabling Multiple Selections

If you want to allow more than one item to be selected, you must explicitly request it. To do so, you must set the selection mode to **SelectionMode.MULTIPLE** by calling **setSelectionMode( )** on the **ListView** model. It is shown here:

final void setSelectionMode(SelectionMode *mode*)

In this case, *mode* must be either **SelectionMode.MULTIPLE** or **SelectionMode.SINGLE**.

When multiple-selection mode is enabled, you can obtain the list of the selections two ways: as a list of selected indices or as a list of selected items. We will use a list of selected items, but the procedure is similar when using a list of the indices of the selected items. (Note, indexing of items in a **ListView** begins at zero.)

To get a list of the selected items, call **getSelectedItems( )** on the selection model. It is shown here:

ObservableList<T> getSelectedItems( )

It returns an **ObservableList** of the items. Because **ObservableList** extends **java.util.List**, you can access the items in the list just as you would any other **List** collection.

To experiment with multiple selections, you can modify the preceding program as follows. First, make **lvTransport final** so it can be accessed within the change event handler. Next, add this line:

```
lvTransport.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
```

It enables multiple-selection mode for **lvTransport**. Finally, replace the change event handler with the one shown here:

```
lvSelModel.selectedItemProperty().addListener(
                                  new ChangeListener<String>() {
  public void changed(ObservableValue<? extends String> changed,
                      String oldVal, String newVal) {

    String selItems = "";
    ObservableList<String> selected =
              lvTransport.getSelectionModel().getSelectedItems();
```

```
    // Display the selections.
    for(int i=0; i < selected.size(); i++)
      selItems += "\n      " + selected.get(i);

    response.setText("All transports selected: " + selItems);
  }
});
```

After making these changes, the program will display all selected forms of transports, as the following output shows:

ComboBox

A control related to the list view is the combo box, which is implemented in JavaFX by the **ComboBox** class. A combo box displays one selection, but it will also display a drop-down list that allows the user to select a different item. You can also allow the user to edit a selection. **ComboBox** inherits **ComboBoxBase**, which provides much of its functionality. Unlike the **ListView**, which can allow multiple selections, **ComboBox** is designed for single-selection.

ComboBox is a generic class that is declared like this:

```
class ComboBox<T>
```

Here, **T** specifies the type of entries. Often, these are entries of type **String**, but other types are also allowed.

ComboBox defines two constructors. The first is the default constructor, which creates an empty **ComboBox**. The second lets you specify the list of entries. It is shown here:

```
ComboBox(ObservableList<T> list)
```

In this case, *list* specifies a list of the items that will be displayed. It is an object of type **ObservableList**, which defines a list of observable objects. As explained in the previous section, **ObservableList** inherits **java.util.List**. As also previously explained, an easy way to create an **ObservableList** is to use the factory method **observableArrayList( )**, which is a static method defined by the **FXCollections** class.

A **ComboBox** generates an action event when its selection changes. It will also generate a change event. Alternatively, it is also possible to ignore events and simply obtain the current selection when needed.

You can obtain the current selection by calling **getValue( )**, shown here:

```
final T getValue( )
```

If the value of a combo box has not yet been set (by the user or under program control), then **getValue( )** will return **null**. To set the value of a **ComboBox** under program control, call **setValue( )**:

final void setValue(T *newVal*)

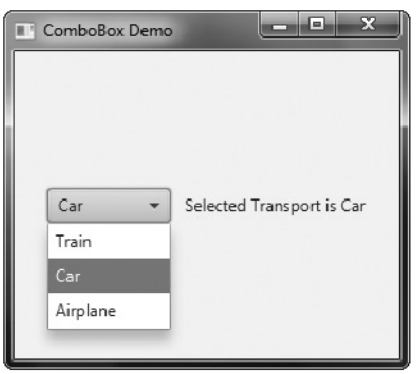
Here, *newVal* becomes the new value.

