CHAPTER 12

Graphical User Interfaces Using JavaFX

CHAPTER CONTENTS

Introduction

Many applications we use every day have a Graphical User Interface, or GUI (pronounced Goo-ey). These GUIs allow the user to communicate to the application by entering text into boxes; pressing buttons; or selecting items from a list, a set of radio buttons, or checkboxes.

GUIs allow the user to drive the application by selecting the next function to be performed, entering the needed data, or setting program preferences, such as colors or fonts. Applications with GUIs are usually easier to learn and use because the interface is familiar to the user.

One way to create a GUI is to use the Swing components. Recently, Oracle introduced JavaFX, a new approach to making GUIs. All the classes needed to create a JavaFX application are automatically included with Java SE; nothing more needs to be downloaded. In this chapter, we present some of the many JavaFX classes, along with the main concepts associated with developing a JavaFX application.

12.1 The Structure of a JavaFX Application

The top-level structure in a JavaFX application is the **stage**, which corresponds to a window. A stage can have one or more **scenes**, which are top-level containers for **nodes** that make up the window contents. A node can be a user interface control, such as a button or a drop-down list; a layout; an image or other media; a graphical shape; a web browser; a chart; or a group. In this chapter, we concentrate on user interface controls, layouts, and images.

To create a JavaFX GUI, we add nodes to a scene. These nodes are arranged in a hierarchy, called a **scene graph**, in which some nodes are children of other nodes. The top node is called the **root**.

JavaFX applications can be built in several ways. If we know which controls our interface needs and how they should be arranged, we can use **FXML**, a scripting language based on **XML** (**Extensible Markup Language**). In this case, we create an FXML file where we specify the layout container and nodes and their properties. For dynamic GUIs where the number or types of controls are determined at runtime, we can define the number, type, properties, and positioning of controls programmatically (that is, through Java code).

In this chapter, we start by using FXML and use Java code later. By defining our GUI using FXML, we separate the GUI from the logic of the code. Also, FXML allows nonprogrammers to contribute to a team by defining the GUI without knowledge of programming.

Examples 12.1 and 12.2 show the basic structure of a JavaFX application that uses FXML.

REFERENCE POINT

You can get more information about the JavaFX classes at https://docs.oracle.com/javase/8/javafx/api/toc.htm

```
1 /*  JavaFX Shell Application
2     Anderson, Franceschi
3 */
4
5 import java.net.URL;
6 import javafx.application.Application;
7 import javafx.fxml.FXMLLoader;
8 import javafx.scene.layout.HBox;
9 import javafx.scene.Scene;
10 import javafx.stage.Stage;
11
12 public class FXShellApplication extends Application
13 {
14   @Override
15   // start is main entry point for the application.
16   // It receives a Stage object – the main window for the
17   // GUI application
18   public void start( Stage stage ) // throws Exception
19   {
20     try
21     {
22       // Locate the FXML resource
23       URL url
24          = getClass( ).getResource( "fxml_shell.xml" );
25
26       // Load the FXML resource, instantiate root Node;
27       // use appropriate layout class for root Node;
28       // here we use HBox
29       HBox root = FXMLLoader.load( url );
30
31       // create a scene associated with the root
32       // and set its width and height
33       Scene scene = new Scene( root, 300, 275 );
34
35       // assign the scene to the stage object
36       stage.setScene( scene );
37
```

```
38        // set title of stage (optional)
39        stage.setTitle( "JavaFX Shell" );
40
41        // make the stage visible
42        stage.show( );
43      }
44      // The FXMLLoader load method throws an exception if
45      // the FXML file is invalid or the URL was not found
46      catch( Exception e )
47      {
48        e.printStackTrace( );
49      }
50    }
51
52    public static void main( String [ ] args )
53    {
54      launch( args );
55    }
56  }
```

Example 12.1 A JavaFX Shell Application

Lines 5 through 10 import the classes we need in most applications. Notice that the JavaFX classes are stored in packages having the prefix of *javafx*.

At line 12, we define our class as inheriting from the *abstract Application* class, which is the entry point for all JavaFX applications. When a JavaFX application is launched, the runtime first calls the *init* method, and then *start*. When the application closes its last window or if it calls the *Platform.exit* method, the runtime calls the *stop* method. The *Application* class provides concrete implementations of the *init* and *stop* methods, so overriding *init* and *stop* is optional. The *Application* class defines the *start* method as *abstract*, however, so we must override the *start* method. Table 12.1 shows some methods of the *Application* class.

Table 12.1 Some Methods of the *Application* Class

Package	javafx.application
Return value	**Method name and argument list**
`void`	`init( )` called by the JavaFX runtime when the application is launched. Overriding this method is optional.

Table 12.1 (continued)

Return value	Method name and argument list
`void`	`launch( String ... args )` *static* method that launches the application. Passes any command-line arguments to the application.
`Application.Parameters`	`getParameters( )` returns any command-line arguments as an *Application.Parameters* object.
`void`	`start( Stage primaryStage )` the main entry point for JavaFX applications. The application places its scene onto the *primaryStage*. This *abstract* method needs to be implemented.
`void`	`stop( )` called when the application signals that it is ready to end. Overriding this method is optional.

Lines 14 through 50 define the *start* method. In line 14, we use the *@Override* annotation. It is good practice to use this annotation whenever we intend to override an inherited method. Using the annotation causes the compiler to alert us if our method header does not correctly override the method in the superclass.

SOFTWARE ENGINEERING TIP

Using the *@Override* annotation whenever you intend to override an inherited method reduces errors because the compiler alerts you if your method header is not correct.

A lot is happening in lines 22 through 29. The FXML file contains the definitions for our GUI components. In order to associate the FXML file with the application, we need to create a URL object containing the file's location; then, we load the file as an application resource. We inherit the *getClass* method from the *Object* class. It returns the *Class* object associated with our application. We then use that object reference to call the *getResource* method in the *Class* class, shown in Table 12.2.

Table 12.2 The *getResource* Method of the *Class<T>* Class

Package	java.lang
Return value	**Method name and argument list**
`URL`	`getResource( String resource )` returns a URL object representing the location of the *resource*, or *null* if the *resource* is not found

Once we have a URL for the FXML file, we call the *static load* method of the *FXMLLoader* class, shown in Table 12.3, to instantiate the layout and components that we defined in our FXML file. This method throws an *IOException* if the format of the FXML file is invalid. In this example, the top-level node is an *HBox* layout object, which arranges its child nodes horizontally across the window. The root and its children make up the scene graph. The return value from the *load* method is a reference to the top-level, or root, node in the FXML file.

Table 12.3 The *load* Method of the *FXMLLoader* Class

Package	javafx.fxml
Return value	**Method name and argument list**
`<T> T`	`load( URL location )` *static* method that instantiates the nodes defined in the FXML file specified by *location* and returns a reference to the root node

After we have the resources loaded, we instantiate a *Scene* object (line 33), which is the container for our GUI content. Table 12.4 shows a constructor of the *Scene* class. Some of the other *Scene* constructors do not require width and height parameters. With those constructors, the scene is sized to fit its contents. For this scene, we specify the layout node returned from the *load* method as the root, and we size the scene to be 300 pixels wide and 275 pixels high.

Table 12.4 A Constructor of the *Scene* Class

Package	javafx.scene
`Scene( Parent root, double width, double height )` instantiates the *Scene* with a top-level node of *root* and sets the width and height to the pixel values specified	

Now we are ready to set the stage, literally. Notice that the *start* method receives a *Stage* object as a parameter (line 18). This corresponds to the top-level window. At line 36, we allocate our newly created scene to the *stage* object. At line 39, we set the text to appear in the title bar, and at line 42, we make the window visible. Table 12.5 shows some useful methods of the *Stage* class.

Table 12.5 Useful Methods of the *Stage* Class

Package	javafx.stage
Return value	**Method name and argument list**
`void`	`setScene( Scene scene )`
	specifies the scene to be hosted by the *Stage*.
`void`	`setTitle( String title )`
	sets the text to appear in the window title bar.
`void`	`show( )`
	makes the window visible. By default, windows are not visible.

We placed most of the code in the *start* method into a *try* block because several exceptions can occur. If the *getResource* method of the *Class* class does not find the FXML file, it returns *null*. This causes a *NullPointerException* when the *FXMLLoader load* method tries to access the file. The *load* method also throws an *IOException* if the FXML file format is invalid. To simplify the examples in this chapter, we use one *catch* block (lines 44–49) that specifies *Exception*, which is a superclass of both *IOException* and *NullPointerException*. In the *catch* block, we print the stack trace to help with debugging errors in our applications. It is good software engineering practice, however, in the final application to catch each exception separately and output a meaningful message to the user.

COMMON ERROR TRAP

Windows are hidden by default. Be sure to call the *show* method at the end of the *start* method. Otherwise, the window will not appear.

At lines 52 through 55, we include the *main* method. Its only job is to start the application by calling the *launch* method of the *Application* class. Any parameters that have been sent to the application are passed to the application. An application may access these parameters using the *getParameters* method.

SOFTWARE ENGINEERING TIP

Catch each possible exception separately. If recovering from the exception is not possible, output a meaningful message to the user.

In Example 12.2, we show a shell FXML file. In this file, we use XML to define the layouts and controls that make up the GUI.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <!-- import classes -->
4 <?import javafx.scene.layout.HBox ?>
5
6 <!-- use appropriate layout manager -->
7 <HBox xmlns:fx="http://javafx.com/fxml" >
8    <!-- define GUI components here -->
9 </HBox>
```

Example 12.2 Shell FXML file: *fxml_shell.xml*

REFERENCE POINT

You can get more information about FXML at http://docs.oracle.com/javafx/2/api/javafx/fxml/doc-files/introduction_to_fxml.html. For more information about XML, visit http://www.w3.org/XML/

Line 1 is the header. Like other lines that do not specify XML elements, it begins with <? and ends with ?>. It defines the XML version as 1.0 and the encoding to be UTF-8, which is essentially Unicode, except that the first 128 characters are stored as 8 bits. Lines 3, 6, and 8 are comments, which use the same syntax as an HTML comment; that is, they begin with <!-- and end with -->. These comments can span more than one line.

At line 4 we import the *HBox* class because it is the root element defined in this sample file. Similarly, we need to import any classes referenced in the XML file.

COMMON ERROR TRAP

Be sure to include an *import* statement for any controls or layout containers referenced in the FXML file. Omitting an *import* statement will generate an exception when the FXML Loader loads the file.

At line 7, we have chosen the *HBox* layout class as an example. In other applications, we put the appropriate top-level layout class here, which becomes the root node for our layout. Lines 7 through 9 illustrate the syntax of an **element**, which we use to define the components of the GUI.

Elements begin with a **start tag** with the element's name enclosed in angle brackets (< >). Some elements, such as the *HBox* definition here, have **closing tags**, which are simply the element's name preceded by a forward slash (/), also enclosed in angle brackets (line 9). Some tags, not illustrated here, have an empty closing tag and end with />.

Elements can have **attributes**, which further define the element or set properties of the element. Attribute definitions are inserted between the element's name and its closing tag.

The syntax for a JavaFX attribute is

```
attributeName = "value"
```

or

```
attributeName = 'value'
```

COMMON ERROR TRAP

Be sure to enclose attribute values in single or double quotes. Omitting one or both quotes will generate an exception when the FXML Loader loads the file.

At line 7, we define an attribute for the root layout:

```
xmlns:fx="http://javafx.com/fxml"
```

This attribute defines the **namespace** for FXML elements. A namespace prevents duplication of variable names, called *name collisions*, by defining a scope for variable names.

Figure 12.1 shows the window created when we run *FXShellApplication.java*. Note that the text in the title bar is "JavaFX Shell" as set in Example 12.1, line 39. The window is empty because we have not added any controls. We add some content to a window in the next section.

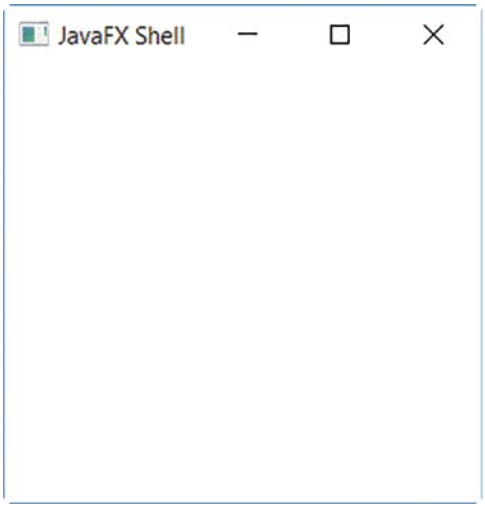


Figure 12.1
The Window Created by *FXShellApplication*

12.2 GUI Controls

JavaFX provides an extensive set of classes that can be used to add a GUI to our applications. A GUI control performs at least one of these functions:

- Displays information
- Collects data from the user
- Allows the user to initiate program functions

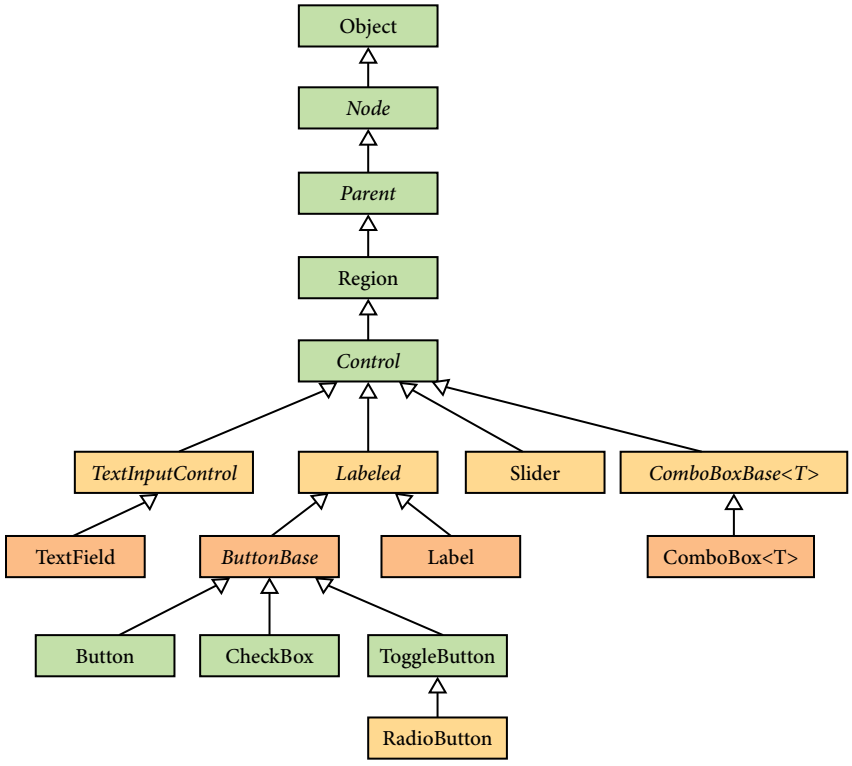
Table 12.6 lists some JavaFX classes that encapsulate GUI controls. All classes listed in Table 12.6 belong to the package *javafx.scene.control*.

Table 12.6 Selected GUI Controls and Their JavaFX Classes

Package	javafx.scene.control
JavaFX Class	**Purpose**
Label	Displays an image or read-only text. Labels are often used to identify the contents of *TextFields*.
TextField	A single-line text box for accepting user input.
Button	Command button that the user clicks to signal that an operation should be performed.
RadioButton	Toggle button that the user clicks to select one option in a group.
CheckBox	Toggle button that the user clicks to select or deselect 0, 1, or more options in a group.
ComboBox	List of options from which the user selects one item.
Slider	Displays a set of continuous values along a horizontal or vertical line. The user can select a value by moving the thumb.

Figure 12.2 shows the hierarchy of some JavaFX controls. Recall that because of the "is a" relationship in inheritance, a subclass object is also an object of each of its superclasses. Thus, all the controls are a *Node* (an element of the scene), a *Parent* (a *Node* that can have children controls), a *Region* (a resizable container that can be styled), and a *Control* (a GUI component). Along the hierarchy, each control has gained methods and properties from its superclasses.

Figure 12.2
The Hierarchy of Some JavaFX Controls



In our applications, we place the controls into the scene by using layout containers, which organize the controls according to each layout's rules. Some of these layout containers are shown in Table 12.7. These classes are in the *javafx.scene.layout* package.

Table 12.7 Commonly Used JavaFX Layout Classes

Package	javafx.scene.layout
Layout Container	**Lays out its children nodes . . .**
HBox	In a single horizontal row.
VBox	In a single vertical column.

Table 12.7 (continued)

Layout Container	Lays out its children nodes . . .
BorderPane	With at most one child in its top, left, right, bottom, and center positions.
GridPane	In a grid of rows and columns. A child can span more than one row or column.
StackPane	In a front-to-back stack.

The hierarchy of the layout classes is shown in Figure 12.3. Like the controls, the layout classes also inherit from *Node*, *Parent,* and *Region*. Instead of inheriting from *Control*, however, these classes inherit from *Pane,* which provides a method for accessing all the children in the scene graph. We use that method later in this chapter.

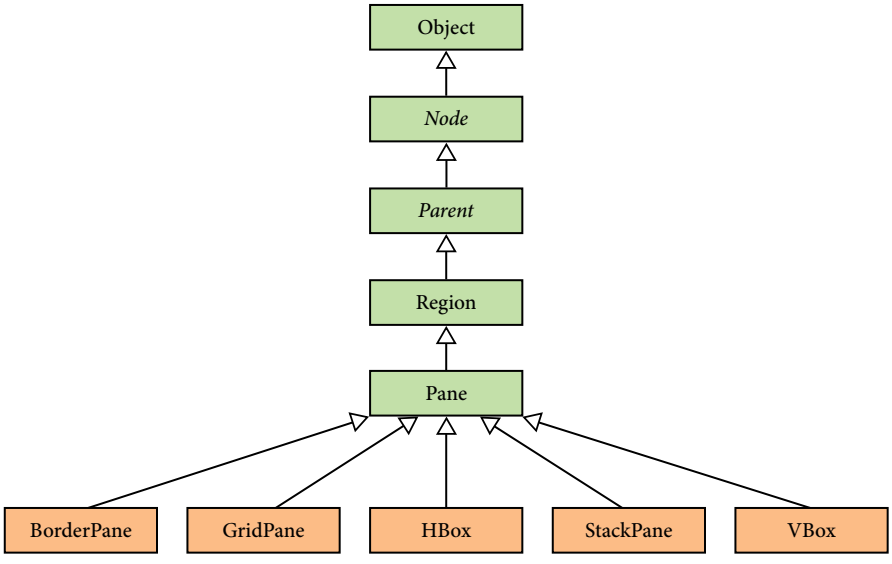


Figure 12.3
The Layout Classes Hierarchy

In the examples in this chapter, we set the root node of our scenes to be one of these layouts and add controls, images, and even other layouts as child nodes. We show how to nest layouts in later examples in this chapter.

12.3 A Simple Control: *Label*

Starting with our shell JavaFX application, let's add a simple control to the window, a *Label,* and an image, which we display through an *ImageView*

node. Example 12.3 shows the application code. Notice that the code is almost identical to Example 12.1, except for the name of the FXML file (line 22), the title bar text (line 32), and that the *root* node is a *VBox* layout (line 24).

```
1 /*  Displaying a Label and image
2     Anderson, Franceschi
3 */
4
5 import java.net.URL;
6 import javafx.application.Application;
7 import javafx.fxml.FXMLLoader;
8 import javafx.scene.layout.VBox;
9 import javafx.scene.Scene;
10 import javafx.stage.Stage;
11
12 public class Dinner extends Application
13 {
14
15   @Override
16   public void start( Stage stage )
17   {
18     try
19     {
20       // find the XML resource
21       URL url
22          = getClass( ).getResource( "fxml_dinner.xml" );
23       // load the XML resource and instantiate the root node
24       VBox root = FXMLLoader.load( url );
25
26       // create a scene
27       Scene scene = new Scene( root, 350, 275 );
28
29       // set the scene
30       stage.setScene( scene );
31       // set title of stage
32       stage.setTitle( "What's for dinner?" );
33       // show the stage
34       stage.show( );
35     }
36     catch( Exception e )
37     {
38       e.printStackTrace( );
39     }
40   }
41
```

```
42   public static void main( String [ ] args )
43   {
44     launch( args );
45   }
46 }
```

Example 12.3 *Dinner.java*

Again, we have an accompanying FXML file, shown in Example 12.4. We begin this file like the shell XML file, with the definition of the XML version and encoding scheme (line 1); then we import all the classes we reference (lines 3–5). In this file, we define the layout to be a *VBox* (lines 7–12). We use attributes to assign values to properties of the *VBox* layout (line 7); we set the *alignment* property to center the child nodes in the window and the *spacing* property to insert 25 pixels between the nodes in the layout.

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2
 3 <?import javafx.scene.control.*?>
 4 <?import javafx.scene.image.*?>
 5 <?import javafx.scene.layout.*?>
 6
 7 <VBox id="root" alignment="CENTER" spacing="25" >
 8   <Label text="Sushi tonight?" textFill="BLUE" />
 9   <ImageView>
10      <Image url="@sushi.jpg" />
11   </ImageView>
12 </VBox>
```

Example 12.4 The *fxml_dinner.xml* File

At line 8, we define a *Label* control and use attributes to set its text to "Sushi tonight?" and its text color (*textFill*) to blue. All attribute values must be plain text, enclosed in double quotes. If an object or primitive value is needed, the FXML Loader performs any necessary conversion. Thus for the text color, we specify the attribute value as "BLUE," and the FXML Loader converts the text to the object *Color.BLUE*.

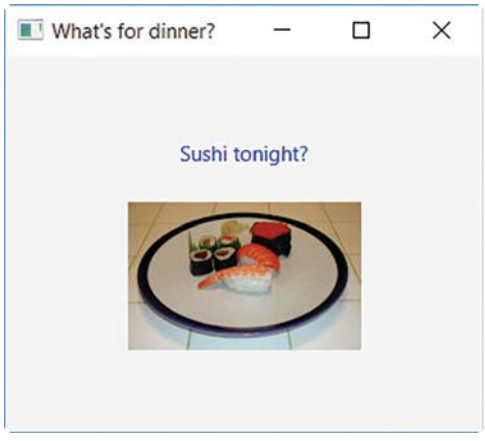
At lines 9 through 11, we define a second child node as an *ImageView* element. We set the *Image* property of the *ImageView* to the file "sushi.jpg" (line 10). We preface the filename with "@" to indicate that we are specifying the filename relative to the current folder. Thus, the file *sushi.jpg* is

REFERENCE POINT

To find the properties that can be set for any layout or control, see the JavaFX class documentation: https://docs.oracle.com/javase/8/javafx/api/toc.htm. Remember also that controls inherit properties from their superclasses.

stored in the same folder as the *.java* file. When this application runs, we see the window shown in Figure 12.4.

Figure 12.4
The Window Produced by Examples 12.3 and 12.4



Skill Practice
with these end-of-chapter questions

12.16.1 Multiple Choice Exercises
Questions 1, 2, 3, 4, 5

12.16.4 Identifying Errors in Code
Question 46

12.16.5 Debugging Area—Using Messages from the Java Compiler and Java JVM
Questions 50, 51, 53

12.4 Event Handling: Managing User Interactions

Now we know how to open a window in an application, and we know how to display a label and an image. The user, however, cannot yet interact with our application. We need to add some interesting GUI controls, such as text entry fields and a button. By interacting with those controls, the user can enter data and initiate operations in our programs.

GUI programming uses an **event-driven model** of programming, as opposed to the procedural model of programming that we have used thus far. By event-driven, we mean that by using a GUI, we put the user in control of what happens next. For example, we might display some text entry fields, some buttons, and a selectable list of items. Then our program "sits back" and waits for the user to interact with the controls. When the user presses a button or selects an item from the list, our application responds by performing the operation the user requested. Then the application sits back again and waits for the user to press another button or select another item from the list. These user actions generate **events**. Thus, the processing of our application consists of responding to events caused by the user interacting with our GUI controls.

When the user interacts with a GUI control, the control **fires an event**. To handle that event, we register our application's interest in being notified when a particular event occurs, and we provide code—an **event handler**, also called a **listener**—to execute when the event occurs.

JavaFX supports and encourages the **Model-View-Controller** architecture for writing GUI applications. In this architecture,

- The **Model** manages the data of the application and its state.
- The **View** presents the user interface.
- The **Controller** handles events generated by the user and communicates those changes to the Model, which updates its state accordingly and communicates any changes back to the Controller. The Controller then updates the View to reflect those changes.

