Chapter 34 & 35

Introduction to JavaFX GUI Programming
&
Exploring JavaFX Controls

AWT → SWINGS → JavaFX

- AWT
  - Original Framework. Has Several Limitations
- SWINGS
  - Successful for two decades. Suitable for Enterprise Applications
- JavaFX
  - Mobile Apps
  - Java’s next generation client platform and GUI framework.
  - provides a powerful, streamlined, flexible framework that simplifies the creation of modern, visually exciting GUIs

JavaFX Basic Concepts

- JavaFX facilitates a more visually dynamic approach to GUIs

JavaFX Packages

- The JavaFX elements are contained in packages that begin with the javafx prefix.

- Examples: javafx.application, javafx.stage, javafx.scene, and javafx.scene.layout.

- Beginning with JDK 9, the JavaFX packages are organized into modules, such as javafx.base, javafx.graphics, and javafx.controls.

JavaFX Basic Concepts..

The Stage and Scene Classes

To create a JavaFX application, you will, at minimum, add at least one **Scene** object to a **Stage**.

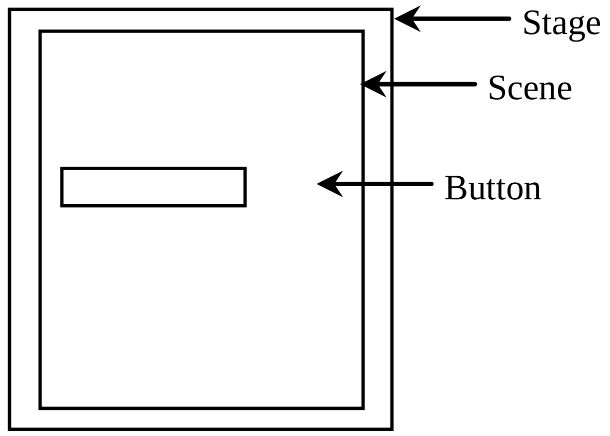
A stage is a container for scenes and a scene is a container for the items that comprise the scene. These elements are encapsulated in the JavaFX API by the **Stage** and **Scene** classes.

Stage is a top-level container. All JavaFX applications automatically have access to one **Stage**, called the *primary stage*. The primary stage is supplied by the run-time system when a JavaFX application is started. Although you can create other stages, for many applications, the primary stage will be the only one required.

As mentioned, **Scene** is a container for the items that comprise the scene. These can consist of controls, such as push buttons and check boxes, text, and graphics. To create a scene, you will add those elements to an instance of **Scene**.

Basic Structure of JavaFX

- Application
- Override the start(Stage) method
- Stage, Scene, and Nodes



JavaFX Basic Concepts..

Nodes and Scene Graphs

The individual elements of a scene are called *nodes*. For example, a push button control is a node.

However, nodes can also consist of groups of nodes. Furthermore, a node can have a child node. In this case, a node with a child is called a *parent node* or *branch node*. Nodes without children are terminal nodes and are called leaves.

The collection of all nodes in a scene creates what is referred to as a *scene graph*, which comprises a *tree*.

There is one special type of node in the scene graph, called the *root node*. This is the top-level node and is the only node in the scene graph that does not have a parent. Thus, with the exception of the root node, all other nodes have parents, and all nodes either directly or indirectly descend from the root node.

The base class for all nodes is **Node**. There are several other classes that are, either directly or indirectly, subclasses of **Node**. These include **Parent, Group, Region**, and **Control**, to name a few.

JavaFX Basic Concepts..

Layouts

- JavaFX provides several layout panes that manage the process of placing elements in a scene.
- For example, the **FlowPane** class provides a flow layout and the **GridPane** class supports a row/column grid-based layout.
- Several other layouts, such as **BorderPane** are available. The layout panes are packaged in **javafx.scene.layout**.

JavaFX Basic Concepts..

The Application Class and the Life-Cycle Methods

A JavaFX application must be a subclass of the **Application** class, which is packaged in **javafx.application**. Thus, your application class will extend **Application**.

The **Application** class defines three life-cycle methods that your application can override. These are called **init( ), start( )**, and **stop( )**, and are shown here, in the order in which they are called:

void init( )
abstract void start(Stage primaryStage)
void stop( )

The **init( )** method is called when the application begins execution. It is used to perform various initializations, however, it *cannot* be used to create a stage or build a scene. If no initializations are required, this method need not be overridden because an empty, default version is provided.

JavaFX Basic Concepts..

The Application Class and the Life-Cycle Methods..

The **start( )** method is called after **init( )**. This is where your application begins and it *can* be used to construct and set the scene. Notice that it is passed a reference to a **Stage** object. This is the stage provided by the run-time system and is the primary stage. (You can also create other stages, but you won't need to for simple applications.) Notice that this method is abstract. Thus, it must be overridden by your application.

When your application is terminated, the **stop( )** method is called. It is here that you can handle any cleanup or shutdown chores. In cases in which no such actions are needed, an empty, default version is provided.

JavaFX Basic Concepts..

Launching a JavaFX Application

To start a free-standing JavaFX application, you must call the **launch( )** method defined by **Application**. It has two forms. Here is the one used in this chapter:

```
public static void launch(String … args)
```

Here, *args* is a possibly empty list of strings that typically specify command-line arguments. When called, **launch( )** causes the application to be constructed, followed by calls to **init( )** and **start( )**. The **launch( )** method will not return until after the application has terminated. This version of **launch( )** starts the subclass of **Application** from which **launch( )** is called. The second form of **launch( )** lets you specify a class other than the enclosing class to start.

Before moving on, it is necessary to make an important point: JavaFX applications that have been packaged by using the **javafxpackager** tool (or its equivalent in an IDE) do not need to include a call to **launch( )**. However, its inclusion often simplifies the test/debug cycle, and it lets the program be used without the creation of a JAR file.

