**Introducing GUI Programming with JavaFX**

JavaFX is one of the Java’s client platforms and GUI framework. JavaFX provides a powerful, streamlined, flexible framework that simplifies the creation of modern, visually exciting GUIs.

The original JavaFX was based on a scripting language called *JavaFX Script*. However, JavaFX Script has been discontinued. Beginning with the release of JavaFX 2.0, JavaFX has been programmed in Java itself and provides a comprehensive API. JavaFX also supports FXML, which can be (but is not required to be) used to specify the user interface.

**JavaFX Basic Concepts**

* In general, the JavaFX framework has all of the good features of Swing. For example, JavaFX is **lightweight**. It can also support an **MVC (Model View Controller) architecture**.
* JavaFX offers a more streamlined, easier-to-use, updated approach.
* JavaFX also greatly simplifies the rendering of objects because it handles repainting automatically. It is no longer necessary for your program to handle this task manually.
* JavaFX facilitates a more visually dynamic approach to GUIs.

**The JavaFX Packages**

The JavaFX elements are contained in packages that begin with the **javafx** prefix. At the time of this writing, there are more than 30 JavaFX packages in its API library. Here are four examples: **javafx.application**, **javafx.stage**, **javafx.scene**, and **javafx.scene.layout**. JavaFX offers a wide array of functionality.

**The Stage and Scene Classes**

The central metaphor implemented by JavaFX is the *stage*. As in the case of an actual stage play, a stage contains a *scene*. A stage defines a space and a scene defines what goes in that space. Or, put another way, a stage is a container for scenes and a scene is a container for the items that comprise the scene. As a result, all JavaFX applications have at least one stage and one scene. These elements are encapsulated in the JavaFX API by the **Stage** and **Scene** classes.

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

**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.

**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**.

**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. 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.

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.

**Launching a JavaFX Application**

To start a free-standing JavaFX application, you must call the **launch( )** method defined by **Application**.

**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.

**A JavaFX Application Skeleton**

All JavaFX applications share the same basic skeleton. 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 pane 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.");

}

}

Although the skeleton is quite short, it can be compiled and run. It produces the

window shown here:



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.

In a real program, the life-cycle methods would not normally output anything to **System.out**. They do so here simply to illustrate when each method is called. Furthermore, you will need to override the **init( )** and **stop( )** methods only if your application must perform special startup or shutdown actions. Otherwise, you can use the default implementations of these methods provided by the **Application** class.

Let’s examine this program in detail. It begins by importing four packages. The first is **javafx.application**, which contains the **Application** class. The **Scene** class is packaged in **javafx.scene**, and **Stage** is packaged in **javafx.stage**. The **javafx.scene.layout** package provides several layout panes. The one used by the program is **FlowPane**. Next, the application class **JavaFXSkel** is created. Notice that it extends **Application**. As explained, **Application** is the class from which all JavaFX applications are derived. **JavaFXSkel** contains two methods. The first is **main( )**. It is used to launch the application via a call to **launch( )**. Notice that the **args** parameter to **main( )** is passed to the **launch( )** method. Although this is a common approach, you can pass a different set of parameters to **launch( )**, or none at all. One other point: As explained earlier, **launch( )** is required by a free-standing application, but not in other cases. When it is not needed, **main( )** is also not needed. However, for reasons already explained, both **main( )** and **launch( )** are included in the JavaFX programs in this book.

When the application begins, the **init( )** method is called first by the JavaFX run-time system. For the sake of illustration, it simply displays a message on **System.out**, but it would normally be used to initialize some aspect of the application. Of course, if no initialization is required, it is not necessary to override **init( )** because an empty, default implementation is provided. It is important to emphasize that **init( )** cannot be used to create the stage or scene portions of a GUI. Rather, these items should be constructed and displayed by the **start( )** method.

After **init( )** finishes, the **start( )** method executes. It is here that the initial scene is created and set to the primary stage. Let’s look at this method line-by-line. First, notice that **start( )** has a parameter of type **Stage**. When **start( )** is called, this parameter will receive a reference to the primary stage of the application. It is to this stage that you will set a scene for the application. After displaying a message on the console that **start( )** has begun execution, it sets the title of the stage using this call to **setTitle( )**:

myStage.setTitle("JavaFX Skeleton.");

Although this step is not necessarily required, it is customary for stand-alone applications. This title becomes the name of the main application window. Next, a root node for a scene is created. The root node is the only node in a scene graph that does not have a parent. In this case, a **FlowPane** is used for the root node, but there are several other classes that can be used for the root.

FlowPane rootNode = new FlowPane();

As mentioned, a **FlowPane** is a layout manager that uses a flow layout. This is a layout in which elements are positioned line-by-line, with lines wrapping as needed. (Thus, it works much like the **FlowLayout** class used by the AWT and Swing.) In this case, a horizontal flow is used, but it is possible to specify a vertical flow. Although not needed by this skeletal application, it is also possible to specify other layout properties, such as a vertical and horizontal gap between elements, and an alignment. You will see an example of this later in this chapter.

The following line uses the root node to construct a **Scene**:

Scene myScene = new Scene(rootNode, 300, 200);

**Scene** provides several versions of its constructor. The one used here creates a scene that has the specified root with the specified width and height. It is shown here:

Scene(Parent *rootnode*, double *width*, double *height*)

Notice that the type of *rootnode* is **Parent**. It is a subclass of **Node** and encapsulates nodes that can have children. Also notice that the width and the height are **double** values. This lets you pass fractional values, if needed. In the skeleton, the root is **rootNode**, the width is 300 and the height is 200.

The next line in the program sets **myScene** as the scene for **myStage**:

myStage.setScene(myScene);

Here, **setScene( )** is a method defined by **Stage** that sets the scene to that specified by its argument. In cases in which you don’t make further use of the scene, you can combine the previous two steps, as shown here:

myStage.setScene(new Scene(rootNode, 300, 200));

Because of its compactness, this form will be used by most of the subsequent examples. The last line in **start( )** displays the stage and its scene:

myStage.show();

In essence, **show( )** shows the window that was created by the stage and screen. When you close the application, its window is removed from the screen and the **stop( )** method is called by the JavaFX run-time system. In this case, **stop( )** simply displays a message on the console, illustrating when it is called. However, **stop( )** would not normally display anything. Furthermore, if your application does not need to handle any shutdown actions, there is no reason to override **stop( )** because an empty, default implementation is provided.

**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 set of controls. The simplest control is the label because it just displays a message, which, in this example, is text.

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.

**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.

The following program puts the preceding discussion into action by creating a simple JavaFX application that displays a 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();

}

}

This program produces the following window:

****

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

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

removes **myLabel** from the scene.

**Using Buttons and Events**

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.

**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 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. 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( )

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.

**NOTE** Although not used in this introduction to JavaFX, an application can also implement an *event filter,* which can be used to manage events. A filter is added to a node by calling addEventFilter( ), which is defined by Node. A filter can consume an event, thus preventing further processing.

**Introducing the 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.

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

**Button(String *str*)**

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

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

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

**Demonstrating Event Handling and the Button**

The following program demonstrates event handling. It uses two buttons and a label. Each

time a button is pressed, the label is set to display which button was pressed.

package application;

//Demonstrate JavaFX events and buttons.

import javafx.application.\*;

import javafx.scene.\*;

import javafx.stage.\*;

import javafx.scene.layout.\*;

import javafx.scene.control.\*;

import javafx.event.\*;

import javafx.geometry.\*;

public class javaFXEvent 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 using Anonymous Inner Class Implementation.

btnAlpha.setOnAction(new EventHandler<ActionEvent>() {

public void handle(ActionEvent ae) {

response.setText("Alpha was pressed.");

}

});

//Handle the action events for the Beta button using Lambda expression .

btnBeta.setOnAction((ae) ->

response.setText("Beta was pressed.")

);

//Add the label and buttons to the scene graph.

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

//Show the stage and its scene.

myStage.show();

}

}Sample output from this program is shown here:

****

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

Button btnAlpha = new Button("Alpha");

Button btnBeta = new Button("Beta");

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

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

// Handle the action events for the Alpha button.

btnAlpha.setOnAction(new EventHandler<ActionEvent>() {

public void handle(ActionEvent ae) {

response.setText("Alpha was pressed.");

}

});

As explained, buttons respond to events of type **ActionEvent**. To register a handler for these events, the **setOnAction( )** method is called on the button. It uses an anonymous inner class to implement the **EventHandler** interface. (Recall that **EventHandler** defines only the **handle( )** method.) Inside **handle( )**, the text in the **response** label is set to reflect the fact that the Alpha button was pressed. Notice that this is done by calling the **setText( )** method on the label. Events are handled by the Beta button in the same way. Note that **response** is declared as a field within **FXEventDemo**, rather than as a local variable. This is because it is accessed within the button event handlers, which are anonymous inner classes. After the event handlers have been set, the **response** label and the buttons **btnAlpha** and **btnBeta** are added to the scene graph by using a call to **addAll( )**:

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

The **addAll( )** method adds a list of nodes to the invoking parent node. Of course, these nodes could have been added by three separate calls to **add( )**, but the **addAll( )** method is more convenient to use in this situation. There are two other things of interest in this program that relate to the way the controls are displayed in the window. First, when the root node is created, this statement is used:

FlowPane rootNode = new FlowPane(10, 10);

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

rootNode.setAlignment(Pos.CENTER);

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

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

btnAlpha.setOnAction( (ae) ->

response.setText("Alpha was pressed.")

);

Note: A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

**Drawing Directly on a Canvas**

JavaFX’s graphics methods are found in the **GraphicsContext** class, which is part of **java.scene.canvas**. These methods can be used to draw directly on the surface of a canvas, which is encapsulated by the **Canvas** class in **java.scene.canvas**. When you draw something, such as a line, on a canvas, JavaFX automatically renders it whenever it needs to be redisplayed.

Before you can draw on a canvas, you must perform two steps. First, you must create a **Canvas** instance. Second, you must obtain a **GraphicsContext** object that refers to that canvas. You can then use the **GraphicsContext** to draw output on the canvas. The **Canvas** class is derived from **Node**; thus it can be used as a node in a scene graph.

**Canvas** defines two constructors. One is the default constructor, and the other is the one shown here:

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

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

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

**GraphicsContext getGraphicsContext2D( )**

The graphics context for the canvas is returned.

**GraphicsContext** defines a large number of methods that draw shapes, text, and images, and support effects and transforms. A few of its methods are described here.

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

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

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

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

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

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

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

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

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

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

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

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

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

It displays *str* starting at the location specified by *topX*, *topY*, filling the text with the current fill. You can set the font and font size of the text being displayed by using **setFont( )**. You can obtain the font used by the canvas by calling **getFont( )**. By default, the system font is used. You can create a new font by constructing a **Font** object. **Font** is packaged in **javafx.scene.text**. For example, you can create a default font of a specified size by using this constructor:

**Font(double *fontSize*)**

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

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

**void setFill(Paint *newFill*)**

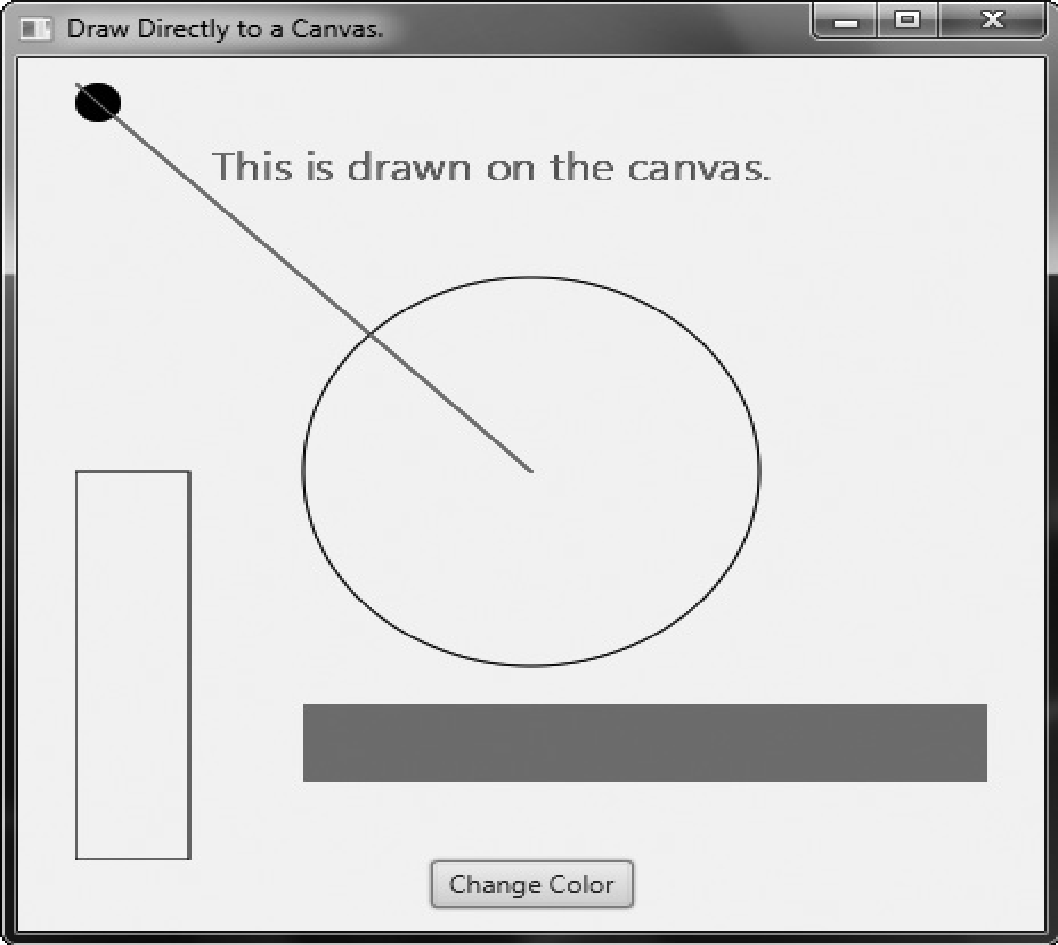
**void setStroke(Paint *newStroke*)**

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

The following program uses the aforementioned methods to demonstrate drawing on a canvas. It first displays a few graphic objects on the canvas. Then, each time the Change Color button is pressed, the color of three of the objects changes color. If you run the program, you will see that the shapes whose color is not changed are unaffected by the change in color of the other objects. Furthermore, if you try covering and then uncovering the window,

you will see that the canvas is automatically repainted, without any other actions on the part

of your program. Sample output is shown here:

****

// Demonstrate drawing.

import javafx.application.\*;

import javafx.scene.\*;

import javafx.stage.\*;

import javafx.scene.layout.\*;

import javafx.scene.control.\*;

import javafx.event.\*;

import javafx.geometry.\*;

import javafx.scene.shape.\*;

import javafx.scene.canvas.\*;

import javafx.scene.paint.\*;

import javafx.scene.text.\*;

public class DirectDrawDemo extends Application {

GraphicsContext gc;

Color[] colors = { Color.RED, Color.BLUE, Color.GREEN, Color.BLACK };

int colorIdx = 0;

public static void main(String[] args) {

// Start the JavaFX application by calling launch().

launch(args);

}

// Override the start() method.

public void start(Stage myStage) {

// Give the stage a title.

myStage.setTitle("Draw Directly to a Canvas.");

// Use a FlowPane for the root node.