The following program demonstrates a combo box by reworking the previous list view example. It handles the action event generated by the combo box.

```
// Demonstrate a combo box.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.geometry.*;
import javafx.collections.*;
import javafx.event.*;


public class ComboBoxDemo extends Application {

  ComboBox<String> cbTransport;
  Label response;

  public static void main(String[] args) {

    // Start the JavaFX application by calling launch().
    launch(args);
  }

  // Override the start() method.
  public void start(Stage myStage) {

    // Give the stage a title.
    myStage.setTitle("ComboBox Demo");

    // Use a FlowPane for the root node. In this case,
    // vertical and horizontal gaps of 10.
    FlowPane rootNode = new FlowPane(10, 10);

    // Center the controls in the scene.
    rootNode.setAlignment(Pos.CENTER);

    // Create a scene.
    Scene myScene = new Scene(rootNode, 280, 120);

    // Set the scene on the stage.
    myStage.setScene(myScene);

    // Create a label.
    response = new Label();
```

```
    // Create an ObservableList of entries for the combo box.
    ObservableList<String> transportTypes =
      FXCollections.observableArrayList( "Train", "Car", "Airplane" );

    // Create a combo box.
    cbTransport = new ComboBox<String>(transportTypes);

    // Set the default value.
    cbTransport.setValue("Train");

    // Set the response label to indicate the default selection.
    response.setText("Selected Transport is " + cbTransport.getValue());

    // Listen for action events on the combo box.
    cbTransport.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        response.setText("Selected Transport is " + cbTransport.getValue());
      }
    });

    // Add the label and combo box to the scene graph.
    rootNode.getChildren().addAll(cbTransport, response);

    // Show the stage and its scene.
    myStage.show();
  }
}
```

Sample output is shown here:



As mentioned, **ComboBox** can be configured to allow the user to edit a selection. Assuming that it contains only entries of type **String**, it is easy to enable editing capabilities. Simply call **setEditable( )**, shown here:

final void setEditable(boolean *enable*)

If *enable* is **true**, editing is enabled. Otherwise, it is disabled. To see the effects of editing, add this line to the preceding program:

```
cbTransport.setEditable(true);
```

After making this addition, you will be able to edit the selection.

ComboBox supports many additional features and functionality beyond those mentioned here. You might find it interesting to explore further. Also, an alternative to a combo box in some cases is the **ChoiceBox** control. You will find it easy to use because it is has similarities to both **ListView** and **ComboBox**.

TextField

Although the controls discussed earlier are quite useful and are frequently found in a user interface, they all implement a means of selecting a predetermined option or action. However, sometimes you will want the user to enter a string of his or her own choosing. To accommodate this type of input, JavaFX includes several text-based controls. The one we will look at is **TextField**. It allows one line of text to be entered. Thus, it is useful for obtaining names, ID strings, addresses, and the like. Like all text controls, **TextField** inherits **TextInputControl**, which defines much of its functionality.

TextField defines two constructors. The first is the default constructor, which creates an empty text field that has the default size. The second lets you specify the initial contents of the field. Here, we will use the default constructor.

Although the default size is sometimes adequate, often you will want to specify its size. This is done by calling **setPrefColumnCount( )**, shown here:

final void setPrefColumnCount(int *columns*)

The *columns* value is used by **TextField** to determine its size.

You can set the text in a text field by calling **setText( )**. You can obtain the current text by calling **getText( )**. In addition to these fundamental operations, **TextField** supports several other capabilities that you might want to explore, such as cut, paste, and append. You can also select a portion of the text under program control.

One especially useful **TextField** option is the ability to set a prompting message inside the text field when the user attempts to use a blank field. To do this, call **setPromptText( )**, shown here:

final void setPromptText(String *str*)

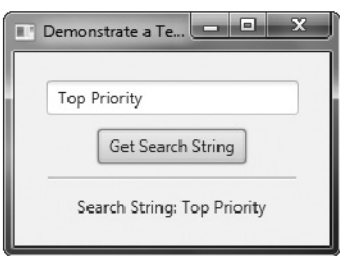
In this case, *str* is the string displayed in the text field when no text has been entered. It is displayed using low-intensity (such as a gray tone).