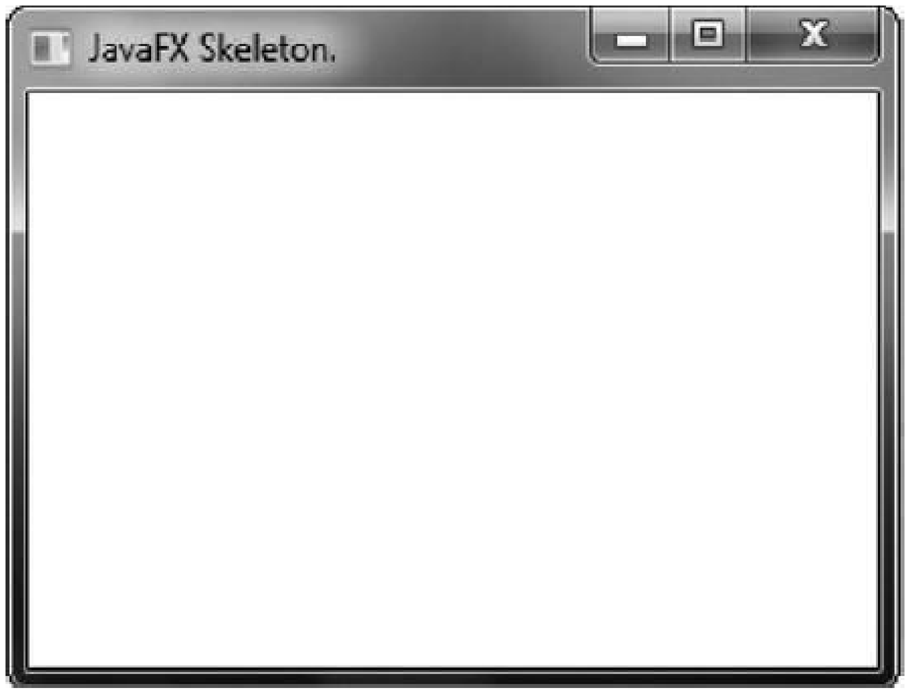
JavaFX Application Skeleton

All JavaFX applications share the same basic skeleton. Therefore, before looking at any more JavaFX features, it will be useful to see what that skeleton looks like.

In addition to showing the general form of a JavaFX application, the skeleton also illustrates how to launch the application and demonstrates when the life-cycle methods are called.

A message noting when each life-cycle method is called is displayed on the console. The complete skeleton is shown here:

```
// A JavaFX application skeleton.

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

public class JavaFXSkel extends Application {

  public static void main(String[] args) {

    System.out.println("Launching JavaFX application.");

    // Start the JavaFX application by calling launch().
    launch(args);
  }
  // Override the init() method.
  public void init() {
    System.out.println("Inside the init() method.");
  }

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

    System.out.println("Inside the start() method.");

    // Give the stage a title.
    myStage.setTitle("JavaFX Skeleton.");

    // Create a root node. In this case, a flow layout pan
    // is used, but several alternatives exist.
    FlowPane rootNode = new FlowPane();

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

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

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

  }

  // Override the stop() method.
  public void stop() {
    System.out.println("Inside the stop() method.");
  }
}
```



It also produces the following output on the console:

```
Launching JavaFX application.
Inside the init() method.
Inside the start() method.
```

When you close the window, this message is displayed on the console:

```
Inside the stop() method.
```

The Application Thread

init( ) method cannot be used to construct a stage or scene. You also cannot create these items inside the application's constructor. The reason is that a stage or scene must be constructed on the *application thread*.

However, the application's constructor and the **init( )** method are called on the main thread, also called the *launcher thread*. Thus, they can't be used to construct a stage or scene.

Instead, you must use the **start( )** method, as the skeleton demonstrates, to create the initial GUI because **start( )** is called on the application thread.

Furthermore, any changes to the GUI currently displayed must be made from the application thread. Fortunately, in JavaFX, events are sent to your program on the application thread. Therefore, event handlers can be used to interact with the GUI.

The **stop( )** method is also called on the application thread.

A Simple JavaFX Control: Label

The primary ingredient in most user interfaces is the control because a control enables the user to interact with the application. JavaFX supplies a rich assortment of controls.

The simplest control is the label because it just displays a message, which, in this example, is text. Although quite easy to use, the label is a good way to introduce the techniques needed to begin building a scene graph.

The JavaFX label is an instance of the **Label** class, which is packaged in **javafx.scene.control**. **Label** inherits **Labeled** and **Control**, among other classes.

The **Labeled** class defines several features that are common to all labeled elements (that is, those that can contain text), and **Control** defines features related to all controls.

A Simple JavaFX Control: Label

Label defines three constructors. The one we will use here is

Label(String *str*)

Here, *str* is the string that is displayed.

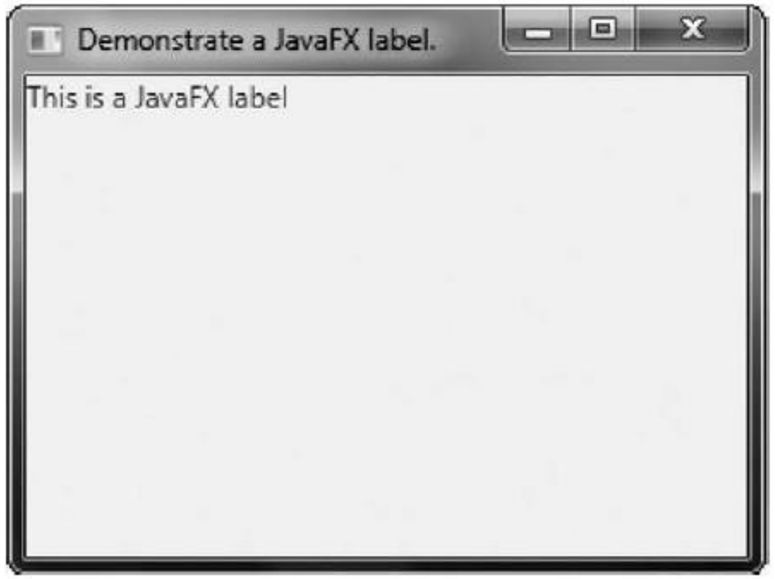
Once you have created a label (or any other control), it must be added to the scene's content, which means adding it to the scene graph. To do this, you will first call **getChildren( )** on the root node of the scene graph. It returns a list of the child nodes in the form of an **ObservableList<Node>**.