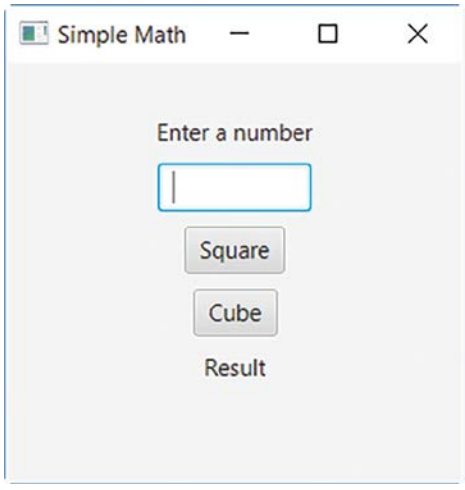
These three components can be placed in the same or in different files.

12.5 Text Fields and Command Buttons

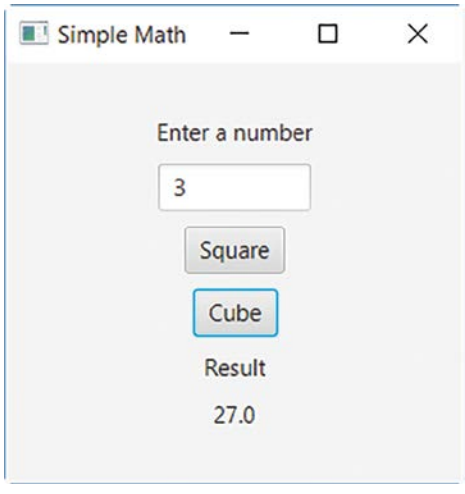
Let's illustrate event handling and the Model-View-Controller architecture with an example that allows a user to square or cube a number. We provide a *TextField* for entering the number; two *Button*s, one for squaring the number and the other for cubing the number; and *Labels* for displaying a prompt and for displaying the squared or cubed result.

The interface we want to create is shown in Figure 12.5A. Figure 12.5B shows the window after the user has entered "3" into the *TextField* and clicked the "Cube" button.

FIGURE 12.5
(A) The Initial Window
(B) The Window After User Interaction



(A)



(B)

For this application, our Model, shown in Example 12.5, is minimal. It provides two *static* methods, *square* (lines 7–14) and *cube* (lines 16–23), to perform the calculations.

```
1 /*  Simple Math Class
2     Anderson, Franceschi
3 */
4
5 public class SimpleMath
6 {
7   /** square method
```

```
 8   *  @param operand  number to square
 9   *  @return the square of operand
10   */
11   public static double square( double operand )
12   {
13     return operand * operand;
14   }
15
16   /** cube method
17   *  @param operand  number to cube
18   *  @return the cube of operand
19   */
20   public static double cube( double operand )
21   {
22     return operand * operand * operand;
23   }
24 }
```

Example 12.5 The Model, *SimpleMath.java*

The controls and their layout are defined in the XML file, shown in Example 12.6, which acts as our View. Because we want to respond to the user clicking either button, we define a controller at line 6 using the *fx:controller* attribute of the layout container with its value set to the class name of the Java file that contains our controller. Thus, in this case, the controller is *SimpleMathController.java*.

The two buttons are defined at lines 12 through 15. The *text* attribute specifies the wording to appear on the button. The *onAction* attribute specifies a method in the controller that should be executed when the user clicks the button. Thus, our controller must define a method named *calculate*. Usually, we code the method as accepting an *ActionEvent* object, as shown below, although this is not required.

```
void calculate( ActionEvent event )
```

The *ActionEvent* object is automatically created when the user clicks a button, selects an item from a list or a menu, or presses the *Enter* key in a *TextField*. The *ActionEvent* object contains data we can query, such as which control fired the event.

The *calculate* method is our event handler or listener, and by setting the *onAction* attribute for a control, we **register** the event handler on our control: our buttons, in this example. Thus, when the user clicks on either of the buttons, the *calculate* method executes.

We have also used the *fx:id* attribute to define a name for all the controls that the controller needs to access. Assigning an *fx:id* to a control puts the control in the *fx* namespace, making the control accessible outside the file in which the control is defined. This allows the Controller to access controls using their *fx:id* value.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <?import javafx.scene.control.*?>
4 <?import javafx.scene.layout.*?>
5
6 <VBox fx:controller="SimpleMathController"
7       xmlns:fx="http://javafx.com/fxml"
8       alignment="center" spacing="10" >
9
10   <Label text="Enter a number" />
11   <TextField fx:id="operand" maxWidth="100" />
12   <Button fx:id="square" text="Square"
13           onAction="#calculate" />
14   <Button fx:id="cube" text="Cube"
15           onAction="#calculate" />
16   <Label text="Result" />
17   <Label fx:id="result" />
18
19 </VBox>
```

Example 12.6 The View, *fxml_simple_math_xml*

Our controller is shown in Example 12.7. It defines as instance variables all the controls it needs to access using their *fx:id* names that we defined in the FXML file (lines 7–10). We use the *@FXML* annotation in front of the instance variable definitions to indicate that these controls, although defined as *private*, are also accessible to FXML.

The *calculate* method is defined at lines 12–27. We also precede the method header with the *@FXML* annotation, again to give FXML access to this *protected* method. The *calculate* method takes as a parameter an *ActionEvent* object. As mentioned, this object is created when the user clicks on the button and contains information about the event that occurred.

At line 16, we use the *getText* method to extract the text that the user typed into the *operand TextField*. The *getText* method is shown along with its companion *setText* method in Table 12.8. Because the *getText* method

returns a *String*, we call the *static parseDouble* method of the *Double* wrapper class to convert the *String* to a *double*. We enclose this operation in a *try/catch* block (lines 14–26) because the *parseDouble* method throws a *NumberFormatException* if the text the user entered cannot be converted to a *double*.

Table 12.8 Useful Methods of the *TextField* Class

Package	javafx.scene.control
Return value	**Method name and argument list**
`String`	`getText( )`
	returns the text typed into the *TextField*
`void`	`setText( String newText )`
	sets the text in the *TextField* to *newText*

If the conversion goes well, we then determine which button the user pressed. The *ActionEvent* class provides a useful method, *getSource*, shown in Table 12.9, which we can use to determine which button the user actually clicked. At line 17, we test whether the user clicked the *square* button. If so, we call the *static square* method in our *SimpleMath* Model, passing the converted number and setting the *result Label* to the return value. Because the *setText* method requires a *String* argument, we concatenate an empty *String* to the return value, which results in a *String*.

Similarly, at lines 19 and 20, we test whether the user clicked the *cube Button*; if so, we call the *static cube* method of the *SimpleMath* class and set the *result Label* to the return value.

```
 1 import javafx.event.ActionEvent;
 2 import javafx.fxml.FXML;
 3 import javafx.scene.control.*;
 4
 5 public class SimpleMathController
 6 {
 7   @FXML private TextField operand;
 8   @FXML private Label result;
 9   @FXML private Button square;
10   @FXML private Button cube;
11
12   @FXML protected void calculate( ActionEvent event )
```

```
13   {
14     try
15     {
16       double op = Double.parseDouble( operand.getText( ) );
17       if ( event.getSource( ) == square )
18         result.setText( "" + SimpleMath.square( op ) );
19       else if ( event.getSource( ) == cube )
20         result.setText( "" + SimpleMath.cube( op ) );
21     }
22     catch( NumberFormatException nfe )
23     {
24       operand.setText( "" );
25       result.setText( "???" );
26     }
27   }
28 }
```

Example 12.7 The Controller, *SimpleMathController.java*

Table 12.9 The *getSource* Method of the *ActionEvent* Class

Package	javafx.event
Return value	**Method name and argument list**
Object	getSource( ) returns the object on which the event was triggered

Finally, *SimpleMathPractice*, the launch class, shown in Example 12.8, performs the usual operations of loading the *fxml_simple_math.xml* file, setting the scene and stage parameters, and displaying the prepared window.

```
1 /*  Simple Math Operations Using Buttons
2     Anderson, Franceschi
3 */
4
5 import java.net.URL;
6 import javafx.application.Application;
7 import javafx.fxml.FXMLLoader;
8 import javafx.scene.layout.VBox;
9 import javafx.scene.Scene;
10 import javafx.stage.Stage;
11
12 public class SimpleMathPractice extends Application
13 {
14   @Override
```

```
15   public void start( Stage stage ) // throws Exception
16   {
17     try
18     {
19       URL url =
20          getClass( ).getResource( "fxml_simple_math.xml" );
21       VBox root = FXMLLoader.load( url );
22       Scene scene = new Scene( root, 300, 275 );
23       stage.setTitle( "Simple Math" );
24       stage.setScene( scene );
25       stage.show( );
26     }
27     catch( Exception e )
28     {
29       e.printStackTrace( );
30     }
31   }
32
33   public static void main( String [ ] args )
34   {
35     launch( args );
36   }
37 }
```

Example 12.8 ***SimpleMathPractice.java***

12.6 Radio Buttons and Checkboxes

If you have ever completed a survey on the web, you are probably familiar with radio buttons and checkboxes.

Radio buttons prompt the user to select one of several mutually exclusive options. Clicking on any radio button deselects any previously selected radio button. Thus, in a group of radio buttons, a user can select only one option at a time.

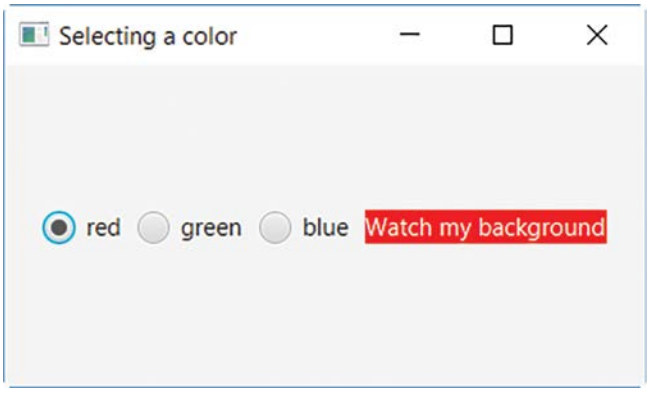
Checkboxes are often associated with the instruction "check all that apply"; that is, the user is asked to select 0, 1, or more options. A checkbox is a toggle button in that if the option is not currently selected, clicking on a checkbox selects the option; if the option is currently selected, clicking on the checkbox deselects the option.

We present two similar examples to illustrate how to use the *RadioButton* and *CheckBox* classes and how they differ. Both examples allow the user to select the background color for a label. We display three color options:

red, green, and blue. Using radio buttons, only one color can be selected at a time. Thus, by clicking on a radio button, the user causes the background of the label to be displayed in one of three colors. Using checkboxes, the user can select any combination of the three color options, so the label color can be set to any of eight possible combinations.

We start with the example using radio buttons. The window when the application starts running is shown in Figure 12.6.

FIGURE 12.6
The GUI for the *ChangingColors* Application



We begin with the Model, *ColorSelector.java*, shown in Example 12.9. Again, this model is minimal, having only one *static* method, *colorToHexString* (lines 11–25). For convenience in passing parameters, we define three *public static* constants at lines 7 through 9. The controller uses these constants as arguments for the *colorToHexString* method.

```
1 /*  ColorSelector class
2     Anderson, Franceschi
3 */
4
5 public class ColorSelector
6 {
7   public static final int RED = 0;
8   public static final int GREEN = 1;
9   public static final int BLUE = 2;
10
11  /** colorToHexString method
12  *  @param selection  the selected color
13  *  @return the hex representation of the selected color
14  */
```

```
15   public static String colorToHexString( int selection )
16   {
17     String result = "#";
18     if ( selection == RED )
19       result += "FF0000";
20     else if ( selection == GREEN )
21       result += "00FF00";
22     else if ( selection == BLUE )
23       result += "0000FF";
24     return result;
25   }
26 }
```

Example 12.9 The Model, *ColorSelector.java*

Next, we look at the View, the FXML file that defines the layout and the controls for the GUI, which is shown in Example 12.10. For this application, we use an *HBox* layout (lines 6–28), which arranges controls horizontally. We set attributes (line 8) to specify that the controls should be centered with 10 pixels between each control. We also specify the class of the Controller that handles the events caused by the user selecting a radio button, as *ChangingColorsController* (line 6).

In order for the radio buttons to be mutually exclusive (i.e., selecting one radio button deselects any previously selected radio button), we define a *ToggleGroup* (lines 10–12). We use the <*fx:define*> element to create objects, such as this *ToggleGroup*, that are not in the scene graph but need to be referenced by a control. In the definition of each radio button (lines 14–22), we add the radio button to the group by setting its *toggleGroup* property to the id of the *ToggleGroup*. We preface the id with "$" to indicate that *colorGroup* is a variable rather than a predefined property value.

For each radio button, we also use the *onAction* property to specify that the *colorChosen* method in the *ChangingColorsController* should be executed when the user clicks the radio button.

At line 14, we set the red radio button's *selected* property to *true*. This automatically sets the radio button as selected when the program starts.

Finally, we define the label whose background we color as the user clicks on the radio buttons (lines 24–26). Here we introduce *style* definitions. If you are familiar with Cascading Style Sheets (CSS) used with HTML pages,

REFERENCE POINT

To learn more about using CSS for styling controls, read the JavaFX CSS Reference Guide at https://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html

this should look familiar to you. Although CSS is beyond the scope of this textbook, we can explain that the *background-color* property sets the background color of a control. Its value can be either a CSS named constant (such as *red*) or the RGB hexadecimal value of the color preceded by a "#" (such as *#FF0000*).

We set the label's background to the color red (*#FF0000*) to be consistent with the *red* radio button being selected when the application starts.

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

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<HBox fx:controller="ChangingColorsController"
      xmlns:fx="http://javafx.com/fxml"
      alignment="center" spacing="10" >

  <fx:define>
    <ToggleGroup fx:id="colorGroup" />
  </fx:define>

  <RadioButton fx:id="red" text="red" selected="true"
               toggleGroup="$colorGroup"
               onAction="#colorChosen" />
  <RadioButton fx:id="green" text="green"
               toggleGroup="$colorGroup"
               onAction="#colorChosen" />
  <RadioButton fx:id="blue" text="blue"
               toggleGroup="$colorGroup"
               onAction="#colorChosen" />

  <Label fx:id="label" text="Watch my background"
         textFill="WHITE"
         style="-fx-background-color:#FF0000" />

</HBox>
```

Example 12.10 ***fxml_changing_colors.xml***

Table 12.10 summarizes the special FXML language elements that we have used.

Table 12.10 FXML Language Elements and Their Meaning

FXML Language Elements and Their Meaning	
FXML Element	**Meaning**
fx:define	Used to create objects not in the scene graph that need to be referenced later
FXML Attributes	**Meaning**
fx:id	Defines a name that can be referenced across the FXML application
fx:controller	Defines a class that contains event handling code for one or more controls
onAction	Defines a method name (preceded by "#") in the controller that should be executed when the user interacts with the control
FXML Annotation	**Meaning**
@FXML	Allows FXML to access a *private* or *protected* class, method, or data
FXML Prefixes	**Meaning**
-fx-	Used with the style attribute to distinguish a JavaFX style attribute from a CSS attribute
$	Used as a prefix to a variable name when the variable is used as a property value
@	Used as a prefix for a URI to specify that the path of the URI starts with the current folder

The Controller, *ChangingColorsController.java*, is shown in Example 12.11. At lines 11 through 14, we define instance variables for all the controls, using the *fx:id* values we defined in the FXML file. The *colorChosen* event handler method, defined at lines 16 through 31, takes as a parameter the *ActionEvent* that is generated when the user clicks on a radio button. We want to set the background color of the label to correspond with the radio button that was clicked. Thus, we set the *style* property of the label. We start by setting a *String* to characters that are common for any color selected (line 18). We then use the *getSource* method of the *ActionEvent* to determine which radio button was clicked and call the *colorToHexString*

method in our Model, passing as a parameter one of the *public static* constants (*RED*, *GREEN*, or *BLUE*) defined in the Model. We then append the returned hexadecimal *String* to *style*. At line 30, we call the *setStyle* method of the *Label* class, passing it the completed *String*. Note that for any property that can be set using attributes in the FXML file, JavaFX usually also provides accessor and mutator methods in the GUI control's class for setting or getting the property programmatically.

```
1 /*  ChangingColorsController class
2     Anderson, Franceschi
3 */
4
5 import javafx.event.ActionEvent;
6 import javafx.fxml.FXML;
7 import javafx.scene.control.*;
8
9 public class ChangingColorsController
10 {
11   @FXML private RadioButton red;
12   @FXML private RadioButton green;
13   @FXML private RadioButton blue;
14   @FXML private Label label;
15
16   @FXML protected void colorChosen( ActionEvent event )
17   {
18     String style = "-fx-background-color: ";
19
20     if ( event.getSource( ) == red )
21      style +=
22       ColorSelector.colorToHexString( ColorSelector.RED );
23     else if ( event.getSource( ) == green )
24      style +=
25       ColorSelector.colorToHexString( ColorSelector.GREEN );
26     else if ( event.getSource( ) == blue )
27      style +=
28       ColorSelector.colorToHexString( ColorSelector.BLUE );
29
30     label.setStyle( style );
31   }
32 }
```

Example 12.11 The Controller, *ChangingColorsController.java*

REFERENCE POINT

To find which properties can be set or retrieved programmatically for any layout or control, see the JavaFX class documentation: https://docs.oracle.com/javase/8/javafx/api/toc.htm. Remember also that controls inherit methods from their superclasses.

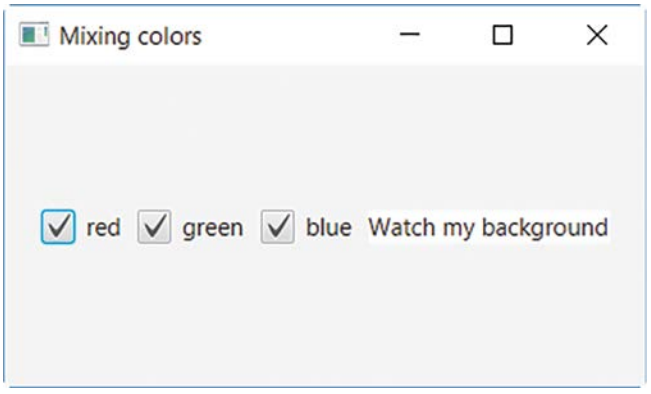
The last part of the application is *ChangingColors*, the launching class, shown in Example 12.12. It follows the format of the previous application launching classes.

```
1 /*  Select a Color Using RadioButtons
2     Anderson, Franceschi
3 */
4
5 import java.net.URL;
6 import javafx.application.Application;
7 import javafx.fxml.FXMLLoader;
8 import javafx.scene.layout.HBox;
9 import javafx.scene.Scene;
10 import javafx.stage.Stage;
11
12 public class ChangingColors extends Application
13 {
14   @Override
15   public void start( Stage stage )
16   {
17     try
18     {
19       URL url =
20        getClass( ).getResource( "fxml_changing_colors.xml" );
21       HBox root = FXMLLoader.load( url );
22       Scene scene = new Scene( root, 400, 200 );
23       stage.setTitle( "Selecting a color" );
24       stage.setScene( scene );
25       stage.show( );
26     }
27     catch( Exception e )
28     {
29       e.printStackTrace( );
30     }
31   }
32
33   public static void main( String [ ] args )
34   {
35     launch( args );
36   }
37 }
```

Example 12.12 The Application Launcher: *ChangingColors.java*

We now demonstrate a similar application using *CheckBoxes*. For this example, we create the interface shown in Figure 12.7.

FIGURE 12.7
The GUI for the *MixingColors* Application



Example 12.13 shows the Model, *ColorMixer.java*. The Model's job is to manage the color mixing based on which colors are selected or deselected. As in our *RadioButton* example, the *ColorMixer* also provides *public static* constants (lines 7–9) for the controller's convenience in passing parameters. To manage which colors are selected, the *ColorMixer* defines an array of three *boolean* values (line 11), with each element representing whether the corresponding color is selected *(true*) or deselected (*false*).

The constructor (lines 13–23) instantiates the array and sets all elements to *true*, representing the color white (all colors selected). The *RED*, *GREEN*, and *BLUE* constants also come in handy here as logical names for the array indexes.

The *toggleColor* method (lines 25–32) alternates the state of a color between *true* and *false*. This is consistent with the behavior of *CheckBoxes,* where each click of the *CheckBox* changes its state between selected and deselected.

Finally, the *hexStringColor* method (lines 34–47) composes a *String* containing the hexadecimal equivalent of the current color. Using the conditional operator, we set each color's contribution to the *String* as either *"FF"* (full color) if selected or *"00"* (no color) if deselected.

```
1 /*  ColorMixer class
2     Anderson, Franceschi
3 */
4
```

```
5  public class ColorMixer
6  {
7    public static final int RED = 0;
8    public static final int GREEN = 1;
9    public static final int BLUE = 2;
10
11   private boolean [ ] rgb;
12
13   /** default constructor
14   *   sets all elements in the rgb array to true
15   *   to represent the color white
16   */
17   public ColorMixer( )
18   {
19     rgb = new boolean[3];
20     rgb[RED] = true;
21     rgb[GREEN] = true;
22     rgb[BLUE] = true;
23   }
24
25   /** toggleColor method
26   *   toggles the color on/off
27   *   @param color  the color to be toggled
28   */
29   public void toggleColor( int color )
30   {
31     rgb[color] = !rgb[color];
32   }
33
34   /** hexStringColor
35   *  @return the hexadecimal representation
36   *          of the color mix
37   */
38   public String hexStringColor( )
39   {
40     String result = "#";
41
42     result += ( rgb[RED] ? "FF" : "00" );
43     result += ( rgb[GREEN] ? "FF" : "00" );
44     result += ( rgb[BLUE] ? "FF" : "00" );
45
46     return result;
47   }
48 }
```

Example 12.13 The Model, *ColorMixer.java*

Next, the FXML file, *fxml_mixing_colors.xml,* shown in Example 12.14, defines the GUI. At lines 11 through 16, we define the three *CheckBoxes* to represent the red, green, and blue color components. We also preselect each *CheckBox* and set the *Label* background to white (line 19) to be consistent with the original state of the Model.

Our controller for this application, *MixingColorsController*, is named at line 6, and we specify that the *mix* method in the controller is the handler for a user selecting or deselecting any of the *CheckBoxes* (lines 12, 14, and 16).

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2
 3 <?import javafx.scene.control.*?>
 4 <?import javafx.scene.layout.*?>
 5
 6 <HBox fx:controller="MixingColorsController"
 7       xmlns:fx="http://javafx.com/fxml"
 8       alignment="center" spacing="10"
 9       style="-fx-background-color:#CCCCCC" >
10
11   <CheckBox fx:id="red" selected="true"
12             text="red" onAction="#mix" />
13   <CheckBox fx:id="green" selected="true"
14             text="green" onAction="#mix" />
15   <CheckBox fx:id="blue" selected="true"
16             text="blue" onAction="#mix" />
17
18   <Label fx:id="label" text="Watch my background"
19          style="-fx-background-color:#FFFFFF" />
20
21 </HBox>
```

Example 12.14 ***fxml_mixing_colors.xml***

The controller, *MixingColorsController.java,* is shown in Example 12.15. The methods in this Model are not *static*, so we define an instance variable, *mixer*, representing the Model (line 11) and the controller instantiates the Model in the constructor (lines 18–21). The *mix* event handler method (lines 23–35) calls the *getSource* method of the *ActionEvent* object to determine which *CheckBox* was clicked. It calls the *toggleColor* method in the Model to invert the state of the color in the *rgb* array. In this way, we do not

care whether the *CheckBox* was just selected or deselected. Once the clicked color component is updated, we compose the new color *String* by calling the *hexStringColor* method of the model (line 33); then we use that *String* to set the *label*'s background color.

```
1 /*  MixingColorsController class
2     Anderson, Franceschi
3 */
4
5 import javafx.event.ActionEvent;
6 import javafx.fxml.FXML;
7 import javafx.scene.control.*;
8
9 public class MixingColorsController
10 {
11   private ColorMixer mixer;
12
13   @FXML private CheckBox red;
14   @FXML private CheckBox green;
15   @FXML private CheckBox blue;
16   @FXML private Label label;
17
18   public MixingColorsController( )
19   {
20     mixer = new ColorMixer( );
21   }
22
23   @FXML protected void mix( ActionEvent event )
24   {
25     if ( event.getSource( ) == red )
26       mixer.toggleColor( ColorMixer.RED );
27     else if ( event.getSource( ) == green )
28       mixer.toggleColor( ColorMixer.GREEN );
29     else if ( event.getSource( ) == blue )
30       mixer.toggleColor( ColorMixer.BLUE );
31
32     String style = "-fx-background-color: ";
33     style += mixer.hexStringColor( );
34     label.setStyle( style );
35   }
36 }
```

Example 12.15 The Controller, *MixingColorsController.java*

The final piece to this application is the standard launch class, shown in Example 12.16.

