6.1 Introduction to JavaFX

Java's support of graphics and GUIs has evolved in various ways over the years. When the language was first introduced, it used a set of classes called the AWT, which stands for Abstract Windowing Toolkit. Later, the Swing API was introduced, which replaced the GUI components of the AWT with more versatile versions.

The JavaFX API has now replaced the AWT and Swing for developing graphical programs. JavaFX combines the best aspects of the previous approaches and adds many additional features. Oracle, the company that manages the Java language, no longer supports those older technologies, including Swing.

This chapter focuses on GUIs in which the user interacts with the program through controls such as buttons and text fields. JavaFX also provides the means for generating graphics such as drawn shapes and the management of images. These purely graphic elements of JavaFX are discussed in Appendix F.

Listing 6.1 shows a small JavaFX program in a class called HelloJavaFX. This program displays a window containing two text elements.

The first thing to note is that the HelloJavaFX class extends the JavaFX Application class. This process is making use of the object-oriented concept of inheritance, which we introduced in Chapter 1 and will explore in more detail in Chapter 8. All JavaFX programs extend the Application class.

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class HelloJavaFX extends Application {
```
LISTING 6.1 continued

// Creates and displays two Text objects in a JavaFX window.
// 
public void start(Stage primaryStage)
{
    Text hello = new Text(50, 50, "Hello, JavaFX!");
    Text question = new Text(120, 80, "How's it going?");

    Group root = new Group(hello, question);
    Scene scene = new Scene(root, 300, 120, Color.LIGHTGREEN);

    primaryStage.setTitle("A JavaFX Program");
    primaryStage.setScene(scene);
    primaryStage.show();
}

// Launched the JavaFX application. This method is not required
// in IDEs that launch JavaFX applications automatically.
// 
public static void main(String[] args)
{
    launch(args);
}

DISPLAY

A JavaFX Program

Hello, JavaFX!

How's it going?

See a full-color version of this figure at the end of the text.
Note that there are two methods in the HelloJavaFX class: a main method as we’ve seen in other programs, and the start method. The main method is used to call the launch method of the Application class. After performing some background set-up, the launch method calls the start method. You’ll typically use the start method to set up and display the primary window of the app.

In a development environment that fully supports JavaFX, the launch method is called automatically. Therefore, if you are using such an IDE, you do not need to write the main method. We will typically leave the main method out of our examples, though you may need to include it.

In this example, the start method creates two Text objects, adds them to a Group, which is then used as the root node of a Scene. The scene is displayed in the primary Stage. While that may seem like a lot of moving parts, there’s a pattern to the organization that you’ll quickly get used to.

JavaFX embraces a theatre metaphor. A Stage is a window. A program can make use of multiple stages if desired. The primary Stage object is created automatically and passed into the start method. In this example, the last three lines of the start method set the title that is displayed in the window’s title bar, set the scene to be displayed in the window, and then call the show method to display the window on the monitor.

The constructor of the Scene class accepts four parameters: a root node to be displayed, the preferred width and height of the scene, and its background color. In this example, the root element is a Group object, which is to be displayed in an area that is 300 pixels wide and 120 pixels high. The background is specified using a Color object that represents a light green color.

A scene displays a single element, often referred to as the root node. The root node may contain other elements, which themselves may contain other elements, and so on, creating a hierarchy of elements that make up the scene. In this example, the root node is a Group object, though there are other options as we’ll see in later examples. The group contains two Text objects, each of which represents a character string and the location at which the text should be displayed.

Unlike a traditional two-dimensional coordinate system, the origin point (0, 0) of the Java coordinate system is in the upper left corner of a graphical component. The x-axis coordinates get larger as you move to the right, and the y-axis coordinates get larger as you move down. See Appendix F for details.

The display point of the first Text object in this example is (50, 50), so the first character of the string “Hello JavaFX!” is displayed 50 pixels to the right, and 50 pixels down from the top left corner of the window. The second Text object is 120 pixels to the right, and 80 pixels down from the origin.
The **HalloJavaFX** example is not interactive—it simply displays some text objects. But it shows some of the basic infrastructure elements that we will use in all JavaFX applications. Let’s now look at an example of a truly interactive GUI.

### GUI Elements

Unlike a text-based program, or even a graphical program with no interaction, a program that has a GUI provides a heightened level of user interaction that often makes a program more effective and interesting.

Three kinds of objects cooperate to create a GUI in JavaFX:

- controls
- events
- event handlers

A GUI **control** is a screen element that displays information and/or allows the user to interact with a program in a certain way. Examples of GUI controls include buttons, text fields, scroll bars, and sliders.

An **event** is an object that represents some occurrence in which we may be interested. Often, events correspond to user actions, such as pushing a button or typing a key on the keyboard. GUI controls generate events to indicate the user action related to that control. For example, a button control will generate an event to indicate that the button has been pushed. A program that is oriented around a GUI, responding to events from the user, is called **event-driven**.

An **event handler** is an object that contains a method that is called when an event occurs. The programmer sets up the relationship between the component that generates an event and the handler that will respond to the event.

For the most part, we will use controls and events that are predefined in the JavaFX API. To set up a GUI program, we present and tailor the necessary controls and provide handlers to perform whatever actions we desire when events occur.

For example, the **PushCounter** program shown in Listing 6.2 presents the user with a single button (labeled “Push Me!”). Each time the button is pushed, a counter is updated and displayed.

This program displays a **Button** object and a **Text** object. The **Button** class represents a push button that allows the user to initiate an action with a mouse click. The **Button** constructor accepts a **String** parameter that specifies the text shown on the button.
```java
/******************************
// HelloJavaFX.java Java Foundations
//
// Demonstrates a graphical user interface in JavaFX.
******************************/

public class PushCounter extends Application {
    private int count;
    private Text countText;

    /*******************************************************************************/
    // Presents a GUI containing a button and a label that displays
    // how many times the button is pushed.
    /*******************************************************************************/
    public void start(Stage primaryStage) {
        count = 0;
        countText = new Text("Pushes: 0");
        Button push = new Button("Push Me!");
        push.setOnAction(this::processButtonPress);
        FlowPane pane = new FlowPane(push, countText);
        pane.setAlignment(Pos.CENTER);
        pane.setHgap(20);
        pane.setStyle("-fx-background-color: cyan");
        Scene scene = new Scene(pane, 300, 100);
        primaryStage.setTitle("Push Counter");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    /*******************************************************************************/
    // Updates the counter and label when the button is pushed.
    /*******************************************************************************/
    public void processButtonPress(ActionEvent event) {
        count++;
        countText.setText("Pushes: " + count);
    }
}
```
A `Button` object generates an `action event` when it is pushed. The button’s `setOnAction` method is used to specify the event handler for the button.

The `:` operator is used to specify a `method reference`, which was introduced to the language in Java 8. In this example, the method reference refers to the `processButtonPress` method in this class (the same one as the `start` method). The `this` reference refers to the object that is currently executing the method, so in this example the `PushCounter` class itself serves as the event handler for the button. There are other ways to specify the event handler relationship, which are discussed later in this section.

The `processButtonPress` method increments the counter variable and updates the text displayed. Note that the variables for the `Button` and `Text` objects are declared as instance data (at the class level), so that they can be referenced in both methods in the class.

The `Button` and `Text` controls are added to a `FlowPane`, which is used as the root node of the scene. In the previous example, a `Group` object as the root—a `Group` does not have any inherent organization of the elements it holds. But the `FlowPane` class is a layout pane (one of several defined in the JavaFX API). A `layout pane` is a container that governs how controls are arranged and presented visually. The nodes in a `FlowPane` are laid out horizontally in rows (which is the
default) or vertically in columns. When there is no more room, the next node in the pane flows into the next row or column. In this example, the button and text are arranged horizontally, centered, with a gap of 20 pixels.

By the way, the Text object used in this program could be replaced with a Label control to create a similar effect. Labels, however, are most appropriately used when labelling other controls, providing advanced keyboard navigation. Labels have many style properties that Text objects don’t have and can also contain images. In this example, a Text object is sufficient.

Alternate Ways to Specify Event Handlers

The PushCounter program used a method reference to define the event handler for the action event generated by the Button object. Let’s look at other ways to define an event handler.

An event handler is actually an object that implements the EventHandler interface. An interface is a list of methods that the implementing class must define. In this case, the EventHandler interface requires an object to define a method called handle to process the event. So, an alternative approach to creating an event handler is to define a full class that implements the EventHandler interface, perhaps as a private inner class within the PushCounter class:

```java
private class ButtonHandler implements EventHandler<ActionEvent> {
    public void handle(ActionEvent event) {
        count++;
        countText.setText("Pushes: " + count);
    }
}
```

Then the call to setOnAction for the button could specify such an object:

```java
push.setOnAction(new ButtonHandler());
```

Interfaces are discussed in more detail in Chapter 9.

Instead of defining a separate class, the event handler could be defined using a lambda expression:

```java
push.setOnAction((event) -> {
    count++;
    countText.setText("Pushes: " + count);
});
```
A lambda expression is defined by a set of parameters in parentheses, the -> arrow operator, followed by an expression. If one expression is insufficient, a block is used. The lambda expression in this example accepts the event object, which is passed to a block that contains our handler code.

A lambda expression can be used whenever an object of a functional interface is required. A functional interface is one that contains a single abstract method. The EventHandler interface is a functional interface.

The method reference approach used in the PushCounter program is equivalent to a lambda expression that supplies the parameter to the method. So this::
processButtonPress is equivalent to event -> processButtonPress(event).

We find the method reference approach to be the cleanest and easiest to follow, so we will often use that approach in our examples.

**Determining Event Sources**

Let's look at an example in which one event handler is used to process the events from multiple sources. Listing 6.3 shows a program that displays two buttons, labeled Red and Blue. When either button is pushed, the background color of the pane is changed accordingly. A FlowPane is used to layout the two buttons side by side.