ObservableList is packaged in **javafx.collections**, and it inherits **java.util.List**, which means that it supports all of the features available to a list as defined by the Collections Framework. Using the returned list of child nodes, you can add the label to the list by calling **add( )**, passing in a reference to the label.

```
// Demonstrate a JavaFX label.

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

public class JavaFXLabelDemo extends Application {

  public static void main(String[] args) {

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

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

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

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

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

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

        // Create a label.
        Label myLabel = new Label("This is a JavaFX label");

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

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

Output:



This is a JavaFX label

In the program, pay special attention to this line:

```
rootNode.getChildren().add(myLabel);
```

It adds the label to the list of children for which **rootNode** is the parent. Although this line could be separated into its individual pieces if necessary, you will often see it as shown here.

It is useful to point out that **ObservableList** provides a method called **addAll( )** that can be used to add two or more children to the scene graph in a single call. (You will see an example of this shortly.)

To remove a control from the scene graph, call **remove( )** on the **ObservableList**. For example,

```
rootNode.getChildren().remove(myLabel);
```

removes **myLabel** from the scene.

https://openjfx.io/

Compile: javac --module-path “PATH to your javafx lib folder”--add-modules javafx.controls,javafx.fxml JavaFXSkel.java

Run: Java --module-path C:\javafx-sdk-17.0.1\lib --add-modules javafx.controls,javafx.fxml JavaFXSkel

Using Button and Events

Although the program in the preceding section presents a simple example of using a JavaFX control and constructing a scene graph, it does not show how to handle **events**.

As you know, most GUI controls generate events that are handled by your program. For example, buttons, check boxes, and lists all generate events when they are used.

One commonly used control is the button. This makes button events one of the most frequently handled. Therefore, a button is a good way to demonstrate the fundamentals of event handling in JavaFX. For this reason, the fundamentals of event handling and the button are introduced together.

Using Buttons and Events..

Event Basics

The base class for JavaFX events is the **Event** class, which is packaged in **javafx.event**. **Event** inherits **java.util.EventObject**, which means that JavaFX events share the same basic functionality as other Java events.

Several subclasses of **Event** are defined. The one that we will use here is **ActionEvent**. It handles action events generated by a button.

In general, JavaFX uses what is, in essence, the delegation event model approach to event handling. To handle an event, you must first register the handler that acts as a listener for the event. When the event occurs, the listener is called. It must then respond to the event and return.

Events are handled by implementing the **EventHandler** interface, which is also in **javafx.event**. It is a generic interface with the following form:

```
interface EventHandler<T extends Event>
```

Here, **T** specifies the type of event that the handler will handle. It defines one method, called **handle( )**, which receives the event object as a parameter.

Using Buttons and Events..

Event Basics (contd.…)

It is shown here:

void handle(T *eventObj*)

Here, *eventObj* is the event that was generated. Typically, event handlers are implemented through anonymous inner classes or lambda expressions, but you can use stand-alone classes for this purpose if it is more appropriate to your application (for example, if one event handler will handle events from more than one source).

Although not required by the examples in this chapter, it is sometimes useful to know the source of an event. This is especially true if you are using one handler to handle events from different sources. You can obtain the source of the event by calling **getSource( )**, which is inherited from **java.util.EventObject**. It is shown here:

Object getSource( )

Using Buttons and Events..

Event Basics (contd….)

Other methods in **Event** let you obtain the event type, determine if the event has been consumed, consume an event, fire an event, and obtain the target of the event. When an event is consumed, it stops the event from being passed to a parent handler.

One last point: In JavaFX, events are processed via an *event dispatch chain*.

When an event is generated, it is passed to the root node of the chain. The event is then passed down the chain to the target of the event. After the target node processes the event, the event is passed back up the chain, thus allowing parent nodes a chance to process the event, if necessary. This is called *event bubbling*. It is possible for a node in the chain to consume an event, which prevents it from being further processed.

Using Buttons and Events..

Introducing Button Control

In JavaFX, the push button control is provided by the **Button** class, which is in **javafx.scene.control**. **Button** inherits a fairly long list of base classes that include **ButtonBase, Labeled, Region, Control, Parent**, and **Node**. If you examine the API documentation for **Button**, you will see that much of its functionality comes from its base classes. Furthermore, it supports a wide array of options.

However, here we will use its default form. Buttons can contain text, graphics, or both. In this chapter, we will use text-based buttons. An example of a graphics-based button is shown in the next chapter.

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

Button(String *str*)

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

Using Buttons and Events..

Introducing Button Control..

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

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

Here, *handler* is the handler being registered. As mentioned, often you will use an anonymous inner class or lambda expression for the handler. The **setOnAction( )** method sets the property **onAction**, which stores a reference to the handler.

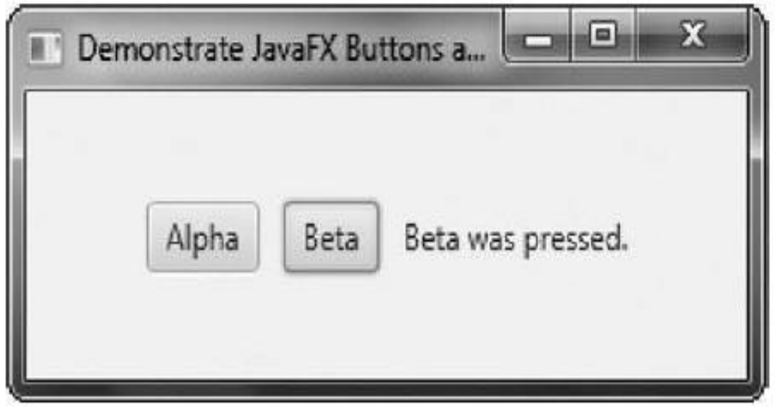
As with all other Java event handling, your handler must respond to the event as fast as possible and then return. If your handler consumes too much time, it will noticeably slow down the application. For lengthy operations, you must use a separate thread of execution.

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

public class JavaFXEventDemo extends Application {

  Label response;

  public static void main(String[] args) {

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

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

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

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

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

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

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

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

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

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

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

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

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

- Sample Output



Beta was pressed.

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

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

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

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

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

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

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

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

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

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

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

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

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

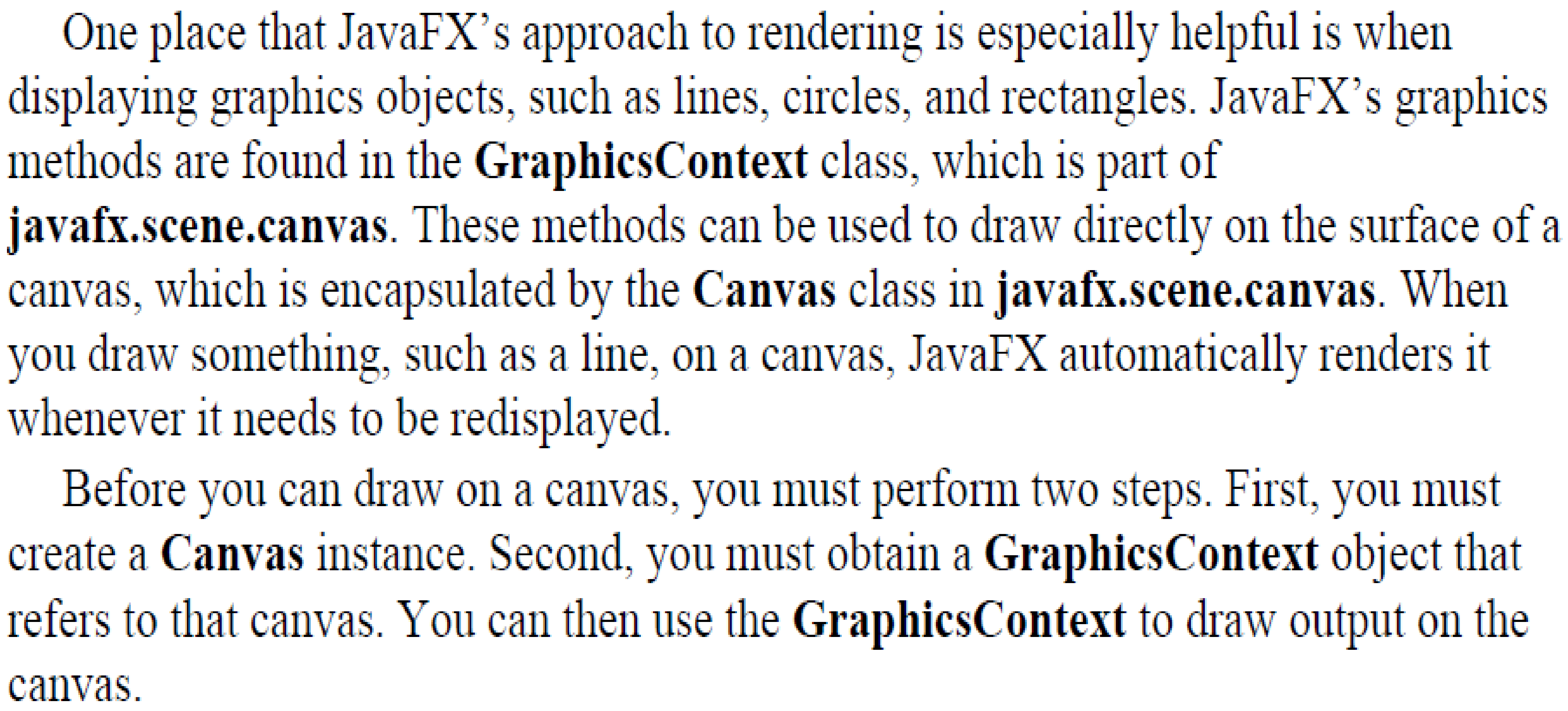
```
rootNode.setAlignment(Pos.CENTER);
```

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

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

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

Drawing Directly on a Canvas



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

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

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

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

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

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

GraphicsContext getGraphicsContext2D( )

The graphics context for the canvas is returned.

GraphicsContext defines a large number of methods that draw shapes, text, and images, and support effects and transforms.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
Font(double fontSize)
```

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

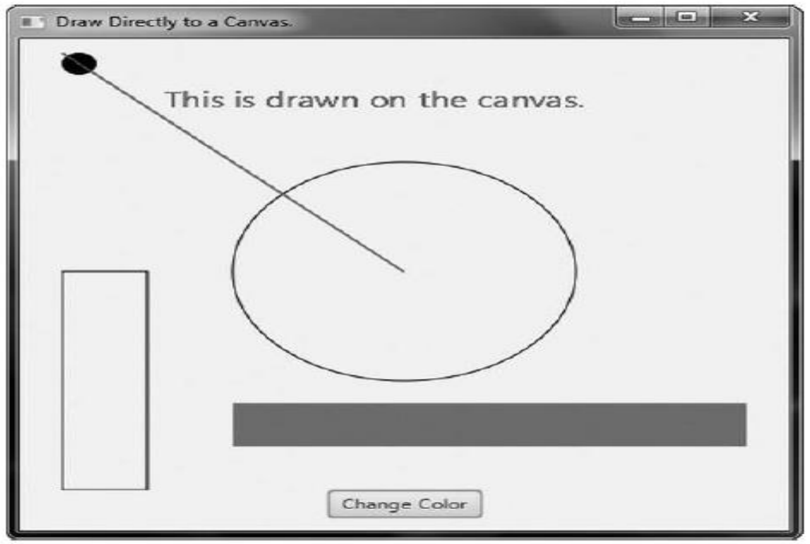
You can specify the fill and stroke using these two methods defined by **GrahpicsContext**:

```
void setFill(Paint newFill)
void setStroke(Paint newStroke)
```