```
1 /*  Mixing Colors Using CheckBoxes
2     Anderson, Franceschi
3 */
4
5 import java.net.URL;
6 import javafx.application.Application;
7 import javafx.fxml.FXMLLoader;
8 import javafx.scene.layout.HBox;
9 import javafx.scene.Scene;
10 import javafx.stage.Stage;
11
12 public class MixingColors extends Application
13 {
14   @Override
15   public void start( Stage stage )
16   {
17     try
18     {
19       URL url =
20          getClass( ).getResource( "fxml_mixing_colors.xml" );
21       HBox root = FXMLLoader.load( url );
22       Scene scene = new Scene( root, 400, 200 );
23       stage.setTitle( "Mixing colors" );
24       stage.setScene( scene );
25       stage.show( );
26     }
27     catch( Exception e )
28     {
29       e.printStackTrace( );
30     }
31   }
32
33   public static void main( String [ ] args )
34   {
35     launch( args );
36   }
37 }
```

Example 12.16 The Launch Class, *MixingColors.java*

12.7 Programming Activity 1: Working with Buttons

In this activity, you will work with two *Buttons* that control a simulated electrical switch. Specifically, you will write the code to perform the following operations:

1. If the user clicks on the "OPEN" button, open the switch.
2. If the user clicks on the "CLOSE" button, close the switch.

The framework for this Programming Activity will animate your code so that you can check its accuracy. Figures 12.8 and 12.9 show the application

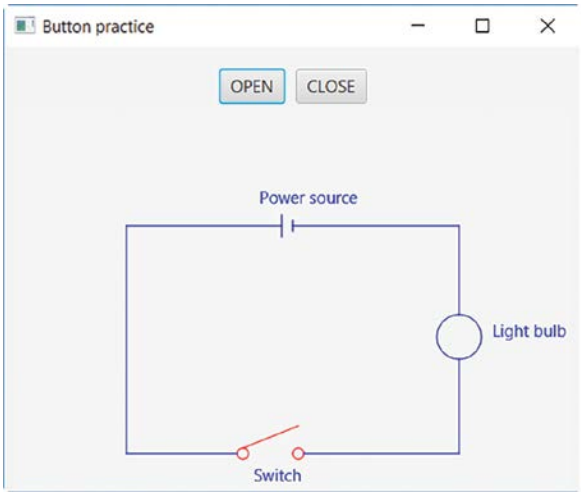


FIGURE 12.8
User Clicked on "OPEN"

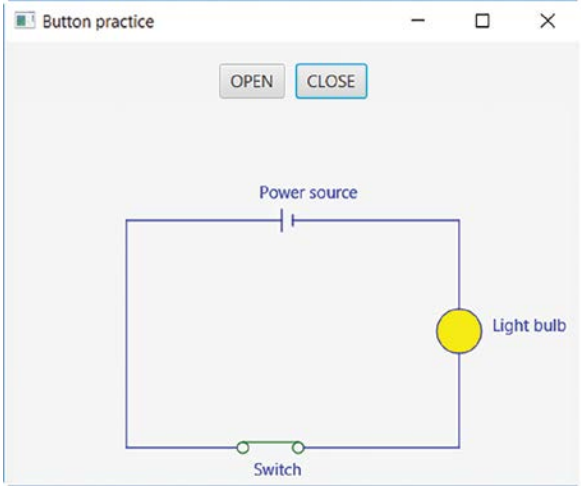


FIGURE 12.9
User Clicked on "CLOSE"

after the user has clicked on the button labeled "OPEN" and on the button labeled "CLOSE," respectively.

Instructions

Copy the source files in the Programming Activity 1 folder for this chapter to a folder on your computer. Note that all files should be in the same folder.

Open the *fxml_button_practice.xml* file. Searching for five asterisks (*****) in the source code will position you to the first code section and then to the second location where you will add your code. In task 1, you will specify the name of the controller as *ButtonPracticeController*. In task 2, you will define an *HBox* layout with the two buttons. Example 12.17 shows the section of the *fxml_button_practice.xml* file where you will add your code.

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

<?import javafx.scene.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<!-- ***** 1. Student code starts here -->
<!-- add the code to this element definition to specify
     the controller as ButtonPracticeController -->
<BorderPane fx:id="bp"
            xmlns:fx="http://javafx.com/fxml" >

  <top>
    <!-- ***** 2. Student code restarts here -->
    <!-- add code to define an HBox layout that is centered
         with an id of hBox and 10 pixels between controls.
         Inside the HBox, add 2 Buttons, with text OPEN and
         CLOSE, and ids of open and close. Clicking on either
         button triggers a call to the event handler flip
         method -->



  </top>

</BorderPane>
```

Example 12.17 Location of Student Code in *fxml_button_practice.xml*

Note that we define an *HBox* layout container inside a *BorderPane* layout container, illustrating that layouts can be nested. We demonstrate nested layouts again later in the chapter.

Next, open *ButtonPracticeController.java* and search for five asterisks (*****). This will position you at the third and final task for this Programming Activity, writing the event handler for the buttons. Example 12.18 shows the section of the source code where you will add your code.

```
44   // ***** 3. Student code restarts here
45   // Code the flip method.
46   // To open the switch, call the open method
47   //   with circuit, the object reference of the
48   //   Circuit object.
49   //   The open method does not take any parameters.
50   // To close the switch, call the close method
51   //   with circuit.
52   //   The close method does not take any parameters.
53   // The last statement of the method should be
54   //   animate( );
```

Example 12.18 Location of Student Code in *ButtonPracticeController.java*

Our framework will animate your code so that you can watch your code work. For this to happen, be sure that you call the *animate* method as the last statement in the *flip* method.

Troubleshooting

If your *flip* method does not animate, check these tips:

- Verify that the last statement in your *flip* method is:

  ```
  animate( );
  ```

- Verify that your listener is registered on the buttons (*onAction*).
- Verify that you have correctly identified the button that fired the event using the *getSource* method.

DISCUSSION QUESTIONS ?

1. Explain why the *getSource* method is useful here.
2. Could you implement this application with *RadioButtons* instead of *Buttons*? What definition would you need to add to the *FXML* file? How would the *flip* method change, if at all?
3. Which class is the Model?

12.8 Combo Boxes

A *ComboBox* implements a drop-down list. When the combo box appears, either no item or one item is displayed, along with a down arrow icon. When the user presses on the down arrow, the combo box "drops" open and displays a list of items, with a scroll bar for viewing more items. The user can select one item from the list. When the user selects an item, the list closes and only the selected item is displayed.

In this example, we allow the user to select a country from a combo box, and in response, we display an image of a typical food from the selected country. Figure 12.10 shows the window when our application begins.

FIGURE 12.10
The Food Sampling Application

We begin as usual with the Model, *FoodSampler.java*, shown in Example 12.19. Lines 9 through 16 define two arrays. The *countryList* array contains the country names to be displayed in the *ComboBox*, and the *foods* array contains the corresponding images of food stored in the same order as the country names. The instance variable *selectedIndex*, defined at line 17, holds the index of the selected country. We use this index to retrieve the appropriate food image to display. With *selectedIndex* set to 0, the Model's initial state is that the country name "France" is selected and *cheese.jpg* is the corresponding image.

The remainder of the Model consists of accessor methods for the *countryList* array (lines 19–25) and *selectedIndex* (lines 27–33), a mutator method for *selectedIndex* (lines 35–41), and an accessor for the image corresponding to *selectedIndex* (lines 43–50).

```
1 /*  FoodSampler class
2     Anderson, Franceschi
3 */
4
5 import javafx.scene.image.Image;
6
7 public class FoodSampler
8 {
9   private String [ ] countryList =
10     { "France", "Greece", "Italy", "Japan", "USA" };
11   private Image [ ] foods =
12     { new Image( "cheese.jpg" ),
13       new Image( "fetasalad.jpg" ),
14       new Image( "pizza.jpg" ),
15       new Image( "sushi.jpg" ),
16       new Image( "hamburger.jpg" ) };
17   private int selectedIndex = 0;
18
19   /** getCountryList method
20   *  @return a reference to the countryList array
21   */
22   public String [ ] getCountryList( )
23   {
24     return countryList;
25   }
26
27   /** the accessor for selectedIndex
28   *  @return the index selected
29   */
30   public int getSelectedIndex( )
31   {
32     return selectedIndex;
33   }
34
35   /** the mutator for selectedIndex
36   * @param selection the new value for selectedIndex
37   */
38   public void updateSelection( int selection )
39   {
40     selectedIndex = selection;
```

```
41   }
42
43   /** the accessor for the image to display
44   *  @return the image from the Image array
45   *          with the index of selected
46   */
47   public Image getImageSelected( )
48   {
49     return foods[selectedIndex];
50   }
51 }
```

Example 12.19 The Model, *FoodSampler.java*

Next we look at the FXML file, *fxml_food_samplings.xml,* shown in Example 12.20. At line 7, we identify the Controller as *FoodSamplingsController.java*. Lines 11 through 13 define our *ComboBox* control and that the *itemSelected* method in the Controller class will handle the event fired by a user selecting an item from the *ComboBox* drop-down list. We also set the *visibleRowCount* property to 3, specifying that when the user opens the *ComboBox*, three items should be displayed along with a scrollbar for viewing the other hidden items. The maximum value for the *visibleRowCount* property is 10. If the list contains fewer items than specified in the *visibleRowCount* property, the *ComboBox* displays all the items in the list.

At lines 15 and 16 we also define an *ImageView* control to display the appropriate food image. We do not specify the initial image here; we let the Controller get the appropriate image from the Model.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <?import javafx.scene.control.*?>
4 <?import javafx.scene.image.*?>
5 <?import javafx.scene.layout.*?>
6
7 <HBox fx:controller="FoodSamplingsController"
8       xmlns:fx="http://javafx.com/fxml"
9       alignment="center" spacing="10" >
10
11   <ComboBox fx:id="countries" visibleRowCount="3"
12             onAction="#itemSelected" >
13   </ComboBox>
14
15   <ImageView fx:id="foodImage">
```

```
16    </ImageView>
17
18 </HBox>
```

Example 12.20 The *fxml_food_samplings.xml* File

We turn now to the Controller, *FoodSamplingsController.java*, shown in Example 12.21. New to this Controller is the *initialize* method, which is called after the scene graph has been created, and can be used to add items to the scene graph that could not be fully defined in the FXML file. Because the Model stores the list of countries and corresponding images, we cannot specify the items for *ComboBox* in the FXML file. However, this approach also has some advantages: By adding the items to the *ComboBox* at runtime, we can easily handle changes in the list without needing to change our Controller or FXML code, and we can use this application with another Model that has a different set of items and images.

A *ComboBox* control is a generic class, so when the Controller defines the *ComboBox* (line 14), we need to specify the class type of the items to be displayed; in this example, we are displaying *Strings*.

A *ComboBox*, like other controls that allow users to select items, uses a *SelectionModel* to handle the user's interactions with the control. The *ComboBox uses* the *SingleSelectionModel*, which is a subclass of *SelectionModel*, to restrict the user to a choice of only one item from the list. The *SingleSelectionModel* class is also a generic class; thus, we specify the class type to be *String* in line 16. Table 12.11 shows some useful methods of the *SingleSelectionModel* class for setting and getting the user's selection.

Table 12.11 Useful Methods of the *SingleSelectionModel* Class

Package	javafx.scene.control
Return value	**Method name and argument list**
`int`	`getSelectedIndex( )`
	returns the index of the currently selected item.
`T`	`getSelectedItem( )`
	returns the currently selected item.
`void`	`select( int index )`
	selects the item at *index*. Any previously selected item is deselected.

In order for our Controller event handler method, *itemSelected,* to be notified when the user selects an item, we need to put our items into an *ObservableList* object. Table 12.12 shows the *getItems* method of the *ComboBox*, which returns an *ObservableList*. The *addAll* method of the *ObservableList* class, shown in Table 12.13, can be used to insert items into the list. The *addAll* method uses the *varargs* syntax to accept a variable number of arguments.

Table 12.12 A Useful Method of the *ComboBox<T>* Class

Package	javafx.scene.control
Return value	**Method name and argument list**
`ObservableList`	`getItems( )`
	returns the values of the items in the *ComboBox* as an *ObservableList*

Table 12.13 A Useful Method of the *ObservableList<E>* Class

Package	javafx.collections
Return value	**Method name and argument list**
`boolean`	`addAll( E... elements )`
	adds the elements to the list

Putting this all together, in the *initialize* method, we instantiate the Model (line 20), and then call the *getItems* method, which returns an empty *ObservableList.* Next, we call the *addAll* method of the *ObservableList* class to insert into the *ComboBox* the array of country names returned by the Model's *getCountryList* method (lines 22–23).

At line 26, we get a reference to the *SingleSelectionModel;* we then use that reference to select the item using the index returned from the Model's *getSelectedIndex* method (line 29). At this point, the value returned will be 0, because in the Model, we initialize the *selectedIndex* instance variable to 0. We then set the image for our *foodImage ImageView* to the filename returned by the Model's *getImageSelected* method (line 30). Thus, when the window first appears, the country France is selected and we display the *cheese.jpg* image.

The *itemSelected* event handler, which executes whenever the user selects a new item from the *ComboBox* (lines 33–41), retrieves the index of the selected item by calling the *getSelectedIndex* method, using the *selectionModel* reference. Using that index, we notify the model that the user has changed the selection. Then we determine which new image to display by calling the Model's *getImageSelected* method.

```
1 /*  FoodSamplingsController class
2     Anderson, Franceschi
3 */
4
5 import javafx.event.ActionEvent;
6 import javafx.fxml.FXML;
7 import javafx.scene.control.*;
8 import javafx.scene.image.*;
9
10 public class FoodSamplingsController
11 {
12   private FoodSampler sampler;
13
14   @FXML private ComboBox<String> countries;
15   @FXML private ImageView foodImage;
16   private SingleSelectionModel<String> selectionModel;
17
18   public void initialize( )
19   {
20    sampler = new FoodSampler( );
21
22    // populate combobox with data from the Model
23    countries.getItems( ).addAll( sampler.getCountryList( ) );
24
25    // get a reference to the SingleSelectionModel
26    selectionModel = countries.getSelectionModel( );
27
28    // initialize View with initial data from Model
29    selectionModel.select( sampler.getSelectedIndex( ) );
30    foodImage.setImage( sampler.getImageSelected( ) );
31   }
32
33   @FXML protected void itemSelected( ActionEvent event )
34   {
35    // retrieve index of country selected
36    int index = selectionModel.getSelectedIndex( );
37    // update the Model
38    sampler.updateSelection( index );
39    // update the View with Image from the Model
40    foodImage.setImage( sampler.getImageSelected( ) );
41   }
42 }
```

Example 12.21 The Controller, *FoodSamplingsController.java*

The last piece of the application is the launch class, shown in Example 12.22.

```
1 /*  Using ComboBox to show a sampling of international foods
2     Anderson, Franceschi
3 */
4
5 import java.net.URL;
6 import javafx.application.Application;
7 import javafx.fxml.FXMLLoader;
8 import javafx.scene.layout.HBox;
9 import javafx.scene.Scene;
10 import javafx.stage.Stage;
11
12 public class FoodSamplings extends Application
13 {
14   @Override
15   public void start( Stage stage )
16   {
17    try
18    {
19      URL url =
20         getClass( ).getResource( "fxml_food_samplings.xml" );
21      HBox root = FXMLLoader.load( url );
22      Scene scene = new Scene( root, 450, 200 );
23      stage.setTitle( "Food samplings of various countries" );
24      stage.setScene( scene );
25      stage.show( );
26    }
27    catch( Exception e )
28    {
29      e.printStackTrace( );
30    }
31   }
32
33   public static void main( String [ ] args )
34   {
35     launch( args );
36   }
37 }
```

Example 12.22 ***FoodSamplings.java***

12.9 Sliders

The *Slider* control is capable of displaying a set of continuous values along a horizontal or vertical line, called a **track**. The user "slides" the knob, called the **thumb**, along the track to select a value from the set. To illustrate the

Slider control, we create an application that allows the user to control the level of gray in a photo. A gray color is created when the red, green, and blue components have the same value. Each color component's value can range from 0.0 to 1.0.

Figure 12.11 shows the GUI when the application begins. The top photo is the original version of the image; you may recognize this as the *sushi.jpg* photo from the *ComboBox* example in the last section. The lower photo has been converted to a middle level of gray. As the user slides the knob left, we darken the photo; as the user slides the knob to the right, we lighten the photo. The vertical lines and numbers shown under the slider are **tick marks** and **tick values**. The values represent a multiplier we use to convert the red, green, and blue color values of the photo to a new level of gray.

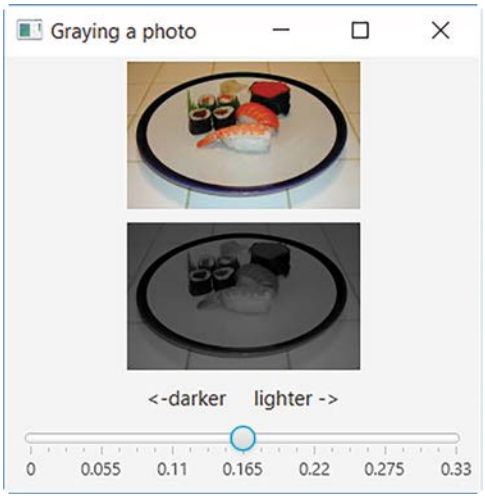


FIGURE 12.11
The Photo Graying GUI

We define the *Slider* control in the FXML file, shown in Example 12.23. At lines 20 through 23, we specify the minimum multiplier value as 0, the maximum as .33, and the initial value in the middle of that range as .165. We set properties to show the tick marks and tick labels and to display the tick marks at intervals of .055 units.

In line 10, we add a *style* property to define padding values to the *VBox*. We add 10 pixels of padding on the left and right sides of the *VBox,* but no padding on the top and bottom.

The *Label* control defined at line 18 aids the user by showing that moving the thumb to the left will darken the image and moving the thumb to the right will lighten the image. Because angle brackets have syntactic meaning

in FXML, we use the special character encoding sequences *<lt;* and *>gt;* to represent the left and right angle brackets, respectively. Table 12.14 shows the encoding sequences for the XML and FXML special characters. The five XML encoding sequences start with an ampersand and end with a semicolon. In addition, FXML adds three special characters; the encoding sequences for these FXML special characters consist simply of a leading backslash followed by the special character. These encoding sequences are useful whenever a control's text value contains any of these special characters as data.

SOFTWARE ENGINEERING TIP

Labels can help guide the user through the interface.

Table 12.14 XML and FXML Special Characters

Character	XML Encoding Sequence
<	&lt;
>	&gt;
&	&amp;
"	&quot;
'	&apos;
Character	**FXML Encoding Sequence**
@	\@
$	\$
%	**\%**

COMMON ERROR TRAP

Be sure to use the XML and FXML encoding sequences when specifying text values that contain characters having syntactic meaning in XML and FXML. Otherwise, the FXML Loader will generate an error when loading the FXML file.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <?import javafx.scene.control.*?>
4 <?import javafx.scene.image.*?>
5 <?import javafx.scene.layout.*?>
6
7 <VBox fx:controller="PhotoGrayerController"
8     xmlns:fx="http://javafx.com/fxml"
9     alignment="center" spacing="10"
10    style="-fx-padding:0 10 0 10;" >
11
12   <ImageView fx:id="originalImageView" >
13     <Image url = "Sushi.jpg" />
14   </ImageView>
15
```

```
16   <ImageView fx:id="grayImageView" />
17
18   <Label text="&lt;-darker     lighter -&gt;" />
19
20   <Slider fx:id="slider" min="0" max=".33"
21           value=".165" showTickMarks="true"
22           showTickLabels="true" majorTickUnit="0.055" />
23
24 </VBox>
```

Example 12.23 The View, *fxml_photo_grayer.xml*

By looking at the Model (Example 12.24), we can see how the *Slider*'s multiplier values are used to adjust the image's color. The Model, *PhotoGrayer.java,* extends the *Image* class, which is used to load graphical images (GIF, BMP, JPG, and PNG) from a URL. In our constructor (lines 13–21), we pass the image filename to the constructor of the *Image* superclass, shown in Table 12.15.

The purpose of the *gray* method (lines 23–50) is to create an image composed of varying levels of gray. The gray level of each pixel in the new image is computed by multiplying the red, green, and blue components of the corresponding pixel in the orginal image by the parameter *coeff*.

Let's look at the *gray* method in detail. To get the size of the image, we call the *getWidth* and *getHeight* methods, shown in Table 12.15, which we have inherited from the *Image* class. Using this information, we create an empty *WritableImage* object of the same size. A *WritableImage* object can be used to create an image from pixels. Table 12.16 shows the constructor and useful methods of the *WritableImage* class, which also inherits from the *Image* class.

We then use the *getPixelWriter* factory method (line 34) to obtain a *PixelWriter* object to write pixel data into the *WritableImage*.

At line 35, we call the *getPixelReader* factory method to get a *PixelReader* object to read pixel data from the original image. A *PixelReader* object provides access to each pixel in the original image (see Table 12.17), while a *PixelWriter* object allows us to set the color of each pixel in the *WritableImage* (see Table 12.18).

Using a nested *for* loop (lines 37–48), we use that *PixelReader* reference to move through each pixel in the original image. We first retrieve the color of

each pixel by calling the *getColor* method of the *PixelReader*, passing it the current *x* and *y* values (lines 40–41).

Using the current color of the original image, we then multiply the sum of the red, green, and blue components by the *coeff* parameter (lines 42–44) and pass the resulting value to the *gray* method of the *Color* class (line 45); this returns a *Color* object where the red, green, and blue components have the *grayValue* parameter's value. The methods of the *Color* class that we use in this example are shown in Table 12.19. Finally, we use the *PixelWriter* to set the color of the corresponding pixel within the *WritableImage* (line 46) to the newly computed gray color.

