10.7.3: JavaFX

IT1406

Level I - Semester 1

10.7.3.1. JavaFx basics

- JavaFX is a Java library used to develop Desktop applications as well as Rich Internet Applications (RIA).
- The applications built in JavaFX, can run on multiple platforms including Web, Mobile and Desktops.
- JavaFX is intended to replace swing in Java applications as a GUI framework. However, It provides more functionalities than swing. Like Swing, JavaFX also provides its own components and doesn't depend upon the operating system.
- It is lightweight and hardware accelerated. It supports various operating systems including Windows, Linux and Mac OS.

10.7.3.1. JavaFx basics

- **JavaFX Packages** - JavaFX elements are contained in packages that begin with the **javafx** prefix.
- In general, a JavaFX application will have three major components namely Stage, Scene and Nodes.
- A **stage** defines a space and contains all the objects of a JavaFX application.
- It is represented by **Stage class** of the package **javafx.stage.**
- All JavaFX applications automatically have access to one **Stage**, called the ***primary stage.***

10.7.3.1. JavaFx basics

- The ***primary stage*** is supplied by the run-time system when a JavaFX application is started.
- Although can create other stages, for many applications, the ***primary stage*** will be the only one required.
- A **scene** defines what goes in the **stage** space.
- The **class Scene** of the package **javafx.scene** represents the scene object. At an instance, the scene object is added to only one stage.

Summary

- A **stage** is **a container for scenes** and a **scene** is a **container for the items that comprise the scene.**

10.7.3.1. JavaFx basics

- As a result, all JavaFX applications have **at least one stage and one scene.**

Nodes and Scene Graphs

- The individual elements of a scene are called **nodes**.
- Nodes can also consist of groups of nodes.
- 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.*

10.7.3.1. JavaFx basics

- 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.

10.7.3.1. JavaFx basics

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.
- The **layout panes** are packaged in **javafx.scene.layout.**

Application Class and Life-cycle Methods

- A JavaFX application must be a subclass of the **Application class,** which is packaged in **javafx.application.**

10.7.3.1. JavaFx basics

- The **Application class** defines **three life-cycle methods** that an application can override.
- These are called **init(), start(),** and **stop()** and are shown 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.

10.7.3.1. JavaFx basics

- 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 an application begins and it can be used to construct and set the scene.
- It is passed a reference to a **Stage** object. This is the stage provided by the run-time system and is the primary stage.
- This method is *abstract*. Thus, it must be overridden by your application.
- When the **application is terminated**, the **stop()** method is called.

10.7.3.1. JavaFx basics

Launching a JavaFX Application

- In the main method, the application is launched using the **launch()** method

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

- Here, **args** is a possibly empty list of strings that typically specify command-line arguments.
- When called, **launch()** causes application to be constructed, followed by calls to **init()** and **start().** The **launch()** method will not return until after the application has terminated.

10.7.3.2. JavaFx application skeleton

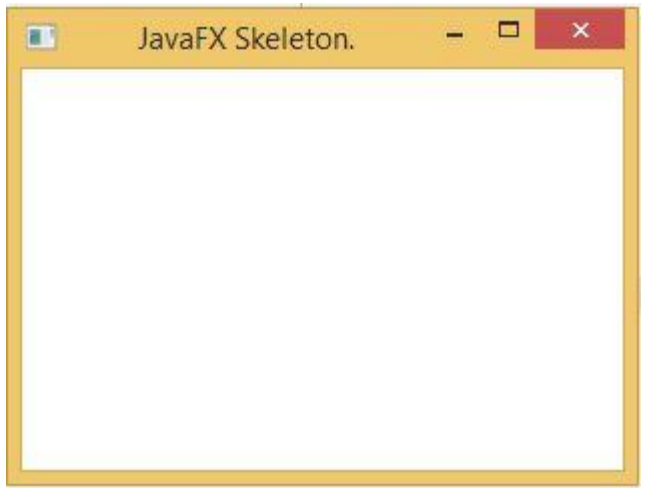
- All JavaFX applications share the same basic skeleton and illustrates how to launch the application and demonstrates when the life-cycle methods are called.
- A message noting when each life-cycle method is called is displayed on the console.
- The complete skeleton is shown here