```java
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.FlowPane;
import javafx.stage.Stage;

//*****************************************************************************
// RedOrBlue.java     Java Foundations
//*****************************************************************************
// Demonstrates the use of one handler for multiple buttons.
//*****************************************************************************
public class RedOrBlue extends Application {
    private Button redButton, blueButton;
    private FlowPane pane;
```
Listing 6.3 continued

```java
// Presents a GUI with two buttons that control the color of the
// pane background.
//
public void start(Stage primaryStage)
{
    redButton = new Button("Red!");
    redButton.setOnAction(this::processColorButton);

    blueButton = new Button("Blue!");
    blueButton.setOnAction(this::processColorButton);

    pane = new FlowPane(redButton, blueButton);
    pane.setAlignment(Pos.CENTER);
    pane.setHgap(20);
    pane.setStyle("-fx-background-color: white");

    Scene scene = new Scene(pane, 300, 100);

    primaryStage.setTitle("Red or Blue?");
    primaryStage.setScene(scene);
    primaryStage.show();
}

// Determines which button was pressed and sets the pane color
// accordingly.
//
public void processColorButton(ActionEvent event)
{
    if (event.getSource() == redButton)
        pane.setStyle("-fx-background-color: crimson");
    else
        pane.setStyle("-fx-background-color: deepskyblue");
}
```
The two buttons use the same method as their event handler. Whenever either button is pressed, the `processColorButton` method is called. It uses an `if` statement to check which button generated the event. If it's the Red button, the background color of the pane is set to red. Otherwise, it must have been the Blue button, so the background color is set to blue.

As in the previous example, the `ActionEvent` object that represents the event is passed into the event handler method. But in that example we ignored the event parameter. In this case, however, we use its `getSource` method which returns the control that generated the event.

Note that the variables representing the two buttons and the pane are declared as instance data at the class level so that they can be accessed in both the `start` method and the event handler method.

We could have created two separate event handler methods, one for the Red button and one for the Blue button. In that case, there would be no need to determine which button generated the event. Whether to have multiple event handlers or determine the event source when it occurs is a design decision that may depend on the situation.
6.2 Other GUI Controls

In addition to push buttons, there are a variety of other interactive components that can be used in a GUI, each with a particular role to play. Choosing the right control for the job is an important design decision. Let’s examine a few more GUI controls.

Text Fields

A text field allows the user to enter one line of text that can be used by the program as needed. The FahrenheitConverter program shown in Listing 6.4 presents a GUI that includes a text field into which the user can type a Fahrenheit temperature. When the user presses the Return (or Enter) key, the program displays the equivalent Celsius temperature.

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

/*****************************************
**** FahrenheitConverter.java Java Foundations
****
**** Demonstrates the use of a TextField and a GridPane.
*******************************************/
public class FahrenheitConverter extends Application
{
    // Launches the temperature converter application.
    public void start(Stage primaryStage)
    {
        Scene scene = new Scene(new FahrenheitPane(), 300, 150);

        primaryStage.setTitle("Fahrenheit Converter");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```
LISTING 6.4 continued

DISPLAY

Fahrenheit Converter

Fahrenheit: 75
Celsius: 23

In this example, the details of the user interface are set up in a separate class, shown in Listing 6.5. The FahrenheitPane class extends the GridPane class, which is a layout pane from the JavaFX API that displays nodes in a rectangular grid.

LISTING 6.5

import javafx.event.ActionEvent;
import javafx.geometry.HPos;
import javafx.geometry.Pos;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.text.Font;

//***************************************************************
// FahrenheitPane.java Java Foundations
// Demonstrates the use of a TextField and a GridPane.
//***************************************************************
public class FahrenheitPane extends GridPane
{
    private Label result;
    private TextField fahrenheit;

    // Sets up a GUI containing a labeled text field for converting
    // temperatures in Fahrenheit to Celsius.
    public FahrenheitPane()
    {
        Font font = new Font(18);
        Label inputLabel = new Label("Fahrenheit:");
        inputLabel.setFont(font);
        GridPane.setHalignment(inputLabel, HPos.RIGHT);
        Label outputLabel = new Label("Celsius:");
        outputLabel.setFont(font);
        GridPane.setHalignment(outputLabel, HPos.RIGHT);
        result = new Label("---");
        result.setFont(font);
        GridPane.setHalignment(result, HPos.CENTER);
        fahrenheit = new TextField();
        fahrenheit.setFont(font);
        fahrenheit.setPrefWidth(50);
        fahrenheit.setAlignment(Pos.CENTER);
        fahrenheit.setOnAction(this::processReturn);
        setAlignment(Pos.CENTER);
        setHgap(20);
        setVgap(10);
        setStyle("-fx-background-color: yellow");
        add(inputLabel, 0, 0);
        add(fahrenheit, 1, 0);
        add(outputLabel, 0, 1);
        add(result, 1, 1);
    }
The user interface is made up of three Label objects and one TextField object. The font size of each element is set using a Font object and calls to the setFont method of each node. Fonts are discussed in more detail in Appendix F.

At the end of the FahrenheitPane constructor, the four elements are added to the pane. (Through inheritance, the FahrenheitPane is a GridPane, and inherits the add method.) The parameters to the add method specify to which grid cell the node is added. The first value is the row and the second is the column. The rows and columns of a grid pane both start at 0.

The processReturn method is used to define the event handler that is triggered when the user presses return while the cursor is in the text field. It is associated with the text field with a call to its setOnAction method.

The processReturn method obtains the text from the text field using a call to the getText method, which returns a character string. The text is converted to an integer using the parseInt method of the Integer wrapper class. Then the method performs the calculation to determine the equivalent Celsius temperature and sets the text of the appropriate label with the result.

Check Boxes

A check box is a button that can be toggled on or off using the mouse, indicating that a particular condition is set or unset. For example, you might use a check box to indicate whether the user has acknowledged and accepted the Terms of Use for your program.
Although you might have a group of check boxes indicating a set of options, each check box operates independently. That is, each can be set to on or off and the status of one does not automatically influence the others. For example, you might use a series of check boxes to indicate which toppings should be included on a pizza. They could be checked or unchecked in any combination.

The program made up of the classes in Listings 6.6 and 6.7 displays two check boxes and a Text object. The check boxes determine whether the text is displayed in bold, italic, both, or neither. Any combination of bold and italic is valid.

The details of the GUI are specified in the StyleOptionsPane class in Listing 6.7. A check box is defined by the CheckBox class from the JavaFX API. When a check box is selected or deselected, it generates an action event. In this example, both check boxes are processed by the same event handler method.

```java
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.stage.Stage;

// StyleOptions.java Java Foundations
// Demonstrates the use of check boxes.
public class StyleOptions extends Application
{
    public void start(Stage primaryStage)
    {        
        StyleOptionsPane pane = new StyleOptionsPane();
        pane.setAlignment(Pos.CENTER);
        pane.setStyle("-fx-background-color: skyblue");

        Scene scene = new Scene(pane, 400, 150);

        primaryStage.setTitle("Style Options");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```
Examine how the `processCheckBoxAction` method handles a change in the state of either check box. Instead of bothering to determine which check box generated the event, or keep track of whether a box was selected or deselected, the event handler simply examines the current state of both check boxes and resets the font accordingly. Local variables are used to set the font weight and posture, which are initially assumed to be unselected. Then the `isSelected` method of each check box is called, which returns `true` if the check box is currently selected. Finally, the font of the text is set appropriately.