When the user presses ENTER while inside a **TextField**, an action event is generated. Although handling this event is often helpful, in some cases, your program will simply obtain the text when it is needed, rather than handling action events. Both approaches are demonstrated by the following program. It creates a text field that requests a search string. When the user presses ENTER while the text field has input focus, or presses the Get Search String button, the string is obtained and displayed. Notice that a prompting message is also included.

```
// Demonstrate a text field.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class TextFieldDemo extends Application {

  TextField tf;
  Label response;

  public static void main(String[] args) {

    // Start the JavaFX application by calling launch().
    launch(args);
  }

  // Override the start() method.
  public void start(Stage myStage) {

    // Give the stage a title.
    myStage.setTitle("Demonstrate a TextField");

    // Use a FlowPane for the root node. In this case,
    // vertical and horizontal gaps of 10.
    FlowPane rootNode = new FlowPane(10, 10);

    // Center the controls in the scene.
    rootNode.setAlignment(Pos.CENTER);

    // Create a scene.
    Scene myScene = new Scene(rootNode, 230, 140);

    // Set the scene on the stage.
    myStage.setScene(myScene);

    // Create a label that will report the contents of the
    // text field.
    response = new Label("Search String: ");

    // Create a button that gets the text.
    Button btnGetText = new Button("Get Search String");

    // Create a text field.
    tf = new TextField();

    // Set the prompt.
    tf.setPromptText("Enter Search String");
```

```
    // Set preferred column count.
    tf.setPrefColumnCount(15);

    // Handle action events for the text field. Action
    // events are generated when ENTER is pressed while
    // the text field has input focus. In this case, the
    // text in the field is obtained and displayed.
    tf.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        response.setText("Search String: " + tf.getText());
      }
    });

    // Get text from the text field when the button is pressed
    // and display it.
    btnGetText.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        response.setText("Search String: " + tf.getText());
      }
    });

    // Use a separator to better organize the layout.
    Separator separator = new Separator();
    separator.setPrefWidth(180);

    // Add controls to the scene graph.
    rootNode.getChildren().addAll(tf, btnGetText, separator, response);

    // Show the stage and its scene.
    myStage.show();
  }
}
```

Sample output is shown here:



Other text controls that you will want to examine include **TextArea**, which supports multiline text, and **PasswordField**, which can be used to input passwords. You might also find **HTMLEditor** helpful.

ScrollPane

Sometimes, the contents of a control will exceed the amount of screen space that you want to give to it. Here are two examples: A large image may not fit within reasonable boundaries, or you might want to display text that is longer than will fit within a small window. Whatever the reason, JavaFX makes it easy to provide scrolling capabilities to any node in a scene graph. This is accomplished by wrapping the node in a **ScrollPane**. When a **ScrollPane** is used, scrollbars are automatically implemented that scroll the contents of the wrapped node. No further action is required on your part. Because of the versatility of **ScrollPane**, you will seldom need to use individual scrollbar controls.

ScrollPane defines two constructors. The first is the default constructor. The second lets you specify a node that you want to scroll. It is shown here:

ScrollPane(Node *content*)

In this case, *content* specifies the information to be scrolled. When using the default constructor, you will add the node to be scrolled by calling **setContent( )**. It is shown here:

final void setContent(Node *content*)

After you have set the content, add the scroll pane to the scene graph. When displayed, the content can be scrolled.

NOTE You can also use **setContent( )** to change the content being scrolled by the scroll pane. Thus, what is being scrolled can be changed during the execution of your program.

Although a default size is provided, as a general rule, you will want to set the dimensions of the *viewport*. The viewport is the viewable area of a scroll pane. It is the area in which the content being scrolled is displayed. Thus, the viewport displays the visible portion of the content. The scrollbars scroll the content through the viewport. Thus, by moving a scrollbar, you change what part of the content is visible.