We return the new image at line 49.

```
 1 /*  Adjusts the gray value of an image
 2     Anderson, Franceschi
 3 */
 4
 5 import javafx.scene.image.Image;
 6 import javafx.scene.image.PixelReader;
 7 import javafx.scene.image.PixelWriter;
 8 import javafx.scene.image.WritableImage;
 9 import javafx.scene.paint.Color;
10
11 public class PhotoGrayer extends Image
12 {
13   /** constructor
14   * @param file  filename of the image
15   *    passes the filename to the Image class
16   *    constructor
17   */
18   public PhotoGrayer( String file )
19   {
20     super( file );
21   }
22
23   /** gray method
24   *  @param  the multipler to determine the
25   *          color for each pixel
26   *  @return a WritableImage
27   */
28   public WritableImage gray( double coeff )
29   {
```

```
30      int width = ( int ) getWidth( );
31      int height = ( int ) getHeight( );
32      WritableImage grayImage
33           = new WritableImage( width, height );
34      PixelWriter pw = grayImage.getPixelWriter( );
35      PixelReader pr = getPixelReader( );
36
37      for ( int x = 0; x < width; x++ )
38      {
39        for ( int y = 0; y < height; y++ )
40        {
41          Color currentColor = pr.getColor( x, y );
42          double grayValue =  coeff *
43            ( currentColor.getRed( ) + currentColor.getGreen( )
44              + currentColor.getBlue( ) );
45          currentColor = Color.gray( grayValue );
46          pw.setColor( x, y, currentColor );
47        }
48      }
49      return grayImage;
50    }
51  }
```

Example 12.24 The Model, *PhotoGrayer.java*

Table 12.15 A Constructor and Useful Methods of the *Image* Class

Package	javafx.scene.image
Constructor	
`Image( String URL )`	constructs an *Image* from the file named in *URL*
Return value	**Method name and argument list**
`PixelReader`	`getPixelReader( )` returns a *PixelReader* object for accessing the pixels in the *Image*
`int`	`getHeight( )` returns the height of the image in pixels
`int`	`getWidth( )` returns the width of the image in pixels

Table 12.16 A Constructor and a Useful Method of the *WritableImage* Class

Package	javafx.scene.image
Constructor	
`WritableImage( int width, int height )`	constructs an empty *WritableImage* of the specified *width* and *height*
Return value	**Method name and argument list**
`PixelWriter`	`getPixelWriter( )` returns a *PixelWriter* object for writing the pixels in the *Image*

Table 12.17 A Method of the *PixelReader* Interface

Package	javafx.scene.image
Return value	**Method name and argument list**
`Color`	`getColor( int x, int y )` returns the *Color* of the pixel at coordinate (*x*, *y*) in the image

Table 12.18 A Method of the *PixelWriter* Interface

Package	javafx.scene.image
Return value	**Method name and argument list**
`void`	`setColor( int x, int y, Color c )` sets the pixel at coordinate (*x*, *y*) to the color *c*

Table 12.19 Useful Methods of the *Color* Class

Package	javafx.scene.paint
Return value	**Method name and argument list**
`double`	`getRed( )` returns the red component of the color in the range 0.0–1.0
`double`	`getGreen( )` returns the green component of the color in the range 0.0–1.0
`double`	`getBlue( )` returns the blue component of the color in the range 0.0–1.0.
`Color`	`gray( double value )` returns a *Color* object where the red, green, and blue components are set to *value*, which can range from 0.0 to 1.0

The Controller, *PhotoGrayerController.java,* shown in Example 12.25, handles any changes to the *Slider*'s value made by the user dragging the thumb in either direction. In the *initialize* method (lines 14–25), we instantiate the Model (line 16), sending the filename of the photo to the constructor.

```
 1 import javafx.beans.value.*;
 2 import javafx.event.*;
 3 import javafx.fxml.FXML;
 4 import javafx.scene.control.*;
 5 import javafx.scene.image.*;
 6
 7 public class PhotoGrayerController
 8 {
 9   private PhotoGrayer photoGrayer;
10
11   @FXML private ImageView grayImageView;
12   @FXML private Slider slider;
13
14   public void initialize( )
15   {
16     photoGrayer = new PhotoGrayer( "sushi.jpg" );
17
18     // initialize grayImageView
19     Image grayImage = photoGrayer.gray( slider.getValue( ) );
20     grayImageView.setImage( grayImage );
21
22     // set up event handling for slider
23     SliderHandler sh = new SliderHandler( );
24     slider.valueProperty( ).addListener( sh );
25   }
26
27   private class SliderHandler
28                 implements ChangeListener<Number>
29   {
30     @Override
31     public void changed( ObservableValue<? extends Number> o,
32                          Number oldValue, Number newValue )
33     {
34       // update grayImageView
35       grayImageView.setImage(
36              photoGrayer.gray( newValue.doubleValue( ) ) );
37     }
38   }
39 }
```

Example 12.25 The Controller, *PhotoGrayerController.java*

At lines 18 through 20, we initialize the second photo by calling the *gray* method in the Model, sending it the current slider value by calling the *getValue* method of the *Slider* class, as shown in Table 12.20.

The final initialization task is to register the event handler for the *Slider* control. Before we cover that, let's look at lines 27 through 38, where we define the event handler as a *private* **inner class** named *SliderHandler*. A *private* inner class is defined inside a *public* class and has access to all the members of the *public* class. Thus, declaring our event handler as a *private* inner class simplifies our code by giving the event handler direct access to our application's GUI components.

Similar to the *ComboBox*, for which we set up an *ObservableList*, the slider value is an *ObservableValue*, meaning that we can register an event handler that will be notified when the value changes. The *SliderHandler* class implements the *ChangeListener* interface, which has one method that we must override, shown in Table 12.21 and here:

```
void changed( ObservableValue<? extends T> observable,
                T oldValue, T newValue )
```

The syntax *<? extends T>* means that the observable value can be of any type that is a subclass of the generic type *T* or the type *T* itself. At lines 30 through 37, where we override the method above, we specify the *T* parameter to be *Number* (in *java.lang*), which is an *abstract* superclass extended by Java's numeric wrapper classes: *Byte*, *Short*, *Integer*, *Long*, *Float*, and *Double*. This works because our observable value is a *Double*.

Inside the method, we set the new color for the *ImageView grayImageView* by calling the *gray* method of the Model and sending it the new value of the *Slider*, obtained by calling the *doubleValue* method of the *Double* class, shown in Table 12.22.

Now we're ready to look at lines 23 and 24. At line 23, we instantiate an object of the *SliderHandler* inner class. At line 24, we call the *valueProperty* method of the *Slider* class (shown in Table 12.20), which returns the value of the *Slider* as a *DoubleProperty* object. Because the *DoubleProperty* class implements the *ObservableValue* interface, we can call the *addListener* method of that class to register our event handler. The *ObservableValue* interface, shown in Table 12.23, is also generic. The syntax *<? super T>* means that the observable item can be of any type that is a superclass of the generic type *T* or the type *T* itself.

Table 12.20 Some Useful Methods of the *Slider* Class

Package	javafx.scene.control
Return value	**Method name and argument list**
`double`	`getValue( )` returns the current value of the *Slider* as a *double*
`DoubleProperty`	`valueProperty( )` returns the current value of the *Slider* as a *DoubleProperty,* which is a *Property* wrapper class for a *double,* and implements the *ObservableValue<Number>* interface

Table 12.21 The Method of the *ChangeListener<T>* Interface

Package	javafx.beans.value
Return value	**Method name and argument list**
`void`	`changed( ObservableValue<? extends T> observable, T oldValue, T newValue )` The *ChangeListener* must implement this method to handle changes to the *ObservableValue*

Table 12.22 A Useful Method of the *Double* Class

Package	java.lang
Return value	**Method name and argument list**
`double`	`doubleValue( )` *abstract* method in the *Number* class overridden by the *Double* class; it returns the value of a *Double* object as a *double*.

Table 12.23 The Methods of the *ObservableValue*<T> Interface

Package	javafx.beans.value
Return value	**Method name and argument list**
`void`	`addListener ( ChangeListener<? super T> listener )` adds a *ChangeListener* whose code will execute whenever the value of the *ObservableValue* changes
`T`	`getValue( )` returns the current value of the *ObservableValue* object
`void`	`removeListener( ChangeListener<? super T> listener )` removes the listener so that it is no longer notified of changes to the value of the *ObservableValue*

The last piece of the application is the launch class, shown in Example 12.26.

```
1 /*  Slider demo
2     Anderson, Franceschi
3 */
4
5 import java.net.URL;
6 import javafx.application.Application;
7 import javafx.fxml.FXMLLoader;
8 import javafx.scene.layout.VBox;
9 import javafx.scene.Scene;
10 import javafx.stage.Stage;
11
12 public class PhotoGraying extends Application
13 {
14   @Override
15   public void start( Stage stage )
16   {
17     try
18     {
19       URL url =
20         getClass( ).getResource( "fxml_photo_grayer.xml" );
21       VBox root = FXMLLoader.load( url );
22       Scene scene = new Scene( root, 350, 320 );
23       stage.setTitle( "Graying a photo" );
24       stage.setScene( scene );
25       stage.show( );
26     }
27     catch( Exception e )
28     {
29       e.printStackTrace( );
30     }
31   }
32
33   public static void main( String [ ] args )
34   {
35     launch( args );
36   }
37 }
```

Example 12.26 ***PhotoGraying.java***

Skill Practice
with these end-of-chapter questions

12.16.1 Multiple Choice Exercises

Questions 6, 7, 8, 9

12.16.2 Reading and Understanding Code

Questions 18, 19, 20, 21, 22, 23, 24, 25, 26

12.16.3 Fill In the Code

Questions 30, 31, 32

12.16.4 Identifying Errors in Code

Question 45

12.16.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

Question 52

12.16.6 Write a Short Program

Questions 55, 56, 57, 59, 60, 61, 62, 63, 66

12.16.7 Programming Projects

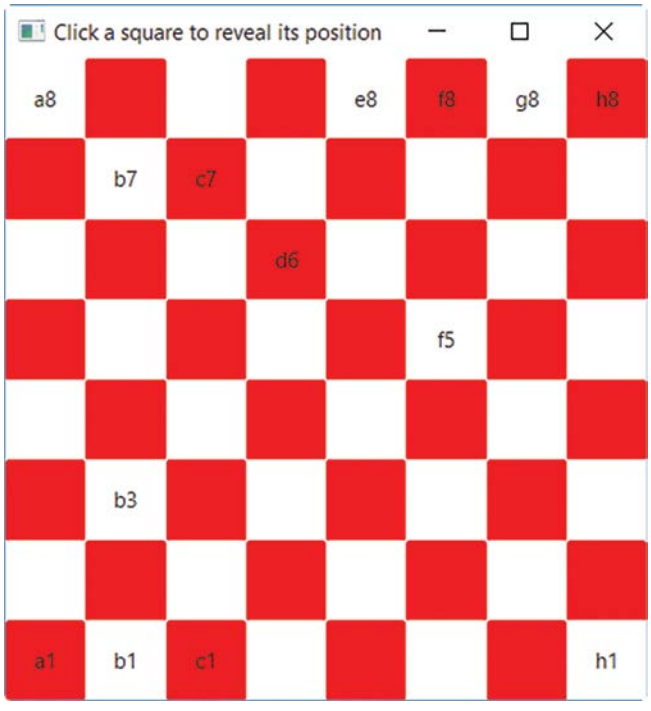
Questions 69, 70, 72, 74, 76, 77, 79, 81, 82, 84, 85

12.10 Building a GUI Programmatically

We have been defining GUIs using FXML, which works well when we know the number and type of controls in our GUI. Sometimes, however, the GUI is dynamic and because we may not know how many buttons or other controls we need until runtime, we must define the GUI, that is, instantiate the controls and set their properties, programmatically. For example, the GUI may include buttons that represent URL links stored in a file or on a web server. In that case, the number of buttons can vary each time we run our application. In other applications, it may be more convenient to build the GUI by code rather than with FXML. For example, if the GUI represents a tic-tac-toe game or a chessboard, it is convenient to represent the GUI as a two-dimensional array of buttons, which we define and instantiate programmatically.

We illustrate this approach with an example where we build a chessboard and reveal the position of a square as its column letter and row number when the user clicks on it. Figure 12.12 shows the application running after the user has clicked on several buttons.

FIGURE 12.12
Running the Chessboard Example



Before defining the View and the Controller, we define our Model, the *BoardGame* class, shown in Example 12.27. We define a board with an array of *chars* representing the column letters and an array of *ints* representing the row numbers. We declare two instance variables for these arrays at lines 12 and 13. We include an array of colors (line 14) so that clients of this class can use custom colors to color the board. The *getSquareText* method (lines 29–37) returns a *String* representation of a square on the board as the concatenation of the letter and digit at the row and column indexes requested by the two parameters.

The *getSquareColor* method (lines 39–49) returns a hexadecimal representation of the color of the square on the board given its row and column numbers, which are sent as parameters to the method. This method can be used to color the board with alternating colors retrieved from the array *colors*.

We provide accessors for the number of rows and the number of columns (lines 81–87 and 89–95).

```
1 /*  BoardGame class
2     Anderson, Franceschi
3 */
4
5 import javafx.scene.paint.Color;
6
7 public class BoardGame
8 {
9   public static char [ ] hexDigits =
10         { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
11           'A', 'B', 'C', 'D', 'E', 'F' };
12   private int [ ] rows;
13   private char [ ] columns;
14   private Color [ ] colors; // to color the rows and columns
15
16   /** Overloaded Constructor
17   *  @param newRows     the rows
18   *  @param newColumns  the columns
19   *  @param newColors   the colors
20   */
21   public BoardGame( int [ ] newRows, char [ ] newColumns,
22                     Color [ ] newColors )
23   {
24     rows = newRows;
25     columns = newColumns;
26     colors = newColors;
27   }
28
29   /** getSquareText method
30   *  @param row      the row index
31   *  @param column   the column index
32   *  @return a String, a concatenation of columns[column] and rows[row]
33   */
34   public String getSquareText( int row, int column )
35   {
36     return "" + columns[column] + rows[row];
37   }
38
39   /** getSquareColor method
40   *  @param row      the row index
41   *  @param column   the column index
42   *  @return a String, a hex representation of
```

```
43   *                    colors[(row + col) % colors.length]
44   */
45   public String getSquareColor( int row, int col )
46   {
47     Color squareColor = colors[(row + col) % colors.length];
48     return toHexString( squareColor );
49   }
50
51   /** toHex utility method
52   *  @param d   a double between 0 and 1 inclusive
53   *  @return a String, the hex representation of d times 255
54   */
55   public String toHex( double d )
56   {
57     int i = ( int ) ( d * 255 );
58     if ( i > 255 )
59       i = 255;
60     else if ( i < 0 )
61       i = 0;
62
63     int q = i / 16;
64     int r = i % 16;
65
66     return "" + hexDigits[q] + hexDigits[r];
67   }
68
69   /** toHexString method
70   *  @param color  a Color
71   *  @return a String, a hex representation of color
72   */
73   public String toHexString( Color color )
74   {
75     String colorText = "#" + toHex( color.getRed( ) )
76                            + toHex( color.getGreen( ) )
77                            + toHex( color.getBlue( ) );
78     return colorText;
79   }
80
81   /** getNumberOfRows method
82   *  @return the number of rows
83   */
84   public int getNumberOfRows( )
85   {
86     return rows.length;
87   }
88
```

```
89   /** getNumberOfColumns method
90   *  @return the number of columns
91   */
92   public int getNumberOfColumns( )
93   {
94     return columns.length;
95   }
96 }
```

Example 12.27 The *BoardGame* Class

Even though we are defining the GUI by code, it is possible to separate the View class from the Controller class, as in the previous examples. However, in order to keep this example simple, we put the View and the Controller in the same class.

Regardless of the layout container used, the constructor of our application needs to perform the following operations:

- Instantiate components
- Add components to the layout container

For this example, we use a *GridPane* layout container that allows us to arrange components in a grid. We can visualize a *GridPane* as a table made up of cells in rows and columns. Each cell can contain one component. These cells can have different sizes. In this application, as in many, however, each cell has the same size. Components are placed on the grid at a specified column and row using one of the *add* methods of the *GridPane* class, such as the one shown in Table 12.24. Note that the column parameter is given before the row parameter.

By default, the rows and columns in the grid are sized to their preferred size, which is based on the content of each cell and independent of the size of the *GridPane*. A *GridPane* maintains lists of row and column constraints that it uses to size each cell in the grid. There are no row or column constraints when a *GridPane* is first instantiated, so these two lists are originally empty. If we want to control the height of each row in the grid, we add as many row constraints as there are rows in the grid; similarly, if we want to control the width of each column in the grid, we add as many column constraints as there are columns in the grid. In this example, we want all the rows to have the same height and all the columns to have the same width. We also want all the cells to fill all the space in the *GridPane*.

The *getRowConstraints* and *getColumnConstraints* methods of the *GridPane* class, shown in Table 12.24, retrieve the list of row and column constraints, respectively.

Table 12.24 Useful Methods of the *GridPane* Class

Package	javax.scene.layout
Return value	**Method name and argument list**
`void`	`add( Node child, int columnIndex, int rowIndex )` adds *child* to this *GridPane* at row *rowIndex* and column *columnIndex*
`ObservableList<RowConstraints>`	`getRowConstraints( )` returns a list of row constraints for this *GridPane*
`ObservableList<ColumnConstraints>`	`getColumnConstraints( )` returns a list of column constraints for this *GridPane*

We use the *add* method of the *ObservableList* interface, inherited from the *List* interface, to add a row or a column constraint to the appropriate list. When we add a constraint to the list of row or column constraints, it is automatically added to the list of constraints in the *GridPane*. This method is shown in Table 12.25.

Table 12.25 The *add* and *clear* Methods of the *List* Interface

Package	java.util
Return value	**Method name and argument list**
boolean	`add( E e )` appends *e* to the end of the list
void	`clear( )` removes all the elements from the list

The *RowConstraints* and *ColumnConstraints* classes can be used to define the height of a row and width of a column in a *GridPane*, respectively. Often, we want to set the dimensions of a row or column as a percentage of the available space within the *GridPane*. We do this with the *setPercentHeight* and

setPercentWidth methods of the *RowConstraints* and *ColumnConstraints* classes. These methods are shown in Tables 12.26 and 12.27.

Table 12.26 The *setPercentHeight* Method of the *RowConstraints* Class

Package	javax.scene.layout
Return value	**Method name and argument list**
`void`	`setPercentHeight( double value )` sets the height of a row as a percentage of the total height of the *GridPane*

Table 12.27 The *setPercentWidth* Method of the *ColumnConstraints* Class

Package	javax.scene.layout
Return value	**Method name and argument list**
`void`	`setPercentWidth( double value )` sets the width of a column as a percentage of the total width of the *GridPane*

Example 12.28 shows how to define a GUI by code using a *GridPane* to display a chessboard. Each position on a chessboard is identified by a letter and a number. From the standpoint of the white player, the lower left square is a1, and from the standpoint of the black player, the lower left square is h8. When the user clicks on a square on the chessboard, our application displays its position.

The *BoardView* class, shown in Example 12.27, extends *GridPane*. In this way, we can set a *BoardView* object as the root of the scene that we place on the stage.

A two-dimensional array of *Buttons*, named *squares* (line 13), makes up the chessboard. We declare an instance variable of type *BoardGame,* our Model, at line 12. The constructor (lines 15–57) defines the GUI and sets up event handling. We call the constructor of *GridPane* at line 17 and retrieve the number of rows and columns from the Model at lines 20 through 22. At lines 23 through 30, we define and add row and column constraints. After instantiating a *RowConstraints* object at line 23, we set its percentage height to 100%, divided by the number of rows. We then add that row constraint to the list of row constraints as many times as there are rows (lines

27–28). In this way, each row has the same height. We define the column constraints similarly, so that each column has the same width.

At line 33, we declare and instantiate the listener *bh*, a *ButtonHandler* object reference. Because we are interested in events related to buttons, our *ButtonHandler private* inner class implements the *EventHandler* interface and overrides the *handle* method. At line 32, we instantiate the two-dimensional array *squares*. We have a two-dimensional array of *Buttons*, so we use nested *for* loops at lines 35 through 56 to instantiate the *Buttons*, add them to the *GridPane*, and register the listener on all the buttons. Using FXML, to include a *Button* in a layout container, we define a *Button* element and specify attributes for that *Button* element in an *xml* file. In order to create and define a *Button* programmatically, we instantiate the *Button* using a constructor and call various methods to set its properties. These setter methods typically accept a parameter that represents the new value for the *Button* property. Some of these methods are shown in Table 12.28. Remember from Figure 12.2 that the *Button* class is a subclass of *ButtonBase*, *Labeled*, *Control*, *Region*, *Parent*, *Node*, and *Object*, and inherits some properties from its superclasses.

At lines 42 through 44, we color the squares according to the coloring pattern and colors set in the Model. In order to do this, we call the *setStyle* method of the *Button* class, inherited from *Node*, and also shown in Table 12.28. The *setStyle* method accepts a CSS-like *String* parameter composed of a semicolon-separated list of attribute-value pairs, with each attribute and value separated by a colon (:). The format of the CSS-style *String* parameter is as follows:

```
"styleAttribute1:value1;styleAttribute2:value2;styleAttribute3:value3; ..."
```

For example, if we want to set the background color to blue and the text's font size to 25, we could use the following *String* as the parameter to the *setStyle* method:

```
"-fx-background-color:blue;-fx-font-size:25"
```

REFERENCE POINT

For a complete listing of JavaFX style attributes, see https://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html

At lines 49 through 51, we ensure that each *Button* will fill the available space within the *GridPane* by setting its maximum width and height to the maximum possible value by calling the *setMaxWidth* and *setMaxHeight* methods (see Table 12.28), using the *static* constant of the *Double* wrapper class (*Double.MAX_VALUE*) that represents the maximum value that a *double* data type can hold. If we remove these statements, the *Buttons* will be sized to their default size, which is based on their contents: in this case,

the button's text and the font size of that text. At lines 53 and 54, we call the *setOnAction* method (see Table 12.28) to register the *ButtonHandler bh* on each *Button*.

Table 12.28 Selected Methods of and Inherited by the *Button* Class

Package	javax.scene.control
Constructor	
`Button( )` creates an empty *Button*	
Return value	**Method name and argument list**
`void`	`setStyle( String style )` sets the value of the *style* property of this *Button* to *style*, which is expected to be a CSS-like *String*. This method is inherited from *Node*.
`void`	`setMaxWidth( double value )` sets the value of the *maxWidth* property of this *Button* to *value*. This method is inherited from *Region*.
`void`	`setMaxHeight( double value )` sets the value of the *maxHeight* property of this *Button* to *value*. This method is inherited from *Region*.
`void`	`setOnAction( EventHandler<ActionEvent> handler )` registers *handler* as the event handler for this *Button*. When the button is clicked, the *handle* method of handler's class will be called automatically. This method is inherited from *ButtonBase*.

In the *handle* method, we use nested *for* loops at lines 65 through 75 to identify the source of the event, that is, which button the user clicked. We then set the text of that button to its board position (line 71), retrieving the corresponding value from the Model. Having found the source of the event, we then exit the event handler via the *return* statement (line 72) to interrupt the *for* loops and thus avoid unnecessary processing.

```
1 /*  Using GridPane to organize our window
2     Anderson, Franceschi
3 */
4
5 import javafx.event.ActionEvent;
```

```
import javafx.event.EventHandler;
import javafx.scene.control.Button;
import javafx.scene.layout.*;

public class BoardView extends GridPane
{
  private BoardGame game;
  private Button [ ][ ] squares;

  public BoardView( BoardGame newGame )
  {
    super( );
    game = newGame;

    // set up grid according to Model
    int rows = game.getNumberOfRows( );
    int columns = game.getNumberOfColumns( );
    RowConstraints row = new RowConstraints( );
    row.setPercentHeight( 100.0 / rows );
    ColumnConstraints col = new ColumnConstraints( );
    col.setPercentWidth( 100.0 / columns );
    for ( int i = 0; i < rows; i++ )
      getRowConstraints( ).add( row );
    for ( int j = 0; j < columns; j++ )
      getColumnConstraints( ).add( col );

    squares = new Button[rows][columns];
    ButtonHandler bh = new ButtonHandler( );

    for ( int i = 0; i < rows; i++ )
    {
      for ( int j = 0; j < columns; j++ )
      {
        // instantiate the button
        squares[i][j] = new Button( );

        // color the button
        squares[i][j].setStyle( "-fx-background-color:"
          + game.getSquareColor( i, j ) );

        // add the button
        add( squares[i][j], j, i );

        // make button fill up available width and height
```

```
50          squares[i][j].setMaxWidth( Double.MAX_VALUE );
51          squares[i][j].setMaxHeight( Double.MAX_VALUE );
52
53          // register listener on button
54          squares[i][j].setOnAction( bh );
55        }
56      }
57    }
58
59    // private inner class event handler
60    private class ButtonHandler implements EventHandler<ActionEvent>
61    {
62      @Override
63      public void handle( ActionEvent event )
64      {
65        for ( int i = 0; i < squares.length; i++ )
66        {
67          for ( int j = 0; j < squares[i].length; j++ )
68          {
69            if ( event.getSource( ) == squares[i][j] )
70            {
71              squares[i][j].setText( game.getSquareText( i, j ) );
72              return;
73            }
74          }
75        }
76      }
77    }
78  }
```

Example 12.28 The *BoardView* Class

Finally, the *ChessBoard* class, shown in Example 12.29, includes the *main* and *start* methods to create a *BoardView* showing a *BoardGame* game. The *letters*, *digits*, and *boardColors* arrays (lines 14–16) define the board. Try to change either the *letters*, *digits*, or *boardColors* array and run the application again. The grid is resized correctly and with new colors, showing the reusability of our classes.

```
1 /*  ChessBoard class
2     Anderson, Franceschi
3 */
4
```

```
 5 import javafx.application.Application;
 6 import javafx.scene.paint.Color;
 7 import javafx.scene.Scene;
 8 import javafx.stage.Stage;
 9
10 public class ChessBoard extends Application
11 {
12   public void start( Stage stage )
13   {
14     char [ ] letters = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h' };
15     int [ ] digits = { 8, 7, 6, 5, 4, 3, 2, 1 };
16     Color [ ] boardColors = { Color.WHITE, Color.RED };
17
18     BoardGame game = new BoardGame( digits, letters, boardColors );
19     BoardView view = new BoardView( game );
20
21     Scene scene = new Scene( view, 450, 450 );
22     stage.setScene( scene );
23     stage.setTitle( "Click a square to reveal its position" );
24     stage.show( );
25   }
26
27   public static void main( String [ ] args )
28   {
29     launch( args );
30   }
31 }
```

Example 12.29 The *ChessBoard* Class

Figure 12.13 shows the application if we eliminate the row and column constraints by commenting out lines 23 through 30 of Example 12.27. Similarly, we will have the same result if we comment out lines 49 through 51, because the buttons will have their computed preferred size, which is based on their content. The cells do not fill up the width and height of the *GridPane*; the buttons have their default minimum size when they are empty. When we click on a button, it expands in width to display its new text, and the width of the whole column increases accordingly.