There are two types of layout panes used in this program. The `HBox` and `VBox` layout panes arrange their nodes in a single row (horizontally) or a single column (vertically), respectively.
LISTING 6.7

import javafx.event.ActionEvent;
import javafx.geometry.Pos;
import javafx.scene.control.CheckBox;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.text.Text;
import javafx.scene.text.Font;
import javafx.scene.text.FontPosture;
import javafx.scene.text.FontWeight;

//****************************************************************************
// StyleOptionsPane.java         Java Foundations
//
// Demonstrates the use of check boxes.
//****************************************************************************
public class StyleOptionsPane extends VBox {
    private Text phrase;
    private CheckBox boldCheckBox, italicCheckBox;

    // Sets up this pane with a Text object and check boxes that
    // determine the style of the text font.
    public StyleOptionsPane() {
        phrase = new Text("Say it with style!");
        phrase.setFont(new Font("Helvetica", 36));

        boldCheckBox = new CheckBox("Bold");
        boldCheckBox.setOnAction(this::processCheckBoxAction);
        italicCheckBox = new CheckBox("Italic");
        italicCheckBox.setOnAction(this::processCheckBoxAction);

        HBox options = new HBox(boldCheckBox, italicCheckBox);
        options.setAlignment(Pos.CENTER);
        options.setSpacing(20); // between the check boxes
        setSpacing(20); // between the text and the check boxes
        getChildren().addAll(phrase, options);
    }

    // Updates the font style of the displayed text.
    //
public void processCheckBoxAction(ActionEvent event) {
    FontWeight weight = FontWeight.NORMAL;
    FontPosture posture = FontPosture.REGULAR;

    if (boldCheckBox.isSelected())
        weight = FontWeight.BOLD;

    if (italicCheckBox.isSelected())
        posture = FontPosture.ITALIC;

    phrase.setFont(Font.font("Helvetica", weight, posture, 36));
}

The StyleOptionsPane class extends VBox, which is used to center the text above the two check boxes. A separate HBox is set up to put the check boxes side by side horizontally.

Since the nodes aren’t added using the VBox constructor, they are added after the fact. But you don’t add nodes to a pane directly. Instead, you call the getChildren method, which returns all nodes already in the pane (which is none in this case) and then call the addAll method to add the new nodes.

Radio Buttons

A radio button is used with other radio buttons to provide a set of mutually exclusive options. Unlike a check box, a radio button is not particularly useful by itself. It has meaning only when grouped with other radio buttons. Only one option in a group of radio buttons is valid. At any point in time, only one button of a radio button group is selected (on). When a radio button is pushed, the other button in the group that is currently on is automatically toggled off.

The term “radio buttons” comes from the way the pre-set station buttons worked on an old-fashioned car radio. At any point, one button was pushed in to specify the current station. When another was pushed to change the station, the current one automatically popped out.

The program made up of the classes shown in Listings 6.8 and 6.9 displays a group of radio buttons and a Text object. The radio buttons determine which phrase is displayed. Because only one phrase is displayed at a time, the use of radio buttons is appropriate.
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.stage.Stage;

// QuoteOptions.java Java Foundations
// Demonstrates the use of radio buttons.
// --------------------------------------------------------------
public class QuoteOptions extends Application
{
    // Creates and presents the program window.
    // --------------------------------------------------------------
    public void start(Stage primaryStage)
    {
        QuoteOptionsPane pane = new QuoteOptionsPane();
        pane.setAlignment(Pos.CENTER);
        pane.setStyle("-fx-background-color: lightgreen");

        Scene scene = new Scene(pane, 500, 150);

        primaryStage.setTitle("Quote Options");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}

DISPLAY

Quote Options

- Philosophy
- Carpentry
- Comedy

Measure twice. Cut once.
A radio button control is represented by the JavaFX RadioButton class. A ToggleGroup object is used to create a set of mutually exclusive radio buttons. To add a button to a group, you pass the group object to the radio button’s setToggleGroup method.

One event handler is used to process all three radio buttons. A radio button produces an action event when it is selected. The processRadioButtonAction method uses a nested if statement to determine which button is currently selected and sets the text accordingly.

```java
import javafx.event.ActionEvent;
import javafx.geometry.Pos;
import javafx.scene.control.RadioButton;
import javafx.scene.control.ToggleGroup;
import javafx.scene.layout.HBox;
import javafx.scene.layout.StackPane;
import javafx.scene.layout.VBox;
import javafx.scene.text.Text;
import javafx.scene.text.Font;
```
LISTING 6.9 continued

// QuoteOptionsPane.java Java Foundations
//
// Demonstrates the use of radio buttons.
//************************************************************************************
public class QuoteOptionsPane extends HBox
{
    private Text quote;
    private String philosophyQuote, carpentryQuote, comedyQuote;
    private RadioButton philosophyButton, carpentryButton, comedyButton;

    // Sets up this pane with a Text object and radio buttons that
    // determine which phrase is displayed.
    //************************************************************************************
    public QuoteOptionsPane()
    {
        philosophyQuote = "I think, therefore I am.";
        carpentryQuote = "Measure twice. Cut once.";
        comedyQuote = "Take my wife, please.";

        quote = new Text(philosophyQuote);
        quote.setFont(new Font("Helvetica", 24));

        StackPane quotePane = new StackPane(quote);
        quotePane.setPrefSize(300, 100);

        ToggleGroup group = new ToggleGroup();

        philosophyButton = new RadioButton("Philosophy");
        philosophyButton.setSelected(true);
        philosophyButton.setToggleGroup(group);
        philosophyButton.setOnAction(this::processRadioButtonAction);

        carpentryButton = new RadioButton("Carpentry");
        carpentryButton.setToggleGroup(group);
        carpentryButton.setOnAction(this::processRadioButtonAction);

        comedyButton = new RadioButton("Comedy");
        comedyButton.setToggleGroup(group);
        comedyButton.setOnAction(this::processRadioButtonAction);
    }
}
Like the previous example, this program uses an HBox and a VBox to organize the GUI elements. This time, however, the VBox is used to organize the buttons and is put into an HBox to lay it out next to the text.

Color and Date Pickers

The JavaFX API includes the ColorPicker class which represents a control that lets the user select a color. The control appears as a single field displaying the current color and its corresponding RGB value in hexadecimal.

When clicked, a color picker displays a drop-down palette of colors from which to choose. If none of the palette colors will do, you can also pick a custom color from a more complicated selection pane, or specify the color using RGB values or another color representation model.
Similarly, a DatePicker object allows the user to select a calendar date. Like the color picker, a date picker appears as a single field. It displays the currently selected date in m/d/y format by default. When clicked, the date picker displays a drop-down calendar that allows the user to change months and years, and ultimately click on a specific date.

The program in Listing 6.10 demonstrates a date picker and a color picker. When a date is selected, a message below the picker fields displays the corresponding day of the week. When a color is selected, the message fill color changes accordingly.

```
import java.time.LocalDate;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.ColorPicker;
import javafx.scene.control.DatePicker;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.FontPosture;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.stage.Stage;

//*****************************************************************************
// PickerDemo.java    Java Foundations
//
// Demonstrates the use of color picker and date picker controls.
//*****************************************************************************
public class PickerDemo extends Application
{
    private Text message;
    private DatePicker datePicker;
    private ColorPicker colorPicker;

    //--------------
    // Allows the user to select a date and a color. A Text object
    // displays the day of the week in the color specified.
    //--------------
    public void start(Stage primaryStage)
    {
        datePicker = new DatePicker(LocalDate.now());
        datePicker.setOnAction(this::processDateChoice);
    }
```
Listing 6.10 continued

colorPicker = new ColorPicker(Color.BLACK);
colorPicker.setOnAction(this::processColorChoice);

message = new Text("HAPPY " + LocalDate.now().getDayOfWeek());
message.setFont(Font.font("Helvetica", FontWeight.BOLD,
    FontPosture.REGULAR, 24));

HBox pickers = new HBox(datePicker, colorPicker);
pickers.setSpacing(15);
pickers.setAlignment(Pos.CENTER);

VBox root = new VBox();
root.setStyle("-fx-background-color: white");
root.setSpacing(20);
root.setAlignment(Pos.CENTER);
root.getChildren().addAll(pickers, message);

Scene scene = new Scene(root, 400, 150);

primaryStage.setTitle("Picker Demo");
primaryStage.setScene(scene);
primaryStage.show();

//gets the value of the date from the date picker and updates the
//message with the corresponding day of the week.

public void processDateChoice(ActionEvent event)
{
    LocalDate date = datePicker.getValue();
    message.setText("HAPPY " + date.getDayOfWeek());
}

//gets the color specified in the color picker and sets the
//color of the displayed message.

public void processColorChoice(ActionEvent event)
{
    message.setFill(colorPicker.getValue());
}
LISTING 6.10 continued

DISPLAY

Picker Demo

1/24/2018 [ ] [ ] #cc3333

HAPPY WEDNESDAY

This program makes use of the java.time.LocalDate class, which represents a calendar date. The LocalDate class has a static method called now that returns the current date, and an object method called getDayOfWeek that returns the day of the week corresponding to the date.

If no date is specified when a DatePicker object is instantiated, the field will initially be blank. In this program, however, the current date is passed to the DatePicker constructor to set the initial date. If no color is specified when a ColorPicker is created, the default color is white. This example passes the color black to the ColorPicker constructor to match the initial color of the message.

In this program, two separate action event handler methods are used to process a selection made using the date picker and color picker. Both use the getValue method of the appropriate picker to get the current value selected by the user. The getValue method of a color picker returns a Color object, while the getValue method of a date picker returns a LocalDate object.