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

Example program to demonstrate Drawing

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



```
// Demonstrate drawing.

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

public class DirectDrawDemo extends Application {

  GraphicsContext gc;

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

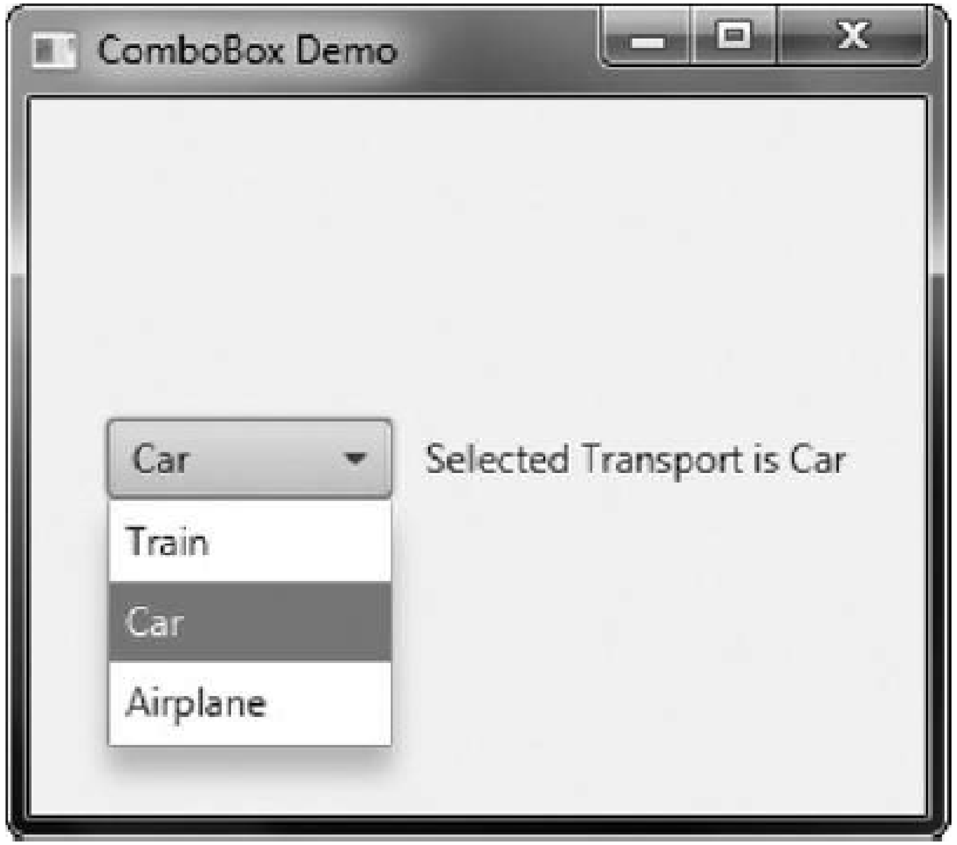
```
public static void main(String[] args) {

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

JavaFX ComboBox

ComboBox



Selected Transport is Car

ComboBox

- A combo box displays one selection, but it will also display a drop-down list that allows the user to select a different item.

- **ComboBox** inherits **ComboBoxBase** which provides much of its functionality.

- Unlike the **ListView**, which can allow multiple selections, **ComboBox** is designed for single-selection.

- **ComboBox** is a generic class that is declared like this:

  ```
  class ComboBox<T>
  ```

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

ComboBox constructors

- The default constructor creates an empty **ComboBox**.

- ComboBox (ObservableList<T>*list*)

  list specifies a list of the items that will be displayed

- list is an object of type **ObservableList**, which defines a list of observable objects.

- **ObservableList** inherits **java.util.List**.

- An easy way to create an **ObservableList** is to use the factory method **observableArrayList( )**, which is a static method defined by the **FXCollections** class.

Events Generated by Combobox

- A **ComboBox** generates an action event when its selection changes
- It will also generate a change event
- Alternatively, it is also possible to ignore events and simply obtain the current selection when needed
- To obtain the current selection call **getValue( )**, shown here:

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

  final void setValue(T *newVal*)

```
// Demonstrate a combo box.

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


public class ComboBoxDemo extends Application {

  ComboBox<String> cbTransport;
  Label response;

  public static void main(String[] args) {

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

ToggleButton

A toggle button looks just like a push button, but it acts differently because it has two states: **pushed and released.**

That is, when you press a toggle button, it stays pressed rather than popping back up as a regular push button does. When you press the toggle button a second time, it releases (pops up).

Therefore, each time a toggle button is pushed, it toggles between these two states.

In JavaFX, a toggle button is encapsulated in the **ToggleButton** class. Like **Button**, **ToggleButton** is also derived from **ButtonBase**. It implements the **Toggle** interface, which defines functionality common to all types of two-state buttons.

ToggleButton defines three constructors.

ToggleButton(String str)

here, *str* is the text displayed in the button.

When the button is pressed, the option is selected. When the button is released, the option is deselected. For this reason, a program usually needs to determine the toggle button's state.

To do this, use the **isSelected( )** method, shown here:

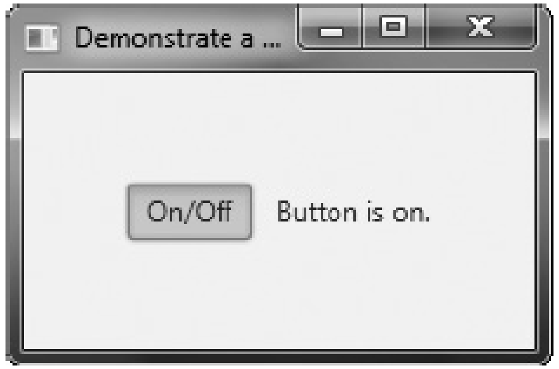
final boolean isSelected( )

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

```
// Demonstrate a toggle button.
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
public class ToggleButtonDemo extends Application {
    ToggleButton tbOnOff;
    Label response;
    public static void main(String[] args) {
    // Start the JavaFX application by calling launch().
        launch(args);
  }
// Override the start() method.
```