FlowPane rootNode = new FlowPane();

// Center the nodes in the scene.

rootNode.setAlignment(Pos.CENTER);

// Create a scene.

Scene myScene = new Scene(rootNode, 450, 450);

// Set the scene on the stage.

myStage.setScene(myScene);

// Create a canvas.

Canvas myCanvas = new Canvas(400, 400);

// Get the graphics context for the canvas.

gc = myCanvas.getGraphicsContext2D();

// Create a push button.

Button btnChangeColor = new Button("Change Color");

// Handle the action events for the Change Color button.

btnChangeColor.setOnAction(new EventHandler<ActionEvent>() {

public void handle(ActionEvent ae) {

// Set the stroke and fill color.

gc.setStroke(colors[colorIdx]);

gc.setFill(colors[colorIdx]);

// Redraw the line, text, and filled rectangle in the new color. This leaves the color of the other //nodes unchanged.

gc.strokeLine(0, 0, 200, 200);

gc.fillText("This is drawn on the canvas.", 60, 50);

gc.fillRect(100, 320, 300, 40);

// Change the color.

colorIdx++;

if(colorIdx == colors.length) colorIdx= 0;

}

});

// Draw initial output on the canvas.

gc.strokeLine(0, 0, 200, 200);

gc.strokeOval(100, 100, 200, 200);

gc.strokeRect(0, 200, 50, 200);

gc.fillOval(0, 0, 20, 20);

gc.fillRect(100, 320, 300, 40);

// Set the font size to 20 and draw text.

gc.setFont(new Font(20));

gc.fillText("This is drawn on the canvas.", 60, 50);

// Add the canvas and button to the scene graph.

rootNode.getChildren().addAll(myCanvas, btnChangeColor);

// Show the stage and its scene.

myStage.show();

}

}

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

**NOTE** javafx.scene.shape contains several classes that can also be used to draw various types of

graphical shapes, such as circles, arcs, and lines. These are represented by nodes and can, therefore,be directly part of the scene graph. You will want to explore these on your own.

**Exploring JavaFXControls**

**Using Image and ImageView**

Several of JavaFX’s controls let you include an image. For example, in addition to text, you can specify an image in a label or a button. Furthermore, you can embed stand-alone images in a scene directly. At the foundation for JavaFX’s support for images are two classes: **Image** and **ImageView**. **Image** encapsulates the image, itself, and **ImageView** manages the display of an image. Both classes are packaged in **javafx.scene.image**. The **Image** class loads an image from either an **InputStream**, a URL, or a path to the image file. **Image** defines several constructors; this is the one we will use:

**Image(String *url*)**

Here, *url* specifies a URL or a path to a file that supplies the image. The argument is assumed

to refer to a path if it does not constitute a properly formed URL. Otherwise, the image is loaded from the URL. The examples that follow will load images from files on the local filesystem. Other constructors let you specify various options, such as the image’s width and height. One other point: **Image** is not derived from **Node**. Thus, it cannot, itself, be part of a scene graph. Once you have an **Image**, you will use **ImageView** to display it. **ImageView** is derived from **Node**, which means that it can be part of a scene graph. **ImageView** defines three constructors. The first one we will use is shown here:

**ImageView(Image *image*)**

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

// Load and display an image.

import javafx.application.\*;

import javafx.scene.\*;

import javafx.stage.\*;

import javafx.scene.layout.\*;

import javafx.geometry.\*;

import javafx.scene.image.\*;

public class ImageDemo extends Application {

public static void main(String[] args) {

// Start the JavaFX application by calling launch().

launch(args);

}

// Override the start() method.

public void start(Stage myStage) {

// Give the stage a title.

myStage.setTitle("Display an Image");

// Use a FlowPane for the root node.

FlowPane rootNode = new FlowPane();

// Use center alignment.

rootNode.setAlignment(Pos.CENTER);

// Create a scene.

Scene myScene = new Scene(rootNode, 300, 200);

// Set the scene on the stage.

myStage.setScene(myScene);

// Create an image.

Image hourglass = new Image("hourglass.png");

// Create an image view that uses the image.

ImageView hourglassIV = new ImageView(hourglass);

// Add the image to the scene graph.

rootNode.getChildren().add(hourglassIV);

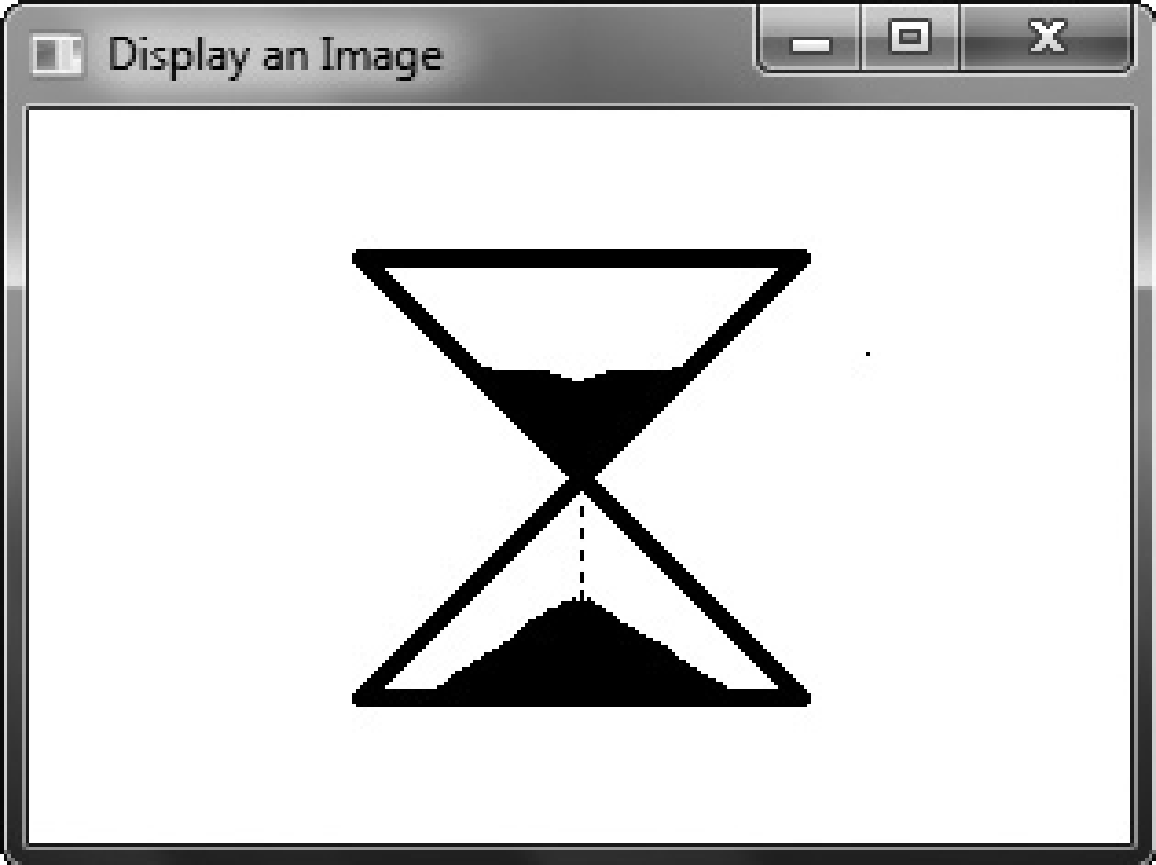
// Show the stage and its scene.

myStage.show();

}

}

Sample output from the program is shown here:



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

// Create an image.

Image hourglass = new Image("HourGlass.png");

// Create an image view that uses the image.

ImageView hourglassIV = new ImageView(hourglass);

As explained, an image by itself cannot be added to the scene graph. It must first beembedded in an **ImageView**.In cases in which you won’t make further use of the image, you can specify a URL orfilename when creating an **ImageView**. In this case, there is no need to explicitly create an**Image**. Instead, an **Image** instance containing the specified image is constructed automaticallyand embedded in the **ImageView**. Here is the **ImageView** constructor that does this:

**ImageView(String *url*)**

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

**Adding an Image to a Label**

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

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

Here, *str* specifies the text message and *image* specifies the image. Notice that the image is of type **Node**. This allows great flexibility in the type of image added to the label, but for our purposes, the image type will be **ImageView**. Here is a program that demonstrates a label that includes a graphic. It creates a label that displays the string "Hourglass" and shows the image of an hourglass that is loaded from

the **hourglass.png** file.

// Demonstrate an image in a label.

import javafx.application.\*;

import javafx.scene.\*;

import javafx.stage.\*;

import javafx.scene.layout.\*;

import javafx.scene.control.\*;

import javafx.geometry.\*;

import javafx.scene.image.\*;

public class LabelImageDemo extends Application {

public static void main(String[] args) {

// Start the JavaFX application by calling launch().

launch(args);

}

// Override the start() method.

public void start(Stage myStage) {

// Give the stage a title.

myStage.setTitle("Use an Image in a Label");

// Use a FlowPane for the root node.

FlowPane rootNode = new FlowPane();

// Use center alignment.

rootNode.setAlignment(Pos.CENTER);

// Create a scene.

Scene myScene = new Scene(rootNode, 300, 200);

// Set the scene on the stage.

myStage.setScene(myScene);

// Create an ImageView that contains the specified image.

ImageView hourglassIV = new ImageView("hourglass.png");

// Create a label that contains both an image and text.

Label hourglassLabel = new Label("Hourglass", hourglassIV);

// Add the label to the scene graph.

rootNode.getChildren().add(hourglassLabel);

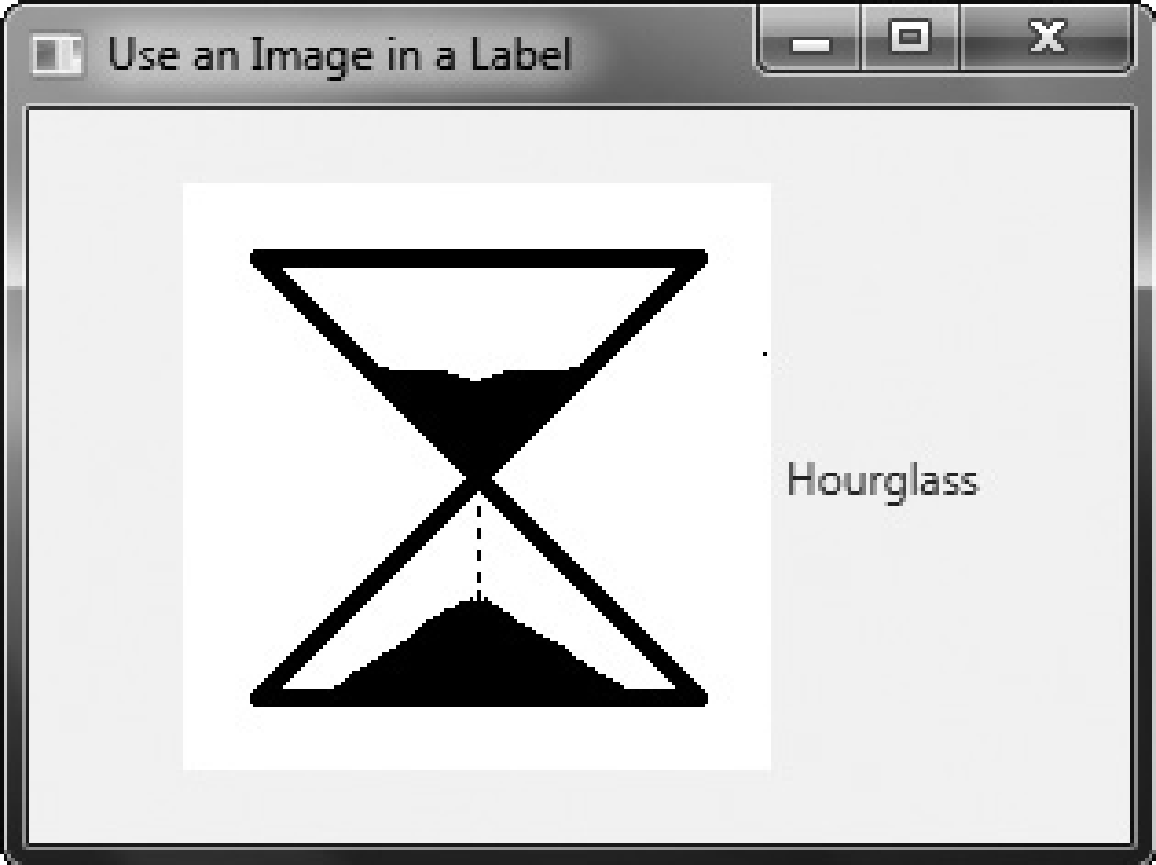
// Show the stage and its scene.

myStage.show();

}

}

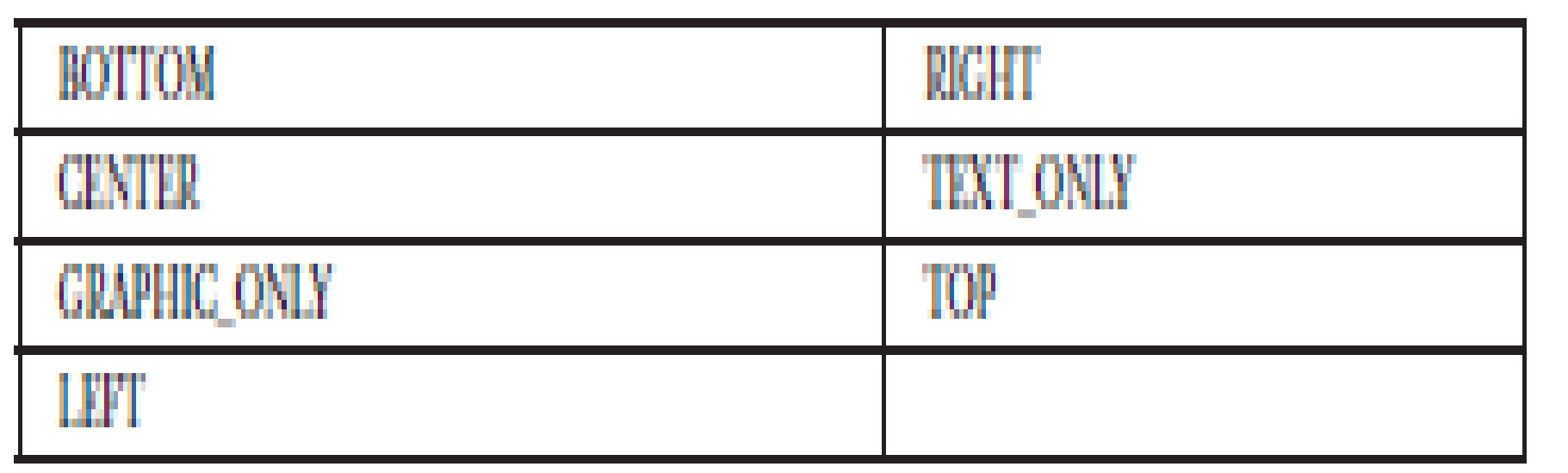
Here is the window produced by the program:



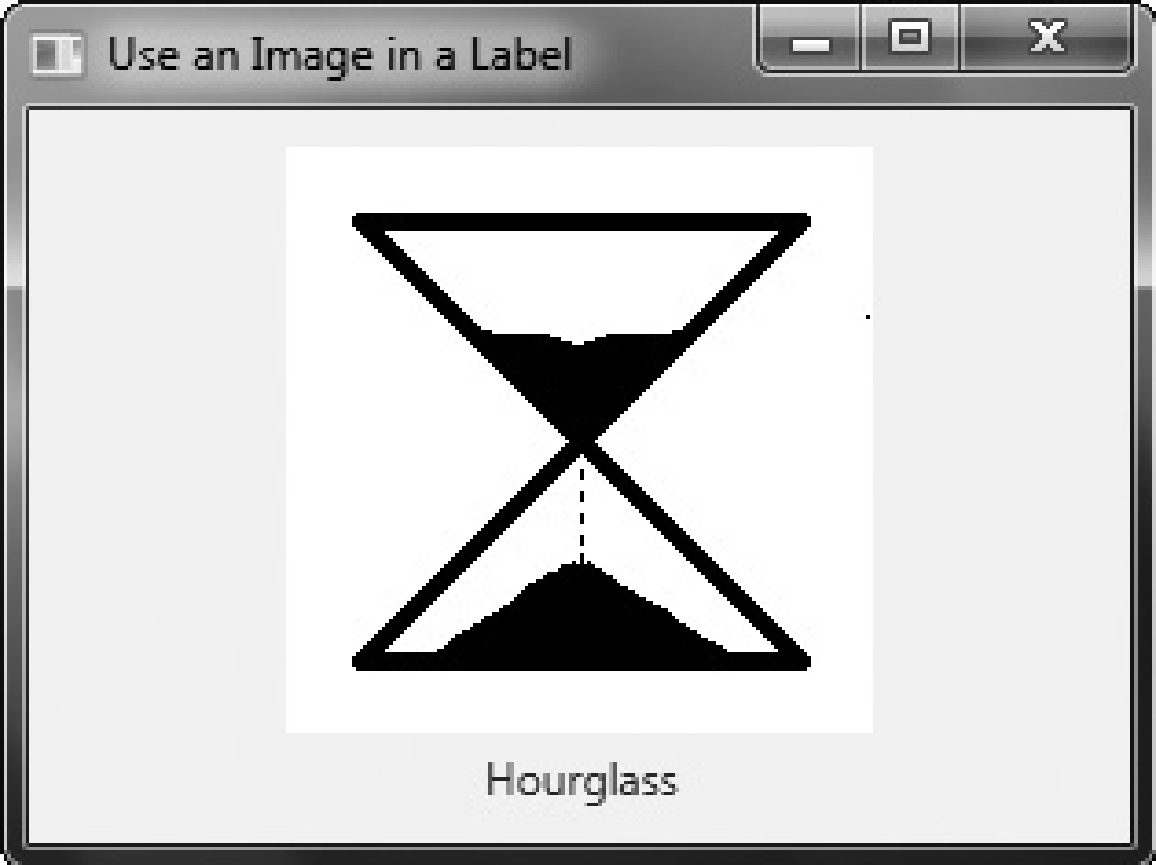
As you can see, both the image and the text are displayed. Notice that the text is to the right of the image. This is the default. You can change the relative positions of the image and text by calling **setContentDisplay( )** on the label. It is shown here:

final void setContentDisplay(ContentDisplay *position*)

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



With the exception of **TEXT\_ONLY** and **GRAPHIC\_ONLY**, the values specify the location of the image. For example, if you add this line to the preceding program: hourglassLabel.setContentDisplay(ContentDisplay.TOP); the image of the hourglass will be above the text, as shown here:



The other two values let you display either just the text or just the image. This might be useful if your application wants to use an image at some times, and not at others, for example.(If you want only an image, you can simply display it without using a label, as described in the previous section.)You can also add an image to a label after it has been constructed by using the **setGraphic( )** method. It is shown here:

**final void setGraphic(Node *image*)**

Here, *image* specifies the image to add.

**Using an Image with a Button**

**Button** is JavaFX’s class for push buttons. The procedure for adding an image to a button is similar to that used to add an image to a label. First obtain an **ImageView** of the image. Then add it to the button. One way to add the image is to use this constructor:

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

Here, *str* specifies the text that is displayed within the button and *image* specifies the image. You can specify the position of the image relative to the text by using **setContentDisplay( )**in the same way as just described for **Label**. Here is an example that displays two buttons that contain images. The first shows an hourglass. The second shows an analog clock. When a button is pressed, the selected timepiece is reported. Notice that the text is displayed beneath the image.

// Use an image with a button.

import javafx.application.\*;

import javafx.scene.\*;

import javafx.stage.\*;

import javafx.scene.layout.\*;

import javafx.scene.control.\*;

import javafx.event.\*;

import javafx.geometry.\*;

import javafx.scene.image.\*;

public class ButtonImageDemo extends Application {

Label response;

public static void main(String[] args) {

// Start the JavaFX application by calling launch().

launch(args);

}

// Override the start() method.

public void start(Stage myStage) {

// Give the stage a title.

myStage.setTitle("Use Images with Buttons");

// Use a FlowPane for the root node. In this case,

// vertical and horizontal gaps of 10.

FlowPane rootNode = new FlowPane(10, 10);

// Center the controls in the scene.

rootNode.setAlignment(Pos.CENTER);

// Create a scene.

Scene myScene = new Scene(rootNode, 250, 450);

// Set the scene on the stage.

myStage.setScene(myScene);

// Create a label.

response = new Label("Push a Button");

// Create two image-based buttons.

Button btnHourglass = new Button("Hourglass",new ImageView("hourglass.png"));

Button btnAnalogClock = new Button("Analog Clock", new ImageView("analog.png"));

// Position the text under the image.

btnHourglass.setContentDisplay(ContentDisplay.TOP);

btnAnalogClock.setContentDisplay(ContentDisplay.TOP);

// Handle the action events for the hourglass button.

btnHourglass.setOnAction(new EventHandler<ActionEvent>() {

public void handle(ActionEvent ae) {

response.setText("Hourglass Pressed");

}

});

// Handle the action events for the analog clock button.

btnAnalogClock.setOnAction(new EventHandler<ActionEvent>() {

public void handle(ActionEvent ae) {

response.setText("Analog Clock Pressed");

}

});

// Add the label and buttons to the scene graph.

rootNode.getChildren().addAll(btnHourglass, btnAnalogClock, response);

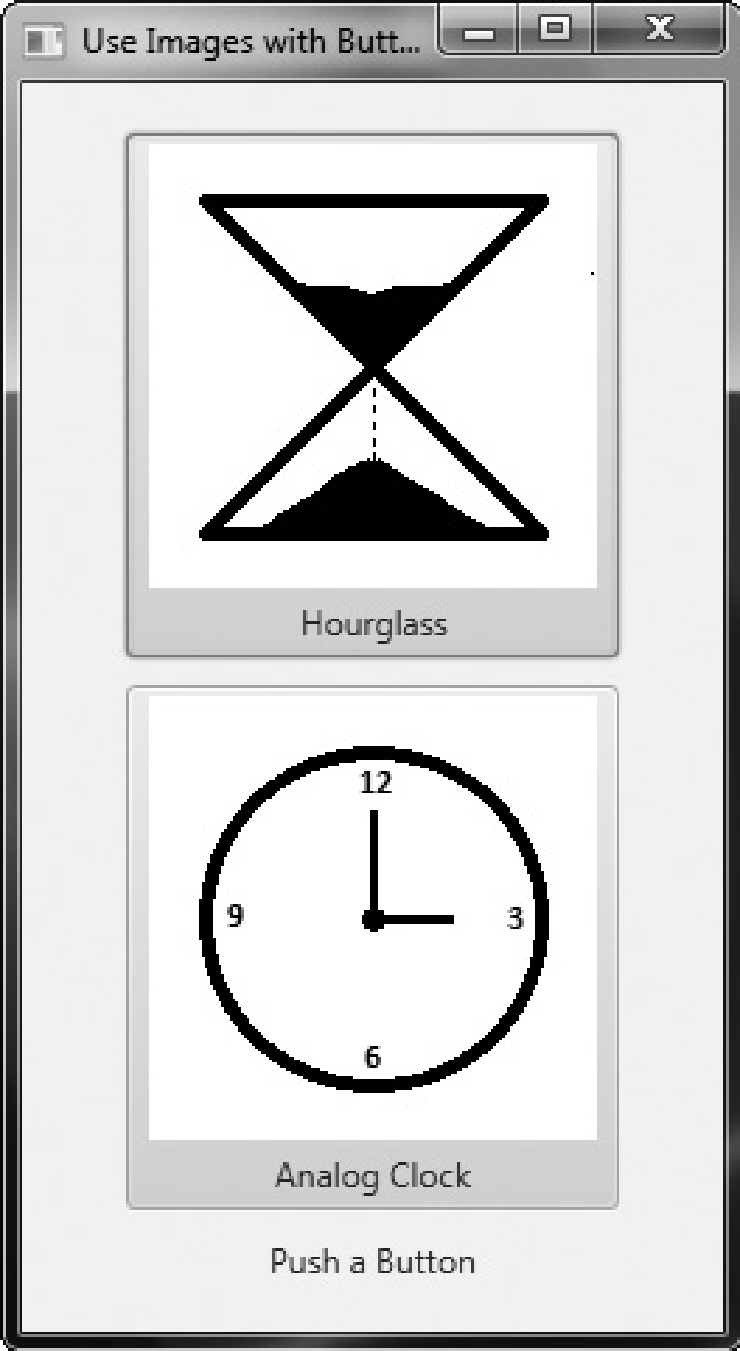
// Show the stage and its scene.

myStage.show();

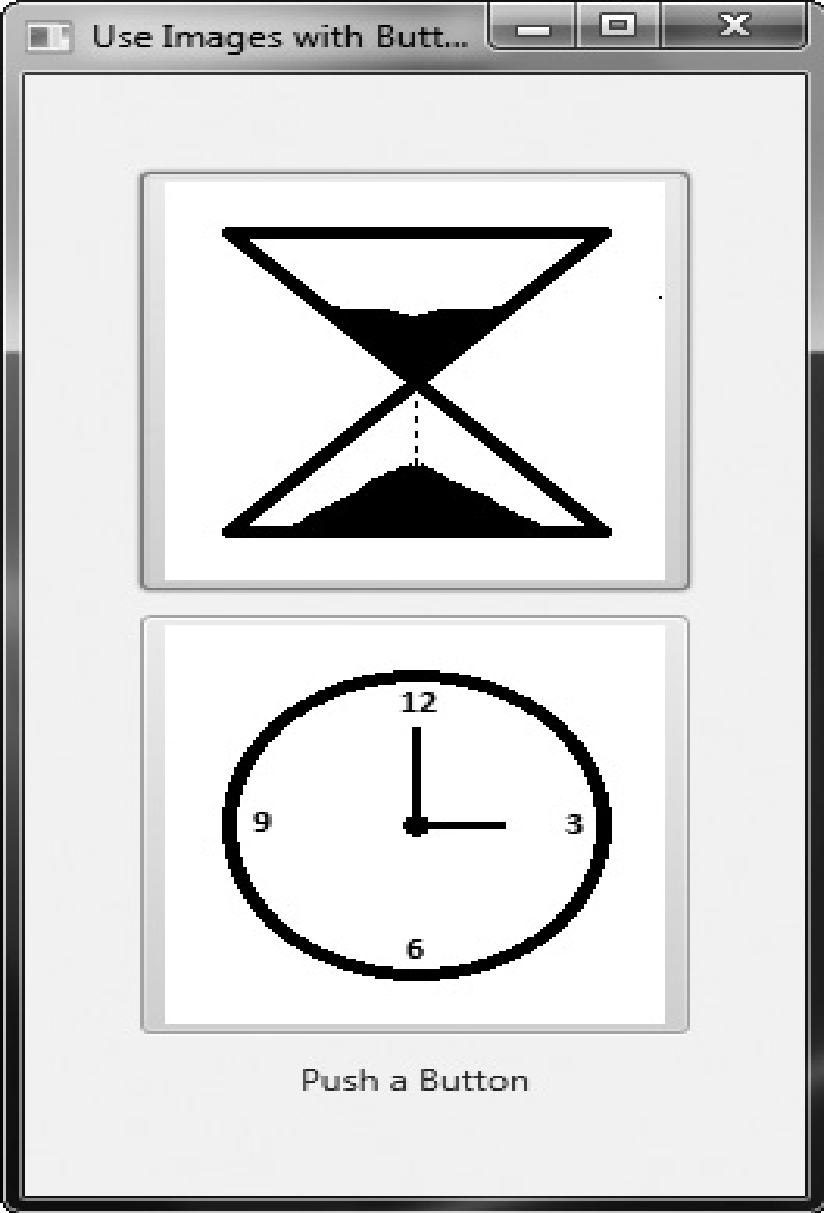
}

}

The output produced by this program is shown here:



If you want a button that contains only the image, pass a null string for the text when constructing the button and then call **setContentDisplay()**, passing in the parameter **ContentDisplay.GRAPHIC\_ONLY**. For example, if you make these modifications to theprevious program, the output will look like this:



**ToggleButton**

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

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

**ToggleButton(String *str*)**

Here, *str* is the text displayed in the button. Another constructor allows you to include an image. Like other buttons, a **ToggleButton** generates an action event when it is pressed. Because **ToggleButton** defines a two-state control, it is commonly used to let the user select an option. When the button is pressed, the option is selected. When the button is released, the option is deselected. For this reason, a program usually needs to determine the toggle button’s state. To do this, use the **isSelected( )** method, shown here:

**final boolean isSelected( )**

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

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

// Demonstrate a toggle button.

import javafx.application.\*;

import javafx.scene.\*;

import javafx.stage.\*;

import javafx.scene.layout.\*;

import javafx.scene.control.\*;

import javafx.event.\*;

import javafx.geometry.\*;

public class ToggleButtonDemo extends Application {

ToggleButton tbOnOff;

Label response;

public static void main(String[] args) {

// Start the JavaFX application by calling launch().

launch(args);

}

// Override the start() method.

public void start(Stage myStage) {

// Give the stage a title.

myStage.setTitle("Demonstrate a Toggle Button");

// Use a FlowPane for the root node. In this case,

// vertical and horizontal gaps of 10.