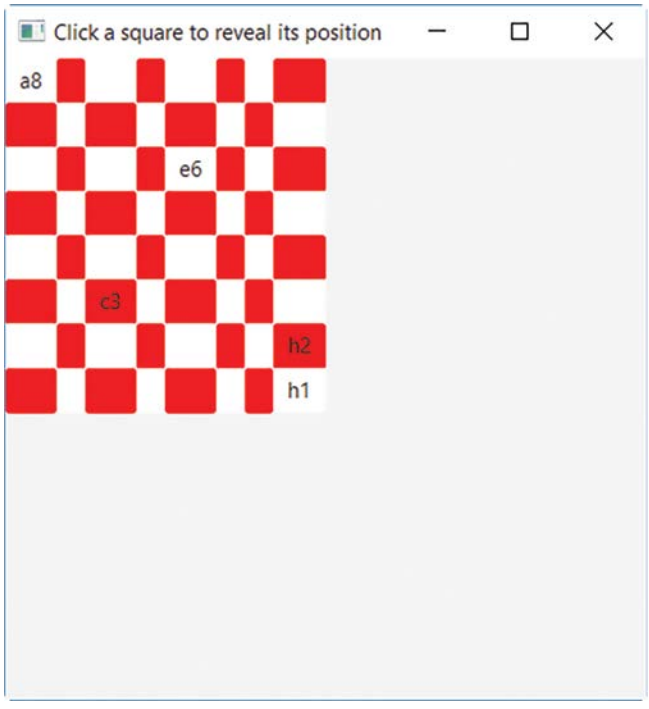


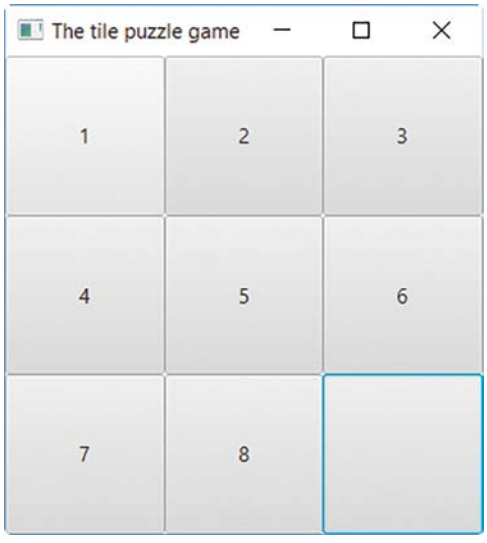
FIGURE 12.13
Running the Chessboard Example Without Row and Column Constraints

12.11 Layout Containers: Dynamically Setting Up the GUI Using *GridPane*

Layout managers can be set dynamically, based on user input. For example, the user could enter the number of rows or columns of the grid. Based on user input, we can also rearrange the components using another layout container, such as *HBox*. The user could also instruct us to remove components and add others. Our next example, the Tile Puzzle game, will illustrate some of these capabilities.

In the Tile Puzzle game, eight tiles displaying the digits 1 through 8 are scrambled on a 3-by-3 grid, leaving one cell empty. Any tile adjacent to the empty cell can be moved to the empty cell by clicking on the numbered tile. The goal is to rearrange the tiles so that the numbers are in the correct order, as shown in Figure 12.14.

FIGURE 12.14
The Winning Position of a 3-by-3 Tile Puzzle Game



The Tile Puzzle game can also be played on a 4-by-4, 5-by-5, and more generally, *n*-by-*n* grid. In this example, we set up a 3-by-3 grid for the first game and then randomly select a 3-by-3, 4-by-4, 5-by-5, or 6-by-6 grid for subsequent games. Later, we can modify this example to allow the user to specify the size of the grid.

Before designing the GUI class, we first code the Model, a class to encapsulate the functionality of the tile puzzle game; the *TilePuzzle* class (Example 12.30) handles the creation of a game of a given size, enables play, and enforces the rules of the game.

The instance variable *tiles* (line 7), a two-dimensional array of *Strings*, stores the state of the puzzle. Each element of *tiles* represents a cell in the puzzle grid. The instance variable *side*, declared at line 8, represents the size of the grid. The instance variables *emptyRow* and *emptyCol*, declared at lines 9 and 10, identify the empty cell in the puzzle grid.

The constructor (lines 12–18) calls the *setUpGame* method (lines 20–44), passing the size of the grid (*newSide*) as an argument. We also call the *setUpGame* method before starting each new game in the GUI class.

Inside the *setUpGame* method, we assign *newSide* to *side*. Rather than randomly generating each tile label, which would complicate this example, we assign the labels to the tiles in descending order using nested *for* loops (lines 33–41). We set the empty cell to the last cell in the grid at lines 42 and 43.

The *tryToPlay* method (lines 71–89) first checks if the play is legal by calling the *possibleToPlay* method (line 77). The play is legal if the tile the user clicked is next to the empty tile. If the *possibleToPlay* method returns *true*, we proceed with the play and return *true*; otherwise, we return *false*. Playing means swapping the values of the empty cell (*emptyRow*, *emptyCol*) and the cell that was just played, represented by the two parameters of the *tryToPlay* method, *row* and *col*.

The *possibleToPlay* method is coded at lines 91 through 101. If the play is legal—that is, if the tile played is within one cell of the empty cell—the method returns *true*; otherwise, the method returns *false*.

The *won* method (lines 103–119) checks if the tiles are in order. If so, the *won* method returns *true*; otherwise, the method returns *false*.

```
1 /** TilePuzzle class
2 *   Anderson, Franceschi
3 */
4
5 public class TilePuzzle
6 {
7  private String [ ][ ] tiles;
8  private int side; // grid size
9  private int emptyRow;
10  private int emptyCol;
11
12  /** constructor
13  * @param newSide grid size
14  */
15  public TilePuzzle( int newSide )
16  {
17   setUpGame( newSide );
18  }
19
20  /** setUpGame
21  * @param newSide grid size
22  */
23  public void setUpGame( int newSide )
24  {
25   if ( newSide < 3 )
26    side = 3;
27   else
28    side = newSide;
29   emptyRow = side - 1;
30   emptyCol = side - 1;
```

```
31   tiles = new String[side][side];
32
33   // initialize tiles
34   for ( int i = 0; i < side; i++ )
35   {
36    for ( int j = 0; j < side; j++ )
37    {
38     tiles[i][j] = String.valueOf( ( side * side )
39                                   - ( side * i + j + 1 ) );
40    }
41   }
42   // set empty cell label to blank
43   tiles[side - 1][side - 1] = "";
44  }
45
46  /** getSide
47  * @return side
48  */
49  public int getSide( )
50  {
51    return side;
52  }
53
54  /** getTiles
55  * @return a copy of tiles
56  */
57  public String[ ][ ] getTiles( )
58  {
59   String[ ][ ] copyOfTiles = new String[side][side];
60
61   for ( int i = 0; i < side; i++ )
62   {
63    for ( int j = 0; j < side; j++ )
64    {
65     copyOfTiles[i][j] = tiles[i][j];
66    }
67   }
68   return copyOfTiles;
69  }
70
71  /** tryToPlay
72  * enable play if play is legal
73  * @return true if the play is legal, false otherwise
74  */
75  public boolean tryToPlay( int row, int col )
76  {
```

```
 77   if ( possibleToPlay( row, col ) )
 78   {
 79    // play: switch empty String and tile label at row, col
 80    tiles[emptyRow][emptyCol] = tiles[row][col];
 81    tiles[row][col] = "";
 82    // update emptyRow and emptyCol
 83    emptyRow = row;
 84    emptyCol = col;
 85    return true;
 86   }
 87   else
 88    return false;
 89  }
 90
 91  /** possibleToPlay
 92  * @return true if the play is legal, false otherwise
 93  */
 94  public boolean possibleToPlay( int row, int col )
 95  {
 96   if ( ( col == emptyCol && Math.abs( row - emptyRow ) == 1 )
 97          || ( row == emptyRow && Math.abs( col - emptyCol ) == 1 ) )
 98    return true;
 99   else
100    return false;
101  }
102
103  /** won
104  * @return true if correct tile order, false otherwise
105  */
106  public boolean won( )
107  {
108   for ( int i = 0; i < side ; i++ )
109   {
110    for ( int j = 0; j < side; j++ )
111    {
112     if ( !( tiles[i][j].equals(
113             String.valueOf( i * side + j + 1 ) ) )
114         && ( i != side - 1 || j != side - 1 ) )
115      return false;
116    }
117   }
118   return true;
119  }
120 }
```

Example 12.30 The *TilePuzzle* Class

As in the chessboard example, when the user clicks on a button, we need to identify the row and the column of that button. In order to avoid looping through all the buttons, we create a class that *extends* the *Button* class and we add two instance variables for the row and column indexes. In this way, when a button belonging to a two-dimensional array of buttons is clicked, we can access its *row* and *column* instance variables in order to identify which button was clicked.

Example 12.31 shows the *GridButton* class, which extends the *Button* class and adds two instance variables, *row* and *column* (lines 9–10). The constructor at lines 12 through 22 allows the client to create a *GridButton* object with a specified text, row, and column. Note that this class is not specific to the tile puzzle application and can be reused for other applications involving an array of buttons placed on a grid.

```
1 /** GridButton class
2 *   Anderson, Franceschi
3 */
4
5 import javafx.scene.control.Button;
6
7 public class GridButton extends Button
8 {
9  private int row;
10 private int column;
11
12 /** Constructor
13 * @param title text for button
14 * @param newRow row
15 * @param newColumn column
16 */
17 public GridButton( String title, int newRow, int newColumn )
18 {
19  super( title );
20  setRow( newRow );
21  setColumn( newColumn );
22 }
23
24 /** getRow method, accessor for row
25 * @return row
26 */
27 public int getRow( )
28 {
29   return row;
30 }
```

```
31
32  /** getColumn method, accessor for column
33  * @return column
34  */
35  public int getColumn( )
36  {
37    return column;
38  }
39
40  /** setRow method, mutator for row
41  * @param newRow, an int
42  */
43  public void setRow( int newRow )
44  {
45    if ( newRow >= 0 )
46      row = newRow;
47  }
48
49  /** setColumn method
50  * @param newColumn, an int
51  */
52  public void setColumn( int newColumn )
53  {
54    if ( newColumn >= 0 )
55      column = newColumn;
56  }
57 }
```

Example 12.31 The *GridButton* Class

Figure 12.15 shows the UML diagram for the *TilePuzzle* and *TilePuzzleViewController* classes.

Example 12.32 shows the *TilePuzzleViewController* class; as in the *BoardView* class in the previous example, it is a subclass of *GridPane* (line 12) and combines the View and the Controller.

Each tile in the game is a *GridButton*. The instance variable *squares* (line 14) hold a two-dimensional array of *GridButtons* so that each element of *squares* is a cell in the game grid. The *TilePuzzle* instance variable *game*, declared at line 15, represents the Model. When the user plays, we call the various methods of the *TilePuzzle* class to enforce the rules of the game.

The constructor (lines 17–22) calls the constructor of *GridPane*, instantiates *game*, and calls the *setUpGameGUI* method. The constructor and the

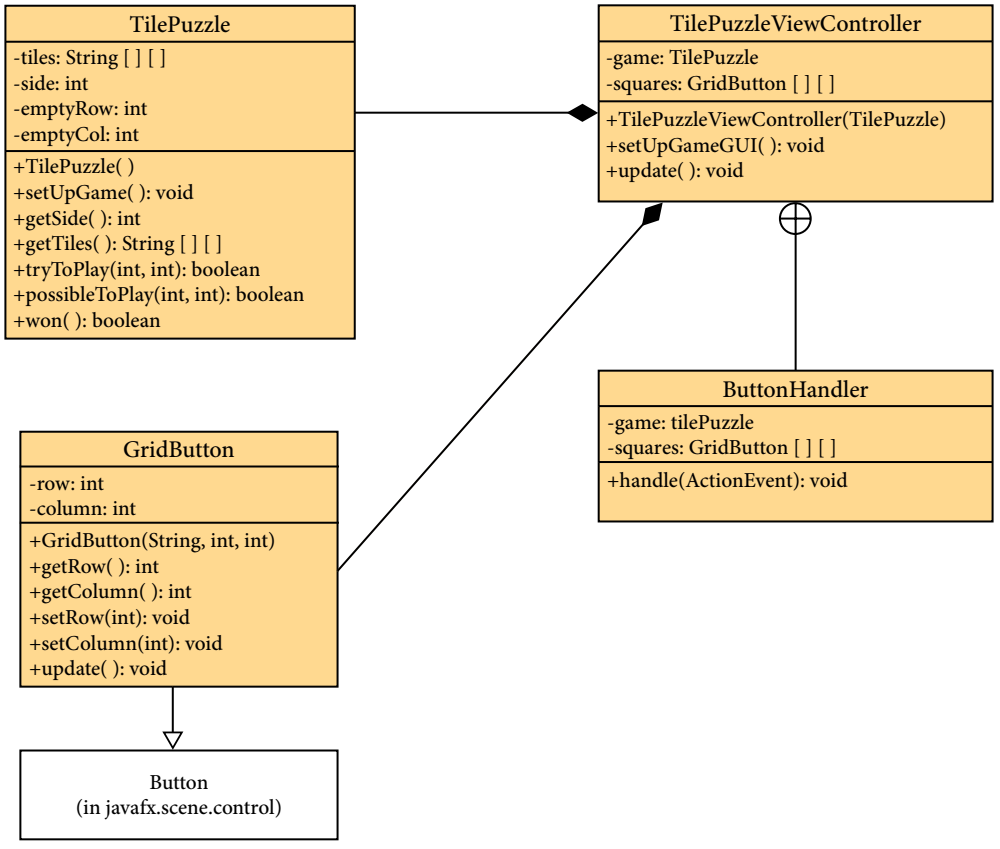


FIGURE 12.15

The UML Diagram for *TilePuzzle* and *TilePuzzleViewController*

setUpGameGUI method define the View. The *private* class *ButtonHandler* (lines 80–90) and the *update* method (lines 63–78) make up the Controller.

The *setUpGameGUI* method (lines 24–61) displays the game in its starting position. We first remove all the components and the row and column constraints from this *GridPane* (lines 26–29). In order to access the children of this *GridPane*, we call the *getChildren* method inherited from *Pane* and shown in Table 12.29. It returns an *ObservableList* of *Nodes* that are the children of this *GridPane*. With it, we call the *clear* method that *ObservableList* inherits from the *List* interface. That removes all the *Buttons* from this *GridPane*. Because the next game may be 3-by-3, 4-by-4, or 5-by-5 grids, we still have to remove the current row and column constraints of this *GridPane*. We access them using the *getRowConstraints* and

getColumnConstraints methods of the *GridPane* class, shown earlier in Table 12.24. They also return a list of *Observable* objects, *RowConstraints,* and *ColumnConstraints*, respectively. We remove these constraints by calling the *clear* method, shown in Table 12.25. We then reset the row and column constraints of this *GridPane* layout container at lines 31 through 39. We instantiate the *squares* array and our event handler at lines 41 and 42. After that, we use nested *for* loops (lines 44–60) to instantiate each button, add it to the container, force the button to fill the available width and height, and register the event handler. The *squares* array parallels the *tiles* array of the *TilePuzzle* class; the *squares* buttons are labeled with the values of *tiles*.

Table 12.29 The *getChildren* Method of the *GridPane* Class Inherited from the *Pane* Class

Package	javax.scene.layout
Return value	**Method name and argument list**
`ObservableList<Node>`	`getChildren( )` returns the children of this pane as an *ObservableList* of *Nodes*

When a button is clicked, our *ButtonHandler handle* method is executed. To determine which button was clicked, we call the *getSource* method using the *ActionEvent* event object that was sent to the method (line 86). Because the *getSource* method returns an *Object*, we need to typecast the return value to a *GridButton*. We then need to determine whether the button clicked was a legal play. To do this, we call the *tryToPlay* method at line 87 with the *row*-and-*column* instance variable values of the button. If that method returns *true*, the play was legal and the model has changed (i.e., a tile has been moved). We then call the *update* method to make the view (i.e., the buttons) reflect the model (line 88).

The *update* method first updates the *squares* button array (lines 65–67). We then test if the current move solved the puzzle by calling the *won* method (line 69) of the *TilePuzzle* class. If the *won* method returns *true*, we congratulate the user by popping up a dialog box at lines 71 and 72. Note that we use the *JOptionPane* class of the *javax.swing* package; at the time of this edition, there is no equivalent class in the JavaFX classes.

We then randomly generate a grid size between three and six (lines 73–74) for the next game and call the *setUpGame* method (line 75) and the *setUpGameGUI* method (line 76) to begin a new game with the new grid size.

```
1 /*  Using GridPane dynamically
2     Anderson, Franceschi
3 */
4
5 import java.util.Random;
6 import javafx.event.ActionEvent;
7 import javafx.event.EventHandler;
8 import javafx.scene.control.Button;
9 import javafx.scene.layout.*;
10 import javax.swing.JOptionPane;
11
12 public class TilePuzzleViewController extends GridPane
13 {
14   private GridButton [ ][ ] squares;
15   private TilePuzzle game; // the tile puzzle game
16
17   public TilePuzzleViewController( TilePuzzle newGame )
18   {
19     super( );
20     game = newGame;
21     setUpGameGUI( );
22   }
23
24   public void setUpGameGUI( )
25   {
26     // remove all components and constraints
27     getChildren( ).clear( );
28     getRowConstraints( ).clear( );
29     getColumnConstraints( ).clear( );
30
31     // set up grid constraints
32     RowConstraints row = new RowConstraints( );
33     row.setPercentHeight( 100.0 / game.getSide( ) );
34     ColumnConstraints col = new ColumnConstraints( );
35     col.setPercentWidth( 100.0 / game.getSide( )  );
36     for ( int i = 0; i < game.getSide( ); i++ )
37       getRowConstraints( ).add( row );
38     for ( int j = 0; j < game.getSide( ); j++ )
39       getColumnConstraints( ).addAll( col );
40
41     squares = new GridButton [game.getSide( )][game.getSide( )];
42     ButtonHandler bh = new ButtonHandler( );
43
44     for ( int i = 0; i < game.getSide( ); i++ )
45     {
46       for ( int j = 0; j < game.getSide( ); j++ )
```

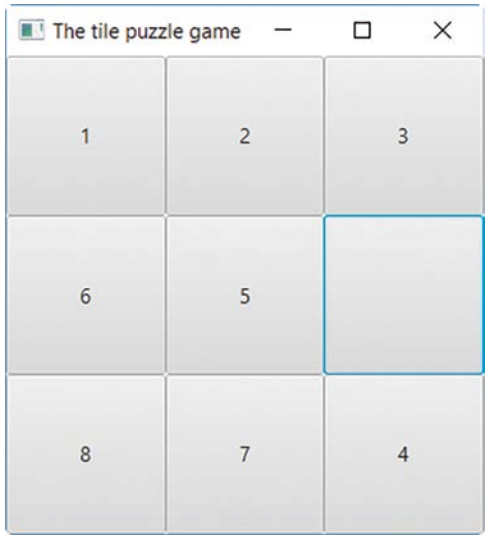
```
47        {
48          squares[i][j] = new GridButton( game.getTiles( )[i][j], i, j );
49
50          // add the button
51          add( squares[i][j], j, i );
52
53          // make button fill up available width and height
54          squares[i][j].setMaxWidth( Double.MAX_VALUE );
55          squares[i][j].setMaxHeight( Double.MAX_VALUE );
56
57          // register listener on button
58          squares[i][j].setOnAction( bh );
59        }
60      }
61    }
62
63    public void update( )
64    {
65      for ( int i = 0; i < game.getSide( ); i++ )
66        for ( int j = 0; j < game.getSide( ); j++ )
67          squares[i][j].setText( game.getTiles( )[i][j] );
68
69      if ( game.won( ) )
70      {
71        JOptionPane.showMessageDialog( null,
72           "Congratulations! You won!\nSetting up new game" );
73        Random random = new Random( );
74        int sideOfPuzzle = 3 + random.nextInt( 4 );
75        game.setUpGame( sideOfPuzzle );
76        setUpGameGUI( );
77      }
78    }
79
80    // private inner class event handler
81    private class ButtonHandler implements EventHandler<ActionEvent>
82    {
83      @Override
84      public void handle( ActionEvent event )
85      {
86        GridButton button = ( GridButton ) event.getSource( );
87        if ( game.tryToPlay( button.getRow( ), button.getColumn( ) ) )
88          update( );
89      }
90    }
91  }
```

Example 12.32 The *TilePuzzleViewController* Class

Finally, the *PlayTilePuzzle* class, shown in Example 12.33, includes the *main* and *start* methods to create a *TilePuzzleViewController* application showing a *TilePuzzle* game. Figure 12.16 shows a run of this game after the user has moved some tiles.

```
/*  PlayTilePuzzle class
    Anderson, Franceschi
*/

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

public class PlayTilePuzzle extends Application
{

  @Override
  public void start( Stage stage )
  {
    TilePuzzle puzzle= new TilePuzzle( 3 );
    TilePuzzleViewController root
      = new TilePuzzleViewController( puzzle );

    Scene scene = new Scene( root, 350, 350 );
    stage.setTitle( "The tile puzzle game" );
    stage.setScene( scene );
```

FIGURE 12.16
The Tile Puzzle Game in Progress



```
22      stage.show( );
23    }
24
25    public static void main( String [ ] args )
26    {
27      launch( args );
28    }
29  }
```

Example 12.33 The *PlayTilePuzzle* Class

12.12 *BorderPane* Layout and Writing Event Handlers Using Lambda Expressions

A *BorderPane* layout container organizes its nodes into five positions: *top*, *bottom*, *left*, *right*, and *center*, with each position holding, at most, one node. The size of each position expands or contracts depending on the size of the node in that position, the sizes of the nodes in the other positions, and whether the other positions contain a node. The *center* position expands to fill any remaining space; if not enough total layout space is allocated, some positions may overlap other positions. Thus, for each position, you can add zero or one node. In contrast to the previously discussed layout containers, the order in which we define the positions and nodes for a *BorderPane* is not important. Figure 12.17 shows the positions in a sample *BorderPane* layout.

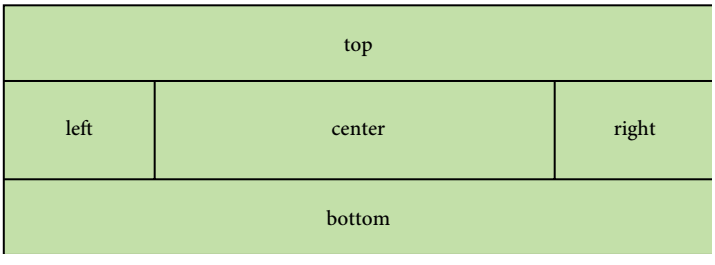


FIGURE 12.17
The Positions of a *BorderPane* Layout Container

To illustrate a *BorderPane,* we create an example that allows the user to mix red, green, and blue color components to create a new color. We place *Buttons* in the nodes in the *left*, *bottom*, and *right* positions to represent the colors red, green, and blue. We place a *Label* in the node in the *center* position to display the current color mix as a hexadecimal number, and we color the background of the node to correspond with the color created by the

current red, green, and blue components. Color component values range between 00 and FF (hexadecimal). When the user clicks on one of the buttons, we change the corresponding amount of that color component by a specified increment. For example, let's assume that the value of the red color component is A0 and the incremental value is 10 (decimal). If the user clicks on the *RED* button, the value of the red color component will become AA.

In the node in the *top* position, we place a *Button* labeled "CHANGE DIRECTION" that will make the incremental amount toggle between a positive and a negative value. For example, if the incremental amount is 10 and the user clicks the CHANGE DIRECTION button, the incremental amount becomes –10.

Figure 12.18 shows the GUI when the application begins. We start with a white background color in the *center* position, with the hexadecimal equivalent of FFFFFF.

FIGURE 12.18
Using a *BorderPane* Layout

Example 12.34 shows our Model, *RGBColorMixer.java*. Its job is to manage the current color values and to toggle the delta value between negative and positive. We store the color values as *ints* between 0 and 255 decimal (line 12). In the constructor (lines 14–26), we initialize the delta value, sent as a parameter, assigning a default value of 1 if the parameter value is 0 or outside the range of –255 to +255. If the delta is negative, we set *red*, *green*, and *blue* to the maximum value, 255. Otherwise, *red*, *green*, and *blue* are auto-initialized to 0.