You can set the viewport dimensions by using these two methods:

final void setPrefViewportHeight(double *height*)

final void setPrefViewportWidth(double *width*)

In its default behavior, a **ScrollPane** will dynamically add or remove a scrollbar as needed. For example, if the component is taller than the viewport, a vertical scrollbar is added. If the component will completely fit within the viewport, the scrollbars are removed.

One nice feature of **ScrollPane** is its ability to pan the contents by dragging the mouse. By default, this feature is off. To turn it on, use **setPannable( )**, shown here:

final void setPannable(boolean *enable*)

If *enable* is **true**, then panning is allowed. Otherwise, it is disabled.

You can set the position of the scrollbars under program control using **setHvalue( )** and **setVvalue( )**, shown here:

final void setHvalue(double *newHval*)

final void setVvalue(double *newVval*)

The new horizontal position is specified by *newHval*, and the new vertical position is specified by *newVval*. Be default, scrollbar positions start at zero.

ScrollPane supports various other options. For example, it is possible to set the minimum and maximum scrollbar positions. You can also specify when and if the scrollbars are shown by setting a scrollbar policy. The current position of the scrollbars can be obtained by calling **getHvalue( )** and **getVvalue( )**.

The following program demonstrates **ScrollPane** by using one to scroll the contents of a multiline label. Notice that it also enables panning.

```
// Demonstrate a scroll pane.
// This program scrolls the contents of a multiline
// label, but any node can be scrolled.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class ScrollPaneDemo extends Application {

  ScrollPane scrlPane;

  public static void main(String[] args) {

    // Start the JavaFX application by calling launch().
    launch(args);
  }

  // Override the start() method.
  public void start(Stage myStage) {

    // Give the stage a title.
    myStage.setTitle("Demonstrate a ScrollPane");

    // Use a FlowPane for the root node.
    FlowPane rootNode = new FlowPane(10, 10);

    // Center the controls in the scene.
    rootNode.setAlignment(Pos.CENTER);

    // Create a scene.
    Scene myScene = new Scene(rootNode, 200, 200);

    // Set the scene on the stage.
    myStage.setScene(myScene);

    // Create a label that will be scrolled.
    Label scrlLabel = new Label(
                          "A ScrollPane streamlines the process of\n" +
```

```
                        "adding scroll bars to a window whose\n" +
                        "contents exceed the window's dimensions.\n" +
                        "It also enables a control to fit in a\n" +
                        "smaller space than it otherwise would.\n" +
                        "As such, it often provides a superior\n" +
                        "approach over using individual scroll bars.");

    // Create a scroll pane, setting scrlLabel as the content.
    scrlPane = new ScrollPane(scrlLabel);

    // Set the viewport width and height.
    scrlPane.setPrefViewportWidth(130);
    scrlPane.setPrefViewportHeight(80);

    // Enable panning.
    scrlPane.setPannable(true);

    // Create a reset label.
    Button btnReset = new Button("Reset Scroll Bar Positions");

    // Handle action events for the reset button.
    btnReset.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
        // Set the scroll bars to their zero position.
        scrlPane.setVvalue(0);
        scrlPane.setHvalue(0);
      }
    });

    // Add the label to the scene graph.
    rootNode.getChildren().addAll(scrlPane, btnReset);

    // Show the stage and its scene.
    myStage.show();
  }
}
```

Sample output is shown here:

TreeView

One of JavaFX's most powerful controls is the **TreeView**. It presents a hierarchical view of data in a tree-like format. In this context, the term *hierarchical* means some items are subordinate to others. For example, a tree is commonly used to display the contents of a file system. In this case, the individual files are subordinate to the directory that contains them. In a **TreeView**, branches can be expanded or collapsed on demand by the user. This allows hierarchical data to be presented in a compact, yet expandable form. Although **TreeView** supports many customization options, you will often find that the default tree style and capabilities are suitable. Therefore, even though trees support a sophisticated structure, they are still quite easy to work with.