FlowPane rootNode = new FlowPane(10, 10);

// Center the controls in the scene.

rootNode.setAlignment(Pos.CENTER);

// Create a scene.

Scene myScene = new Scene(rootNode, 220, 120);

// Set the scene on the stage.

myStage.setScene(myScene);

// Create a label.

response = new Label("Push the Button.");

// Create the toggle button.

tbOnOff = new ToggleButton("On/Off");

// Handle action events for the toggle button.

tbOnOff.setOnAction(new EventHandler<ActionEvent>() {

public void handle(ActionEvent ae) {

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

else response.setText("Button is off.");

}

});

// Add the label and buttons to the scene graph.

rootNode.getChildren().addAll(tbOnOff, response);

// Show the stage and its scene.

myStage.show();

}

}



**RadioButton**

Another type of button provided by JavaFX is the *radio button*. Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any one time. They are supported by the **RadioButton** class, which extends both **ButtonBase** and **ToggleButton**. It also implements the **Toggle** interface. Thus, a radio button is a specialized form of a togglebutton. To create a radio button, we will use the following constructor:

**RadioButton(String *str*)**

Here, *str* is the label for the button. Like other buttons, when a **RadioButton** is used, an action event is generated. For their mutually exclusive nature to be activated, radio buttons must be configured into a group. Only one of the buttons in the group can be selected at any time. For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected. A button group is created by the **ToggleGroup** class, which is packaged in **javafx.scene.control**. **ToggleGroup** provides only a default constructor. Radio buttons are added to the toggle group by calling the **setToggleGroup( )** method, defined by **ToggleButton**, on the button. It is shown here:

**final void setToggleGroup(ToggleGroup *tg*)**

Here, *tg* is a reference to the toggle button group to which the button is added. After all radio buttons have been added to the same group, their mutually exclusive behavior will be enabled. In general, when radio buttons are used in a group, one of the buttons is selected when the group is first displayed in the GUI. Here are two ways to do this.

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

**final void setSelected(boolean *state*)**

If *state* is **true**, the button is selected. Otherwise, it is deselected. Although the button is selected, no action event is generated. A second way to initially select a radio button is to call **fire( )** on the button. It is shown here:

**void fire( )**

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

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

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

import javafx.application.\*;

import javafx.scene.\*;

import javafx.stage.\*;

import javafx.scene.layout.\*;

import javafx.scene.control.\*;

import javafx.event.\*;

import javafx.geometry.\*;

public class RadioButtonDemo extends Application {

Label response;

public static void main(String[] args) {

// Start the JavaFX application by calling launch().

launch(args);

}

// Override the start() method.

public void start(Stage myStage) {

// Give the stage a title.

myStage.setTitle("Demonstrate Radio Buttons");

// Use a FlowPane for the root node. In this case, vertical and horizontal gaps of 10.

FlowPane rootNode = new FlowPane(10, 10);

// Center the controls in the scene.

rootNode.setAlignment(Pos.CENTER);

// Create a scene.

Scene myScene = new Scene(rootNode, 220, 120);

// Set the scene on the stage.

myStage.setScene(myScene);

// Create a label that will report the selection.

response = new Label("");

// Create the radio buttons.

RadioButton rbTrain = new RadioButton("Train");

RadioButton rbCar = new RadioButton("Car");

RadioButton rbPlane = new RadioButton("Airplane");

// Create a toggle group.

ToggleGroup tg = new ToggleGroup();

// Add each button to a toggle group.

rbTrain.setToggleGroup(tg);

rbCar.setToggleGroup(tg);

rbPlane.setToggleGroup(tg);

// Handle action events for the radio buttons.

rbTrain.setOnAction(new EventHandler<ActionEvent>() {

public void handle(ActionEvent ae) {

response.setText("Transport selected is train.");

}

});

rbCar.setOnAction(new EventHandler<ActionEvent>() {

public void handle(ActionEvent ae) {

response.setText("Transport selected is car.");

}

});

rbPlane.setOnAction(new EventHandler<ActionEvent>() {

public void handle(ActionEvent ae) {

response.setText("Transport selected is airplane.");

});

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

rbTrain.fire();

// Add the label and buttons to the scene graph.

rootNode.getChildren().addAll(rbTrain, rbCar, rbPlane, response);

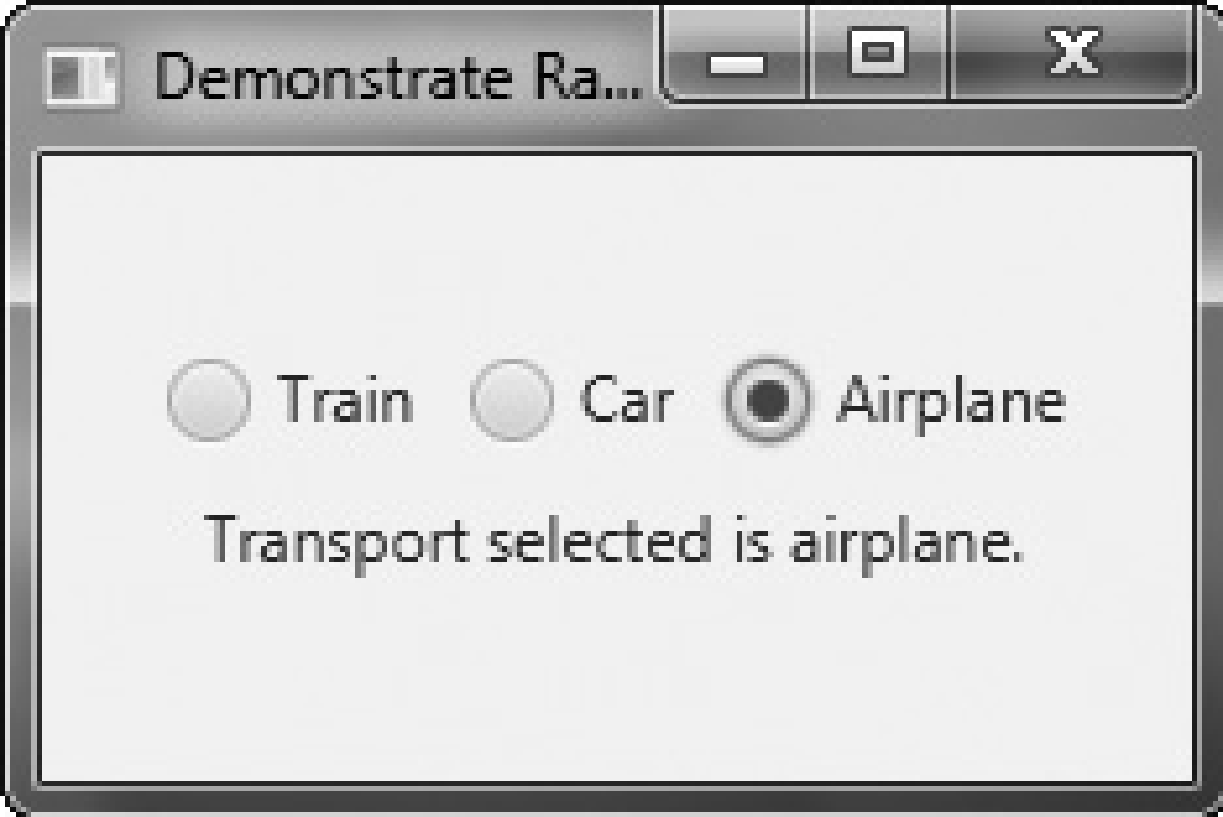
// Show the stage and its scene.

myStage.show();

}

}

Sample output is shown here:



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

RadioButton rbTrain = new RadioButton("Train");

RadioButton rbCar = new RadioButton("Car");

RadioButton rbPlane = new RadioButton("Airplane");

Next, a **ToggleGroup** is constructed:

ToggleGroup tg = new ToggleGroup();

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

rbTrain.setToggleGroup(tg);

rbCar.setToggleGroup(tg);

rbPlane.setToggleGroup(tg);

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

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

**An Alternative Way to Handle Radio Buttons**

Although handling events generated by radio buttons is often useful, sometimes it is more appropriate to ignore those events and simply obtain the currently selected button when that information is needed. This approach is demonstrated by the following program. It adds a button called Confirm Transport Selection. When this button is pressed, the currently selected radio button is obtained and then the selected transport is displayed in a label. When you try the program, notice that changing the selected radio button does not cause the confirmed transport to change until you press the Confirm Transport Selection button.

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

import javafx.application.\*;

import javafx.scene.\*;

import javafx.stage.\*;

import javafx.scene.layout.\*;

import javafx.scene.control.\*;

import javafx.event.\*;

import javafx.geometry.\*;

public class RadioButtonDemo2 extends Application {

Label response;

ToggleGroup tg;

public static void main(String[] args) {

// Start the JavaFX application by calling launch().

launch(args);

}

// Override the start() method.

public void start(Stage myStage) {

// Give the stage a title.

myStage.setTitle("Demonstrate Radio Buttons");

// Use a FlowPane for the root node. In this case, vertical and horizontal gaps of 10.

FlowPane rootNode = new FlowPane(10, 10);

// Center the controls in the scene.

rootNode.setAlignment(Pos.CENTER);

// Create a scene.

Scene myScene = new Scene(rootNode, 200, 140);

// Set the scene on the stage.

myStage.setScene(myScene);

// Create two labels.

Label choose = new Label(" Select a Transport Type ");

response = new Label("No transport confirmed");

// Create push button used to confirm the selection.

Button btnConfirm = new Button("Confirm Transport Selection");

// Create the radio buttons.

RadioButton rbTrain = new RadioButton("Train");

RadioButton rbCar = new RadioButton("Car");

RadioButton rbPlane = new RadioButton("Airplane");

// Create a toggle group.

tg = new ToggleGroup();

// Add each button to a toggle group.

rbTrain.setToggleGroup(tg);

rbCar.setToggleGroup(tg);

rbPlane.setToggleGroup(tg);

// Initially select one of the radio buttons.

rbTrain.setSelected(true);

// Handle action events for the confirm button.

btnConfirm.setOnAction(new EventHandler<ActionEvent>() {

public void handle(ActionEvent ae) {

// Get the radio button that is currently selected.

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

// Display the selection.

response.setText(rb.getText() + " is confirmed.");

}

});

// Use a separator to better organize the layout.

Separator separator = new Separator();

separator.setPrefWidth(180);

// Add the label and buttons to the scene graph.

rootNode.getChildren().addAll(choose, rbTrain, rbCar, rbPlane,

separator, btnConfirm, response);

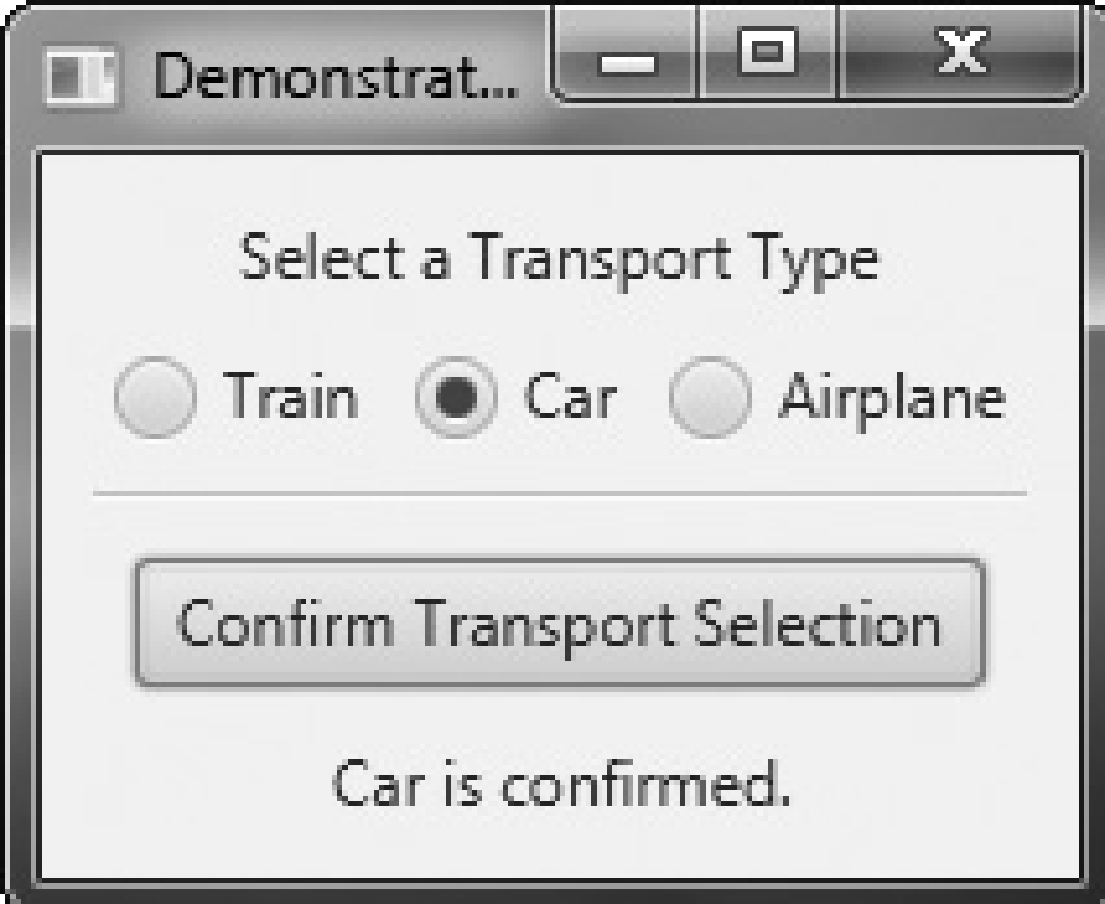
// Show the stage and its scene.

myStage.show();

}

}

The output from the program is shown here:



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

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

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

**final Toggle getSelectedToggle( )**

It returns a reference to the **Toggle** that is selected. In this case, the return value is cast to**RadioButton** because this is the type of button in the group.The second thing to notice is the use of a visual separator, which is created by thissequence:

**Separator separator = new Separator();**

**separator.setPrefWidth(180);**

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

**Exercise 1**

package example;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* (Traffic lights) Write a program that simulates a traffic light. The program \*

\* lets the user select one of three lights: red, yellow, or green. When a radio \*

\* button is selected, the light is turned on. Only one light can be on at a time \*