6.3 Mouse and Key Events

In addition to events that are generated when the user interacts with a control, there are events that are fired when the user interacts with the computer’s mouse and keyboard. We can design a program to capture and respond to these events as well.
### Mouse Events

Let's examine the events that are generated when using a mouse, described in Figure 6.1. The coordinates of the mouse are captured in the event object when any of these events occur.

When you click the mouse button while the mouse pointer is over a JavaFX node, three events occur: one when the mouse button is pushed down (mouse pressed), and two when it is let up (mouse released and mouse clicked).

A node will generate a mouse entered event when the mouse pointer passes into its graphical space. Likewise, it generates a mouse exited event when the mouse pointer is moved off of the node.

A stream of mouse moved events occur while the mouse is in motion. If the mouse button is pressed down while the mouse is being moved, mouse dragged events are generated. These events are generated very quickly while the mouse is in motion, allowing a program to track and respond to the ongoing movement of the mouse.

There is a corresponding convenience method for setting the handler for each of the mouse events, such as setOnMousePressed, setOnMouseReleased, etc.

The program shown in Listing 6.11 responds to one mouse event. When the mouse button is clicked anywhere on the scene, a line is displayed from the origin point (0, 0) in the upper left corner to the location of the mouse pointer. Also, the distance between those two points is calculated and displayed.

```java
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color;
```
import javafx.scene.shape.Line;
import javafx.scene.text.Text;
import javafx.stage.Stage;

/**
 * ClickDistance.java
 * Demonstrates the handling of a mouse click event.
 */
public class ClickDistance extends Application {
    private Line line;
    private Text distanceText;

    public void start(Stage primaryStage)
    {
        line = new Line(0, 0, 0, 0);
        distanceText = new Text(150, 30, "Distance: --");
        Group root = new Group(distanceText, line);
        Scene scene = new Scene(root, 400, 300, Color.LIGHTYELLOW);
        scene.setOnMouseClicked(this::processMouseClicked);
        primaryStage.setTitle("Click Distance");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public void processMouseClicked(MouseEvent event)
    {
        double clickX = event.getX();
        double clickY = event.getY();
        line.setEndX(clickX);
        line.setEndY(clickY);
        double distance = Math.sqrt(clickX * clickX + clickY * clickY);
    }
}
Listing 6.11 continued

```java
String distanceStr = String.format("%.2f", distance);
distanceText.setText("Distance: " + distanceStr);
```

Display

Click Distance

Distance: 318.28

Click Distance

Distance: 289.36

See a full-color version of these figures at the end of the text.
The `processMouseClicked` method is set as the event handler, which is passed to the `MouseEvent` object that represents the event. Calling the `getX` and `getY` methods of the event returns the coordinates where the mouse was clicked. Using those values, the end point of the line is reset and the distance to the origin point is calculated and displayed.

Now let’s look at an example that responds to two mouse-oriented events. The `RubberLines` program shown in Listing 6.12 allows the user to draw a line between two points by clicking the mouse button to establish one end point and dragging the mouse to the other end point. The line is constantly redrawn as the mouse is being dragged, giving the illusion that the user is stretching the line into existence. This effect is called rubberbanding.

Two event handlers are established in this program: one to handle the mouse being pressed and the other to handle the mouse being dragged. When the mouse button is pressed, a new `Line` object is created and added to the root node of the scene. The line is initially only one pixel long, corresponding to the location of the mouse.

As the mouse is being dragged, multiple mouse drag events are generated. Each time, the end point of the line is updated to the current position of the mouse pointer. These changes happen so quickly that it appears as if one line is being stretched. When the user releases the mouse button, the drag effects stop, and the line’s position is now fixed. The user can then draw another line if desired.

**Listing 6.12**

```java
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.input.MouseEvent;
import javafx.scene.paint.Color;
import javafx.scene.shape.Line;
import javafx.stage.Stage;

/**
 * Demonstrates the handling of mouse press and mouse drag events.
 */
public class RubberLines extends Application
{
    private Line currentLine;
    private Group root;
```
Listing 6.12 continued

// Displays an initially empty scene, waiting for the user to
// draw lines with the mouse.

public void start(Stage primaryStage)
{
    root = new Group();

    Scene scene = new Scene(root, 500, 300, Color.BLACK);
    scene.setOnMousePressed(this::processMousePress);
    scene.setOnMouseDragged(this::processMouseDrag);

    primaryStage.setTitle("Rubber Lines");
    primaryStage.setScene(scene);
    primaryStage.show();
}

// Adds a new line to the scene when the mouse button is pressed.

public void processMousePress(MouseEvent event)
{
    currentLine = new Line(event.getX(), event.getY(), event.getX(),
                            event.getY());
    currentLine.setStroke(Color.CYAN);
    currentLine.setStrokeWidth(3);
    root.getChildren().add(currentLine);
}

// Updates the end point of the current line as the mouse is
// dragged, creating the rubber band effect.

public void processMouseDrag(MouseEvent event)
{
    currentLine.setEndX(event.getX());
    currentLine.setEndY(event.getY());
}
Key Events

A key event is generated when a keyboard key is pressed. Key events allow a program to respond immediately to the user when he or she is typing or pressing other keys such as the arrow keys. If key events are being handled, there is no need to wait for the user to press the Enter key as there is in other keyboard input situations.

There are three types of key events, as listed in Figure 6.2. The methods `setOnKeyPressed`, `setOnKeyReleased`, and `setOnKeyTyped` can be used to set the event handlers for these methods.

<table>
<thead>
<tr>
<th>Key Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key pressed</td>
<td>A keyboard key is pressed down.</td>
</tr>
<tr>
<td>key released</td>
<td>A keyboard key is released.</td>
</tr>
<tr>
<td>key typed</td>
<td>A keyboard key that generates a character typed (pressed and released).</td>
</tr>
</tbody>
</table>

**FIGURE 6.2** JavaFX key events
A key typed event is slightly different than the other two. A key typed event is not a function of the underlying platform, while the other two are. A key typed event is only generated when a key representing a Unicode character is entered.

The program shown in Listing 6.13 responds to key pressed events. It displays an image of an alien that can be moved around the screen using the arrow keys on the keyboard. When the up arrow key is pressed, for instance, the alien immediately moves upward on the screen.

### Listing 6.13

```java
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.input.KeyEvent;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

//******************************************************************************
// AlienDirection.java  Java Foundations
//
// Demonstrates the handling of keyboard events.
//******************************************************************************
public class AlienDirection extends Application {
    public final static int JUMP = 10;

    private ImageView imageView;

    //******************************************************************************
    // Displays an image that can be moved using the arrow keys.
    //******************************************************************************
    public void start(Stage primaryStage) {
        Image alien = new Image("alien.png");

        imageView = new ImageView(alien);
        imageView.setX(20);
        imageView.setY(20);

        Group root = new Group(imageView);

        Scene scene = new Scene(root, 400, 200, Color.BLACK);
        scene.setOnKeyPressed(this::processKeyPress);
    }
}
```
primaryStage.setTitle("Alien Direction");
primaryStage.setScene(scene);
primaryStage.show();
}

// Modifies the position of the image view when an arrow key is pressed.
public void processKeyPress(KeyEvent event)
{
  switch (event.getCode())
  {
    case UP:
      imageView.setY(imageView.getY() - JUMP);
      break;
    case DOWN:
      imageView.setY(imageView.getY() + JUMP);
      break;
    case RIGHT:
      imageView.setX(imageView.getX() + JUMP);
      break;
    case LEFT:
      imageView.setX(imageView.getX() - JUMP);
      break;
    default:
      break; // do nothing if it's not an arrow key
  }
}
The **start** method of this example loads the alien image and sets up an **ImageView** object to display it. The initial position of the image view is explicitly set.

Key events are processed by the node that has the **keyboard focus**. In this example, the events are processed by the scene itself. So, the **setOnKeyPressed** method of the scene is called to set the handler for keys that are pressed.

When a keyboard key is pressed, the event handler method is called and passed a **KeyEvent** object. The **getCode** method of the event object returns a code that represents the key that was pressed. More specifically, it returns a **KeyCode** object, which is an enumerated type representing the various keys.

A **switch** statement is used to handle each of the four arrow keys to which we want the program to respond. For instance, when the right arrow key is pressed, a specific number of pixels (represented by the constant **jump**) are added to the x position of the image view. If the user presses any key other than the arrow keys, it is ignored.

It should be noted that if a key typed event is generated, its **getCode** method will always return **KeyCode.UNDEFINED**. In that case, the **getCharacter** method of the event can be used to get the character.

### 6.4 Dialog Boxes

A **dialog box** is a window that pops up on top of any currently active window so that the user can interact with it. A dialog box can serve a variety of purposes, such as conveying some information, confirming an
action, or allowing the user to enter some information. Usually, a dialog box has a solitary purpose, and the user's interaction with it is brief.