```
public void start(Stage myStage) {
     // Give the stage a title.
     myStage.setTitle("Demonstrate a Toggle Button");
     // Use a FlowPane for the root node. In this case,
     // vertical and horizontal gaps of 10.
     FlowPane rootNode = new FlowPane(10, 10);
     // Center the controls in the scene.
     rootNode.setAlignment(Pos.CENTER);
     // Create a scene.
     Scene myScene = new Scene(rootNode, 220, 120);
     // Set the scene on the stage.
     myStage.setScene(myScene);
     // Create a label.
     response = new Label("Push the Button.");
     // Create the toggle button.
     tbOnOff = new ToggleButton("On/Off");
```

```
        // Handle action events for the toggle button.
        tbOnOff.setOnAction(new EventHandler<ActionEvent>() {
            public void handle(ActionEvent ae) {
                 if(tbOnOff.isSelected())
response.setText("Button is on.");
                 else response.setText("Button is off.");
             }
         });
         // Add the label and buttons to the scene graph.
         rootNode.getChildren().addAll(tbOnOff, response);
         // Show the stage and its scene.
        myStage.show();
    }
}
```



Button is on.

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

RadioButton

Another type of button provided by JavaFX is the *radio button*.

Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any time. They are supported by the **RadioButton** class, which extends both **ButtonBase** and **ToggleButton**. It also implements the **Toggle** interface.

Thus, a radio button is a specialized form of a toggle button.

They are the primary control employed when the user must select only one option among several alternatives.

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

RadioButton(String str)

here, *str* is the label for the button.

Like other buttons, when a **RadioButton** is used, an action event is generated.

Radio buttons must be configured into a group. Only one of the buttons in the group can be selected at any time.

A button group is created by the **ToggleGroup** class, which is packaged in **javafx.scene.control**. **ToggleGroup** provides only a default constructor.

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

final void setToggleGroup(ToggleGroup tg)

here, *tg* is a reference to the toggle button group to which the button is added.

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

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

final void setSelected(boolean state)

If *state* is **true**, the button is selected. Otherwise, it is deselected.

Although the button is selected, no action event is generated.

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

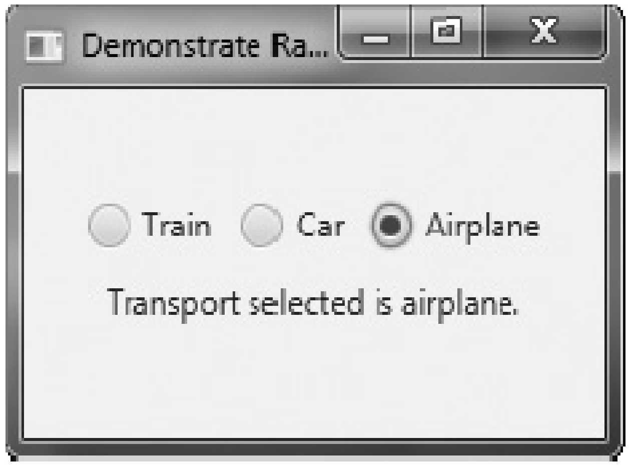
This method results in an action event being generated for the button if the button was previously not selected.

```
/* This program responds to the action events generated by a radio button selection. It also
shows how to fire the button under program control. */
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
public class RadioButtonDemo extends Application {
    Label response;
    public static void main(String[] args) {
    // Start the JavaFX application by calling launch().
        launch(args);
     }
   // Override the start() method.
```

```
public void start(Stage myStage) {
    // Give the stage a title.
    myStage.setTitle("Demonstrate Radio Buttons");
    // Use a FlowPane for the root node. In this case,
    // vertical and horizontal gaps of 10.
    FlowPane rootNode = new FlowPane(10, 10);
    // Center the controls in the scene.
    rootNode.setAlignment(Pos.CENTER);
    // Create a scene.
    Scene myScene = new Scene(rootNode, 220, 120);
     // Set the scene on the stage.
    myStage.setScene(myScene);
    // Create a label that will report the selection.
    response = new Label("");
    // Create the radio buttons.
    RadioButton rbTrain = new RadioButton("Train");
    RadioButton rbCar = new RadioButton("Car");
    RadioButton rbPlane = new RadioButton("Airplane");
```

```
// Create a toggle group.
ToggleGroup tg = new ToggleGroup();
// Add each button to a toggle group.
rbTrain.setToggleGroup(tg);
rbCar.setToggleGroup(tg);
rbPlane.setToggleGroup(tg);
// Handle action events for the radio buttons.
rbTrain.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
          response.setText("Transport selected is train.");
      }
 });
rbCar.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
          response.setText("Transport selected is car.");
       }
  });
```

```
rbPlane.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        response.setText("Transport selected is airplane.");
     }
 });
// Fire the event for the first selection. This causes that radio
//button to be selected and an action event for that button to occur.
rbTrain.fire();
// Add the label and buttons to the scene graph.
rootNode.getChildren().addAll(rbTrain, rbCar, rbPlane, response);
// Show the stage and its scene.
myStage.show();
}
}
```



Transport selected is airplane.

CheckBox

In JavaFX, the check box is encapsulated by the **CheckBox** class. Its immediate superclass is **ButtonBase**. Thus it is a special type of button.

CheckBox supports three states. The first two are checked or unchecked, as you would expect, and this is the default behavior. The third state is *indeterminate* (also called *undefined*).

Here is the **CheckBox** constructor that we will use:

CheckBox(String str)

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

The following program demonstrates check boxes. It displays four check boxes that represent different types of computers. They are labeled Smartphone, Tablet, Notebook, and Desktop. Each time a check-box state changes, an action event is generated. It is handled by displaying the new state (selected or cleared) and by displaying a list of all selected boxes.