\* (see Figure 16.37a). No light is on when the program starts. \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import javafx.application.Application;

import javafx.stage.Stage;

import javafx.scene.Scene;

import javafx.scene.control.RadioButton;

import javafx.scene.control.ToggleGroup;

import javafx.scene.layout.StackPane;

import javafx.scene.layout.VBox;

import javafx.scene.layout.HBox;

import javafx.scene.layout.BorderPane;

import javafx.scene.paint.Color;

import javafx.scene.shape.Circle;

import javafx.scene.shape.Rectangle;

import javafx.geometry.Pos;

public class TrafficLight extends Application {

@Override // Override the start method in the Application calss

public void start(Stage primaryStage) {

// Create a vbox

VBox paneForCircles = new VBox(5);

paneForCircles.setAlignment(Pos.CENTER);

// Create three circles

Circle c1 = getCircle();

Circle c2 = getCircle();

Circle c3 = getCircle();

c1.setFill(Color.RED);

// Place circles in vbox

paneForCircles.getChildren().addAll(c1, c2, c3);

// Create a rectangle

Rectangle rectangle = new Rectangle();

rectangle.setFill(Color.WHITE);

rectangle.setWidth(30);

rectangle.setHeight(100);

rectangle.setStroke(Color.BLACK);

rectangle.setStrokeWidth(2);

StackPane stopSign = new StackPane(rectangle, paneForCircles);

// Create a hbox

HBox paneForRadioButtons = new HBox(5);

paneForRadioButtons.setAlignment(Pos.CENTER);

// Create radio buttons

RadioButton rbRed = new RadioButton("Red");

RadioButton rbYellow = new RadioButton("Yellow");

RadioButton rbGreen = new RadioButton("Green");

// Create a toggle group

ToggleGroup group = new ToggleGroup();

rbRed.setToggleGroup(group);

rbYellow.setToggleGroup(group);

rbGreen.setToggleGroup(group);

rbRed.setSelected(true);

paneForRadioButtons.getChildren().addAll(rbRed, rbYellow, rbGreen);

// Create a border pane

BorderPane pane = new BorderPane();

pane.setCenter(stopSign);

pane.setBottom(paneForRadioButtons);

// Create and register handlers

rbRed.setOnAction(e -> {

if (rbRed.isSelected()) {

c1.setFill(Color.RED);

c2.setFill(Color.WHITE);

c3.setFill(Color.WHITE);

}

});

rbYellow.setOnAction(e -> {

if (rbYellow.isSelected()) {

c1.setFill(Color.WHITE);

c2.setFill(Color.YELLOW);

c3.setFill(Color.WHITE);

}

});

rbGreen.setOnAction(e -> {

if (rbGreen.isSelected()) {

c1.setFill(Color.WHITE);

c2.setFill(Color.WHITE);

c3.setFill(Color.GREEN);

}

});

// Create a scene and place it in the stage

Scene scene = new Scene(pane, 200, 150);

primaryStage.setTitle("Traffic Light"); // Set the stage title

primaryStage.setScene(scene); // Place the scene in the stage

primaryStage.show(); // Display the stage

}

/\*\* Return a circle \*/

private Circle getCircle() {

Circle c = new Circle(10);

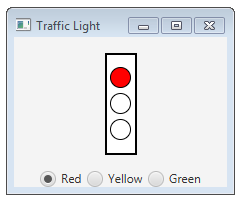
c.setFill(Color.WHITE);

c.setStroke(Color.BLACK);

return c;

}

}



**CheckBox**

The **CheckBox** class encapsulates the functionality of a check box. Its immediate superclass is **ButtonBase**. **CheckBox** supports three states. The first two are checked or unchecked The third state is *indeterminate* (also called *undefined*). It is typically used to indicate that the state of the check box has not been set or that it is not relevant to a specific situation. If you need the indeterminate state, you will need to explicitly enable it.

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

**CheckBox(String *str*)**

It creates a check box that has the text specified by *str* as a label. As with other buttons, a **CheckBox** generates an action event when it is selected. Here is a program that demonstrates check boxes. It displays check boxes that let the user select various deployment options, which are Web, Desktop, and Mobile. Each time a check box state changes, an action event is generated and handled by displaying the new state (selected or cleared) and by displaying a list of all selected boxes.

// Demonstrate Check Boxes.

import javafx.application.\*;

import javafx.scene.\*;

import javafx.stage.\*;

import javafx.scene.layout.\*;

import javafx.scene.control.\*;

import javafx.event.\*;

import javafx.geometry.\*;

public class CheckboxDemo extends Application {

CheckBox cbWeb;

CheckBox cbDesktop;

CheckBox cbMobile;

Label response;

Label allTargets;

String targets = "";

public static void main(String[] args) {

// Start the JavaFX application by calling launch().

launch(args);

}

// Override the start() method.

public void start(Stage myStage) {

// Give the stage a title.

myStage.setTitle("Demonstrate Checkboxes");

// Use a FlowPane for the root node. In this case,

// vertical and horizontal gaps of 10.

FlowPane rootNode = new FlowPane(10, 10);

// Center the controls in the scene.

rootNode.setAlignment(Pos.CENTER);

// Create a scene.

Scene myScene = new Scene(rootNode, 230, 140);

// Set the scene on the stage.

myStage.setScene(myScene);

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

// Create a label that will report the state of the

// selected check box.

response = new Label("No Deployment Selected");

// Create a label that will report all targets selected.

allTargets = new Label("Target List: <none>");

// Create the check boxes.

cbWeb = new CheckBox("Web");

cbDesktop = new CheckBox("Desktop");

cbMobile = new CheckBox("Mobile");

// Handle action events for the check boxes.

cbWeb.setOnAction(new EventHandler<ActionEvent>() {

public void handle(ActionEvent ae) {

if(cbWeb.isSelected())

response.setText("Web deployment selected.");

else

response.setText("Web deployment cleared.");

showAll();

}

});

cbDesktop.setOnAction(new EventHandler<ActionEvent>() {

public void handle(ActionEvent ae) {

if(cbDesktop.isSelected())

response.setText("Desktop deployment selected.");

else

response.setText("Desktop deployment cleared.");

showAll();

}

});

cbMobile.setOnAction(new EventHandler<ActionEvent>() {

public void handle(ActionEvent ae) {

if(cbMobile.isSelected())

response.setText("Mobile deployment selected.");

else

response.setText("Mobile deployment cleared.");

showAll();

}

});

// Use a separator to better organize the layout.

Separator separator = new Separator();

separator.setPrefWidth(200);

// Add controls to the scene graph.

rootNode.getChildren().addAll(heading, separator, cbWeb, cbDesktop,

cbMobile, response, allTargets);

// Show the stage and its scene.

myStage.show();

}

// Update and show the targets list.

void showAll() {

targets = "";

if(cbWeb.isSelected()) targets = "Web ";

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

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

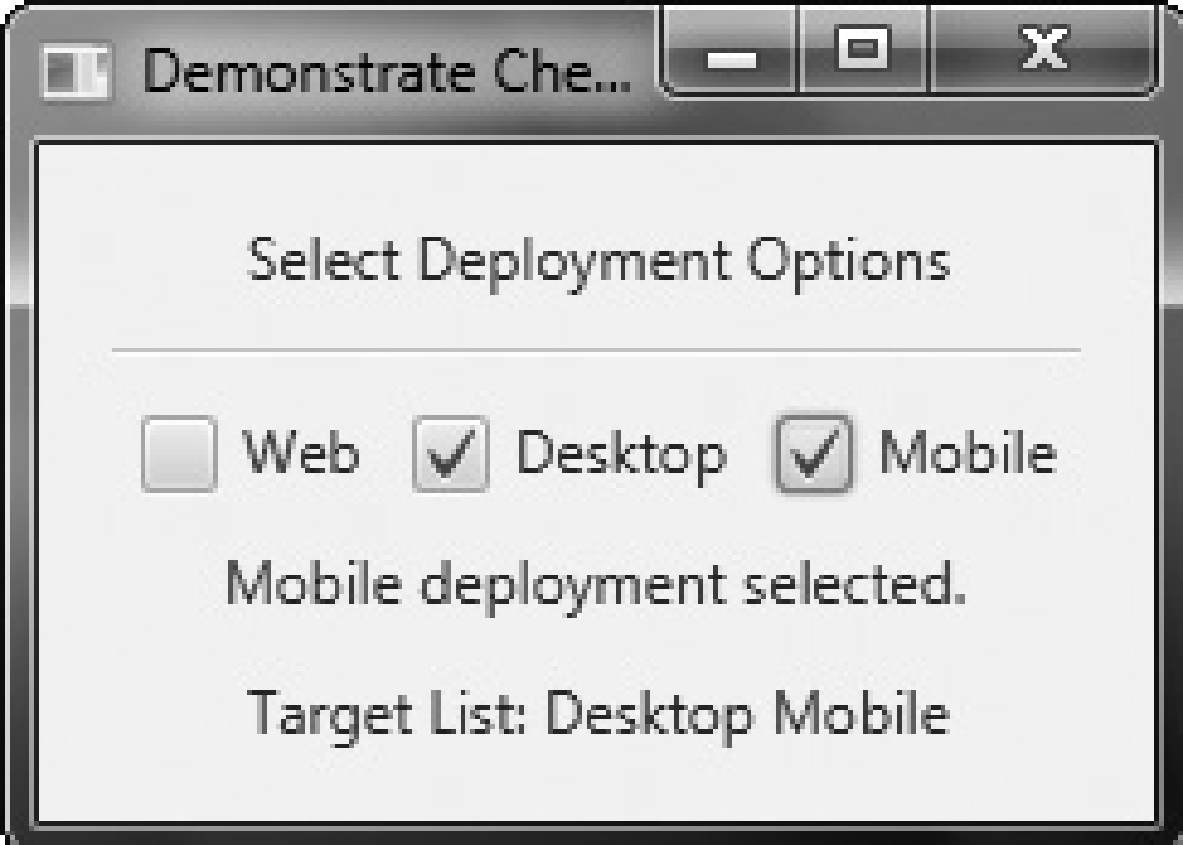
if(targets.equals("")) targets = "<none>";

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

}

}

Sample output is shown here:



The operation of this program is straightforward. Each time a check box is changed, an action command is generated. To determine if the box is checked or unchecked, the **isSelected( )** method is called. As mentioned, by default, **CheckBox** implements two states: checked and unchecked. If you want to add the indeterminate state, it must be explicitly enabled. To do this, call **setAllowIndeterminate( )**, shown here:

**final void setAllowIndeterminate(boolean *enable*)**

In this case, if *enable* is **true**, the indeterminate state is enabled. Otherwise, it is disabled. When the indeterminate state is enabled, the user can select between checked, unchecked, and indeterminate. You can determine if a check box is in the indeterminate state by calling **isIndeterminate( )**, shown here:

**final boolean isIndeterminate( )**

It returns **true** if the checkbox state is indeterminate and **false** otherwise. You can see the effect of a three-state check box by modifying the preceding program. First, enable the indeterminate state on the check boxes by calling **setAllowIndeterminate( )**on each check box, as shown here:

**cbWeb.setAllowIndeterminate(true);**

**cbDesktop.setAllowIndeterminate(true);**

**cbMobile.setAllowIndeterminate(true);**

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

cbWeb.setOnAction(new EventHandler<ActionEvent>() {

public void handle(ActionEvent ae) {

if(cbWeb.isIndeterminate())

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

response.setText("Web deployment selected.");

else

response.setText("Web deployment cleared.");

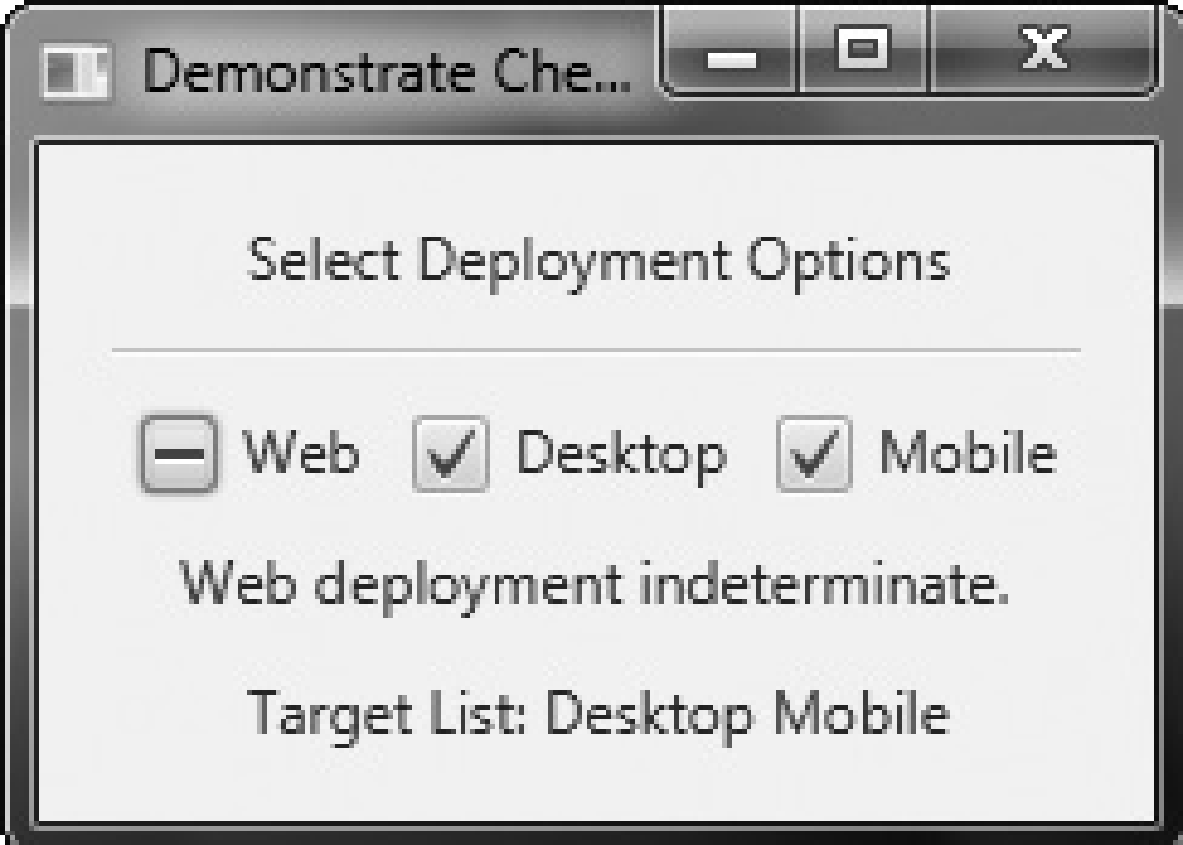
showAll();

}

});

Now, all three states are tested. Update the other two handlers in the same way. After making

these changes, the indeterminate state can be selected, as this sample output shows:



Here, the Web check box is indeterminate.