Support for dialog boxes in GUIs comes from a few classes in the JavaFX API. The `Alert` class provides support for several basic dialog boxes that can be easily created and displayed. There are several types of alerts, specified by the `Alert.AlertType` enumerated type:

- `AlertType.INFORMATION`—conveys information
- `AlertType.CONFIRMATION`—allows the user to confirm an action
- `AlertType.WARNING`—conveys a warning
- `AlertType.ERROR`—indicates something has gone wrong

The differences in the alert types include the title, header, buttons, and graphic used. All of these elements can be tailored if desired.

Two other classes that define dialog boxes in JavaFX are the `TextInputDialog` class and the `ChoiceDialog` class. They allow the user to enter input using a text field and a drop-down choice box, respectively.

The program in Listing 6.14 uses dialog boxes exclusively to interact with the user. It first presents the user with a `TextInputDialog` that prompts the user to enter an integer. After the user presses the OK button, a second dialog box appears, informing the user whether the number entered was even or odd. After the user dismisses that box, a third dialog box appears to determine whether the user would like to test another number.

**Listing 6.14**

```java
import java.util.Optional;
import javafx.application.Application;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.ButtonType;
import javafx.scene.control.TextInputDialog;
import javafx.stage.Stage;

// EvenOdd.java  Java Foundations
//
// Demonstrates the use of information and confirmation alerts, as well
// as text input dialog boxes.

public class EvenOdd extends Application {
```
public void start(Stage primaryStage) throws Exception
{
    boolean doAnother = true;

    while (doAnother)
    {
        TextInputDialog inputDialog = new TextInputDialog();
        inputDialog.setHeaderText(null);
        inputDialog.setTitle(null);
        inputDialog.setContentText("Enter an integer:");
        Optional<String> numString = inputDialog.showAndWait();

        if (numString.isPresent())
        {
            int num = Integer.parseInt(numString.get());

            String result = "That number is 
                            (num % 2 == 0) ? "even."
                            : "odd.");

            Alert answerDialog = new Alert(AlertType.INFORMATION);
            answerDialog.setHeaderText(null);
            answerDialog.setContentText(result);
            answerDialog.showAndWait();

            Alert confirmDialog = new Alert(AlertType.CONFIRMATION);
            confirmDialog.setHeaderText(null);
            confirmDialog.setContentText("Do another?");
            Optional<ButtonType> another = confirmDialog.showAndWait();

            if (another.get() != ButtonType.OK)
                doAnother = false;
            else
                doAnother = false;
        }
    }
}
LISTING 6.14 continued

DISPLAY

? Enter an integer: 38

Message

! That number is even.

Confirmation

? Do another?

See a full-color version of these figures at the end of the text.

The headers for all of the dialog boxes shown in Listing 6.14 are set to null to keep them small and simple. The title (the text in the title bar) on the first one is also set to null, but on the others the default value is used.

After the first dialog box is set up, its showAndWait method is called, which causes the program to block at that point, waiting for the user to enter a value and press a button. When the user presses a button, this method returns an Optional<String> object that represents the text entered in the text field.

If the user entered something into the text field, it is converted to an integer, then the program determines if it is even or odd using a conditional statement. The appropriate text is used to set up an information Alert, then it is displayed using its showAndWait method. This time, its return value is ignored.
The third dialog box is set up and displayed, again using the `showAndWait` method. If the user presses the OK button, the loop executes another time to process another number. If the user presses the Cancel button, or simply closes the dialog window, the `doAnother` variable is set to false, and the loop terminates.

The `Optional` class is simply a container for a particular type of value. The value returned by the `showAndWait` method of a `TextInputDialog` is an `Optional<String>`. For a `confirmation` alert, the value returned by `showAndWait` is an `Optional<ButtonType>`. By setting it up this way, the `showAndWait` method of any dialog box returns the same type of object (an `Optional` object), but it contains whatever type of value is appropriate for that interaction.

**File Choosers**

A specialized dialog box called a file chooser allows the user to select a file from a hard drive or other storage medium. You probably have run many programs that allow you to specify a file using a similar dialog box.

The program in Listing 6.15 displays a file chooser dialog box to the user. When a file is selected, the contents of the file are read and displayed in a window containing a text area.

---

### Listing 6.15

```java
import java.io.File;
import java.io.IOException;
import java.util.Scanner;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.controlTextArea;
import javafx.scene.text.TextFont;
import javafx.stage.FileChooser;
import javafx.stage.Stage;

.GetInstance from Java Foundations
Demonstrates the use of a file chooser dialog box and a text area.

public class DisplayFile extends Application
{
    //-----------------------------------------------
    // Presents a file chooser dialog, reads the selected file and
    // loads it into a text area.
    //-----------------------------------------------
```
Listing 6.15 continued

```java
public void start(Stage primaryStage) throws IOException {
   FileChooser chooser = new FileChooser();
    File selectedFile = chooser.showOpenDialog(primaryStage);
    TextArea content = new TextArea();
    content.setFont(new Font("Courier", 12));
    content.setEditable(false);
    if (selectedFile != null) {
        content.setText("No file chosen.");
    } else {
        Scanner scan = new Scanner(selectedFile);
        String info = "";
        while (scan.hasNext()) {
            info += scan.nextLine() + "\n";
            content.setText(info);
        }
    }
    Scene scene = new Scene(content, 500, 500);
    primaryStage.setTitle("Display File");
    primaryStage.setScene(scene);
    primaryStage.show();
}
```

Display

- Name
  - Bonnie Jean - Burns.txt
  - Hope - Addison.txt
  - Sister Maude - Rossetti.txt
  - The Oak - Tennyson.txt
  - The Road Not Taken - Frost.txt
  - The Things We Dare Not Tell - Lawson.txt
  - This Living Hand - Keats.txt
  - Winter Roses - Whittier.txt

See a full-color version of this figure at the end of the text.
LISTING 6.15

The Road Not Taken - Frost.txt

The Road Not Taken
by Robert Frost

Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;

Then took the other, as just as fair,
And having perhaps the better claim,
Because it was grassy and wanted wear;
Though as for that the passing there
Had worn them really about the same,

And both that morning equally lay
In leaves no step had trodden black.
Oh, I kept the first for another day!
Yet knowing how way leads on to way,
I doubted if I should ever come back.

I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood, and I-
I took the one less traveled by,
And that has made all the difference.

The showOpenDialog method of a FileChooser object presents a dialog box that allows the user to specify a file to be opened. Similarly, the showOpenMultipleDialog method presents a dialog box that lets the user specify multiple files at once, and the showSaveDialog method presents a dialog box that allows the user to specify a file in which to save information.
All of these methods accept a parameter that represents the “owner” window. All input to the owner is blocked while the file dialog is being shown.

The look and feel of a file chooser dialog box is based on the underlying platform on which the program is running—it is not determined by JavaFX.

In this example, after the file is specified, it is read using a Scanner object and its contents are loaded, line by line, into a TextArea object. A text area is a control that presents multiple lines of text (unlike a TextField, which only shows one line). Once the text area content is set, it is displayed in a scene on the primary stage.

A text area is editable by default, allowing the user to change the text. Note that such edits only change the displayed text, not the underlying file. To save the changes, the text must be written back to the file, or saved in another file. A save dialog of the FileChooser class may be helpful in this case.

It should be noted that there is another JavaFX class called DirectoryChooser that is similar to FileChooser but is designed for selecting directories (folders).

6.5 JavaFX Properties

A JavaFX property is an object that holds a value, similar to a wrapper class. But a property is observable, which means the property value can be monitored and changed as needed. Many JavaFX classes store properties rather than regular instance data. For instance, instead of storing an int primitive or even an Integer object, a JavaFX class might store an IntegerProperty object.

A key benefit to using properties is the concept of property binding. A property can be bound to another property, so that when the value of one property changes, the other is automatically updated. For example, the radius of the Circle class is represented by a DoubleProperty object, which could be bound to the property that represents the width of a Scene, so that the circle size changes automatically as the window is resized.

The program in Listing 6.16 displays a small circle in the center of the scene, as well as two Text objects in the upper left corner that display the height and width of the scene. All of these elements are bound to the width and height of the scene in various ways such that, as the window (and therefore the scene) is resized, the position of the circle and the displayed text change automatically.
import javafx.application.Application;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.text.Text;
import javafx.stage.Stage;

// PropertyBindingDemo.java     Java Foundations

// Demonstrates the ability to bind one property to another.

public class PropertyBindingDemo extends Application{

    public void start(Stage primaryStage){
        Group root = new Group();
        Scene scene = new Scene(root, 300, 200, Color.SKYBLUE);

        Circle center = new Circle(6);
        center.centerXProperty().bind(scene.widthProperty().divide(2));
        center.centerYProperty().bind(scene.heightProperty().divide(2));

        StringProperty width = new SimpleStringProperty("Width: ");
        StringProperty height = new SimpleStringProperty("Height: ");

        Text widthText = new Text(20, 30, ")
        widthText.textProperty().bind(width.concat(scene.widthProperty()));

        Text heightText = new Text(20, 60, ")
        heightText.textProperty().bind(height.concat(scene.heightProperty()));

        root.getChildren().addAll(center, widthText, heightText);
    }
}
LISTING 6.16  continued

```java
primaryStage.setTitle("Property Binding Demo");
primaryStage.setScene(scene);
primaryStage.show();
```
The properties that represent the x and y coordinates of the circle’s center are bound to the properties that represent the width and height of the scene, respectively. The x coordinate is always kept at one-half the value of the width property, and the y coordinate is kept at one-half of the height property. So the circle stays in the center of the window as the window is resized.