```
// Demonstrate Check Boxes.
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
public class CheckboxDemo extends Application {
    CheckBox cbSmartphone;
    CheckBox cbTablet;
    CheckBox cbNotebook;
    CheckBox cbDesktop;
    Label response;
    Label selected;
    String computers;
```

```
public static void main(String[] args) {
   // Start the JavaFX application by calling launch().
   launch(args);
 }
// Override the start() method.
public void start(Stage myStage) {
    // Give the stage a title.
    myStage.setTitle("Demonstrate Check Boxes");
   // Use a vertical FlowPane for the root node. In this case,
   // vertical and horizontal gaps of 10.
   FlowPane rootNode = new FlowPane(Orientation.VERTICAL, 10, 10);
   // Center the controls in the scene.
   rootNode.setAlignment(Pos.CENTER);
   // Create a scene.
   Scene myScene = new Scene(rootNode, 230, 200);
   // Set the scene on the stage.
   myStage.setScene(myScene);
   Label heading = new Label("What Computers Do You Own?");
```

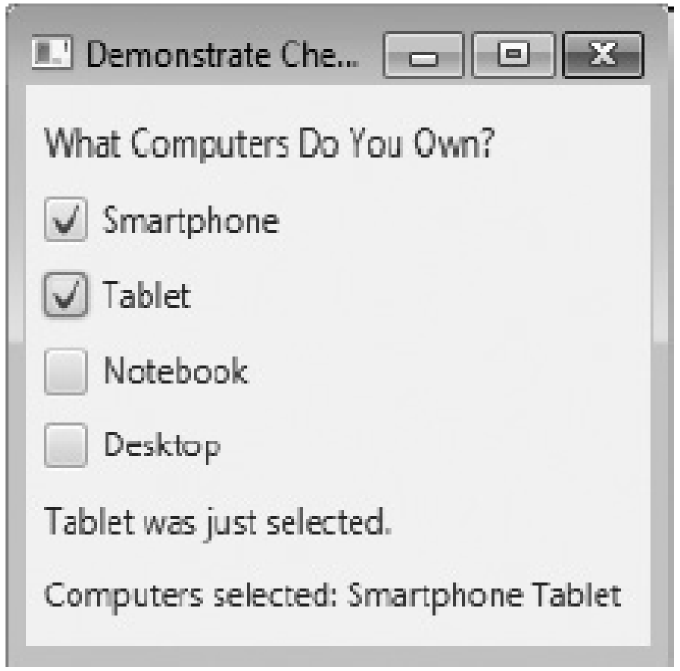
```
// Create a label that will report the state change of a check box.
response = new Label("");
// Create a label that will report all selected check boxes.
selected = new Label("");
// Create the check boxes.
cbSmartphone = new CheckBox("Smartphone");
cbTablet = new CheckBox("Tablet");
cbNotebook = new CheckBox("Notebook");
cbDesktop = new CheckBox("Desktop");
// Handle action events for the check boxes.
cbSmartphone.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        if(cbSmartphone.isSelected())
            response.setText("Smartphone was just selected.");
        else
            response.setText("Smartphone was just cleared.");
        showAll();
     }       });
```

```
cbTablet.setOnAction(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent ae) {
             if(cbTablet.isSelected())
                   response.setText("Tablet was just selected.");
            else
                   response.setText("Tablet was just cleared.");
            showAll();
         }
  });
cbNotebook.setOnAction(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent ae) {
              if(cbNotebook.isSelected())
                    response.setText("Notebook was just selected.");
              else
                    response.setText("Notebook was just cleared.");
             showAll();
         }
 });
```

```
cbDesktop.setOnAction(new EventHandler<ActionEvent>() {
      public void handle(ActionEvent ae) {
           if(cbDesktop.isSelected())
                response.setText("Desktop was just selected.");
          else
                response.setText("Desktop was just cleared.");
          showAll();
       }
 });
// Add controls to the scene graph.
rootNode.getChildren().addAll(heading, cbSmartphone, cbTablet,
                                                    cbNotebook, cbDesktop, response, selected);
// Show the stage and its scene.
myStage.show();
showAll();
}
```

```
// Update and show the selections.
void showAll() {
    computers = "";
    if(cbSmartphone.isSelected())
        computers = "Smartphone ";
    if(cbTablet.isSelected())
        computers += "Tablet ";
    if(cbNotebook.isSelected())
        computers += "Notebook ";
    if(cbDesktop.isSelected())
        computers += "Desktop";
    selected.setText("Computers selected: " +
computers);
  }
}
```

Sample output is shown here:



What Computers Do You Own?

✓ Smartphone

✓ Tablet

Notebook

Desktop

Tablet was just selected.

Computers selected: Smartphone Tablet

Text Field

JavaFX includes several text-based controls. The one we will look at is **TextField**. It allows one line of text to be entered. Thus, it is useful for obtaining names, ID strings, addresses, and the like.

Like all text controls, **TextField** inherits **TextInputControl**, which defines much of its functionality.

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

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

final void setPrefColumnCount(int *columns*)

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

You can set the text in a text field by calling **setText( )**. You can obtain the current text by calling **getText( )**. In addition to these fundamental operations,

TextField supports several other capabilities that you might want to explore, such as cut, paste, and append. You can also select a portion of the text under program control.

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

final void setPromptText(String *str*)

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

When the user presses enter while inside a **TextField**, an action event is generated. Although handling this event is often helpful, in some cases,

The program will simply obtain the text when it is needed, rather than handling action events.

Both approaches are demonstrated by the following program. It creates a text field that requests a search string. When the user presses enter while the text field has input focus, or presses the Get Search String button, the string is obtained and displayed. Notice that a prompting message is also included.

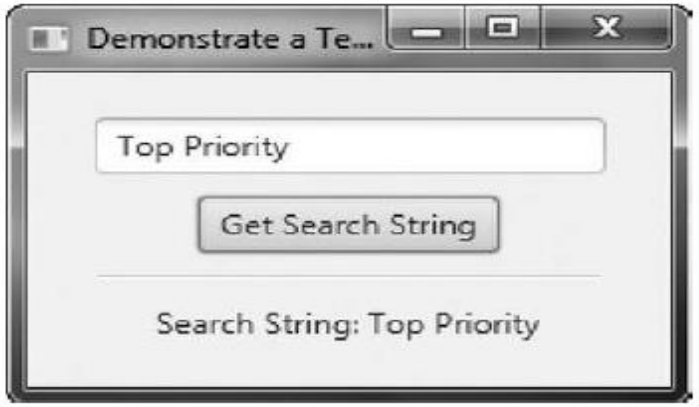
```
// Demonstrate a text field.