**ListView**

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

**ListView** is a generic class that is declared like this:

**class ListView<T>**

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

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

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

Here, *list* specifies a list of the items that will be displayed. It is an object of type **ObservableList**, which defines a list of observable objects. It inherits **java.util.List**. Thus, it supports the standard collection methods. **ObservableList** is packaged in **javafx.collections**. Probably the easiest way to create an **ObservableList** for use in a **ListView** is to use the factory method **observableArrayList( )**, which is a static method defined by the **FXCollections** class (which is also packaged in **javafx.collections**). The version we will use is shown here:

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

In this case, **E** specifies the type of elements, which are passed via *elements*. By default, a **ListView** allows only one item in the list to be selected at any one time.

However, you can allow multiple selections by changing the selection mode. For now, we will use the default, single-selection model. Although **ListView** provides a default size, sometimes you will want to set the preferred height and/or width to best match your needs. One way to do this is to call the **setPrefHeight( )** and **setPrefWidth( )** methods, shown here:

**final void setPrefHeight(double *height*)**

**final void setPrefWidth(double *width*)**

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

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

There are two basic ways in which you can use a **ListView**. First, you can ignore events generated by the list and simply obtain the selection in the list when your program needs it. Second, you can monitor the list for changes by registering a change listener. This lets you respond each time the user changes a selection in the list. This is the approach used here. To listen for change events, you must first obtain the selection model used by the **ListView**. This is done by calling **getSelectionModel( )** on the list. It is shown here:

**final MultipleSelectionModel<T> getSelectionModel( )**

It returns a reference to the model. **MultipleSelectionModel** is a class that defines the model used for multiple selections, and it inherits **SelectionModel**. However, multiple selections are allowed in a **ListView** only if multiple-selection mode is turned on. Using the model returned by **getSelectionModel( )**, you will obtain a reference to the selected item property that defines what takes place when an element in the list is selected. This is done by calling **selectedItemProperty( )**, shown next:

**final ReadOnlyObjectProperty<T> selectedItemProperty( )**

You will add the change listener to this property.

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

// Demonstrate a list view.

import javafx.application.\*;

import javafx.scene.\*;

import javafx.stage.\*;

import javafx.scene.layout.\*;

import javafx.scene.control.\*;

import javafx.geometry.\*;

import javafx.beans.value.\*;

import javafx.collections.\*;

public class ListViewDemo extends Application {

Label response;

public static void main(String[] args) {

// Start the JavaFX application by calling launch().

launch(args);

}

// Override the start() method.

public void start(Stage myStage) {

// Give the stage a title.

myStage.setTitle("ListView Demo");

// Use a FlowPane for the root node. In this case,

// vertical and horizontal gaps of 10.

FlowPane rootNode = new FlowPane(10, 10);

// Center the controls in the scene.

rootNode.setAlignment(Pos.CENTER);

// Create a scene.

Scene myScene = new Scene(rootNode, 200, 120);

// Set the scene on the stage.

myStage.setScene(myScene);

// Create a label.

response = new Label("Select Transport Type");

// Create an ObservableList of entries for the list view.

ObservableList<String> transportTypes =

FXCollections.observableArrayList( "Train", "Car", "Airplane" );

// Create the list view.

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

// Set the preferred height and width.

lvTransport.setPrefSize(80, 80);

// Get the list view selection model.

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

// Use a change listener to respond to a change of selection within a list view.

lvSelModel.selectedItemProperty().addListener(

new ChangeListener<String>() {

public void changed(ObservableValue<? extends String> changed, String oldVal, String newVal) {

// Display the selection.

response.setText("Transport selected is " + newVal);

}

});

rootNode.getChildren().addAll(lvTransport, response);

// Show the stage and its scene.

myStage.show();

}

}

Sample output is shown here:



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

ObservableList<String> transportTypes =

FXCollections.observableArrayList( "Train", "Car", "Airplane" );

It uses the **observableArrayList( )** method to create a list of strings. Then, the **ObservableList**

is used to initialize a **ListView**, as shown here:

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

The program then sets the preferred width and height of the control.Now, notice how the selection model is obtained for **lvTransport**:

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

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

**ListView Scrollbars**

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

ObservableList<String> transportTypes =

FXCollections.observableArrayList( "Train", "Car", "Airplane",

"Bicycle", "Walking" );

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



**Enabling Multiple Selections**

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

**final void setSelectionMode(SelectionMode *mode*)**

In this case, *mode* must be either **SelectionMode.MULTIPLE** or **SelectionMode.SINGLE**.When multiple-selection mode is enabled, you can obtain the list of the selections twoways: as a list of selected indices or as a list of selected items. We will use a list of selecteditems, but the procedure is similar when using a list of the indices of the selected items.(Note, indexing of items in a **ListView** begins at zero.). To get a list of the selected items, call **getSelectedItems( )** on the selection model. It is shown here:

**ObservableList<T> getSelectedItems( )**

It returns an **ObservableList** of the items. Because **ObservableList** extends **java.util.List**,you can access the items in the list just as you would any other **List** collection.To experiment with multiple selections, you can modify the preceding program asfollows. First, make **lvTransport final** so it can be accessed within the change event handler.Next, add this line:

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

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

lvSelModel.selectedItemProperty().addListener(

new ChangeListener<String>() {

public void changed(ObservableValue<? extends String> changed,String oldVal, String newVal) {

String selItems = "";

ObservableList<String> selected =

lvTransport.getSelectionModel().getSelectedItems();

// Display the selections.

for(int i=0; i < selected.size(); i++)

selItems += "\n " + selected.get(i);

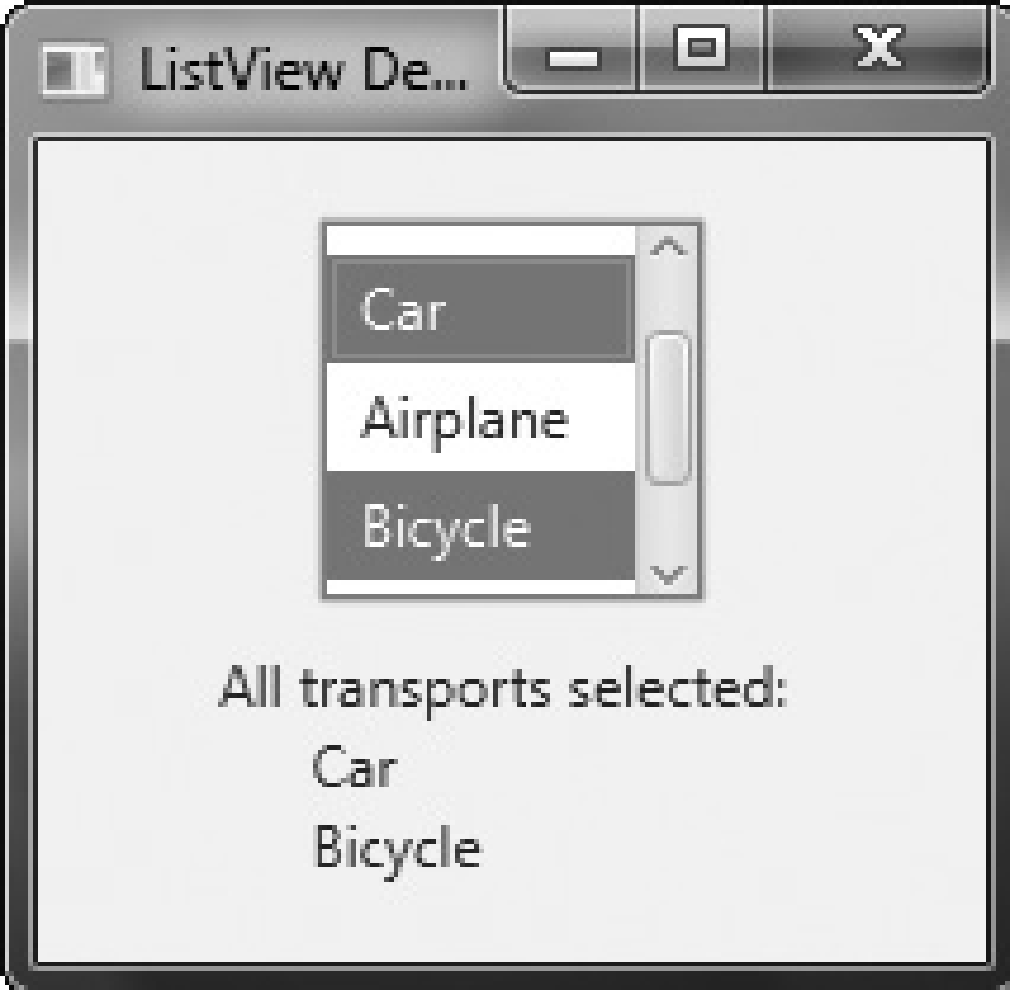
response.setText("All transports selected: " + selItems);

}

});

After making these changes, the program will display all selected forms of transports, as

the following output shows:



Assignments

1. Write javafx program to simulate traffic signal
2. Write javafx program to create a registration form with different controls

**ComboBox**

A control related to the list view is the combo box, which is implemented in JavaFX by the ComboBox class. A combo box displays one selection, but it will also display a drop-down list that allows the user to select a different item. You can also allow the user to edit a selection.

ComboBox inherits ComboBoxBase, which provides much of its functionality. Unlike the ListView, which can allow multiple selections, ComboBox is designed for single-selection.

ComboBox is a generic class that is declared like this:

**class ComboBox<T>**

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

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

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

In this case, list specifies a list of the items that will be displayed. It is an object of type ObservableList, which defines a list of observable objects. 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.

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

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

**final T getValue( )**

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

call setValue( ):

**final void setValue(T newVal)**

Here, newVal becomes the new value.

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

// Demonstrate a combo box.

import javafx.application.\*;

import javafx.scene.\*;

import javafx.stage.\*;

import javafx.scene.layout.\*;

import javafx.scene.control.\*;

import javafx.geometry.\*;

import javafx.collections.\*;

import javafx.event.\*;

public class ComboBoxDemo extends Application {

ComboBox<String> cbTransport;

Label response;

public static void main(String[] args) {

// Start the JavaFX application by calling launch().

launch(args);

}

// Override the start() method.

public void start(Stage myStage) {

// Give the stage a title.

myStage.setTitle("ComboBox Demo");

// Use a FlowPane for the root node. In this case,

// vertical and horizontal gaps of 10.

FlowPane rootNode = new FlowPane(10, 10);

// Center the controls in the scene.

rootNode.setAlignment(Pos.CENTER);

// Create a scene.

Scene myScene = new Scene(rootNode, 280, 120);

// Set the scene on the stage.

myStage.setScene(myScene);

// Create a label.

response = new Label();

/

/ Create an ObservableList of entries for the combo box.

ObservableList<String> transportTypes =

FXCollections.observableArrayList( "Train", "Car", "Airplane" );

// Create a combo box.

cbTransport = new ComboBox<String>(transportTypes);

// Set the default value.

cbTransport.setValue("Train");

// Set the response label to indicate the default selection.

response.setText("Selected Transport is " + cbTransport.getValue());

// Listen for action events on the combo box.

cbTransport.setOnAction(new EventHandler<ActionEvent>() {

public void handle(ActionEvent ae) {

response.setText("Selected Transport is " + cbTransport.getValue());

}

});

// Add the label and combo box to the scene graph.

rootNode.getChildren().addAll(cbTransport, response);

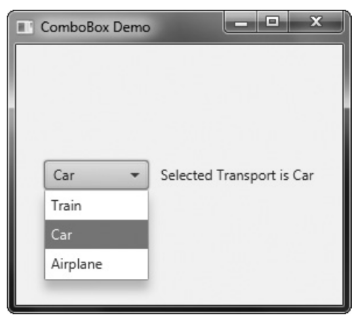
// Show the stage and its scene.

myStage.show();

}

}

Sample output is shown here:



As mentioned, ComboBox can be configured to allow the user to edit a selection. Assuming that it contains only entries of type String, it is easy to enable editing capabilities.

Simply call setEditable( ), shown here:

**final void setEditable(boolean enable)**

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

**cbTransport.setEditable(true);**

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

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

**TextField**

Sometimes you will want the user to enter a string of his or her own choosing. To accommodate this type of input, JavaFX includes several text-based controls. The one we will look at is TextField. It allows one line of text to be entered. Thus, it is useful for obtaining names, ID strings, addresses, and the like. Like all text controls, TextField inherits TextInputControl, which defines much of its functionality.

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

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

**final void setPrefColumnCount(int columns)**

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

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

You can also select a portion of the text under program control.

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

**final void setPromptText(String str)**

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

When the user presses enter while inside a TextField, an **action event** is generated.

Although handling this event is often helpful, in some cases, your program will simply obtain the text when it is needed, rather than handling action events. Both approaches are demonstrated by the following program. It creates a text field that requests a search string.

When the user presses enter while the text field has input focus, or presses the Get Search String button, the string is obtained and displayed. Notice that a prompting message is also included.

// Demonstrate a text field.

import javafx.application.\*;

import javafx.scene.\*;

import javafx.stage.\*;

import javafx.scene.layout.\*;

import javafx.scene.control.\*;

import javafx.event.\*;

import javafx.geometry.\*;

public class TextFieldDemo extends Application {

TextField tf;

Label response;

public static void main(String[] args) {

// Start the JavaFX application by calling launch().

launch(args);

}

// Override the start() method.

public void start(Stage myStage) {

// Give the stage a title.

myStage.setTitle("Demonstrate a TextField");

// Use a FlowPane for the root node. In this case,

// vertical and horizontal gaps of 10.

FlowPane rootNode = new FlowPane(10, 10);

// Center the controls in the scene.

rootNode.setAlignment(Pos.CENTER);

// Create a scene.

Scene myScene = new Scene(rootNode, 230, 140);

// Set the scene on the stage.

myStage.setScene(myScene);

// Create a label that will report the contents of the

// text field.

response = new Label("Search String: ");

// Create a button that gets the text.

Button btnGetText = new Button("Get Search String");

// Create a text field.

tf = new TextField();

// Set the prompt.

tf.setPromptText("Enter Search String");

// Set preferred column count.

tf.setPrefColumnCount(15);

// Handle action events for the text field. Action

// events are generated when ENTER is pressed while

// the text field has input focus. In this case, the

// text in the field is obtained and displayed.

tf.setOnAction(new EventHandler<ActionEvent>() {

public void handle(ActionEvent ae) {

response.setText("Search String: " + tf.getText());

}

});

// Get text from the text field when the button is pressed

// and display it.

btnGetText.setOnAction(new EventHandler<ActionEvent>() {

public void handle(ActionEvent ae) {

response.setText("Search String: " + tf.getText());

}

});

// Use a separator to better organize the layout.