For each color component, we include a similar method *changeRed* (lines 38–50), *changeGreen* (lines 52–63), and *changeBlue* (lines 65–76). The methods ensure that adding the delta to the current value will not exceed 255 or become negative.

In the *invertDelta* method (lines 28–36), we simply toggle the delta between its negative and positive values.

The *toHex* utility method (lines 78–92) converts the *int* value of a color component to a *String* containing two hex characters, obtained using indexes into the *static char* array *hexDigits* (lines 7–9). We call this method from the *toHexString* method (lines 94–103) to compose a *String* containing six hex digits preceded by a # that can be used as the value for a style attribute.

```
1 /** RGBColorMixer class
2 *   Anderson, Franceschi
3 */
4
5 public class RGBColorMixer
6 {
7  public static char [ ] hexDigits =
8              { '0', '1', '2', '3', '4', '5', '6', '7',
9                '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };
10
11  private int delta; // color increment
12  private int red, green, blue;
13
14  /** Constructor
15  * @param newDelta delta value to add to or subtract
16  *        from the color
17  */
18  public RGBColorMixer( int newDelta )
19  {
20    if ( newDelta < -255 || newDelta > 255 || newDelta == 0 )
21      delta = 1;
22    else
23      delta = newDelta;
24    if ( delta < 0 )
25      red = green = blue = 255;
26  }
27
28  /** invertDelta method
29  * sets delta to its opposite
30  * if positive, delta becomes negative
31  * if negative, delta becomes positive
```

```
32  */
33  public void invertDelta( )
34  {
35    delta = -delta;
36  }
37
38  /** changeRed method
39  *   add delta to red
40  */
41  public void changeRed( )
42  {
43
44   if ( red + delta > 255 )
45     red = 255;
46   else if ( red + delta < 0 )
47     red = 0;
48   else
49     red = red + delta;
50  }
51
52  /** changeGreen method
53  * add delta to green
54  */
55  public void changeGreen( )
56  {
57   if ( green + delta > 255 )
58     green = 255;
59   else if ( green + delta < 0 )
60     green = 0;
61   else
62     green = green + delta;
63  }
64
65  /** changeBlue method
66  * add delta to blue
67  */
68  public void changeBlue( )
69  {
70   if ( blue + delta > 255 )
71     blue = 255;
72   else if ( blue + delta < 0 )
73     blue = 0;
74   else
75     blue = blue + delta;
76  }
```

```
77
78    /** toHex method
79    * @return a String, the Hex equivalent of i
80    */
81    public String toHex( int i )
82    {
83      if ( i > 255 )
84        i = 255;
85      else if ( i < 0 )
86        i = 0;
87
88      int q = i / 16;
89      int r = i % 16;
90
91      return "" + hexDigits[q] + hexDigits[r];
92    }
93
94    /** toHexString method
95    * @return a Hex String representation of color
96    */
97    public String toHexString( )
98    {
99      String colorText = "#" + toHex( red )
100                             + toHex( green )
101                             + toHex( blue );
102     return colorText;
103   }
104 }
```

Example 12.34 The Model, *RGBColorMixer.java*

Because we know the number and type of GUI components for our application, we can define the GUI in an FXML file, shown in Example 12.35. The *root* node is a *BorderPane* (lines 6–7). We use FXML elements to set up the five positions, into each of which we can insert, at most, one node. If we directly insert a *Button* into a position, the *Button* is aligned to the left and top corner of the position. Unfortunately, we cannot define an *alignment* property for the *BorderPane* position elements. Our answer, then, is to insert an *HBox* as the single node in each position and to set its *alignment* property to "center." We then use nesting of nodes to insert a *Button* into the *HBoxes* in the *top*, *bottom*, *left*, and *right* positions, labeled "CHANGE DIRECTION," "GREEN," "RED," and "BLUE," respectively (lines 8–27). For the *center* position, we insert a *Label* inside the *HBox* for displaying the

current color value. We also define an *fx:id* for this *HBox* because we need to access it in the controller in order to set its background color to reflect the current color selection (lines 28–32).

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2
 3 <?import javafx.scene.control.*?>
 4 <?import javafx.scene.layout.*?>
 5
 6 <BorderPane fx:controller="RGBColorMixerController"
 7             xmlns:fx="http://javafx.com/fxml" >
 8   <top>
 9     <HBox alignment="center">
10       <Button fx:id="change" text="CHANGE DIRECTION" />
11     </HBox>
12   </top>
13   <bottom>
14     <HBox alignment="center">
15       <Button fx:id="green" text="GREEN" />
16     </HBox>
17   </bottom>
18   <left>
19     <HBox alignment="center">
20       <Button fx:id="red" text="RED" />
21     </HBox>
22   </left>
23   <right>
24     <HBox alignment="center">
25       <Button fx:id="blue" text="BLUE" />
26     </HBox>
27   </right>
28   <center>
29     <HBox fx:id="hBoxCenter" alignment="center">
30       <Label fx:id="label" />
31     </HBox>
32   </center>
33 </BorderPane>
```

Example 12.35 The *fxml_mix_colors.xml* File

Note that we do not define an *onAction* property for the *Buttons i*n the FXML file. Instead, we define the event handlers in the Controller, shown in Example 12.36. We do this to demonstrate how to use anonymous objects and lambda expressions.

In the constructor (lines 14–18), we instantiate an instance of the Model with a delta of −10. Then, in the *initialize* method, we set the background color of the *center* position's *HBox* and text of the *Label* by calling the *setCenterStyle* method (line 23). The *setCenterStyle* method (lines 48–59) calls the *toHexString* and *setStyle* methods in the Model. Because we set the delta to a negative value, the Model sets the red, green, and blue color components to 255, making the initial color white. Thus, when the application begins, the center background is white and the label displays "#FFFFFF" as shown in Figure 12.18. Note that we retrieve the initial values from the Model rather than hard code the background color and label text inside the View.

In the examples in this chapter, we have defined event handlers in the Controller as *protected* methods that are specified in the *onAction* property for a node, and also as *private* inner classes. When an event handler is dedicated to only one control, another option is to define the event handler as an **anonymous class**, that is, a class that we define inline without giving the class a name. Using this option, we define and register the event handler at the same time. Let's explore this option with the event handler for the CHANGE DIRECTION Button. The only action of our event handler is to call the Model's *invertDelta* method.

We can register our event handler using the *setOnAction* method of the *Button* class (inherited from the *ButtonBase* class). To do this, we define a class that implements the *EventHandler<ActionEvent>* interface. This class needs to define one method, *handle*, shown in Table 12.30.

Table 12.30 The *handle* Method of the *EventHandler<T extends Event>* Interface

Package	javafx.event
Return value	**Method name and argument list**
`void`	`handle( T event )`

Thus, we could register and define our event handler as an anonymous class that implements the *EventHandler* interface using this code:

```
change.setOnAction( new EventHandler<ActionEvent>( )
                    {
                      @Override
                      public void handle( ActionEvent event )
                      {
                        mixer.invertDelta( );
                      }
                    }
                  );
```

Thus, the argument we send to the *setOnAction* method is an object of the class that implements the *EventHandler<ActionEvent>* interface. We instantiate the object using *new* and define the class by providing the body to the *handle* method inline. Although this is compact code, misplacing a curly brace, parenthesis, or semicolon can generate errors that are difficult to debug. To write this code in a more readable form, we can use **lambda expressions**, introduced in Java Version 8.

Lambda expressions can be used only with **functional interfaces**, which are interfaces that require the class implementing that interface to define only one method.

A lambda expression contains the following elements:

- A comma-separated list of parameters enclosed in parentheses. The data types of the parameters may be omitted. The parentheses may also be omitted if there is only one parameter.
- The arrow token, ->.
- A method body, which can be a single expression or a block enclosed in curly braces.
  - If the body of the method consists of a single expression, then the JVM evaluates the expression and returns its value. As an alternative, you can use a *return* statement, but that requires curly braces.

Given this syntax, we can replace the code above with:

```
change.setOnAction( event -> mixer.invertDelta( ) );
```

In fact, this is line 26 in Example 12.36. If the user clicks on the CHANGE DIRECTION Button, we execute the Model's *invertDelta* method.

We also use lambda expressions at lines 28 through 46 to define and register the event handlers for the *RED*, *GREEN*, and *BLUE* Buttons. Each event handler calls the appropriate method to change the color component, and then calls *setCenterStyle* to set the background color of the center *HBox* and display the current color value in the *Label*. Because these event handlers consist of more than one statement, we need to place the two statements inside curly braces. Otherwise, the approach is the same. We gain several advantages by using lambda expressions. The code is even more compact than defining an anonymous class; the code is more readable; and, by defining a separate event handler for each color button, we avoid calling the *getSource* method to determine which button the user clicked.

```
1 import javafx.event.*;
2 import javafx.fxml.FXML;
3 import javafx.scene.control.*;
4 import javafx.scene.layout.*;
5
6 public class RGBColorMixerController
7 {
8   private RGBColorMixer mixer;
9
10  @FXML private HBox hBoxCenter;
11  @FXML private Label label;
12  @FXML private Button change, red, green, blue;
13
14  public RGBColorMixerController( )
15  {
16    // initialize model with delta of -10
17    mixer = new RGBColorMixer( -10 );
18  }
19
20  public void initialize( )
21  {
22    // initialize center label
23    setCenterStyle( );
24
25    // register listener for changeDirection button
26    change.setOnAction( event -> mixer.invertDelta( ) );
27
28    // register listeners for red, green, and blue buttons
29    red.setOnAction( event ->
30                     {
31                       mixer.changeRed( );
32                       setCenterStyle( );
33                     } );
34
35    green.setOnAction( event ->
36                       {
37                        mixer.changeGreen( );
38                        setCenterStyle( );
39                       } );
40
41    blue.setOnAction( event ->
42                      {
43                       mixer.changeBlue( );
44                       setCenterStyle( );
45                      } );
46  }
```

```
47
48   /** setCenterStyle method
49   *    sets the background color of the
50   *    HBox and the text of Label
51   *    in the center position
52   */
53   public void setCenterStyle( )
54   {
55     String style = "-fx-background-color: ";
56     style += mixer.toHexString( );
57     hBoxCenter.setStyle( style );
58     label.setText( mixer.toHexString( ) );
59   }
60 }
```

Example 12.36 The Controller, *RGBMixerController.java*

The launch class, shown in Example 12.37, completes the application.

```
1 /*  MixColors class
2     Anderson, Franceschi
3 */
4
5 import java.net.URL;
6 import javafx.application.Application;
7 import javafx.fxml.FXMLLoader;
8 import javafx.scene.layout.BorderPane;
9 import javafx.scene.Scene;
10 import javafx.stage.Stage;
11
12 public class MixColors extends Application
13 {
14   @Override
15   public void start( Stage stage )
16   {
17     try
18     {
19       URL url =
20          getClass( ).getResource( "fxml_mix_colors.xml" );
21       BorderPane root = FXMLLoader.load( url );
22       Scene scene = new Scene( root, 325, 300 );
23       stage.setTitle( "Mixing Colors" );
24       stage.setScene( scene );
25       stage.show( );
```

```
26      }
27      catch( Exception e )
28      {
29        e.printStackTrace( );
30      }
31    }
32
33    public static void main( String [ ] args )
34    {
35      launch( args );
36    }
37  }
```

Example 12.37 The Launch Class, *MixColors*

12.13 Nesting Components

Components can be nested. Indeed, because the layout containers are subclasses of *Parent*, itself a subclass of *Node*, they are both containers and components. As such, they can contain other layout containers, which in turn can contain components. We can use this feature to achieve more precise layouts.

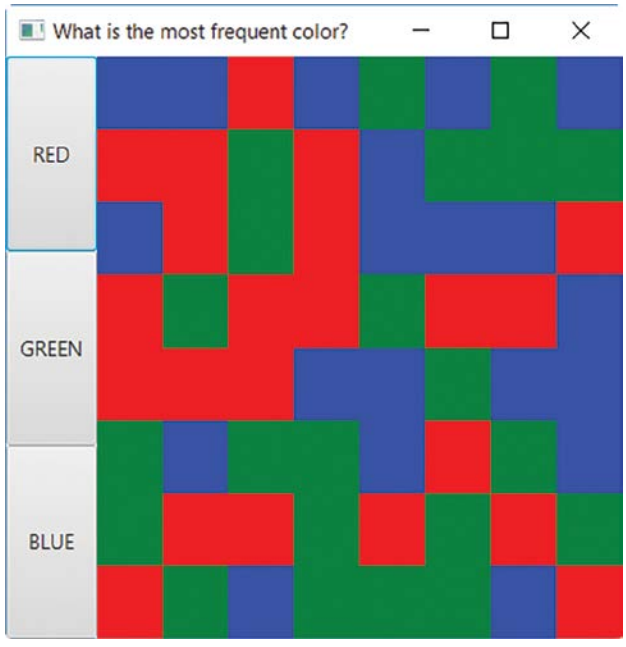
When nesting components, we usually place several components into a layout container and place that layout container into another layout container. Each layout container can be different so that components can be arranged in many ways. We can even have multiple levels of nesting, as needed.

We illustrate nesting of components in an example where, using three colors (red, green, and blue), we randomly generate an 8-by-8 grid of colors. The user tries to guess the most frequently occurring color by clicking on one of the buttons.

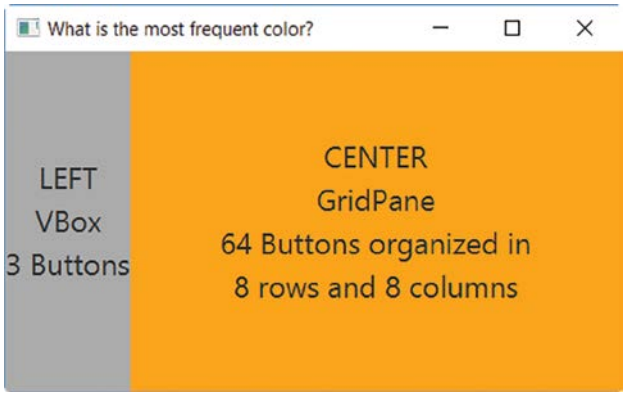
Figure 12.19A shows our window when the application begins, and Figure 12.19B shows the underlying layout of the window. We use a *BorderPane* overall, with a *VBox* containing the three buttons on the left side of the *BorderPane*, and a *GridPane* with eight rows and eight columns containing the 64 colored labels in the center position of the *BorderPane*; thus, the two positions we are using, left and center, adjust to fill the space.

Again, we use the MVC architecture and design our classes for reusability. This time, we also code the Controller so that it is reusable under certain conditions. Inside the *Controller* class, instead of using a class for the Model,

FIGURE 12.19
(A) The Color Frequency Game
(B) The Underlying Nested Layout



(A)



(B)

we use an interface. Thus, the Controller is now reusable, along with the same View, with any Model that implements that interface. Note that we could also make the View an interface, but in order to keep this example relatively simple, we do not. To achieve reusability for the Controller, we use the following pattern for the *Controller* class:

```
public class Controller
{
  private ModelInterface model; // ModelInterface is an interface
  private View view;
  ...
}
```

The *ModelInterface* specifies several methods to be implemented that the Controller will call to manage the game. The *Model* class uses this pattern:

```
public class Model implements ModelInterface
{
  // Implements ModelInterface methods
  ...
}
```

We want the *Controller* class to be reusable with any class whose functionality is based on the user clicking buttons on the left panel with a grid of labels in the center position. We could use this class in an application where the user tries to guess the most frequent color, or the least frequent color, or with a different color system (e.g., HSL—Hue, Saturation and Lightness).

Thus, we design an interface to support this type of application that has buttons on the left and a grid of labels in the center. Example 12.38 shows the *ColorGridGame* interface. It specifies methods to retrieve the title for the grid, its size, its number of colors, the label of a color, and the color index of a label. (It is implied that the colors used in the grid of colors will come from an array of colors, and that there will be an array of *Strings* [labels] that parallels that array of colors.) It also specifies methods to test if an answer is the correct answer, as well as to access each color in the grid. We call these methods from the Controller class with the Model instance variable in order to manage the game. The UML diagram for this example is shown in Figure 12.20.

```
 1 /* ColorGridGame interface
 2  * Anderson, Franceschi
 3 */
 4
 5 import javafx.scene.paint.Color;
 6
 7 public interface ColorGridGame
 8 {
 9  public String getTitle( );
10  public int getSize( );
```

```
11  public int getNumberOfColors( );
12  public String getLabel( int index );
13  public int getIndex( String label );
14  public boolean isCorrect( int index );
15  public String getGridHexColor( int row, int col );
16 }
```

Example 12.38 The *ColorGridGame* Interface

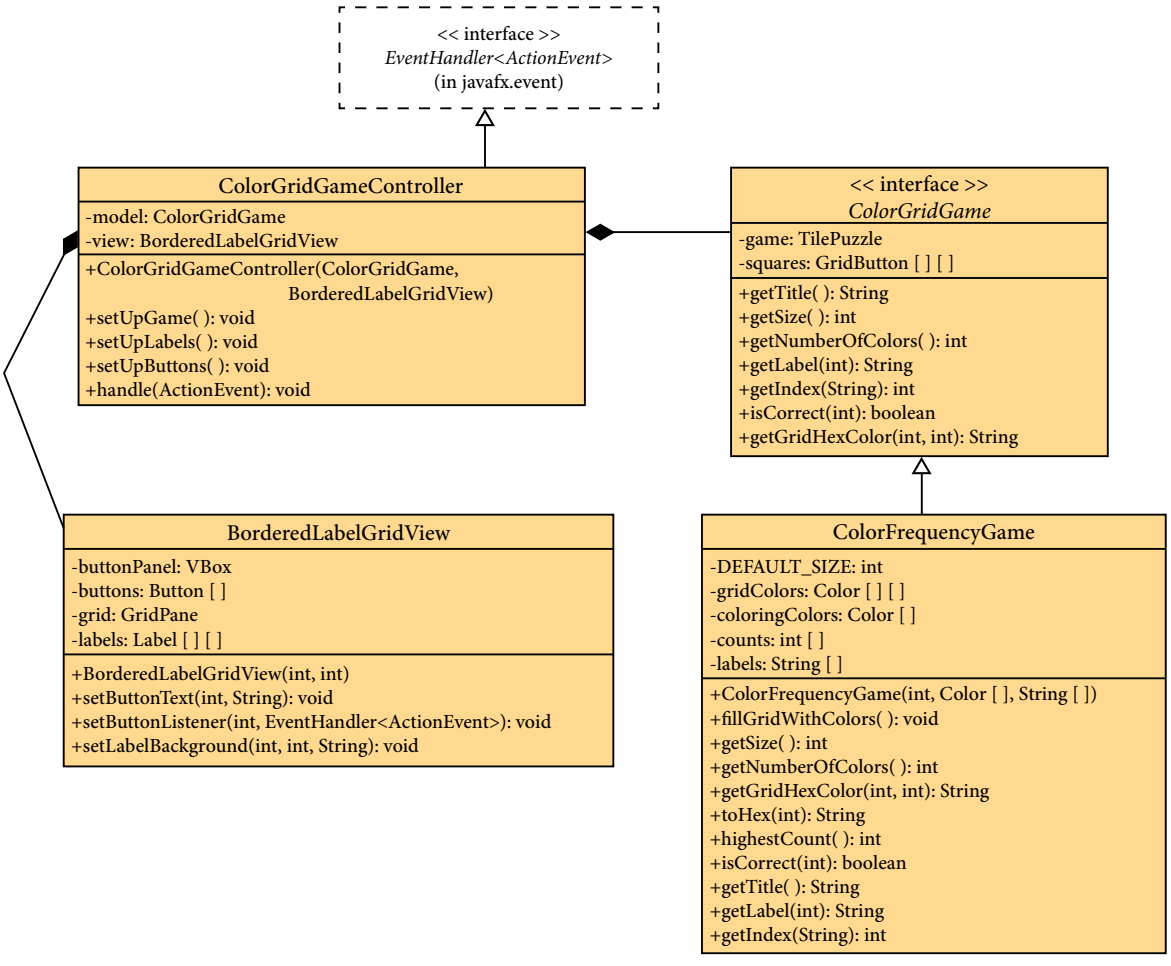


FIGURE 12.20

The UML Diagram for the Color Frequency Game Application

We now code our Model for the color frequency game. In this application, we generate colors for our grid of labels. Example 12.39 shows the *ColorFrequencyGame* class, which *implements ColorGridGame* (line 8). At

lines 10 through 12, we declare a *char* array storing hexadecimal digits. We use this array to convert an integer to its hexadecimal equivalent *String*. Its *gridColors* instance variable (line 15) stores a two-dimensional array, or grid, of colors. The *coloringColors* and *counts* arrays (lines 16–17) are parallel arrays: *coloringColors* stores the colors used for coloring the grid, and *counts* stores the frequency of each color. The instance variable *labels* (line 19) is an array of *Strings* that is meant to be labels for the buttons. The *labels* array also parallels the *coloringColors* array.

The constructor (lines 21–36) instantiates all three arrays and calls the *fillGridWithColors* method at line 34. The *fillGridWithColors* method (lines 38–53) fills the grid of colors with colors randomly generated from the *coloringColors* array and keeps track of their respective frequency in the *counts* array (line 50).

The *getGridHexColor* method (lines 71–83) returns a hexadecimal representation of the color stored at a specified row and column within the array *gridColors*. It uses the utility method *toHex* (lines 85–98), which converts an integer whose value is between 0 and 255 to its hexadecimal equivalent *String*.

To determine whether the user has chosen the correct color, we provide the *isCorrect* method (lines 114–122), which takes as a parameter the index of the selected color. It returns *true* if the value in *counts* at that index is maximal, *false* otherwise. The *highestCount* method (lines 100–112) calculates and returns the maximum value in the *counts* array.

The *getTitle* method (lines 124–130) returns a *String* that can be used as the title of a GUI whose Model is the *ColorFrequencyGame* class. The *getLabel* method (lines 132–139) returns the element of the array *labels* at a specified index, sent as the parameter of the method; that is, the label is to be used with a color located at that index within the array *coloringColors*. The *getIndex* method (lines 141–153) is the opposite method: It returns the index within the array *labels* of a *String*, sent as the parameter of the method; the color corresponding to that label is the element in the array *coloringColors* at that index.