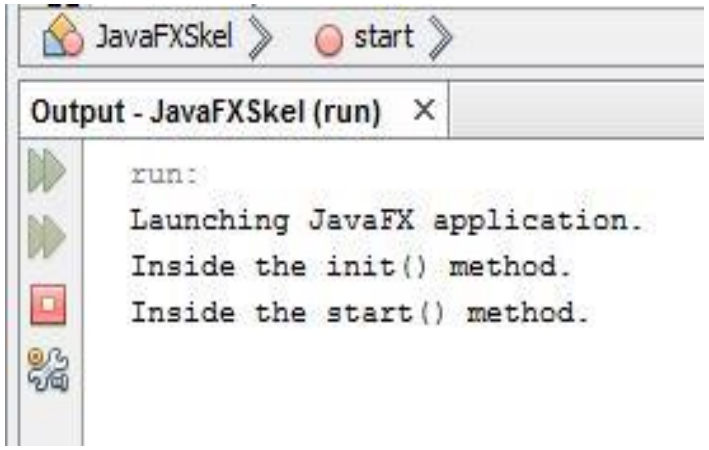
10.7.3.2. JavaFx application skeleton

```
// 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.");
    }
}
```

10.7.3.2. JavaFx application skeleton

- The skeleton produces the following **window** and the **output on the console** shown here:





- When **the window is closed**, this message is displayed on the console:

  Inside the stop() method.

10.7.3.2. JavaFx application skeleton

- 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.**
- The application class **JavaFXSkel** is created. It extends Application**.** 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().**

10.7.3.2. JavaFx application skeleton

- The **args** parameter to main() is passed to the launch() method. Although this is a common approach, a different set of parameters can pass to **launch(),** or none at all.
- When the application begins, the **init()** method is called first by the JavaFX run-time system.
- init() method simply displays a message on System.out, but it would normally be used to initialize some aspect of the application.
- If no initialization is required, it is not necessary to override init() because an empty, default implementation is provided.
- init() cannot be used to create the stage or scene portions of a GUI.

10.7.3.2. JavaFx application skeleton

- After init() finishes, the **start()** method executes. The initial scene is created and set to the primary stage.
- start() has a parameter of type **Stage**. When start() is called, this parameter will receive a reference to the primary stage of 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.");
```

- 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.

10.7.3.2. JavaFx application skeleton

- In this case, a **FlowPane** is used for the root node.

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

- FlowPane 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.)
- 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.

10.7.3.2. JavaFx application skeleton

- It is shown here:

  Scene(Parent rootnode, double width, double height)

- The type of **rootnode** is **Parent**. It is a subclass of Node and encapsulates nodes that can have children.The width and the height are **double** values.
- 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.

10.7.3.2. JavaFx application skeleton

- In cases in which the scene don't make further use, the previous two steps can combine, as shown here:

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

- The last line in start() displays the stage and its scene:

```
myStage.show();
```

- When the application is closed, 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.

10.7.3.2. JavaFx application skeleton

- Furthermore, if the application does not need to handle any shutdown actions, there is no reason to override stop() because an empty, default implementation is provided.

10.7.3.3. Compiling and running JavaFx applications

- One important advantage of JavaFX is that the same program can be run in a variety of different execution environments.
- For example, you can run a JavaFX program as a stand-alone desktop application, inside a web browser, or as a Web Start application.
- In general, a JavaFX program is compiled like any other Java program.
- However, because of the need for additional support for various execution environments, the easiest way to compile a JavaFX application is to use an Integrated Development Environment (IDE) that fully supports JavaFX programming, such as NetBeans.

10.7.3.3. Compiling and running JavaFx applications

- Alternatively, to compile and test a JavaFX application using the command line tools, just **compile** and **run** the application in the normal way, using **javac** and **java.**

10.7.3.4. JavaFx controls and events

A Simple JavaFX Control: Label

- The primary ingredient in most user interfaces is the control, because a control enables the user to interact with the application.
- JavaFX supplies a rich assortment of controls. The simplest control is the **label** because it just displays a message, which, in this example, is text.
- 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 and Control defines features related to all controls.

10.7.3.4. JavaFx controls and events

- Label defines three constructors. The one use here is **Label(String str).** Here, **str** is the string that is displayed.
- Once a label have been created, it must be added to the scene's content, which means adding it to the scene graph.
- To do this, 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.**
- Using the returned list of child nodes,the label can add to the list by calling **add(),** passing in a reference to the label.

10.7.3.4. JavaFx controls and events

- 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();
    }
}
```

10.7.3.4. JavaFx controls and events

- 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.
- 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);

10.7.3.4. JavaFx controls and events

Using Buttons and Events

- One commonly used control is the **button** and it is a good way to demonstrate the fundamentals of event handling in JavaFX.

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. **ActionEvent** handles action events generated by a button.

10.7.3.4. JavaFx controls and events

- 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>

- 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) ⟶ eventOjt is the event

- The source of the event can obtain by calling **getSource(),** which is inherited from **java.util.EventObject.** It is shown here:

  Object getSource()

10.7.3.4. JavaFx controls and events

Introducing the Button Control

- In JavaFX, the **push button control** is provided by the **Button** class, which is in **javafx.scene.control.**
- Buttons can contain text, graphics, or both.
- Button defines three constructors. The one is shown here:

  Button(String *str)*

- ***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.

10.7.3.4. JavaFx controls and events

- A listener can register for this event by using **setOnAction(),** which has this general form:

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

- ***handler*** is the handler being registered. Often 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.

10.7.3.4. JavaFx controls and events

Demonstrating Event Handling and the Button

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

```
// Demonstrate JavaFX events and buttons.

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

```
public class JavaFXEventDemo extends Application {

    Label response;

    public static void main(String[] args) {

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

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

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

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

10.7.3.4. JavaFx controls and events

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

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

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

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

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

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

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

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

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

10.7.3.4. JavaFx controls and events

- Sample output from this program is shown here:

- 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.

10.7.3.4. JavaFx controls and events

- 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.");
        }
    });
```

- Buttons respond to events of type ActionEvent. To register a handler for these events, the **setOnAction()** method is called on the button.
- Inside **handle()**, the text in the **response** label is set to reflect the fact that the Alpha button was pressed.
- This is done by calling the **setText()** method on the label.

10.7.3.4. JavaFx controls and events

- Events are handled by the Beta button in the same way.
- After the event handlers have been set, the **response** label and the buttons **btnAlpha** and **btnBeta** are added to the scene graph by using a call to **addAll( ):**

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

- The addAll() method adds a list of nodes to the invoking parent node.
- When the root node is created, this statement is used:

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

10.7.3.4. JavaFx controls and events

- The **FlowPane** constructor is passed two values. These specify the horizontal and vertical gap that will be left around elements in the scene.
- Sets the alignment of the elements in the FlowPane:

  rootNode.setAlignment(Pos.CENTER);
- The alignment of the elements is centered. This is done by calling **setAlignment()** on the FlowPane.
- **Pos** is an enumeration that specifies alignment constants. It is packaged in **javafx.geometry.**

10.7.3.5. JavaFX controls

Using Image and ImageView

- In addition to text, an image can specify in a label or a button and can embed stand-alone images in a scene directly.
- JavaFX's support two classes of images: **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 will use:

Image(String *url)*

10.7.3.5. JavaFX controls

- ***url*** specifies a URL or a path to a file that supplies the image.
- 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; this is the one will use:

  ImageView(Image image)

- Constructor creates an ImageView that uses image for its image.

10.7.3.5. JavaFX controls

- 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.

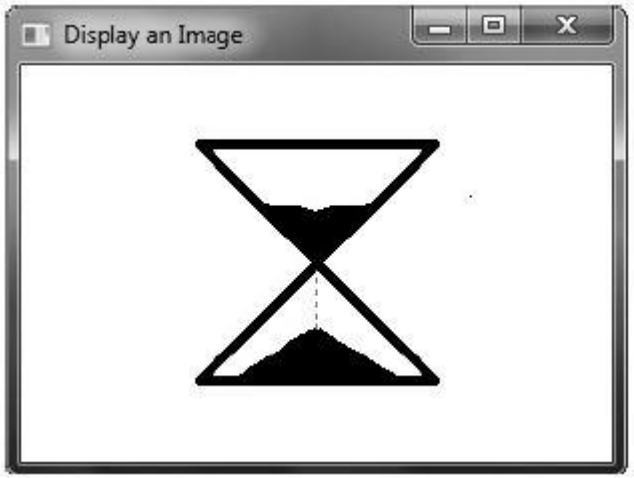
```
// Load and display an image.