Separator separator = new Separator();

separator.setPrefWidth(180);

// Add controls to the scene graph.

rootNode.getChildren().addAll(tf, btnGetText, separator, response);

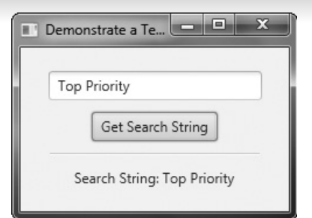
// Show the stage and its scene.

myStage.show();

}

}

Sample output is shown here:

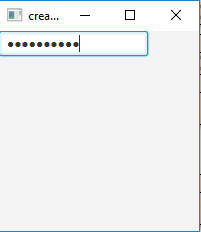


Other text controls that you will want to examine include TextArea, which supports multiline text, and PasswordField, which can be used to input passwords. You might also find HTMLEditor helpful.

PasswordField

PasswordField is a Text field that masks entered characters (the characters that are entered are not shown to the user). It allows the user to enter a single-line of unformatted text, hence it does not allow multi-line input.

|  |
| --- |
| // Java program to create a passwordfield  import javafx.application.Application;  import javafx.scene.Scene;  import javafx.scene.control.\*;  import javafx.scene.layout.\*;  import javafx.event.ActionEvent;  import javafx.event.EventHandler;  import javafx.scene.control.Label;  import javafx.stage.Stage;  public class Passwordfield extends Application  {        // launch the application      public void start(Stage s)      {          // set title for the stage          s.setTitle("creating Passwordfield");            // create a Passwordfield          PasswordField b =new PasswordField();            // create a tile pane          FlowPane r = new FlowPane();          // add password field          r.getChildren().add(b);           // create a scene          Scene sc =new Scene(r,200,200);          // set the scene          s.setScene(sc);          s.show();        }       public static void main(String args[])      {          // launch the application         launch(args);      }  } |

1. **Output**:  
   

**Exercise**

**Write a JavaFX program for login page with validation.**

package application;

//LOGIN PAGE WITH VALIDATION

import javafx.application.Application;

import javafx.geometry.\*;

import javafx.scene.\*;

import javafx.scene.control.\*;

import javafx.scene.layout.\*;

import javafx.stage.\*;

import javafx.scene.control.TextField;

import javafx.scene.control.PasswordField;

public class LoginpageValidation extends Application {

//Start the JavaFX application by calling launch().

public static void main(String[] args) {

launch(args);

}

//Override the start() method.

public void start(Stage myStage) {

//Give the stage a title.

myStage.setTitle("Demonstrate a Toggle Button");

FlowPane rootNode = new FlowPane(100,10);

rootNode.setAlignment(Pos.CENTER);

Scene myScene = new Scene(rootNode, 300, 300);

myStage.setScene(myScene);

Label response=new Label("enter username and password");

TextField t= new TextField("username");

PasswordField p= new PasswordField();

Button btnlogin=new Button("LOGIN");

btnlogin.setOnAction((ae)-> {

if(t.getText().equals("NITTE") && p.getText().equals("NITTE123"))

response.setText("LOGIN SUCCESSFUL. WELCOME "+ t.getText());

else response.setText("LOGIN FAILED. INVALID USERNAME OR PASSWORD");

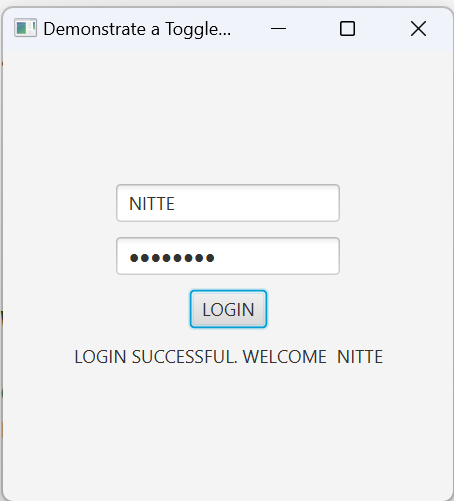
});

rootNode.getChildren().addAll(t,p,btnlogin, response);

myStage.show();

}

}



**ScrollPane**

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

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

**ScrollPane(Node content)**

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

**final void setContent(Node content)**

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

NOTE You can also use setContent( ) to change the content being scrolled by the scroll pane. Thus, what is being scrolled can be changed during the execution of your program. Although a default size is provided, as a general rule, you will want to set the dimensions of the viewport. The viewport is the viewable area of a scroll pane. It is the area in which the content being scrolled is displayed. Thus, the viewport displays the visible portion of the content. The scrollbars scroll the content through the viewport. Thus, by moving a scrollbar, you change what part of the content is visible.

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

**final void setPrefViewportHeight(double height)**

**final void setPrefViewportWidth(double width)**

In its default behavior, a ScrollPane will dynamically add or remove a scrollbar as needed. For example, if the component is taller than the viewport, a vertical scrollbar is added. If the component will completely fit within the viewport, the scrollbars are removed. One nice feature of ScrollPane is its ability to pan the contents by dragging the mouse.

By default, this feature is off. To turn it on, use setPannable( ), shown here:

**final void setPannable(boolean enable)**

If enable is true, then panning is allowed. Otherwise, it is disabled. You can set the **position of the scrollbars** under program control using setHvalue( ) and setVvalue( ), shown here:

**final void setHvalue(double newHval)**

**final void setVvalue(double newVval)**

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

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

**getHvalue( ) and getVvalue( ).**

The following program demonstrates ScrollPane by using one to scroll the contents

of a multiline label. Notice that it also enables panning.

// Demonstrate a scroll pane.

// This program scrolls the contents of a multiline

// label, but any node can be scrolled.

import javafx.application.\*;

import javafx.scene.\*;

import javafx.stage.\*;

import javafx.scene.layout.\*;

import javafx.scene.control.\*;

import javafx.event.\*;

import javafx.geometry.\*;

public class ScrollPaneDemo extends Application {

ScrollPane scrlPane;

public static void main(String[] args) {

// Start the JavaFX application by calling launch().

launch(args);

}

// Override the start() method.

public void start(Stage myStage) {

// Give the stage a title.

myStage.setTitle("Demonstrate a ScrollPane");

// Use a FlowPane for the root node.

FlowPane rootNode = new FlowPane(10, 10);

// Center the controls in the scene.

rootNode.setAlignment(Pos.CENTER);

// Create a scene.

Scene myScene = new Scene(rootNode, 200, 200);

// Set the scene on the stage.

myStage.setScene(myScene);

// Create a label that will be scrolled.

Label scrlLabel = new Label(

"A ScrollPane streamlines the process of\n" +

"adding scroll bars to a window whose\n" +

"contents exceed the window's dimensions.\n" +

"It also enables a control to fit in a\n" +

"smaller space than it otherwise would.\n" +

"As such, it often provides a superior\n" +

"approach over using individual scroll bars.");

// Create a scroll pane, setting scrlLabel as the content.

scrlPane = new ScrollPane(scrlLabel);

// Set the viewport width and height.

scrlPane.setPrefViewportWidth(130);

scrlPane.setPrefViewportHeight(80);

// Enable panning.

scrlPane.setPannable(true);

// Create a reset label.

Button btnReset = new Button("Reset Scroll Bar Positions");

// Handle action events for the reset button.

btnReset.setOnAction(new EventHandler<ActionEvent>() {

public void handle(ActionEvent ae) {

// Set the scroll bars to their zero position.

scrlPane.setVvalue(0);

scrlPane.setHvalue(0);

}

});

// Add the label to the scene graph.

rootNode.getChildren().addAll(scrlPane, btnReset);

// Show the stage and its scene.

myStage.show();

}

}

Sample output is shown here:



**TreeView**

TreeView presents a hierarchical view of data in a tree-like format. In this context, the term hierarchical means some items are subordinate to others. For example, a tree is commonly used to display the contents of a file system. In this case, the individual files are subordinate to the directory that contains them. In a TreeView,

branches can be expanded or collapsed on demand by the user.

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

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

TreeView is a generic class that is defined like this:

**class TreeView<T>**

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

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

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

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

class TreeItem<T>

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

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

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

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

The following example shows how to build and use a TreeView. The tree presents a hierarchy of food. The type of items stored in the tree are strings. The root is labeled Food.

Under it are three direct descendent nodes: Fruit, Vegetables, and Nuts. Under Fruit are three child nodes: Apples, Pears, and Oranges. Under Apples are three leaf nodes: Fuji, Winesap, and Jonathan. Each time a selection is made, the name of the item is displayed.

Also, the path from the root to the item is shown. This is done by the repeated use of getParent( ).

// Demonstrate a TreeView

import javafx.application.\*;

import javafx.scene.\*;

import javafx.stage.\*;

import javafx.scene.layout.\*;

import javafx.scene.control.\*;

import javafx.event.\*;

import javafx.beans.value.\*;

import javafx.geometry.\*;

public class TreeViewDemo extends Application {

Label response;

public static void main(String[] args) {

// Start the JavaFX application by calling launch().

launch(args);

}

// Override the start() method.

public void start(Stage myStage) {

// Give the stage a title.

myStage.setTitle("Demonstrate a TreeView");

// Use a FlowPane for the root node. In this case,

// vertical and horizontal gaps of 10.

FlowPane rootNode = new FlowPane(10, 10);

// Center the controls in the scene.

rootNode.setAlignment(Pos.CENTER);

// Create a scene.

Scene myScene = new Scene(rootNode, 310, 460);

// Set the scene on the stage.

myStage.setScene(myScene);

// Create a label that will report the state of the

// selected tree item.

response = new Label("No Selection");

// Create tree items, starting with the root.

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

// Now add subtrees, beginning with fruit.

TreeItem<String> tiFruit = new TreeItem<String>("Fruit");

// Construct the Apple subtree.

TreeItem<String> tiApples = new TreeItem<String>("Apples");

// Add child nodes to the Apple node.

tiApples.getChildren().add(new TreeItem<String>("Fuji"));

tiApples.getChildren().add(new TreeItem<String>("Winesap"));

tiApples.getChildren().add(new TreeItem<String>("Jonathan"));

// Add varieties to the fruit node.

tiFruit.getChildren().add(tiApples);

tiFruit.getChildren().add(new TreeItem<String>("Pears"));

tiFruit.getChildren().add(new TreeItem<String>("Oranges"));

// Finally, add the fruit node to the root.

tiRoot.getChildren().add(tiFruit);

// Now, add vegetables subtree, using the same general process.

TreeItem<String> tiVegetables = new TreeItem<String>("Vegetables");

tiVegetables.getChildren().add(new TreeItem<String>("Corn"));

tiVegetables.getChildren().add(new TreeItem<String>("Peas"));

tiVegetables.getChildren().add(new TreeItem<String>("Broccoli"));

tiVegetables.getChildren().add(new TreeItem<String>("Beans"));

tiRoot.getChildren().add(tiVegetables);

// Likewise, add nuts subtree.

TreeItem<String> tiNuts = new TreeItem<String>("Nuts");

tiNuts.getChildren().add(new TreeItem<String>("Walnuts"));

tiNuts.getChildren().add(new TreeItem<String>("Peanuts"));

tiNuts.getChildren().add(new TreeItem<String>("Pecans"));

tiRoot.getChildren().add(tiNuts);

// Create tree view using the tree just created.

TreeView<String> tvFood = new TreeView<String>(tiRoot);

// Get the tree view selection model.

MultipleSelectionModel<TreeItem<String>> tvSelModel =

tvFood.getSelectionModel();

// Use a change listener to respond to a selection within a tree view

tvSelModel.selectedItemProperty().addListener(

new ChangeListener<TreeItem<String>>() {

public void changed(

ObservableValue<? extends TreeItem<String>> changed,

TreeItem<String> oldVal, TreeItem<String> newVal) {

// Display the selection and its complete path from the root.

if(newVal != null) {

// Construct the entire path to the selected item.

String path = newVal.getValue();

TreeItem<String> tmp = newVal.getParent();

while(tmp != null) {

path = tmp.getValue() + " -> " + path;

tmp = tmp.getParent();

}

// Display the selection and the entire path.

response.setText("Selection is " + newVal.getValue() +

"\nComplete path is " + path);

}

}

});

// Add controls to the scene graph.

rootNode.getChildren().addAll(tvFood, response);

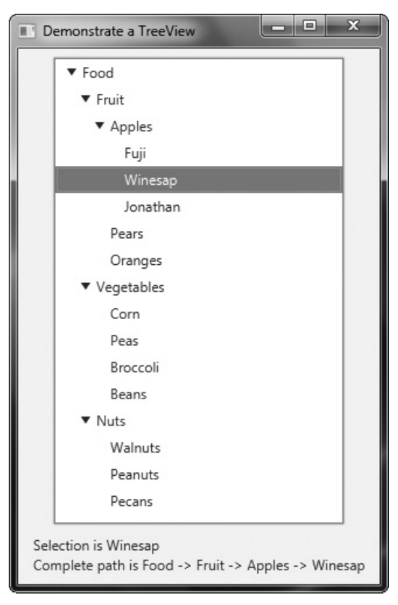
// Show the stage and its scene.

myStage.show();

}

}

Sample output is shown here:



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

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

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

tiRoot.getChildren().add(tiNuts);

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

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

String path = newVal.getValue();

TreeItem<String> tmp = newVal.getParent();

while(tmp != null) {

path = tmp.getValue() + " -> " + path;

tmp = tmp.getParent();

}

The code works like this: First, the value of the newly selected node is obtained. In this example, the value will be a string, which is the node’s name. This string is assigned to the path string. Then, a temporary variable of type TreeItem<String> is created and initialized to refer to the parent of the newly selected node. If the newly selected node does not have a parent, then tmp will be null. Otherwise, the loop is entered, within which each parent’s value (which is its name in this case) is added to path. This process continues until the root node of the tree (which has no parent) is found. Although the preceding shows the basic mechanism required to handle a TreeView, it is important to point out that several customizations and options are supported. TreeView is a powerful control that you will want to examine fully on your own.

**Introducing JavaFX Menus**

**Menu system.**

The JavaFX menu system supports several key elements, including

• **The menu bar**, which is the main menu for an application.

• **The standard menu**, which can contain either items to be selected or other menus(submenus).

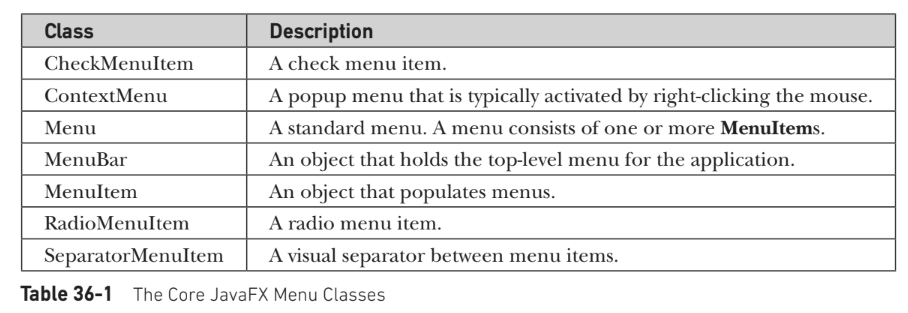
• **The context menu**, which is often activated by right-clicking the mouse. Context

menus are also called popup menus.