TreeView implements a conceptually simple, tree-based data structure. A tree begins with a single *root node* that indicates the start of the tree. Under the root are one or more *child nodes* There are two types of child nodes: *leaf nodes* (also called *terminal nodes*), which have no children, and *branch nodes,* which form the root nodes of *subtrees*. A subtree is simply a tree that is part of a larger tree. The sequence of nodes that leads from the root to a specific node is called a *path*.

One very useful feature of **TreeView** is that it automatically provides scrollbars when the size of the tree exceeds the dimensions of the view. Although a fully collapsed tree might be quite small, its expanded form may be quite large. By automatically adding scrollbars as needed, **TreeView** lets you use a smaller space than would ordinarily be possible.

TreeView is a generic class that is defined like this:

```
class TreeView<T>
```

Here, **T** specifies the type of value held by an item in the tree. Often, this will be of type **String**. **TreeView** defines two constructors. This is the one we will use:

```
TreeView(TreeItem<T> rootNode)
```

Here, *rootNode* specifies the root of the tree. Because all nodes descend from the root, it is the only one that needs to be passed to **TreeView**.

The items that form the tree are objects of type **TreeItem**. At the outset, it is important to state that **TreeItem** does not inherit **Node**. Thus, **TreeItem**s are not general-purpose objects. They can be used in a **TreeView**, but not as stand-alone controls. **TreeItem** is a generic class, as shown here:

```
class TreeItem<T>
```

Here, **T** specifies the type of value held by the **TreeItem**.

Before you can use a **TreeView**, you must construct the tree that it will display. To do this, you must first create the root. Next, add other nodes to that root. You do this by calling either **add( )** or **addAll( )** on the list returned by **getChildren( )**. These other nodes can be leaf nodes or subtrees. After the tree has been constructed, you create the **TreeView** by passing the root node to its constructor.

You can handle selection events in the **TreeView** in a way similar to the way that you handle them in a **ListView**, through the use of a change listener. To do so, first, obtain the selection model by calling **getSelectionModel( )**. Then, call **selectedItemProperty( )** to obtain the property for the selected item. On that return value, call **addListener( )** to add a change listener. Each time a selection is made, a reference to the new selection will be

passed to the **changed( )** handler as the new value. (See **ListView** for more details on handling change events.)

You can obtain the value of a **TreeItem** by calling **getValue( )**. You can also follow the tree path of an item in either the forward or backward direction. To obtain the parent, call **getParent( )**. To obtain the children, call **getChildren( )**.