```
1 /** ColorFrequencyGame class
2     Anderson, Franceschi
3 */
4
5 import java.util.Random;
```

```
import javafx.scene.paint.Color;

public class ColorFrequencyGame implements ColorGridGame
{
 public static char [ ] hexDigits =
    { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
      'A', 'B', 'C', 'D', 'E', 'F' };

 private final int DEFAULT_SIZE = 8;
 private Color [ ][ ] gridColors;
 private Color [ ] coloringColors;
 private int [ ] counts; // color frequencies in gridColors

 private String [ ] labels;

 /** Constructor
 * @param size number of rows and columns in gridColors
 * @param colors coloringColors
 * @param newLabels labels
 */
 public ColorFrequencyGame( int size, Color [ ] colors,
                            String [ ] newLabels )
 {
  if ( size <= 0 )
   size = DEFAULT_SIZE;
  gridColors = new Color[size][size];
  coloringColors = colors;
  counts = new int[coloringColors.length];
  fillGridWithColors( );
  labels = newLabels;
 }

 /** fillGridWithColors method
 * randomly fills gridColors with colors from coloringColors
 */
 public void fillGridWithColors( )
 {
  Random random = new Random( );
  for ( int i = 0; i < gridColors.length; i++ )
  {
   for ( int j = 0; j < gridColors[i].length; j++ )
   {
    int colorIndex = random.nextInt( coloringColors.length );
    gridColors[i][j] = coloringColors[colorIndex];
    counts[colorIndex] += 1;
```

```
51    }
52   }
53  }
54
55  /** getSize method
56  * @return length of gridColors
57  */
58  public int getSize( )
59  {
60   return gridColors.length;
61  }
62
63  /** getNumberOfColors method
64  * @return length of coloringColors
65  */
66  public int getNumberOfColors( )
67  {
68   return coloringColors.length;
69  }
70
71  /** getGridHexColor method
72  * @param row, an int, the row index
73  * @param col, an int, the column index
74  * @return a String, the hex equivalent of gridColor[row][col]
75  */
76  public String getGridHexColor( int row, int col )
77  {
78   String colorText = "#"
79     + toHex( ( int ) ( 255 * gridColors[row][col].getRed( ) ) )
80     + toHex( ( int ) ( 255 * gridColors[row][col].getGreen( ) ) )
81     + toHex( ( int ) ( 255 * gridColors[row][col].getBlue( ) ) );
82   return colorText;
83  }
84
85  /** toHex method
86  * @return a String, the Hex equivalent of i
87  */
88  public String toHex( int i )
89  {
90   if ( i > 255 )
91     i = 255;
92   else if ( i < 0 )
93     i = 0;
94
95   int q = i / 16;
```

```
 int r = i % 16;
 return "" + hexDigits[q] + hexDigits[r];
}

/** highestCount method
* @return the highest color frequency in the grid
*/
public int highestCount( )
{
 int max = counts[0];
 for ( int i = 1; i < counts.length; i++ )
 {
  if ( counts[i] > max )
   max = counts[i];
 }
 return max;
}

/** isCorrect method
* @param index, an int
* @return true if the frequency of the color for index
* is the highest, false otherwise
*/
public boolean isCorrect( int index )
{
 return counts[index] == highestCount( );
}

/** getTitle method
* @return a String representing a title for this object
*/
public String getTitle( )
{
 return "What is the most frequent color?";
}

/** getLabel method
* @param index, an int
* @return labels[index]
*/
public String getLabel( int index )
{
 return labels[index];
}
```

```
141  /** getIndex method
142  * @param label, a String
143  * @return index of label in labels
144  */
145  public int getIndex( String label )
146  {
147   for ( int i = 0; i < labels.length; i++ )
148   {
149    if ( labels[i].equals( label ) )
150     return i;
151   }
152   return -1;
153  }
154 }
```

Example 12.39 The *ColorFrequencyGame* Class

Example 12.40 shows the *BorderedLabelGridView* class. It is a layout container class that *extends BorderPane* (line 16). At its left position, it contains an array of buttons arranged vertically. At its center position, it contains a grid of labels.

Thus, our instance variables are a *VBox* (line 18), which contains an array of *Buttons* (line 19), and a *GridPane* (line 20), which contains a two-dimensional array of *Labels* (line 21). We place the *VBox* in the left position and the *GridPane* in the center position. The other positions within the *BorderPane* are not used.

The constructor (lines 23–67) defines a GUI consisting of an array of buttons on the left, arranged vertically, and a grid of labels in the center, assumed to be a square. Its two parameters represent the number of buttons and the size of the grid. For simplicity, we are not validating these parameters, but both parameters must be greater than 0. The grid of labels is defined at lines 27 through 50 inside the *GridPane*. The array of buttons is defined at lines 52 through 63 inside the *VBox*. At lines 65 and 66, we add the *VBox* and the *GridPane* to this *BorderPane* to the left and center positions, respectively.

We instantiate *grid*, the *GridPane* instance variable, at line 27. At lines 29 through 37, we define row and column constraints so that all the rows in the grid have the same height, all the columns have the same width, and the rows and columns take all the available space in the grid. At lines 39

through 50, we instantiate the labels and add them to the grid, making sure that each label fills the available width and height (lines 45–47).

We instantiate *buttonPanel*, the *VBox* instance variable, at line 52. At lines 54 through 63, we instantiate the buttons and add them to *buttonPanel*, also making sure that each label fills the available width and height (lines 58–61). It is important to set not only the maximum height of the buttons to a high value (line 60), but also their maximum width (line 59). If we do not, the buttons will have uneven width because their text contents are different. It is also important to call the *setVgrow* method (line 61), shown in Table 12.31, so that the buttons are expanded vertically in order to fill the available space within the *VBox* to which they belong. The *setVgrow* method sets the priority level for expanding the height of a node that is contained in the *VBox* container. The *setVgrow* method is *static*, so there is no reference to a specific *VBox* when we call this method; thus, it is assumed that the node is or will be contained in a *VBox*. If we do not call *setVgrow*, the heights of the buttons are not expanded. *Priority* is an *enum* with three possible values: *ALWAYS*, *NEVER*, and *SOMETIMES*. The *ALWAYS* value ensures that the node grows in order to fill the available space in the *VBox* container. We specify the same priority level for all of the buttons so that they grow equally.

Table 12.31 The *setVgrow* Method of the *VBox* Class

Package	javafx.scene.layout
Return value	**Method name and argument list**
`void`	`setVgrow( Node child, Priority value )` *static* method that sets the vertical grow priority of *child*, assuming that *child* is contained in a *VBox*. If set, the *VBox* will allocate additional space for *child* if space is available.

Finally, we provide methods so that the text of each button and the background color of each label can be set from outside the class (lines 69–72 and 79–82). We also provide a method (lines 74–77) to set an *EventHandler* for each button in the array *buttons*. By calling these methods, the Controller can set up event handling and update this View.

```
1 /** BorderedLabelGridView class
2 * Reusable generic layout using a BorderPane's
3 * left and center positions.
4 * The left VBox is made up of a vertical array of buttons.
5 * The center position is made up of a grid of labels.
```

```
 6 * Accessors are provided so that a Controller can access
 7 * the array of buttons and the 2-dim array (grid) of labels.
 8 * Anderson, Franceschi
 9 */
10
11 import javafx.event.*;
12 import javafx.scene.control.*;
13 import javafx.scene.layout.*;
14 import javafx.scene.paint.Color;
15
16 public class BorderedLabelGridView extends BorderPane
17 {
18  private VBox buttonPanel; // left, holds array of buttons
19  private Button [ ] buttons;
20  private GridPane grid; // center, holds grid of labels
21  private Label [ ][ ] labels; // grid of labels
22
23  // numberOfButtons and gridSize must be greater than 0
24  public BorderedLabelGridView( int numberOfButtons, int gridSize )
25  {
26   super( );
27   grid = new GridPane( );
28
29   // set up grid as gridSize by gridSize
30   RowConstraints row = new RowConstraints( );
31   row.setPercentHeight( 100.0 / gridSize );
32   ColumnConstraints col = new ColumnConstraints( );
33   col.setPercentWidth( 100.0 / gridSize );
34   for ( int i = 0; i < gridSize; i++ )
35     grid.getRowConstraints( ).add( row );
36   for ( int j = 0; j < gridSize; j++ )
37     grid.getColumnConstraints( ).addAll( col );
38
39   labels = new Label[gridSize][gridSize];
40   for ( int i = 0; i < labels.length; i++ )
41   {
42    for ( int j = 0; j < labels[i].length; j++ )
43    {
44     labels[i][j] = new Label( );
45     // make label fill up available width and height
46     labels[i][j].setMaxWidth( Double.MAX_VALUE );
47     labels[i][j].setMaxHeight( Double.MAX_VALUE );
48     grid.add( labels[i][j], j, i );
49    }
50   }
51
```

```
52   buttonPanel = new VBox( );
53
54   buttons = new Button[numberOfButtons];
55   for ( int i = 0; i < buttons.length; i++ )
56   {
57    buttons[i] = new Button( );
58    // make button fill up available width and height
59    buttons[i].setMaxWidth( Double.MAX_VALUE );
60    buttons[i].setMaxHeight( Double.MAX_VALUE );
61    VBox.setVgrow( buttons[i], Priority.ALWAYS );
62    buttonPanel.getChildren( ).add( buttons[i] );
63   }
64
65   setLeft( buttonPanel );
66   setCenter( grid );
67  }
68
69  public void setButtonText( int row, String text )
70  {
71   buttons[row].setText( text );
72  }
73
74  public void setButtonListener( int row, EventHandler<ActionEvent> eh )
75  {
76   buttons[row].setOnAction( eh );
77  }
78
79  public void setLabelBackground( int row, int col, String hexColor )
80  {
81   labels[row][col].setStyle( "-fx-background-color: " + hexColor );
82  }
83 }
```

Example 12.40 The *BorderedLabelGridView* Class

Example 12.41 shows the *ColorGridGameController* class. At lines 13 and 14, we declare our two instance variables: (1) *model*, a *ColorGridGame*; and (2) *view*, a *BorderedLabelGridView*. These instance variables enable us to get user input from the View, call the appropriate methods of the Model, and update the View accordingly.

The constructor (lines 16–22) accepts two parameters that it assigns to *model* and *view*, and it calls the *setUpGame* method at line 21. The *setUpGame* method (lines 24–28) calls the *setUpLabels* and *setUpButtons* methods.

The *setUpLabels* method (lines 30–35) uses a nested loop to color the grid of labels in the View based on the grid of colors in the Model. At line 34, we set the background color of the current label by calling the View's *setLabelBackground* method, passing the row index, column index, and background color that we retrieve from the Model by calling the *getGridHexColor* method.

The *setUpButtons* method (lines 37–44) places a label on each button in the View based on the label data in the Model and sets up event handling using a single loop. At line 41, we set the text for the current button, passing its index and the label that we retrieve from the Model by calling the *getLabel* method. At line 42, we register *this* object (i.e., this controller, which implements *EventHandler*) on the current button.

We handle the events for the buttons in the *handle* method at lines 46 through 56. We first retrieve the button that originated the event by calling *getSource* at line 48. The *getSource* method returns an *Object*, so we need to typecast the return value to a *Button*. We then retrieve the index of that button's corresponding label by calling the Model's *getIndex* method, passing the text of the button. Next, we check whether the user won or lost by calling the Model's *isCorrect* method, passing that index. Depending on the result, we pop up a dialog box indicating whether the user has won or lost (lines 51–54). After the game is won or lost, we should disable the buttons. This is left as an exercise.

```
 1 /** ColorGridGameController class
 2 * Anderson, Franceschi
 3 */
 4
 5 import javafx.event.*;
 6 import javafx.scene.control.*;
 7 import javafx.scene.layout.*;
 8 import javafx.scene.paint.Color;
 9 import javax.swing.JOptionPane;
10
11 public class ColorGridGameController implements EventHandler<ActionEvent>
12 {
13  private ColorGridGame model;
14  private BorderedLabelGridView view;
15
16  public ColorGridGameController( ColorGridGame newModel,
17                                  BorderedLabelGridView newView )
18  {
```

```
 model = newModel;
 view = newView;
 setUpGame( );
}

public void setUpGame( )
{
 setUpLabels( );
 setUpButtons( );
}

public void setUpLabels( )
{
 for ( int i = 0; i < model.getSize( ); i++ )
  for ( int j = 0; j < model.getSize( ); j++ )
   view.setLabelBackground( i, j, model.getGridHexColor( i, j ) );
}

public void setUpButtons( )
{
 for ( int i = 0; i < model.getNumberOfColors( ); i++ )
 {
  view.setButtonText( i, model.getLabel( i ) );
  view.setButtonListener( i, this );
 }
}

public void handle( ActionEvent event )
{
 Button button = ( Button ) event.getSource( );
 int index = model.getIndex( button.getText( ) );

 if ( model.isCorrect( index ) )
  JOptionPane.showMessageDialog( null, "You won" );
 else
  JOptionPane.showMessageDialog( null, "Sorry, you lost" );
 // disable buttons here
}
}
```

Example 12.41 The *ColorGridGameController* Class

Finally, the *PlayColorCount* class, shown in Example 12.42, includes the *main* and *start* methods to create a color count application. We declare an

array of three *Colors* at line 16 and a parallel array of *Strings* at line 17, and pass these arrays to the constructor of the *ColorFrequencyGame* at lines 19 through 20. We then create the *BorderedLabelGridView* object *root* at lines 22 and 23. At line 26, we retrieve and set the title of the window. Using *game* and *root*, we create a *ColorGridGameController* at lines 30 and 31. Note that *game* "is a" *ColorGridGame* object because *ColorFrequencyGame* inherits from *ColorGridGame*. Figure 12.21 shows a run of this example after the user has clicked on the button "GREEN." To play the game with more colors, we could simply add more colors and labels to the *colors* and *labels* array.

```
1 /** PlayColorCount class
2 *   Anderson, Franceschi
3 */
4
5 import javafx.application.Application;
6 import javafx.scene.paint.Color;
7 import javafx.scene.Scene;
8 import javafx.stage.Stage;
9
10 public class PlayColorCount extends Application
11 {
12
13   @Override
14   public void start( Stage stage )
15   {
16     Color [ ] colors = { Color.RED, Color.GREEN, Color.BLUE };
17     String [ ] labels = { "RED", "GREEN", "BLUE" };
18
19     ColorFrequencyGame game
20       = new ColorFrequencyGame( 8, colors, labels );
21
22     BorderedLabelGridView root
23       = new BorderedLabelGridView( colors.length, game.getSize( ) );
24
25     Scene scene = new Scene( root, 450, 425 );
26     stage.setTitle( game.getTitle( ) );
27     stage.setScene( scene );
28     stage.show( );
29
30     ColorGridGameController controller
31       = new ColorGridGameController( game, root );
32   }
33
```

```
34   public static void main( String [ ] args )
35   {
36     launch( args );
37   }
38 }
```

Example 12.42 The *PlayColorCount* Class

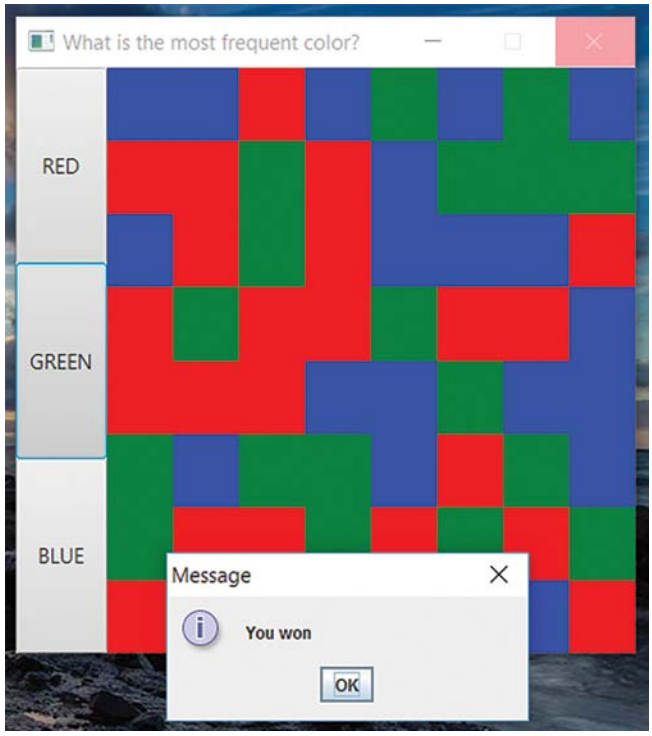


FIGURE 12.21
The User Clicked on the Button GREEN

Skill Practice
with these end-of-chapter questions

12.16.1 Multiple Choice Exercises

Questions 10, 11, 12, 13, 14, 15, 16, 17

12.16.2 Reading and Understanding Code

Questions 27, 28, 29

12.14 Programming Activity 2: Working with Layout Containers

In this Programming Activity, you will complete the implementation of a version of the Tile Puzzle game using a more complex GUI. As it stands now, the application compiles and runs, but it is missing a lot of code. Figure 12.22 shows the window that will open when you run the application without adding your code. Once you have completed the five tasks of this Programming Activity, you should see the window in Figure 12.23 when you run your program and click on the "3-by-3" button. When you click on one of the buttons labeled "3-by-3," "4-by-4," or "5-by-5," the tile puzzle will reset to a grid of that size.

In addition to the *TilePuzzle* class, we provide you with a prewritten *GameView* class, which encapsulates a View for the Tile Puzzle game. We have implemented the *GameView* class as a *GridPane* container, so you can add it to another layout container, such as a *BorderPane*. It has two important methods, shown in Table 12.32. Thus, your job in this Programming Activity is not to write the game code, but to organize components in a window.

FIGURE 12.22
The Starting Window When Running the Prewritten Code

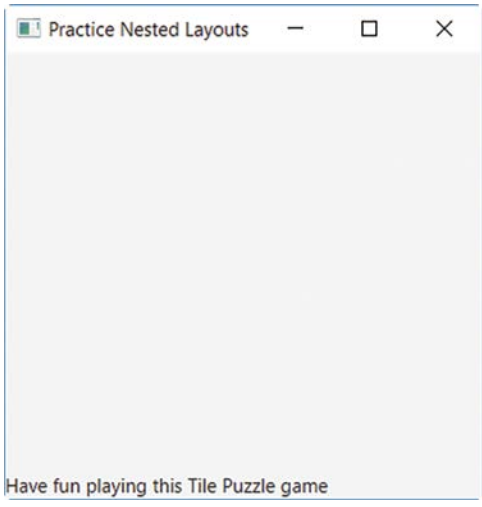


FIGURE 12.23
The Starting Window When the Activity Is Completed

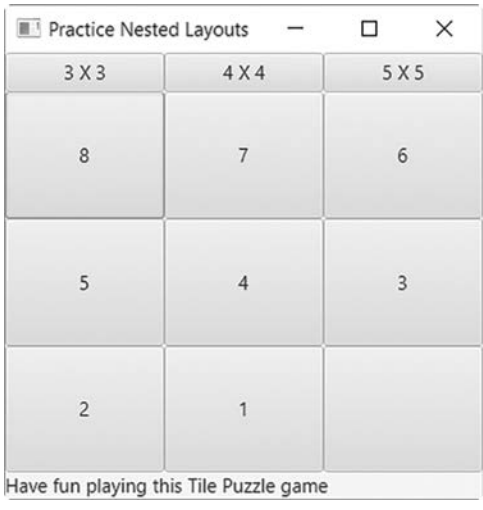


TABLE 12.32 The *GameView* Class API

Constructor	
`GameView( int nSides )`	
instantiates a tile puzzle View having an *nSides*-by-*nSides* grid	
Return value	**Method name and argument list**
`void`	`setUpGame( int nSides )`
	resets the grid as an *nSides*-by-*nSides* grid

You need to edit the *NestedLayoutPractice* class, which *extends BorderPane*. Your job is to:

1. Declare an *Hbox* named *top* and three *Buttons* that will be added to the *top* position of the *BorderPane*.
2. Set the layout containers for the *center* and *top* positions.
3. Add the *top* and the *gameView* layout containers to the *BorderPane*.
4. Code an appropriate *private* listener class.
5. Instantiate the listener and register it on the appropriate components.

Instructions

Copy the source files in the Programming Activity 2 folder for this chapter to a folder on your computer.

1. Write the code to declare the needed instance variables. Load the *NestedLayoutPractice.java* source file and search for five asterisks in a row (*****). This will position you at the instance variables declaration.

```
// ***** Task 1: declare an HBox named top
// also declare three Button instance variables
// that will be added to the HBox top.
// These buttons will determine the grid size of the game:
// 3-by-3, 4-by-4, or 5-by-5


// task 1 ends here
```

2. Next, write the code to set the layout manager of the window and add the component *gameView* in the center position of the window. In the *NestedLayoutPractice.java* source file, search again for five asterisks in a row (*****). This will position you inside the constructor.

```
// ***** Task 2: student code starts here

// instantiate the GameView object


// add gameView to the center of this BorderPane


// task 2 ends here
```

3. Next, write the code to instantiate the *HBox* top component, instantiate the buttons from task 1, add them to *top*, and finally add *top* at the top position of our *BorderPane*. In the *NestedLayoutPractice.java* source file, search again for five asterisks in a row (*****). This will position you inside the constructor.

```
// ***** Task 3: Student code restarts here
// instantiate the HBox component named top

// instantiate the Buttons that determine the grid size


// add the buttons to HBox top
// make them take all the available space


// add HBox top to this BorderPane as its top component


// task 3 ends here
```

4. Next, write the code for the *private* inner class that implements the appropriate listener. In the *NestedLayoutPractice.java* source file, search again for five asterisks in a row (*****). This will position you between the constructor and the end of the class.

```
// ***** Task 4: Student code restarts here
// create a private inner class that implements EventHandler
// your method should identify which of the 3 buttons
//    was the source of the event
// depending on which button was pressed,
//    call the setUpGame method of the GameView class
//    with arguments 3, 4, or 5
// the API of that method is:
//   public void setUpGame( int nSides )


// task 4 ends here
```

5. Next, write the code to declare and instantiate a listener, and register it on the appropriate components. In the *NestedLayoutPractice.java* source file, search again for five asterisks in a row (*****). This will position you inside the constructor.

```
// ***** Task 5: Student code restarts here
// Note: search for and complete Task 4 before performing this task
// declare and instantiate an EventHandler
```

```
// register the handler on the 3 buttons
// that you declared in Task 1


// task 5 ends here
```

After completing each task, compile your code.

When you have finished writing all of the code, compile the source code and run *NestedLayoutPracticeApplication*. Try clicking on the three buttons that you added.

? DISCUSSION QUESTIONS

1. Identify the various layout containers you used and the screen positions they occupy.
2. Explain why the left and right positions are empty on the window.

12.15 Chapter Summary

- A Graphical User Interface (GUI) allows the user to enter data and initiate actions for an application by entering text into boxes; pressing buttons; moving the thumb of a slider; or selecting items from a list, a set of radio buttons, or checkboxes.
- Applications with GUIs are usually easier to learn and use because the interface is familiar to the user.
- JavaFX is a set of classes included with Java SE for creating GUIs.
- The top-level structure in a JavaFX application is the stage, which corresponds to a window. A stage can have one or more scenes, which are top-level containers for nodes that make up the window contents. A node can be a user interface control, such as a button or drop-down list; a layout container; an image or other media; a graphical shape; a web browser; a chart; or a group.
- JavaFX applications can be built in several ways. If we know which controls our interface needs and how they should be arranged, we can specify the layout, controls, and their properties using FXML, a scripting language based on XML. For more complicated GUIs or for dynamic GUIs where the number or type of control is determined at runtime, we can control the number, type, properties, and positioning of controls programmatically.

CHAPTER SUMMARY

CHAPTER SUMMARY

- A JavaFX application extends the *Application* class and, at minimum, implements the *start* method.
- Some JavaFX GUI controls are *Label*, *TextField*, *Button*, *RadioButton*, *CheckBox*, *ComboBox*, and *Slider*. These controls inherit from the *Node* class.
- Controls can be arranged in a window using layout containers, such as *HBox*, *VBox*, *BorderPane*, *GridPane*, or *StackPane*. Layout containers can be nested.
- GUI applications use an event-driven model, where the user determines which functions are performed by interacting with the application controls, consequently firing events. To handle an event, we register our interest in being notified of the event and provide code: an event handler or listener, to be executed when the event occurs.
- JavaFX supports and encourages the Model-View-Controller architecture in GUI applications. The Model manages the data of the application and its state. The View presents the user interface. The Controller handles events generated by the user and communicates those changes to the Model, which updates its state accordingly and communicates any changes back to the Controller. The Controller then updates the View to reflect those changes.
- FXML can be used to define not only the application's layout containers, GUI controls, and their properties, but also the controller class and the method to be executed when an event fires.
- An *ActionEvent* object is created when the user clicks a button, selects an item from a list or a menu, or presses the *Enter* key in a *TextField*. The *getSource* method of the *ActionEvent* object returns a reference to the control that fired the event.
- To make *RadioButtons* mutually exclusive, we define a *ToggleGroup* and then set each *RadioButton*'s *toggleGroup* property accordingly.
- A Controller can have an *initialize* method, which is called after the scene graph has been created. The *initialize* method can be used to add nodes and set properties that could not be fully defined in the FXML file. In the *initialize* method, we also can retrieve initial values from the Model and update the View accordingly so that the View reflects the initial state of the Model when the application starts.

- We put *ComboBox* items into an *ObservableList* and use the *SingleSectionModel* to manage the selection of items.
- The *Slider* control is capable of displaying a set of continuous values along a horizontal or vertical line called a track. The user "slides" the knob, called the thumb, along the track to select a value. We can set properties of a *Slider* to display tick marks and tick values.
- An event handler for a *Slider* control needs to implement the *ChangeListener<T>* interface.
- We can also define our application's GUI components, properties, and event handlers programmatically. This is useful for dynamic GUIs where the number or type of controls is not known until runtime, or when we have an array of controls that are handled identically.
- A *VBox* is a layout container that arranges its components vertically.
- An *HBox* is a layout container that arranges its components horizontally.
- A *GridPane* can be visualized as a table made up of cells in rows and columns. Each cell can contain one component. These cells can have different sizes, which we specify using row and column constraints.
- A *BorderPane* layout container organizes its nodes into five positions—*top*, *bottom*, *left*, *right*, and *center*—with each position holding one node at most.
- Lambda expressions can be used to simplify the definition of an event handler as an anonymous class that *implements* a functional interface, that is, an interface that requires only one method to be implemented.
- A lambda expression contains (1) a comma-separated list of parameters enclosed in parentheses—the data types of the parameters may be omitted, and the parentheses can also be omitted if there is only one parameter; (2) the arrow token, ->; and (3) a method body, which can be a single expression or a block enclosed in curly braces. If the body of the method consists of a single expression, then the JVM evaluates the expression and returns its value. You can also use a return statement, but that requires curly braces.

12.16 Exercises, Problems, and Projects

12.16.1 Multiple Choice Exercises

1. An example of a GUI component class is
   - ❑ *FXML*
   - ❑ *controller*
   - ❑ *TextField*
   - ❑ *Stage*
2. What are the primary uses of GUI components? (Check all that apply.)
   - ❑ Display information
   - ❑ Facilitate the coding of methods
   - ❑ Let the user control the program
   - ❑ Collect information from the user
3. In what package do you find the *Button*, *TextField*, and *ComboBox* classes?
   - ❑ *javafx.scene*
   - ❑ *javafx.scene.control*
   - ❑ *java.scene*
   - ❑ *java.awt*
   - ❑ *java.io*
4. *VBox* is a
   - ❑ Label
   - ❑ Layout container that arranges components vertically
   - ❑ Layout container that arranges components horizontally
   - ❑ Scene
5. The property of a *Label* element that specifies the text inside the label is
   - ❑ *Label*
   - ❑ *Word*
   - ❑ *Phrase*
   - ❑ *text*

6. The property that specifies the name of the Controller class for the View defined in the FXML document is
   - ❑ *control*
   - ❑ *fx:control*
   - ❑ *controller*
   - ❑ *fx:controller*
7. The property of a *Button* element that specifies the method called when the user clicks the button is
   - ❑ *action*
   - ❑ *press*
   - ❑ *onAction*
   - ❑ *onPress*
8. What attribute and annotation do we use with an instance variable of the Controller to reference a GUI component defined in an FXML document?
   - ❑ *id and FXML*
   - ❑ *id and @FXML*
   - ❑ *fx:id and FXML*
   - ❑ *fx:id and @FXML*
9. Assume that we have correctly defined the FXML attribute of a *Button* element that specifies the method to call when the user clicks on that button. What is the return type of that method?
   - ❑ *Button*
   - ❑ *void*
   - ❑ *boolean*
   - ❑ *onAction*
10. With JavaFX, a user interface must be defined using FXML; it cannot be defined programmatically.
    - ❑ True
    - ❑ False

11. We want to set up event handling programmatically when the user clicks on a button. What should the programmer do? (Check all that apply.)

    ❑ Code a class that implements the *EventHandler<ActionEvent>* interface.

    ❑ Declare and instantiate an object reference (a listener) of the class above.

    ❑ Call the *handle* method.

    ❑ Register the listener on the button.

12. Assuming everything has been coded correctly in the previous question, what happens when the user clicks a button?

    ❑ The *handle* method executes.

    ❑ The *Button* constructor executes.

    ❑ The *start* method executes.

13. We want to build a class that implements an interface that listens to key events. What interface should we implement?

    ❑ *EventHandler<ActionEvent>*

    ❑ *EventHandler<KeyEvent>*

    ❑ *EventHandler<MouseEvent>*

    ❑ *KeyHandler<Event>*

14. We are designing a GUI programmatically with three buttons; a different action will be taken depending on which button the user clicks. We want to code only one *private* class implementing the *EventHandler<ActionEvent>* interface. Inside the *handle* method, which method do we call to determine which button was clicked?

    ❑ *getButton*

    ❑ *getSource*

    ❑ *getOrigin*

15. *NewLayout* is a layout container.

    ❑ True

    ❑ False

16. What is the maximum number of top-level controls that a *BorderPane* can manage?
    - ❑ 2
    - ❑ 3
    - ❑ 4
    - ❑ 5
    - ❑ 6
17. FXML elements can be nested.
    - ❑ True
    - ❑ False

12.16.2 Reading and Understanding Code

For Questions 18 to 22, consider the following FXML file representing the View for the application:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<HBox fx:controller="MyController"
      xmlns:fx="http://javafx.com/fxml"
      alignment="center" spacing="10" >
  <Button fx:id="button1" text="Button 1"
          onAction="#go" />
  <Button fx:id="button2" text="Button 2"
          onAction="#go" />
  <Label fx:id="result" />
  <Button fx:id="button3" text="Button 3"
          onAction="#go" />
  <Button fx:id="button4" text="Button 4"
          onAction="#go" />
</HBox>
```