JavaFX menus also support accelerator keys, which enable menu items to be selected without having to activate the menu, and mnemonics, which allow a menu item to be selected by the keyboard once the menu options are displayed. In addition to “normal” menus, JavaFX also supports the toolbar, which provides rapid access to program functionality, often paralleling menu items.

**Menu Basics**

The JavaFX menu system is supported by a group of related classes packaged in javafx.scene.control. The ones used in this chapter are shown in Table 36-1, and they represent the core of the menu system. Although JavaFX allows a high degree of customization if desired, normally you will simply use the menu classes as-is because their default look and feel is generally what you will want.



Here is brief overview of how the classes fit together. To create a main menu for an application, you first need an instance of **MenuBar**. This class is a container for menus. To the MenuBar you add instances of Menu. Each Menu object defines a menu. That is, each Menu object contains one or more selectable items. The items displayed by a Menu are objects of type **MenuItem**. Thus, a MenuItem defines a

selection that can be chosen by the user. In addition to “standard” menu items, you can also include check and radio menu items in a menu. Their operation parallels check box and radio button controls. A check menu item is created by **CheckMenuItem**. A radio menu item is created by **RadioMenuItem**. Both of these classes extend MenuItem. **SeparatorMenuItem** is a convenience class that creates a separator line in a menu. It inherits CustomMenuItem, which is a class that facilitates embedding other types of controls in a menu item. CustomMenuItem extends MenuItem.

One key point about JavaFX menus is that MenuItem does not inherit Node. Thus,

instances of MenuItem can only be used in a menu. They cannot be otherwise incorporated into a scene graph. However, MenuBar does inherit Node, which does allow the menu bar to be added to the scene graph.

Another key point is that MenuItem is a superclass of Menu. This allows the creation of submenus, which are, essentially, menus within menus. To create a submenu, you first create and populate a Menu object with MenuItems and then add it to another Menu object. You

When a menu item is selected, an action event is generated. The text associated with the selection will be the name of the selection. Thus, when using one action event handler to process all menu selections, one way you can determine which item was selected is by examining the name. Of course, you can also use separate anonymous inner classes or lambda expressions to handle each menu item’s action events. In this case, the menu selection is already known and there is no need to examine the name to determine which item was selected.

As an alternative or adjunct to menus that descend from the menu bar, you can also create stand-alone, context menus, which pop up when activated. To create a context menu, first create an object of type ContextMenu. Then, add MenuItems to it. A context menu is often activated by clicking the right mouse button when the mouse is over a control Class Description CheckMenuItem A check menu item.

ContextMenu A popup menu that is typically activated by right-clicking the mouse.

Menu A standard menu. A menu consists of one or more MenuItems. MenuBar An object that holds the top-level menu for the application. MenuItem An object that populates menus. RadioMenuItem A radio menu item. SeparatorMenuItem A visual separator between menu items.

Table 36-1 The Core JavaFX Menu Classes for which a context menu has been defined. It is important to point out that ContextMenu is not derived from MenuItem. Rather, it inherits PopupControl. A feature related to the menu is the toolbar. In JavaFX, toolbars are supported by the ToolBar class. It creates a stand-alone component that is often used to provide fast access to functionality contained within the menus of the application. For example, a toolbar might provide fast access to the formatting commands supported by a word processor.

**An Overview of MenuBar, Menu, and MenuItem**

Before you can create a menu, you need to know some specifics about MenuBar, Menu, and MenuItem. These form the minimum set of classes needed to construct a main menu for an application. MenuItems are also used by context (i.e., popup) menus. Thus, these classes form the foundation of the menu system.

**MenuBar**

MenuBar is essentially a container for menus. It is the control that supplies the main menu of an application. Like all JavaFX controls, it inherits Node. Thus, it can be added to a scene graph. MenuBar has only one constructor, which is the default constructor. Therefore, initially, the menu bar will be empty, and you will need to populate it with menus prior to use. As a general rule, an application has one and only one menu bar. MenuBar defines several methods, but often you will use only one: **getMenus( )**. It returns a list of the menus managed by the menu bar. It is to this list that you will add the menus that you create. The getMenus( ) method is shown here:

**final ObservableList<Menu> getMenus( )**

A Menu instance is added to this list of menus by calling **add( )**. You can also use **addAll( )** to add two or more Menu instances in a single call. The added menus are positioned in the bar from left to right, in the order in which they are added. If you want to add a menu at a specific location, then use this version of add( ):

**void add(int idx, Menu menu)**

Here, menu is added at the index specified by idx. Indexing begins at 0, with 0 being the left-most menu.

In some cases, you might want to remove a menu that is no longer needed. You can do this by calling **remove( )** on the ObservableList returned by getMenus( ). Here are two of its forms:

**void remove(Menu menu)**

**void remove(int idx)**

Here, menu is a reference to the menu to remove, and idx is the index of the menu to remove. Indexing begins at zero. It is sometimes useful to obtain a count of the number of items in a menu bar. To do this, call **size( )** on the list returned by getMenus( ).

NOTE Recall that ObservableList implements the List collections interface, which gives you access to all the methods defined by List.

Once a menu bar has been created and populated, it is added to the scene graph in the normal way.

**Menu**

Menu encapsulates a menu, which is populated with MenuItems. As mentioned, Menu is derived from MenuItem. This means that one Menu can be a selection in another Menu.

This enables one menu to be submenu of another. Menu defines three constructors.

Perhaps the most commonly used is shown here:

**Menu(String name)**

It creates a menu that has the name specified by name. You can specify an image along with text with this constructor:

**Menu(String name, Node image)**

Here, image specifies the image that is displayed. In all cases, the menu is empty until menu items are added to it. Finally, you don’t have to give a menu a name when it is constructed.

To create an unnamed menu, you can use the default constructor:

**Menu( )**

In this case, you can add a name and/or image after the fact by calling **setText( ) or**

**setGraphic( ).**

Each menu maintains a list of menu items that it contains. To add an item to the menu,add items to this list. To do so, first call **getItems( )**, shown here:

**final ObservableList<MenuItem> getItems( )**

It returns the list of items currently associated with the menu. To this list, add menu items by calling either **add( ) or addAll( )**. Among other actions, you can remove an item by calling **remove( )** and obtain the size of the list by calling **size( ).**

One other point: You can add a menu separator to the list of menu items, which is an object of type SeparatorMenuItem. Separators help organize long menus by allowing you to group related items together. A separator can also help set off an important item, such as the Exit selection in a menu.

**MenuItem**

MenuItem encapsulates an element in a menu. This element can be either a selection linked to some program action, such as Save or Close, or it can cause a submenu to be displayed.

MenuItem defines the following three constructors.

**MenuItem( )**

**MenuItem(String name)**

**MenuItem(String name, Node image)**

The first creates an empty menu item. The second lets you specify the name of the item, and the third enables you to include an image.

A MenuItem generates an action event when selected. You can register an action event handler for such an event by calling setOnAction( ), just as you did when handling button events. It is shown again for your convenience:

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

Here, handler specifies the event handler. You can fire an action event on a menu item by calling **fire( ).**

MenuItem defines several methods. One that is often useful is **setDisable( ),** which you can use to enable or disable a menu item. It is shown here:

**final void setDisable(boolean disable)**

If disable is true, the menu item is disabled and cannot be selected. If disable is false, the item is enabled. Using **setDisable( ),** you can turn menu items on or off, depending on program conditions.

**Create a Main Menu**

As a general rule, the most commonly used menu is the main menu. This is the menu defined by the menu bar, and it is the menu that defines all (or nearly all) of the functionality of an application. As you will see, JavaFX streamlines the process of creating and managing the main menu. Here, you will see how to construct a simple main menu. Subsequent sections will show various options.

Constructing the main menu requires several steps. First, create the MenuBar instance that will hold the menus. Next, construct each menu that will be in the menu bar. In general, a menu is constructed by first creating a Menu object and then adding MenuItems to it. After the menus have been created, add them to the menu bar. Then, the menu bar, itself, must be added to the scene graph. Finally, for each menu item, you must add an action event handler that responds to the action event fired when a menu item is selected.

Here is a program that creates a simple menu bar that contains **three menus**. The first is a standard **File menu** that contains **Open, Close, Save, and Exit** selections.

The second menu is called **Options**, and it contains two submenus called **Colors and Priority.**

The third menu is called **Help**, and it has one item: **About**. When a menu item is selected, the name of the selection is displayed in a label.

// Demonstrate Menus

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 MenuDemo 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 Menus");

// Use a BorderPane for the root node.

BorderPane rootNode = new BorderPane();

// Create a scene.

Scene myScene = new Scene(rootNode, 300, 300);

// Set the scene on the stage.

myStage.setScene(myScene);

// Create a label that will report the selection.

response = new Label("Menu Demo");

// Create the menu bar.

MenuBar mb = new MenuBar();

// Create the File menu.

Menu fileMenu = new Menu("File");

MenuItem open = new MenuItem("Open");

MenuItem close = new MenuItem("Close");

MenuItem save = new MenuItem("Save");

MenuItem exit = new MenuItem("Exit");

fileMenu.getItems().addAll(open, close, save,new SeparatorMenuItem(), exit);

// Add File menu to the menu bar.

mb.getMenus().add(fileMenu);

// Create the Options menu.

Menu optionsMenu = new Menu("Options");

// Create the Colors submenu.

Menu colorsMenu = new Menu("Colors");

MenuItem red = new MenuItem("Red");

MenuItem green = new MenuItem("Green");

MenuItem blue = new MenuItem("Blue");

colorsMenu.getItems().addAll(red, green, blue);

optionsMenu.getItems().add(colorsMenu);

// Create the Priority submenu.

Menu priorityMenu = new Menu("Priority");

MenuItem high = new MenuItem("High");

MenuItem low = new MenuItem("Low");

priorityMenu.getItems().addAll(high, low);

optionsMenu.getItems().add(priorityMenu);

// Add a separator.

optionsMenu.getItems().add(new SeparatorMenuItem());

// Create the Reset menu item.

MenuItem reset = new MenuItem("Reset");

optionsMenu.getItems().add(reset);

// Add Options menu to the menu bar.

mb.getMenus().add(optionsMenu);

// Create the Help menu.

Menu helpMenu = new Menu("Help");

MenuItem about = new MenuItem("About");

helpMenu.getItems().add(about);

// Add Help menu to the menu bar.

mb.getMenus().add(helpMenu);

// Create one event handler that will handle menu action events.

EventHandler<ActionEvent> MEHandler =

new EventHandler<ActionEvent>() {

public void handle(ActionEvent ae) {

String name = ((MenuItem)ae.getTarget()).getText();

// If Exit is chosen, the program is terminated.

if(name.equals("Exit")) Platform.exit();

response.setText( name + " selected");

}

};

// Set action event handlers for the menu items.

open.setOnAction(MEHandler);

close.setOnAction(MEHandler);

save.setOnAction(MEHandler);

exit.setOnAction(MEHandler);

red.setOnAction(MEHandler);

green.setOnAction(MEHandler);

blue.setOnAction(MEHandler);

high.setOnAction(MEHandler);

low.setOnAction(MEHandler);

reset.setOnAction(MEHandler);

about.setOnAction(MEHandler);

// Add the menu bar to the top of the border pane and

// the response label to the center position.

rootNode.setTop(mb);

rootNode.setCenter(response);

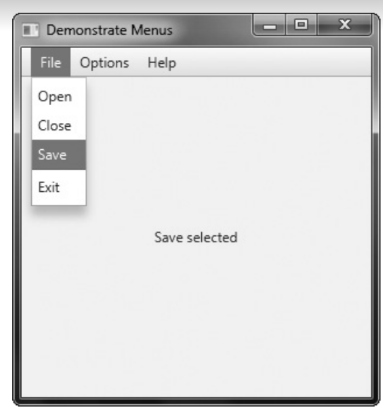
// Show the stage and its scene.

myStage.show();

}

}

Sample output is shown here:



MenuDemo uses a BorderPane instance for the root node. It defines a window that has five areas: top,bottom, left, right, and center. The following methods set the node assigned to these areas:

final void setTop(Node node)

final void setBottom(Node node)

final void setLeft(Node node)

final void setRight(Node node)

final void setCenter(Node node)

Here, node specifies the element, such as a control, that will be shown in each location.Later in the program, the menu bar is positioned in the top location and a label that displays the menu selection is set to the center position. Setting the menu bar to the top position ensures that it will be shown at the top of the application and will automatically be resized to fit the horizontal width of the window. This is why BorderPane is used in the menu examples. Of course, other approaches, such as using a VBox, are also valid.

Platform.exit( )- The Platform class is defined by JavaFX and packaged in javafx.application. Its exit( ) method causes the stop( ) life-cycle method to be called. System.exit( ) does not.

An accelerator can be associated with a Menu or MenuItem. It is specified by calling setAccelerator( ), shown next:

**final void setAccelerator(KeyCombination keyComb)**

Here, keyComb is the key combination that is pressed to select the menu item.

KeyCombination is class that encapsulates a key combination, such as ctrl-s. It is

packaged in javafx.scene.input.

KeyCombination defines two protected constructors, but often you will use the

keyCombination( ) factory method, shown here:

static KeyCombination keyCombination(String keys)

In this case, keys is a string that specifies the key combination. It typically consists of a modifier,

such as ctrl, alt, shift, or meta, and a letter, such as s. There is a special value, called

shortcut, which can be used to specify the ctrl key in a Windows system and the meta key

on a Mac. (It also maps to the typically used shortcut key on other types of systems.)

Therefore, if you want to specify ctrl-s as the key combination for Save, then use the string "shortcut+S". This way, it will work for both Windows and Mac and elsewhere.

The following sequence adds accelerators to the File menu created by the MenuDemo

program in the previous section. After making this change, you can directly select a File

menu option by pressing ctrl-o, ctrl-c, ctrl-s, or ctrl-e.

// Add keyboard accelerators for the File menu.

open.setAccelerator(KeyCombination.keyCombination("shortcut+O"));

close.setAccelerator(KeyCombination.keyCombination("shortcut+C"));

save.setAccelerator(KeyCombination.keyCombination("shortcut+S"));

exit.setAccelerator(KeyCombination.keyCombination("shortcut+E"));

A mnemonic can be specified for both MenuItem and Menu objects, and it is very easy to do. Simply precede the letter in the name of the menu or menu item with an underscore.

For example, in the preceding example, to add the mnemonic F to the File menu, declare fileMenu as shown here:

Menu fileMenu = new Menu("\_File"); // now defines a mnemonic

After making this change, you can select the File menu by typing alt then f. However, mnemonics are active only if mnemonic parsing is true (as it is by default). You can turn mnemonic parsing on or off by using setMnemonicParsing( ), shown here:

final void setMnemonicParsing(boolean enable)

In this case, if enable is true, then mnemonic parsing is turned on. Otherwise, it is turned off.