The `centerXProperty` method returns the `DoubleProperty` object that represents the x coordinate of the circle. It is bound, using a call to the `bind` method of the property, to the property returned by the `widthProperty` method of the scene. That value is divided into half by a call to the `divide` method (properties are objects, so you can use regular arithmetic operators on them). A similar relationship is set up for the y coordinate and the height.

The text displayed by a `Text` object is stored as a `StringProperty` object. In this program, two additional `StringProperty` objects are created to bind them to (a property can only be bound to another property).

The text property displaying the width is bound to a string property containing "Width: " concatenated to the width value of the scene. A similar relationship is set up for the height text.

Note that no explicit event handlers were set up for this program. Property binding is taking care of all dynamic updates to the elements in the scene. However, it’s important to recognize that a property binding cannot always be used in place of an event handler. Binding is used to keep data in sync, whereas an event handler is code executed when an event occurs to accomplish any desired effect. An event handler is, therefore, more versatile. Since we were only keeping data in sync in this program, property bindings were sufficient.

We have stated that a property is observables. To be more precise, a property implements the `ObservableValue` interface (or, more likely, one of its descendants). The `bind` method creates a `Binding` object to keep a particular value in sync with one or more sources. Methods such as `divide` also create bindings.

**Change Listeners**

A property can have a `change listener`, which is similar to an event handler in that it is set up to run whatever specific code you’d like. You would use a change listener if you wanted to respond to a property value changing and needed to do something other than keep two data values in sync.

Properties have an `addListener` method that can be used to set up a change listener for that property. You can specify the listener method as you would with an event handler convenience method:
myProperty.addListener(this::processChange);

A change listener method receives three parameters: the ObservableValue object (the property) whose value changed, the old value, and the new value. The types of the old and new values depend on the type of the value that the property holds. For example, here's a listener method that handles the changes to a StringProperty object:

```java
public void processChange(ObservableValue<String> val,
                        String oldValue, String newValue)
{
    //whatever
}
```

As with event handling methods, the method name can be anything desired. Similarly, here is a change listener method that handles changes to an IntegerProperty object:

```java
public void processChange(ObservableValue<Integer> val,
                        Integer oldValue, Integer newValue)
{
    //whatever
}
```

Listing 6.17 shows a program that is functionally equivalent to the PropertyBindingDemo program, but uses a change listener instead of property binding.

```
import javafx.application.Application;
import javafx.beans.value.ObservableValue;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.text.Text;
import javafx.stage.Stage;

//******************************************************************************************
//@ ChangeListenerDemo.java Java Foundations
//@
// Demonstrates the ability to respond to property changes using
// change listeners. Functionally equivalent to PropertyBindingDemo.
//******************************************************************************************
```
Listing 6.17 continued

```java
public class ChangeListenerDemo extends Application
{
    private Scene scene;
    private Circle center;
    private Text widthText, heightText;

    /**********************************************************************
    // Displays the width and height of the scene, as well as a circle
    // in the center of the scene. The scene is updated using a change
    // listener as the window is resized.
    /**********************************************************************
    public void start(Stage primaryStage)
    {
        Group root = new Group();

        scene = new Scene(root, 300, 200, Color.SKYBLUE);
        scene.widthProperty().addListener(this::processResize);
        scene.heightProperty().addListener(this::processResize);

        center = new Circle(6);
        center.setCenterX(scene.getWidth() / 2);
        center.setCenterY(scene.getHeight() / 2);

        widthText = new Text(20, 30, "Width: " + scene.getWidth());
        heightText = new Text(20, 60, "Height: " + scene.getHeight());

        root.getChildren().addAll(center, widthText, heightText);

        primaryStage.setTitle("ChangeListener Demo");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    /**********************************************************************
    // Updates the position of the circle and the displayed width and
    // height when the window is resized.
    /**********************************************************************
    public void processResize(ObservableValue<? extends Number> property,
                                  Object oldValue, Object newValue)
    {
        center.setCenterX(scene.getWidth() / 2);
        center.setCenterY(scene.getHeight() / 2);
        widthText.setText("Width: " + scene.getWidth());
        heightText.setText("Height: " + scene.getHeight());
    }
}
```
In this version of the program, the scene, circle, and text objects are declared at the class level so that they can be accessed by the listener method. The same listener is used for changes in both the width and height of the scene.

The new property value could have been obtained from the parameters to the listener method, but then we would have had to have a separate listener for the width and height (so we'd know which property to set) and there would have been a lot of casting involved. Instead, the method parameters are ignored and the new values are taken directly from the scene object.

**Sliders**

A *slider* is a GUI control that allows the user to specify a numeric value within a bounded range. The slider value is a property which can be bound to some other property.

A slider is displayed as a track along which the slider knob can be dragged. A slider can be presented either vertically or horizontally and can have optional tick marks and labels indicating the range of values.

The program shown in Listing 6.18 displays an ellipse and allows the user to control the shape of that ellipse using two sliders. The horizontal slider determines the value of the radius along the x axis of the ellipse and the vertical slider determines the value of the radius along the y axis.

```java
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Orientation;
import javafx.scene.Scene;
import javafx.scene.control.Slider;
import javafx.scene.layout.BorderPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Ellipse;
import javafx.stage.Stage;

//***************************************************************************
// EllipseSliders.java    Java Foundations
//
// Demonstrates the use of slider controls and property binding.
//***************************************************************************
```
public class EllipseSliders extends Application
{
    private Ellipse ellipse;
    private Slider xSlider, ySlider;

    // Displays an ellipse with sliders that control the width and
    // height of the ellipse.
    public void start(Stage primaryStage)
    {
        ellipse = new Ellipse(250, 150, 150, 75);
        ellipse.setFill(Color.SALMON);

        xSlider = new Slider(0, 200, 150);
        xSlider.setShowTickMarks(true);
        xSlider.setPadding(new Insets(0, 20, 20, 80));

        ellipse.radiusXProperty().bind(xSlider.valueProperty());

        ySlider = new Slider(0, 100, 75);
        ySlider.setOrientation(Orientation.VERTICAL);
        ySlider.setShowTickMarks(true);
        ySlider.setPadding(new Insets(20, 0, 0, 30));

        ellipse.radiusYProperty().bind(ySlider.valueProperty());

        BorderPane pane = new BorderPane();
        pane.setLeft(ySlider);
        pane.setBottom(xSlider);
        pane.setCenter(ellipse);
        pane.setStyle("-fx-background-color: gray");

        Scene scene = new Scene(pane, 500, 300);

        primaryStage.setTitle("Ellipse Sliders");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
See a full-color version of these figures at the end of the text.
A slider is presented horizontally unless you explicitly set it to vertical using its setOrientation method. The setShowTickMarks method accepts a boolean value and is used to set whether tick marks should be displayed next to the slider bar. The setPadding method determines the spacing around the slider when it is displayed. The Slider class has additional methods that can be used to tailor the look and behavior of a slider.

The changes made to the ellipse are done exclusively through property bindings. There are no explicit event handlers written for this program. The property representing the x radius of the ellipse is bound (using the bind method) to the value of the horizontal slider. Likewise, the property representing the y radius of the ellipse is bound to the value of the vertical slider.

**Spinners**

A spinner is a JavaFX control that allows the user to select a value from a list of predefined values arranged in a sequence. The current value is shown in a text field, and the user steps through the options using a pair of arrow buttons displayed next to, on either side of, or above and below the text field.

The options of a spinner are never displayed in a list, like they are with a drop-down choice box or combo box. In a spinner, only one value, the currently selected value, is displayed at a time. A spinner may be preferred so that the options won’t obscure other elements in the GUI.

The program in Listing 6.19 presents two spinners, one that provides numeric options 1 through 10, and another that allows the user to select from a sequence of strings. The current values of the spinners are reflected in a Text object shown below them.