18. How many buttons will be displayed in the window?
19. How are the buttons and the label organized in the window?
20. What class should we code in order to process clicks on the buttons by the user?
21. What method will execute when the user clicks on one of the buttons?

22. What is the return type of that method?

For Questions 23 through 26, consider the following code (and assume that the FXML file defines the text for cb1, cb2, and cb3 as "Choice 1," "Choice 2," and "Choice 3"):

```
/*  Controller class
*   Anderson, Franceschi
*/
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.*;

public class Controller
{
  @FXML private CheckBox cb1;
  @FXML private CheckBox cb2;
  @FXML private CheckBox cb3;
  @FXML private Label label;
  private int read, write, execute;

  @FXML protected void mix( ActionEvent event )
  {
    CheckBox cb = ( CheckBox ) event.getSource( );
    if ( cb == cb1 )
      read = ( cb.isSelected( ) ? 4 : 0 );
    else if ( cb == cb2 )
      write = ( cb.isSelected( ) ? 2 : 0 );
    else if ( cb == cb3 )
      execute = ( cb.isSelected( ) ? 1 : 0 );

    int mode = read + write + execute;
    label.setText( "mode: " + mode );
  }
}
```

23. What happens when the user checks "Choice 1" only?
24. What happens when the user checks "Choice 1" and then checks "Choice 2"?
25. What happens when the user checks "Choice 1," then "Choice 2," and then "Choice 3"?
26. What happens when the user checks "Choice 3," then "Choice 2," and then "Choice 2" again?

For Questions 27 through 29, consider the following code (and assume that the class extending *Application* exists and is correctly coded):

```
import javafx.event.*;
import javafx.scene.control.Button;
import javafx.scene.layout.*;

public class BoardView extends GridPane
{
  private Button [ ][ ] buttons;

  public BoardView( )
  {
    super( );

    ColumnConstraints col = new ColumnConstraints( );
    col.setPercentWidth( 25 );
    RowConstraints row = new RowConstraints( );
    row.setPercentHeight( 20 );

    for ( int i = 0; i < 5; i++ )
      getRowConstraints( ).add( row );
    for ( int j = 0; j < 4; j++ )
      getColumnConstraints( ).add( col );
    buttons = new Button[5][4];
    ButtonHandler bh = new ButtonHandler( );

    for ( int i = 0; i < 5; i++ )
    {
      for ( int j = 0; j < 4; j++ )
      {
        // instantiate the buttons
        buttons[i][j] = new Button( );
        buttons[i][j].setMaxWidth( Double.MAX_VALUE );
        buttons[i][j].setMaxHeight( Double.MAX_VALUE );

        add( buttons[i][j], j, i );
        buttons[i][j].setOnAction( bh );
      }
    }
  }

  private class ButtonHandler implements EventHandler <ActionEvent>
  {
    public void handle(  ActionEvent event )
    {
      for ( int i = 0; i < buttons.length; i++ )
```

```
            for ( int j = 0; j < buttons[i].length; j++ )
              if ( event.getSource( ) == buttons[i][j] )
                buttons[i][j].setText( "" + ( i + j ) );
        }
      }
    }
```

27. How many rows and columns are in the grid?
28. What happens when the user clicks on the *Button* located at the top left of the grid?
29. What happens when the user clicks on the *Button* located at the bottom right of the grid?

12.16.3 Fill In the Code

For Questions 30 through 32, consider the following FXML document representing a GUI:

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

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<BorderPane fx:controller="MyController"
            xmlns:fx="http://javafx.com/fxml"  >
  <top>
    <HBox alignment="center">
      <Button fx:id="button1" text="INSERT" />
    </HBox>
  </top>

  <left>
    <HBox alignment="center">
      <Button fx:id="button2" text="UPDATE" onAction="#go"/>
    </HBox>
  </left>

  <!-- answer to questions 30 to 32 go here -->




</BorderPane>
```

30. Add a *VBox* element that contains a button at the *right* position within the *BorderPane*. The text of the button should say DELETE;

when the user clicks on it, the *test* method of the *MyController* class should execute. The id of the button should be *button3*.

31. Add an *HBox* element that contains a label at the *center* position of the *BorderPane*. The text of the label should be SELECT. The id of the label should be *label1*.

32. Add a *VBox* element that contains a button at the bottom position within the *BorderPane*. The text of the button should be CREATE; when the user clicks on it, the *table* method of the *MyController* class should execute. The id of the button should be *button4*.

For Questions 33 through 37, consider the following class:

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

public class A extends HBox
{
  private Button b;
  private Label l;

}
```

33. Inside the constructor, this code instantiates the button *b* with the text "Button."

```
// your code goes here
```

34. Inside the constructor, this code instantiates the label *l* with the text "Hello."

```
// your code goes here
```

35. Inside the constructor, this code adds *b* and *l* to this *HBox* so that *b* is on the right and *l* is on the left:

```
// your code goes here
```

36. Inside the constructor, this code registers the listener *mh* on the button *b*:

```
// the MyHandler class is a private class implementing EventHandler
MyHandler mh = new MyHandler( );
// your code goes here
```

37. Inside the *handle* method of a *private* inner class implementing the *EventHandler* interface, this code changes the text of *l* to "Button clicked" if the button *b* was clicked:

```
public void handle( ActionEvent ae )
{
  // your code goes here
}
```

For Questions 38 through 43, consider the following class:

```
import javafx.scene.control.*;
import javafx.scene.layout.*;

public class B extends BorderPane
{
  private VBox left;
  private HBox top;
  private Button [ ] buttons; // length 4
  private TextField [ ] textfields; // length 3
  private Label label1;
  private Label label2;

}
```

Also, assume that none of the instance variables have been instantiated and you are coding inside the constructor.

38. This code instantiates *top* and *left*.

```
// your code goes here
```

39. This code instantiates the text fields with text *TF0*, *TF1*, and *TF2*, and it places them in that order inside *top*. They should fill the whole available width of *top*.

```
// your code goes here
```

40. This code instantiates the buttons with text *Button 0, Button 1*, *Button 2*, and *Button 3*, and it places them in that order inside *left*. They should fill the whole available height of *left*.

```
// your code goes here
```

41. This code adds *left* and *top* at the left and top positions of the *BorderPane*, respectively.

```
// your code goes here
```

42. This code instantiates *labe11* and *label2* with the text *CENTER* and *BOTTOM*, respectively.

```
// your code goes here
```

43. This code adds *label1* and *label2* at the center and bottom positions within the *BorderPane*, respectively.

```
// your code goes here
```

44. Replace the anonymous class definition for a *Button* event handler with a lambda expression.

```
quit.setOnAction( new EventHandler<ActionEvent>( )
                  {
                    @Override
                    public void handle( ActionEvent event )
                    {
                      JOptionPane.showMessageDialog( null, "Goodbye" );
                      System.exit( 0 );
                    }
                  }
);
```

12.16.4 Identifying Errors in Code

45. Where is the error in this code sequence?

```
<?xml version="1.0" encoding="UTF-8"?>
<HBox
      xmlns:fx="http://javafx.com/fxml"
     alignment="center" spacing="10" >
  <Button text="Button 1" />
</HBox>
```

46. Where is the error in this code sequence?

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<HBox>
  <Label text=result />
</HBox>
```

47. Where is the error in this code sequence?

```
import java.scene.layout.*;
public class MyGame extends GridPane
{
}
```

48. Where is the error in this code sequence?

```
import javafx.event.*;
import javafx.scene.control.Button;
import javafx.scene.layout.*;
```

```
public class MyGame extends GridPane
{
  // some code here
  private class MyHandler extends EventHandler<ActionEvent>
  {
    public void handle( ActionEvent ae )
    { }
  }
}
```

49. Where is the error in this code sequence?

```
import javafx.event.*;
import javafx.scene.control.Button;
import javafx.scene.layout.*;
public class MyGame extends GridPane
{
  // some code here
  private class MyHandler implements EventHandler
  {
    public void handle( ActionEvent ae )
    { }
  }
}
```

12.16.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

50. You coded the following class:

```
import java.net.URL;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.layout.VBox;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Test50 extends Application
{
  @Override
  public void start( Stage stage )
  {
    try
    {
      URL url = getClass( ).getResource( "fxml_ex1.xml" );
      VBox root = FXMLLoader.load( url );
      Scene scene = new Scene ( root, 300, 275 );
      stage.setTitle( "Test" );
      stage.setScene( scene );
```

```
      stage.show( );
    }
    catch( Exception e )
    {
      System.out.println( e.getMessage( ) );
    }
  }

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

The code compiles; when you run, the window does not open and you get the following message:

```
Location is required
```

What do think the problem is?

51. You coded the following FXML file, whose name is *fxml_ex51.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<VBox xmlns:fx="http://javafx.com/fxml"
      alignment="center" spacing="20" >
  <Label text=Welcome />
  <Label text="FXML Test" />
</VBox>
```

Assume that the *Application* class is correctly coded. When you run, the window does not open and you get this message:

```
fxml_ex51.xml:6
```

Explain what the problem is and how to fix it.

52. You coded the following FXML file (*fxml_ex52.xml*) and *Controller52* class:

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

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<VBox fx:controller="Controller52"
      xmlns:fx="http://javafx.com/fxml"
      alignment="center" spacing="10" >
  <Button fx:id="button" text="GO"
          onAction="#go" />
</VBox>
```

```
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.*;

public class Controller52
{
  protected void go( ActionEvent event )
  {
    System.out.println( "Inside go" );
  }
}
```

Assume that the *Application* class is correctly coded. The code compiles but when you run, the window does not open and you get the following message:

```
fxml_ex52.xml:10
```

Explain what the problem is and how to fix it.

53. You coded the following in the file *Test53.java*:

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

public class Test53 extends Application
{
  public void start( Stage stage )
  {
    Ex53 root = new Ex53( );
    Scene scene = new Scene( root, 300, 275 );
    stage.setTitle( "Test" );
    stage.setScene( scene );
  }
  public static void main( String [ ] args )
  {
    launch( args );
  }
}
```

Assume that the *Ex53* class is correctly coded. The code compiles and runs, but the window does not show. Explain what the problem is and how to fix it.

54. You coded the following in the *Ex54.java* file:

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

```
public class Ex54 extends VBox
{
  private Button button;
  private Label label;

  public Ex54( )
  {
    super( );
    label = new Label( "HI" );
    button = new Button( "GO" );
    getChildren( ).add( label );
    getChildren( ).add( button );
  }

  private class ButtonHandler implements EventHandler<ActionEvent>
  {
    public void handle( ActionEvent ae )
    {
      label.setText( "Hello" );
    }
  }
}
```

Assume that the *Application* class has been correctly coded. The code compiles and runs. However, when you click the button, the text in the label does not change. Explain what the problem is and how to fix it.

12.16.6 Write a Short Program

55. Write a program that displays a text field and two buttons labeled "uppercase" and "lowercase." When the user clicks on the uppercase button, the text changes to uppercase; when the user clicks on the lowercase button, the text changes to lowercase. Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.

56. Write a program with two radio buttons and a text field. When the user clicks on one radio button, the text changes to lowercase; when the user clicks on the other radio button, the text changes to uppercase. Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.

57. Write a program with two checkboxes and a text field. When no checkbox is selected, the text is displayed in lowercase. When only the first checkbox is selected, the text is shown alternating upper/lowercase. When only the second checkbox is selected, the text is shown alternating lower/uppercase. When both checkboxes are selected, the text is shown in uppercase. Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.
58. Same as 57, except create the GUI programmatically, not using FXML.
59. Write a program that simulates a multiple choice question of your choice; the question should be a "check all that apply" type of question. There should be at least four possible answers, each using a checkbox. When the user selects any checkbox, your program should process the user's answer and show whether the answer is true (all checkbox selections are correct) or false (at least one checkbox selection is incorrect) in a label. Use FXML for the GUI and to set up event handling; include a Model.
60. Same as 59, but include a button to process the answer; do not process the answer when the user selects checkboxes.
61. Write a program that simulates a guessing game in a GUI program. Ask the user for a number between 1 and 6 in a text field, and then roll a die randomly and indicate whether or not the user won. Write the program in such a way that any invalid user input (i.e., not an integer between 1 and 6) is rejected and the user is asked again for input. Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.
62. Write a program that simulates a guessing game in a GUI program. Generate a secretrandom number between 1 and 100; that number is hidden from the user. Ask the user to guess a number between 1 and 100 in a text field, and then tell the user whether the number is too high, too low, or is the correct number. Let the user continue to guess until the correct number is guessed. Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.

63. Modify the *ComboBox* example in this chapter to use *ListView* instead.

64. Write a program that displays a 4-by-6 grid of buttons, each with some unique text. One button is the "winning" button, which your model determines randomly. When the user clicks on the winning button, change its text to "Won." If the user clicks on any other button, change its text to "No." Use code for the GUI and to set up event handling; include a Model.

65. Same as Exercise 64 with the following additions: Keep track of how many times the user clicks on buttons. If the user has not won after five clicks, the text on the last button clicked should be changed to "Lost." Once the user has lost or won, you should disable the game; that is, the buttons should no longer respond to clicks from the user.

66. Write a program that displays a combo box and a label. The combo box displays five U.S. states using two letters for each state (e.g., CA, MD). When the user selects a state from the combo box, the label is populated with the full name of the state (e.g., if the user selects MD, the label is populated with Maryland). Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.

67. Same as 66, except use code for the GUI instead of FXML.

12.16.7 Programming Projects

68. Write a GUI-based tic-tac-toe game for two players. Use code for the GUI and to set up event handling; include a Model.

69. Write a GUI-based program that analyzes a word. The user will type the word in a text field. Provide buttons for the following:

    - ❑ One button, when clicked, displays the length of the word.
    - ❑ Another button, when clicked, displays the number of vowels in the word.
    - ❑ Another button, when clicked, displays the number of upper-case letters in the word.

    Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.

70. Write a GUI-based program that analyzes a soccer game. The user will type the names of two teams and the score of the game in four text fields. You should add appropriate labels and create buttons for the following:
    - ❑ One button, when clicked, displays which team won the game.
    - ❑ Another button, when clicked, displays the game score.
    - ❑ Another button, when clicked, displays by how many goals the winning team won.

    Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.
71. Same as 70, except use code for the GUI, not FXML.
72. Write a GUI-based program that analyzes a round of golf. You will retrieve the data for 18 holes from a text file. Each line in the file will include the par for that hole (3, 4, or 5) and your score for that hole. Your program should read the file and display a combo box listing the 18 holes. When the user selects a hole, the score for that hole should be displayed in a label. Provide buttons for the following:
    - ❑ One button, when clicked, displays whether your overall score was over par, under par, or par.
    - ❑ Another button, when clicked, displays the number of holes for which you made par.
    - ❑ Another button, when clicked, displays how many birdies you scored. (A birdie on a hole is 1 under par.)

    Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.
73. Same as 72, except use code for the GUI, not FXML.
74. Write a GUI-based program that analyzes statistics for tennis players. You will retrieve the data from a text file. Each line in the file will list the name of a player, the player's number of wins for the year, and the player's number of losses for the year. Your program should read the file and display the list of players. When the user selects a player, the

winning percentage of the player should be displayed in a label. Provide buttons for the scenarios that follow:

- ❑ One button, when clicked, displays which player had the most wins for the year.
- ❑ Another button, when clicked, displays which player had the highest winning percentage for the year.
- ❑ Another button, when clicked, displays how many players had a winning record for the year.

Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.

75. Write a GUI-based program that simulates the selection of a basketball team. You will retrieve the data from a text file containing 10 lines. Each line will list the name of a player. Your program needs to read the file and display 10 checkboxes representing the 10 players. A text area will display the team, made up of the players being selected. A basketball team has five players. Your program should not allow the user to change his or her selection after the team has five players. Every time the user checks or unchecks a checkbox, the team in the text area should be updated accordingly. Provide buttons for the following:

- ❑ One button, when clicked, displays how many players are currently on the team.
- ❑ Another button, when clicked, displays how many players remain unselected.

Use code for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.

76. Write a GUI-based program that enables the user to choose a file containing an image. Your application then displays that image in a label. Your GUI should include a button and a label. When the user clicks on the button, open a file-choosing dialog box to enable the user to select a file. (Hint: look up the *FileChooser* class

in the Java Class Library.) The dialog box should show the files in the current directory with an extension of either *jpg* or *gif*.

Use FXML for the GUI and to set up event handling; include a Model.

77. Write a GUI-based program that displays a team on a soccer field. You will retrieve data from a text file containing 11 lines. Each line will contain the name of a player. Your program should read the file and display the following window when it starts. (You can assume that the players in the file are not in any particular order.) Each cell is a button; when the user clicks on a button, the button replaces its text with the name of the player.

Left wing (11)	Striker (9)	Right wing (7)
Left midfielder (6)	Midfielder (10)	Right midfielder (8)
Left defender (3)	Stopper (4) Sweeper (5)	Right defender (2)
	Goalie (1)	

Use FXML for the GUI and to set up event handling; include a Model.

78. Modify the slider example of the chapter so that it uses three sliders instead of one. Each slider represents a coefficient between 0 and 1 for the red, green, and blue amount, respectively. The sum of these three coefficients must be less than or equal to 1. The gray shade of a pixel whose RGB components are (red, green, blue) is:

```
grayShade = red * coeffRed + green * coeffGreen + blue * coeffBlue
```

Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.

79. Same as 78, but create the GUI programmatically instead.

80. Write a GUI-based program that includes three sliders and a label. The sliders are used to define the red, green, and blue components of a color that you should use for the background color of the label. Each slider represents a coefficient between 0 and 255 for the red, green, and blue amount, respectively. As the user moves a slider, the background color of the label changes.

Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.

81. Same as 80, but create the GUI programmatically instead.

82. Write a GUI-based program that generates a UNIX permission command; the UNIX permission command format is:

    ```
    chmod xyz filename
    ```

    where *x*, *y*, and *z* have values between 0 and 7

    Provide the following:

    - ❑ Three combo boxes for the permission level for all, the group, and the owner of a file. In each combo box, the user can choose the permission level, a number between 0 and 7.
    - ❑ One text field, where the user enters the name of a file.
    - ❑ A label that displays the permission command for that file based on the values of the three combo boxes. Every time the user interacts with one of the combo boxes, the label should be updated.

    Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.

83. Same as 82, except that you should update the label every time the user updates the name of the file.

84. Same as 82, but create the GUI programmatically instead of with FXML.

85. Write a GUI-based program that simulates entering the destination in a car's GPS system. The user can enter a destination by typing it in a text field or by choosing a destination from a drop-down list of previous destinations. The list of previous destinations is sorted as follows: the most recent destination is at the top and the least recent is at the bottom of the list. If the user enters a destination in a text field, the user needs to click a button to validate it. A label displays the destination selected. When a destination is either entered or selected by the user, the list of previous destinations should be updated; the current destination goes to the top of the list. There should not be any duplicate destinations in the list. In this version, the list is empty when we start the program. Data are not persistent: Every time we start the program, the list of previous destinations is empty.

    Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.

86. Same as 85, but the list of previous destinations should be persistent. It can be stored in a file and the drop-down list can be populated by the contents of the file.

12.16.8 Technical Writing

87. You are part of a team writing a complex program that includes a GUI. Our team is made up of programmers, as well as one artist and one HTML/XML developer who do not know programming but could learn some basic things quickly. We know that the GUI uses many different components and does not lend itself to using simple data structures like arrays. We know that the GUI is well defined and that the initial data in all its components are always the same. Would you define the GUI with FXML or programmatically? Discuss the pros and cons.

12.16.9 Group Project (for a group of 1, 2, or 3 students)

88. Design and code a program that simulates an auction. You should consider the following:

    A file contains a list of items to be auctioned. You can decide on the format of this file and its contents. For example, the file could look like this:

    ```
    Oldsmobile,oldsmobile.gif,100
    World Cup soccer ticket,soccerTickets.gif,50
    Trip for 2 to Rome,trip.gif,100
    ```

    In the preceding file sample, each line represents an item as follows: The first field is the item's description, the second field is the name of a file containing an image of the item, and the third field is the minimum bid. You can assume that each item's description is unique.

    Items are offered via an online-like auction. (You do not need to include any network programming; your program is a single-computer program.) Users of the program can choose which item to bid on from a list or combo box. Along with displaying the description of the item, your program should show a picture of the item and the current highest bid. (At the beginning, the current highest bid is the minimum bid.) Users bid on an item by selecting the item, typing a name (you can assume a different name), and entering a price for the item. Your

program should remember the highest bidder and the highest bid for each item by writing the information to a file. Furthermore, each time a bid is made, the item's highest bid, displayed on the screen, should be updated if necessary.

Use FXML for the GUI and to set up event handling; include a Model. Be sure that the initial state of the View matches the initial state of the Model.