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

```
public 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);
```

10.7.3.5. JavaFX controls

```
        // 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



10.7.3.5. JavaFX controls

- In the program, loads the image and then creates an ImageView which uses that image.
- An image by itself cannot be added to the scene graph. It must first be embedded in an ImageView.

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

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

- Further don't make use of the image, a URL or filename can specify when creating an ImageView.
- Here is the ImageView constructor that does this:

  ImageView(String url)

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

10.7.3.5. JavaFX controls

- For the example, an Image instance containing the specified image is constructed automatically and embedded in the ImageView.

Adding an Image to a Label

- The **Label** class encapsulates a label. It can display a text message, a graphic, or both.
- To add an image use this form of Label's constructor:

  Label(String str, Node image)

- ***str*** specifies the text message and ***image*** specifies the image. Notice that the image is of type Node.

10.7.3.5. JavaFX controls

- 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.

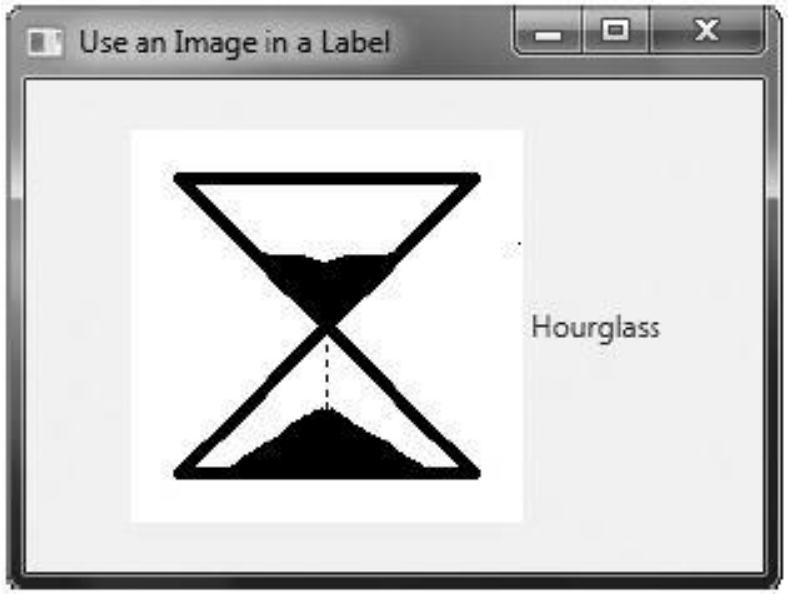
```
// Demonstrate an image in a label.