**Listing 6.19**

```java
import javafx.application.Application;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.geometry.Pos;
```
import javafx.scene.Scene;
import javafx.scene.control.Spinner;
import javafx.scene.control.SpinnerValueFactory.IntegerSpinnerValueFactory;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;

//******************************************************************************
// SpinnerDemo.java         Java Foundations
//
// Demonstrates the use of spinner controls and property binding.
//******************************************************************************
public class SpinnerDemo extends Application
{
    private Spinner<Integer> intSpinner;
    private Spinner<String> stringSpinner;
    private Text text;

    //------------------------------------------------------------------------------
    // Presents an integer spinner and a string spinner, updating some
    // text when either value changes.
    //------------------------------------------------------------------------------
    public void start(Stage primaryStage)
    {
        IntegerSpinnerValueFactory svf =
            new IntegerSpinnerValueFactory(1, 10, 5);
        intSpinner = new Spinner<Integer>(svf);

        ObservableList<String> list = FXCollections.observableArrayList();
        list.addAll("Grumpy", "Happy", "Sneaky", "Sleepy", "Dopey",
            " Bashful", "Doc");
        stringSpinner = new Spinner<String>(list);
        stringSpinner.getStyleClass().add(Spinner.STYLE_CLASS_SPLIT_ARROWS_VERTICAL);

        StringProperty textString = new SimpleStringProperty("");

        text = new Text();
        text.setFont(new Font("Helvetica", 24));
        text.textProperty().bind(textString.concat(
            intSpinner.valueProperty().concat(" and ").concat(
            stringSpinner.valueProperty())));
    }
}
VBox pane = new VBox(intSpinner, stringSpinner, text);
pane.setStyle("-fx-background-color: skyblue");
pane.setAlignment(Pos.CENTER);
pane.setSpacing(25);

Scene scene = new Scene(pane, 300, 250);

primaryStage.setTitle("Spinner Demo");
primaryStage.setScene(scene);
primaryStage.show();

See a full-color version of this figure at the end of the text.
The set of spinner options is defined by a `SpinnerValueFactory`. In this example, the integer spinner is made by creating a `IntegerSpinnerValueFactory` with a minimum value of 1, a maximum value of 10, and an initial value of 5. The value factory is passed to the `Spinner` constructor.

For the string spinner, an `ObservableList` object serves as the value factory. It is filled with the strings representing the options, which is then used to create the spinner itself.

By default, the arrows of a spinner appear on the right side of the text field, pointing up and down (vertical). This is how the arrows appear on the integer spinner in the example. For the string spinner, they are explicitly set to appear above and below the text field by adding a particular spinner style class to the spinner. The `Spinner` class contains several constants that represent different arrow positions.

The `Text` object displayed at the bottom of the window is updated automatically whenever either spinner is updated. A property binding is set up to keep the displayed text in sync with the spinner values.
6.6 Tool Tips and Disabling Controls

Paying attention to details when designing a GUI can often be the difference between a good user experience and a bad one. This section describes two such details: tool tips and the ability to disable controls.

A tool tip is a short line of text that appears when the mouse pointer is paused momentarily over a control or other GUI element. Tool tips are usually used to provide a hint to the user about the control, such as the purpose of a button. They are especially helpful with buttons that display icons instead of text.

A tool tip is represented by the ToolTip class, and can be applied to any node in a scene graph. Tool tips are most often applied to controls, which have a convenience method called `setToolTip` for setting them up:

```java
myButton.setTooltip(new ToolTip("Update the total cost"));
```

Another helpful practice when designing a GUI is to disable a control if it should not be used or currently has no effect. For example, you might disable a slider controlling the volume of the background music until the user checks the check box that indicates background music should be played.

A disabled control appears “greyed out” and doesn’t respond to any user’s attempt to interact with it. Disabled components not only convey to the user which actions are appropriate and which aren’t, but they may also prevent erroneous situations from occurring.

Controls are enabled by default. To disable a control, call its `setDisable` method, passing in the boolean value `true`:

```java
myButton.setDisable(true);
```

To re-enable the control, call `setDisable` again, passing `false`.

The program in Listing 6.20 uses both tool tips and disabled controls. The scene displays the image of a light bulb and two buttons. The buttons control whether the light bulb is “on” or “off.”

**Listing 6.20**

```java
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.geometry.Pos;
import javafx.geometry.Rectangle2D;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.ToolTip;
```
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

// LightBulb.java Java Foundations

// Demonstrates the use of tool tips and disabled controls.

public class LightBulb extends Application
{
    private Button onButton, offButton;
    private ImageView bulbView;

    public void start(Stage primaryStage)
    {
        Image img = new Image("lightBulbs.png");
        bulbView = new ImageView(img);
        bulbView.setViewport(new Rectangle2D(0, 0, 125, 200)); // off

        onButton = new Button("On");
onButton.setPrefWidth(70);
onButton.setTooltip(new Tooltip("Turn me on!"));
onButton.setOnAction(this::processButtonPress);

        offButton = new Button("Off");
offButton.setPrefWidth(70);
offButton.setTooltip(new Tooltip("Turn me off!"));
offButton.setDisable(true);
offButton.setOnAction(this::processButtonPress);

        HBox controlsBox = new HBox(10, onButton, offButton);
        VBox controls = new VBox(10, bulbView, controlsBox);
        primaryStage.setScene(new Scene(controls, 300, 300));
        primaryStage.show();
    }
}

HBox buttons = new HBox(onButton, offButton);
buttons.setAlignment(Pos.CENTER);
buttons.setSpacing(30);

VBox root = new VBox(bulbView, buttons);
root.setAlignment(Pos.CENTER);
root.setStyle("-fx-background-color: black");
root.setSpacing(20);

Scene scene = new Scene(root, 250, 300);

primaryStage.setTitle("Light Bulb");
primaryStage.setScene(scene);
primaryStage.show();

// Determines which button was pressed and sets the image viewport
// appropriately to show either the on or off bulb. Also swaps the
// disable state of both buttons.
public void processButtonPress(ActionEvent event)
{
    if (event.getSource() == onButton)
    {
        bulbView.setViewport(new Rectangle2D(143, 0, 125, 200)); // on
        onButton.setDisable(true);
        offButton.setDisable(false);
    }
    else
    {
        bulbView.setViewport(new Rectangle2D(0, 0, 125, 200)); // off
        offButton.setDisable(true);
        onButton.setDisable(false);
    }
}
LISTING 6.20 continued

DISPLAY

Light Bulb

See a full-color version of this figure at the end of the text.
The two buttons are labeled On and Off. Both buttons have tool tips set, such that when the user rests the mouse pointer on top of either one, appropriate text appears explaining the purpose of the button.

The buttons are also set up so that only one of them is enabled at a time. Initially, the light bulb is off, so the Off button is disabled. This indicates that the
On button is the only appropriate action at that moment. When the user presses the On button, the image of the light bulb changes, the On button is disabled, and the Off button is enabled.

There is actually only one image used in this program; it contains both the "off" and "on" versions of the light bulb side by side (see Figure 6.3). A viewport is used on the ImageView to display only one side or the other at any point. Viewports are discussed further in Appendix F.

One event handler method is used to process both buttons. That method determines which button was pressed, and then changes the viewport and disabled status of the buttons accordingly.
Summary of Key Concepts

- JavaFX is now the preferred approach for developing Java programs that use graphics and GUIs.
- JavaFX uses a theatre metaphor to present a scene on a stage.
- A GUI is made up of controls, events that represent user actions, and handlers that process those events.
- A single event handler can be used to process events generated by multiple controls.
- The HBox and VBox layout panes arrange their nodes in a single row or column.
- A group of radio buttons provide a set of mutually exclusive options.
- Color and date pickers are controls that allow the user to specify a color or calendar date, respectively.
- Moving the mouse and clicking the mouse button generate events to which a program can respond.
- Rubberbanding is the graphical effect caused when a shape seems to resize as the mouse is dragged.
- Key events allow a program to respond immediately to the user pressing keyboard keys.
- A dialog box is a pop-up window that allows brief, specific user interaction.
- The look and feel of a file chooser is based on the underlying platform.
- Many values in JavaFX classes are managed as properties, which can be bound to other properties.
- Property bindings are used specifically to keep data in sync. They are not a replacement for event handlers in general.
- A slider allows the user to specify a numeric value within a bounded range.
- A spinner lets the user select a value from a list of predefined options using arrow buttons.
- A tool tip provides a hint to the user about the purpose of a control.
- Controls should be disabled when their use is inappropriate.
Summary of Terms

**action event**  A JavaFX event that represents a general action such as a button push.

**change listener**  An object similar to an event handler that is set up to execute when a property value changes.

**check box**  A GUI control that can be toggled on or off using the mouse.

**color picker**  A GUI control that lets the user select a color.

**control**  A screen element that displays information and/or allows the user to interact with a program.

**date picker**  A GUI control that lets the user select a calendar date.

**dialog box**  A window that pops up on top of any currently active window so that the user can interact with it.

**disable**  The act of setting the status of a GUI control so that it cannot be used.

**event**  An object that represents an occurrence to which a program might respond.

**event-driven program**  A program designed to respond to events when they occur.

**event handler**  An object that contains a method that is called when an event occurs.

**file chooser**  A specialized dialog box that allows the user to select a file from a hard drive or other storage medium.

**functional interface**  An interface that contains a single abstract method.

**graphical user interface (GUI)**  An interface to a program that uses graphical elements such as windows, menus, buttons, and text fields.

**interface**  A Java structure that specifies a list of methods that the implementing class must define.

**key event**  An object that represents one of several events that occur when the user types keys on the keyboard.

**lambda expression**  A function that can be defined anonymously (without being given an identifying name) and passed as a parameter.