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

public class TextFieldDemo extends Application {

  TextField tf;
  Label response;

  public static void main(String[] args) {

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Sample Output:



Search String: Top Priority

ListView

A **ListView** can display a list of entries from which you can select one or more. Scrollbars are automatically added when the number of items in the list exceeds the number that can be displayed within the control's dimensions. **ListView** is a generic class that is declared like this:

class ListView<T>

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

Here is the **ListView** constructor that we will use:
ListView(ObservableList<T> list)

The list of items to be displayed is specified by *list*. It is an object of type **ObservableList**. **ObservableList** supports a list of objects.

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

To create an ObservableList for use in a ListView use the factory method observableArrayList( ), which is a static method defined by the FXCollections class (which is packaged in javafx.collections).

static <E> ObservableList<E> observableArrayList(E ... elements)
In this case, E specifies the type of elements, which are passed via elements.

to set the preferred height and/or width, size:
final void setPrefHeight(double height)
final void setPrefWidth(double width)
void setPrefSize(double width, double height)

you can monitor the list for changes by registering a change listener. This lets you respond each time the user changes a selection in the list. A change listener is supported by the ChangeListener interface, which is packaged in javafx.beans.value. The ChangeListener interface defines only one method, called **changed( ).** It is shown here:
void changed(ObservableValue<? extends T> changed, T oldVal, T newVal)

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

To listen for change events, you must first obtain the selection model used by the ListView. This is done by calling getSelectionModel( ) on the list. It is shown here:
final MultipleSelectionModel<T> getSelectionModel( )
It returns a reference to the model.

Using the model returned by getSelectionModel( ), you will obtain a reference to the
selected item property that defines what takes place when an element in the list is selected.

This is done by calling selectedItemProperty( ), shown next:
final ReadOnlyObjectProperty<T> selectedItemProperty( )

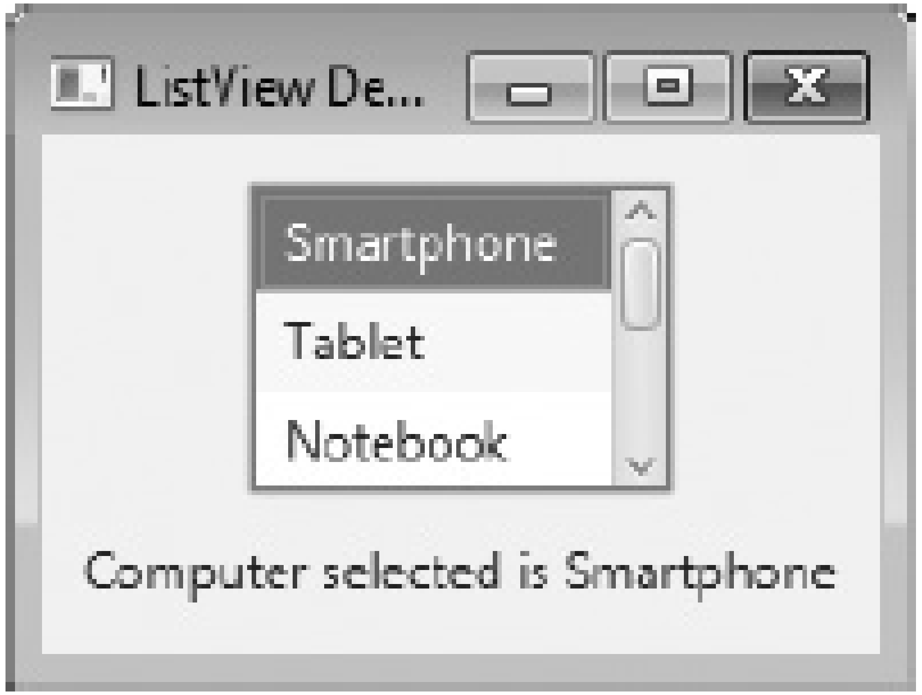
You will add the change listener to this property by using the addListener( ) method on the returned property.
The addListener( ) method is shown here:
void addListener(ChangeListener<? super T> listener)
In this case, T specifies the type of the property.

The following example creates a list view that displays a list of computer types, allowing the user to select one.
When one is chosen, the selection is displayed.

```
// Demonstrate a list view.
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.geometry.*;
import javafx.beans.value.*;
import javafx.collections.*;
public class ListViewDemo extends Application {
    Label response;
    public static void main(String[] args) {
        // Start the JavaFX application by calling launch().
        launch(args);
     }
    // Override the start() method.
```

```
public void start(Stage myStage) {
    // Give the stage a title.
    myStage.setTitle("ListView Demo");
    // Use a FlowPane for the root node. In this case,
    // vertical and horizontal gaps of 10.
    FlowPane rootNode = new FlowPane(10, 10);
    // Center the controls in the scene.
    rootNode.setAlignment(Pos.CENTER);
    // Create a scene.
    Scene myScene = new Scene(rootNode, 200, 120);
    // Set the scene on the stage.
    myStage.setScene(myScene);
    // Create a label.
    response = new Label("Select Computer Type");
    // Create an ObservableList of entries for the list view.
    ObservableList<String> computerTypes =
    FXCollections.observableArrayList("Smartphone", "Tablet", "Notebook", "Desktop" );
// Create the list view.
ListView<String> lvComputers = new ListView<String>(computerTypes);
// Set the preferred height and width.
lvComputers.setPrefSize(100, 70);
// Get the list view selection model.
MultipleSelectionModel<String> lvSelModel = lvComputers.getSelectionModel();
// Use a change listener to respond to a change of selection within  a list view.
lvSelModel.selectedItemProperty().addListener( new ChangeListener<String>() {
        public void changed(ObservableValue<? extends String> changed, String oldVal, String newVal) {
            // Display the selection.
            response.setText("Computer selected is " + newVal);
          }
   });
// Add the label and list view to the scene graph.
rootNode.getChildren().addAll(lvComputers, response);
 // Show the stage and its scene.
      myStage.show();
    }
}
```

Sample output is shown here.
Notice that a vertical scroll bar has been included so that the list can be scrolled to see all of its entries.



Computer selected is Smartphone