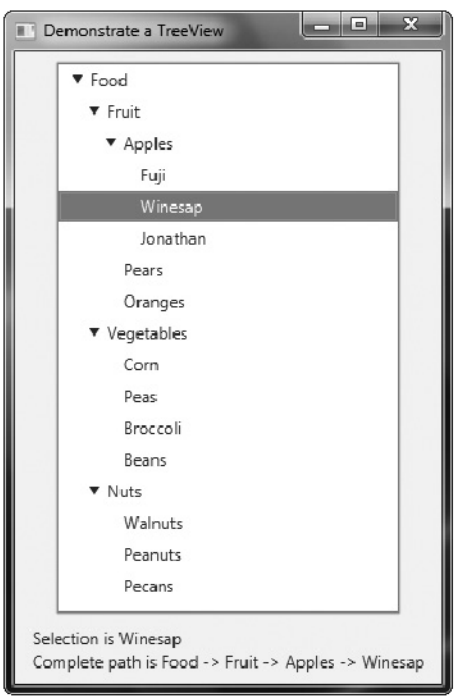
The following example shows how to build and use a **TreeView**. The tree presents a hierarchy of food. The type of items stored in the tree are strings. The root is labeled Food. Under it are three direct descendent nodes: Fruit, Vegetables, and Nuts. Under Fruit are three child nodes: Apples, Pears, and Oranges. Under Apples are three leaf nodes: Fuji, Winesap, and Jonathan. Each time a selection is made, the name of the item is displayed. Also, the path from the root to the item is shown. This is done by the repeated use of **getParent( )**.

```
// Demonstrate a TreeView

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.beans.value.*;
import javafx.geometry.*;

public class TreeViewDemo extends Application {

  Label response;

  public static void main(String[] args) {

    // Start the JavaFX application by calling launch().
    launch(args);
  }

  // Override the start() method.
  public void start(Stage myStage) {

    // Give the stage a title.
    myStage.setTitle("Demonstrate a TreeView");

    // Use a FlowPane for the root node. In this case,
    // vertical and horizontal gaps of 10.
    FlowPane rootNode = new FlowPane(10, 10);

    // Center the controls in the scene.
    rootNode.setAlignment(Pos.CENTER);

    // Create a scene.
    Scene myScene = new Scene(rootNode, 310, 460);

    // Set the scene on the stage.
    myStage.setScene(myScene);
```

```
// Create a label that will report the state of the
// selected tree item.
response = new Label("No Selection");

// Create tree items, starting with the root.
TreeItem<String> tiRoot = new TreeItem<String>("Food");

// Now add subtrees, beginning with fruit.
TreeItem<String> tiFruit = new TreeItem<String>("Fruit");

// Construct the Apple subtree.
TreeItem<String> tiApples = new TreeItem<String>("Apples");

// Add child nodes to the Apple node.
tiApples.getChildren().add(new TreeItem<String>("Fuji"));
tiApples.getChildren().add(new TreeItem<String>("Winesap"));
tiApples.getChildren().add(new TreeItem<String>("Jonathan"));

// Add varieties to the fruit node.
tiFruit.getChildren().add(tiApples);
tiFruit.getChildren().add(new TreeItem<String>("Pears"));
tiFruit.getChildren().add(new TreeItem<String>("Oranges"));

// Finally, add the fruit node to the root.
tiRoot.getChildren().add(tiFruit);

// Now, add vegetables subtree, using the same general process.
TreeItem<String> tiVegetables = new TreeItem<String>("Vegetables");
tiVegetables.getChildren().add(new TreeItem<String>("Corn"));
tiVegetables.getChildren().add(new TreeItem<String>("Peas"));
tiVegetables.getChildren().add(new TreeItem<String>("Broccoli"));
tiVegetables.getChildren().add(new TreeItem<String>("Beans"));
tiRoot.getChildren().add(tiVegetables);

// Likewise, add nuts subtree.
TreeItem<String> tiNuts = new TreeItem<String>("Nuts");
tiNuts.getChildren().add(new TreeItem<String>("Walnuts"));
tiNuts.getChildren().add(new TreeItem<String>("Peanuts"));
tiNuts.getChildren().add(new TreeItem<String>("Pecans"));
tiRoot.getChildren().add(tiNuts);

// Create tree view using the tree just created.
TreeView<String> tvFood = new TreeView<String>(tiRoot);

// Get the tree view selection model.
MultipleSelectionModel<TreeItem<String>> tvSelModel =
                                  tvFood.getSelectionModel();

// Use a change listener to respond to a selection within
// a tree view
tvSelModel.selectedItemProperty().addListener(
                        new ChangeListener<TreeItem<String>>() {
  public void changed(
              ObservableValue<? extends TreeItem<String>> changed,
                  TreeItem<String> oldVal, TreeItem<String> newVal) {
```

```
        // Display the selection and its complete path from the root.
        if(newVal != null) {

          // Construct the entire path to the selected item.
          String path = newVal.getValue();
          TreeItem<String> tmp = newVal.getParent();
          while(tmp != null) {
            path = tmp.getValue() + " -> " + path;
            tmp = tmp.getParent();
          }

          // Display the selection and the entire path.
          response.setText("Selection is " + newVal.getValue() +
                           "\nComplete path is " + path);
        }
      }
     });

    // Add controls to the scene graph.
    rootNode.getChildren().addAll(tvFood, response);

    // Show the stage and its scene.
    myStage.show();
  }
}
```

Sample output is shown here:



There are two things to pay special attention to in this program. First, notice how the tree is constructed. First, the root node is created by this statement:

```
TreeItem<String> tiRoot = new TreeItem<String>("Food");
```

Next, the nodes under the root are constructed. These nodes consist of the root nodes of subtrees: one for fruit, one for vegetables, and one for nuts. Next, the leaves are added to these subtrees. However, one of these, the fruit subtree, consists of another subtree that contains varieties of apples. The point here is that each branch in a tree leads either to a leaf or to the root of a subtree. After all of the nodes have been constructed, the root nodes of each subtree are added to the root node of the tree. This is done by calling **add( )** on the root node. For example, this is how the Nuts subtree is added to **tiRoot**.

```
tiRoot.getChildren().add(tiNuts);
```

The process is the same for adding any child node to its parent node.

The second thing to notice in the program is the way the path from the root to the selected node is constructed within the change event handler. It is shown here:

```
String path = newVal.getValue();
TreeItem<String> tmp = newVal.getParent();
while(tmp != null) {
  path = tmp.getValue() + " -> " + path;
  tmp = tmp.getParent();
}
```

The code works like this: First, the value of the newly selected node is obtained. In this example, the value will be a string, which is the node's name. This string is assigned to the **path** string. Then, a temporary variable of type **TreeItem<String>** is created and initialized to refer to the parent of the newly selected node. If the newly selected node does not have a parent, then **tmp** will be null. Otherwise, the loop is entered, within which each parent's value (which is its name in this case) is added to **path**. This process continues until the root node of the tree (which has no parent) is found.

Although the preceding shows the basic mechanism required to handle a **TreeView**, it is important to point out that several customizations and options are supported. **TreeView** is a powerful control that you will want to examine fully on your own.

Introducing Effects and Transforms

A principal advantage of JavaFX is its ability to alter the precise look of a control (or any node in the scene graph) through the application of an *effect* and/or a *transform*. Both effects and transforms help give your GUI the sophisticated, modern look that users have come to expect. Although it is beyond the scope of this book to examine each effect and transform supported by JavaFX, the following introduction will give you an idea of the benefits they provide.

Effects

Effects are supported by the abstract **Effect** class and its concrete subclasses, which are packaged in **javafx.scene.effect**. Using these effects, you can customize the way a node in a scene graph looks. Several built-in effects are provided. Here is a sampling:

Bloom	Increases the brightness of the brighter parts of a node.
BoxBlur	Blurs a node.
DropShadow	Displays a shadow that appears behind the node.
Glow	Produces a glowing effect.
InnerShadow	Displays a shadow inside a node.
Lighting	Creates shadow effects of a light source.
Reflection	Displays a reflection.

These, and the other effects, are easy to use and are available for use by any **Node**, including controls. Of course, depending on the control, some effects will be more appropriate than others.

To set an effect on a node, call **setEffect( )**, which is defined by **Node**. It is shown here:

final void setEffect(Effect *effect*)

Here, effect is the *effect* that will be applied. To specify no effect, pass **null**. Thus, to add an effect to a node, first create an instance of that effect and then pass it to **setEffect( )**. Once this has been done, the effect will be used whenever the node is rendered (as long as the effect is supported by the environment). To demonstrate the power of effects, we will use two of them: **Glow** and **InnerShadow**. However, the process of adding an effect is essentially the same no matter what effect you choose.

Glow produces an effect that gives a node a glowing appearance. The amount of glow is under your control. To use a glow effect, you must first create a **Glow** instance. This is the constructor that we will use:

Glow(double *glowLevel*)

Here, *glowLevel* specifies the amount of glowing, which must be a value between 0.0 and 1.0.

After a **Glow** instance has been created, the glow level can be changed by using **setLevel( )**, shown here:

final void setLevel(double *glowLevel*)

As before, *glowLevel* specifies the glow level, which must be between 0.0 and 1.0.

InnerShadow produces an effect that simulates a shadow on the inside of the node. It supports various constructors. This is the one we will use:

InnerShadow(double *radius*, Color *shadowColor*)