**layout pane**  A JavaFX node that governs how the nodes it contains are presented visually.

**method reference**  A syntactic technique for specifying a method in an object or class.

**mouse event**  An object that represents one of several events that occur when the user uses the mouse or presses a mouse button.
property  An object that holds a value that can be observed (monitored).

property binding  The act of binding one property to another, so that when the value of one property changes, the other is also changed automatically.

radio button  A GUI control that is used with other radio buttons to provide a set of mutually exclusive options.

root node  A JavaFX node that is displayed in a scene and contains all elements of the scene.

slider  A GUI control that allows the user to specify a numeric value within a bounded range by dragging a knob.

spinner  A GUI control that allows the user to select a value from a list of predefined values arranged in a sequence.

text field  A GUI control that allows the user to enter one line of text.

tool tip  A short line of text that appears when the mouse pointer is paused momentarily over a control or other GUI element.

Self-Review Questions

SR 6.1  What analogy does JavaFX use to represent high level GUI elements?

SR 6.2  What three elements are needed in a JavaFX GUI?

SR 6.3  What does a scene display?

SR 6.4  What is the relationship between an event and an event handler?

SR 6.5  Describe the ways in which a JavaFX event handler can be specified.

SR 6.6  What type of event does a push button generate? A check box? A radio button?

SR 6.7  Compare and contrast check boxes and radio buttons.

SR 6.8  When would you use a slider?

SR 6.9  What is the purpose of a layout pane?

SR 6.10  What is a mouse event? A key event?

SR 6.11  What is a dialog box?

SR 6.12  What's the difference between a JavaFX property and a primitive value?

SR 6.13  What is the purpose of a tool tip?

SR 6.14  When should a GUI control be disabled?
Exercises

EX 6.1 Explain how two controls can be set up to share the same event handler. How can the event handler tell which control generated the event?

EX 6.2 Can one node have multiple event handler methods? Give an example.

EX 6.3 Explain what would happen if the ToggleGroup had not been set on the radio buttons used in the QuoteOptions program.

EX 6.4 In the PushCounter program, why was the count primitive variable and the Text object declared at the class level? Why wasn't anything else, such as the Button?

EX 6.5 Why is the event object that is passed to an event handler method sometimes ignored?

EX 6.6 What kind of event does TextField object generate? When does it generate the event?

EX 6.7 Which user action generates three separate mouse events? Which events? Why?

EX 6.8 Describe the events that can be generated when the user types a key on the keyboard.

EX 6.9 How would you create a rollover effect in a JavaFX program? For example, how would you change the background color of the scene whenever you rolled over an image with the mouse cursor?

EX 6.10 What is a file chooser and how does it relate to a dialog box?

EX 6.11 What is a property binding? Give an example.

EX 6.12 Compare and contrast event handlers and change listeners.

Programming Projects

PP 6.1 Write a JavaFX application that displays a button and a number. Every time the button is pushed, change the number to a random value between 1 and 100.

PP 6.2 Write a JavaFX application that presents a button and a circle. Every time the button is pushed, the circle should be moved to a new random location within the window.

PP 6.3 Write a JavaFX application that presents two buttons and a number (initially 50) to the user. Label the buttons Increment
and Decrement. When the Increment button is pushed, increment the displayed value. Likewise, decrement the value when the Decrement button is pushed.

**PP 6.4** Write a JavaFX application that presents an unlabeled text field in the center of the window surrounded by a circle. When the user enters a radius value in the text field and presses return, redraw the circle accordingly.

**PP 6.5** Write a JavaFX application that presents four labeled text fields, allowing the user to enter values for name, age, favorite color, and hobby. Include a button labeled print. When the button is pushed, the program should print the contents of all fields to standard output using println statements.

**PP 6.6** Write a JavaFX application that allows the user to pick a set of pizza toppings using a set of check boxes. Assuming each topping cost 50 cents, and a plain pizza costs $10, display the cost of the pizza.

**PP 6.7** Write a JavaFX application that allows the user to select a color out of five options provided by a set of radio buttons. Change the color of a displayed square accordingly.

**PP 6.8** Write a JavaFX application that displays the drawing of a traffic light. Allow the user to select which light is on (red, yellow, or green) from a set of radio buttons.

**PP 6.9** Write a JavaFX application that allows the user to display the image of one of the three scooges (Moe, Curly, or Larry) based on a radio button choice.

**PP 6.10** Write a JavaFX application that counts the number of times the mouse button has been clicked on the scene. Display that number at the top of the window.

**PP 6.11** Write a JavaFX application that changes its background color depending on where the mouse pointer is located. If the mouse pointer is on the left half of the program window, display red; if it is on the right half, display green.

**PP 6.12** Write a JavaFX application that draws a circle using a rubberbanding technique. The circle size is determined by a mouse drag. Use the initial mouse press location as the fixed center point of the circle. Compute the distance between the current location of the mouse pointer and the center point to determine the current radius of the circle.
PP 6.13 Write a JavaFX application that serves as a mouse odometer, continually displaying how far, in pixels, the mouse has moved while it is over the program window. Display the current odometer value at the top of the window. Hint: As the mouse moves, use the distance formula to calculate how far the mouse has traveled since the last event, and add that to a running total.

PP 6.14 Write a JavaFX application that displays the side view of a spaceship that follows the movement of the mouse. When the mouse button is pressed down, have a laser beam shoot out of the front of the ship (one continuous beam, not a moving projectile) until the mouse button is released. Define the spaceship using a separate class.

PP 6.15 Modify the AlienDirection program from this chapter so that the image is not allowed to move out of the visible area of the window. Ignore any key event that would cause that to happen.

PP 6.16 Modify the QuoteOptions program so that it provides three additional quote options. Use an array to store the quote strings and a choice box to present the options (instead of radio buttons).

PP 6.17 Write a JavaFX application that displays an image and plays a sound effect with each mouse click. Rotate through four images and five sound effects, so the image/sound effect pairing is different each time.

PP 6.18 Write a JavaFX application that creates polyline shapes dynamically using mouse clicks. Each mouse click adds a new line segment to the current polyline from the previous point to the current mouse position. Allow the user to end the current polyline with a double click. Provide a choice box that allows the user to select the color of the next polyline drawn from a list of five options and provide a button that clears the window and allows the user to begin again. Draw each line segment with a rubberbanding effect.

PP 6.19 Write a JavaFX application that displays a text field, a color picker, and a button. When the user presses the button, display the text obtained from the text field in the color selected by the color picker.

PP 6.20 Modify the RubberLines program so that a color picker is displayed in the upper left corner of the window. Let the value of the color picker determine the color of the next line drawn.
PP 6.21 Write a JavaFX application that displays a Text object and a slider that controls the font size of the text.

PP 6.22 Create a new version of the QuoteOptions program that uses a list view to pick the quote category rather than a set of radio buttons. Provide at least seven categories and corresponding quotes.

PP 6.23 Write a JavaFX application that uses a split pane to display three versions of an image side by side. The first image will be in full color, the second will be in black and white, and the third will have a sepia affect applied. Ensure that the images fill the width of each section of the split pane as the divider bars are moved.

Answers to Self-Review Questions

SRA 6.1 JavaFX uses a theater analogy to describe high-level GUI elements, such as displaying a scene on a stage.

SRA 6.2 A JavaFX GUI is made up of on-screen controls, events that those controls generate, and event handlers that respond to events when they occur.

SRA 6.3 A JavaFX scene displays a root node, which serves as a container for all elements that make up the scene.

SRA 6.4 An event usually represents a user action. An event handler contains methods that are called when an event occurs.

SRA 6.5 In our examples, a JavaFX event handler method is specified using a method reference. But that’s just an alternate way to specify a lambda expression that can be used as the event handler method. Another alternative is to define a full event handler class that implements the EventHandler interface.

SRA 6.6 Push buttons, check boxes, and radio buttons all generate action events, which represent the primary action on the control (the button was pushed).

SRA 6.7 Both check boxes and radio buttons show a toggled state: either on or off. However, radio buttons work as a group in which only one can be toggled on at any point in time. Check boxes, on the other hand, represent independent options.

SRA 6.8 A slider is useful when the user needs to specify a numeric value within specific bounds. Using a slider to get this input, as opposed to a text field or some other control, minimizes user error.
SRA 6.9 A layout pane is used to control the visual presentation of a set of GUI elements. There are several specific layout panes defined in the JavaFX API that display elements in particular ways.

SRA 6.10 A mouse event is one of several events that occur when the user moves the mouse or presses a mouse button. A key event occurs when the user types on the keyboard.

SRA 6.11 A dialog box is a small window that appears for the purpose of conveying information, confirming an action, or accepting input. Generally, dialog boxes are used in specific situations for brief user interactions.

SRA 6.12 A JavaFX property is an object that wraps a primitive value. Furthermore, a property is observable, meaning it can be monitored by a change listener which can take action when the value changes.

SRA 6.13 A tool tip is usually used to explain the purpose of a GUI control or other node by displaying a small amount of text when the mouse cursor is allowed to rest over the node.

SRA 6.14 GUI controls should be disabled when their use is inappropriate. This helps guide the user to proper actions and minimizes error handling and special cases.