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);
```

10.7.3.5. JavaFX controls

```
        // 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();
    }
}
```

Sample output



10.7.3.5. JavaFX controls

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

final void setContentDisplay(ContentDisplay *position*)

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

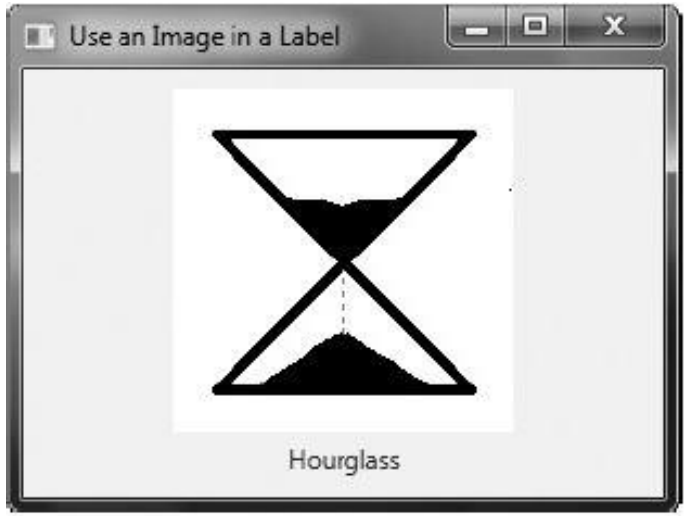
BOTTOM	RIGHT
CENTER	TEXT_ONLY
GRAPHIC_ONLY	TOP
LEFT	

10.7.3.5. JavaFX controls

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

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

- The image of the hourglass will be shown here:



10.7.3.5. JavaFX controls

- An image is added to a label after it has been constructed by using the **setGraphic()** method. It is shown here:

final void setGraphic(Node *image*)

specifies the image to add

10.7.3.5. JavaFX controls

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)

- *str* specifies the text that is displayed within the button and *image* specifies the image.
- The position of the image is specified relative to the text by using setContentDisplay() in the same way as just described for Label.

10.7.3.5. JavaFX controls

- 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. 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");
```

10.7.3.5. JavaFX controls

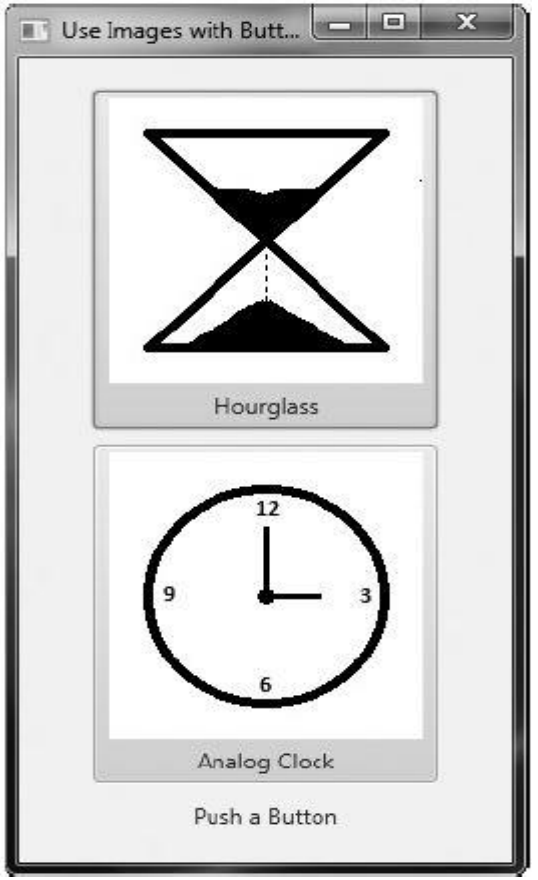
```
// Use a FlowPane for the root node. In this case,
// vertical and horizontal gaps of 10.
FlowPane rootNode = new FlowPane(10, 10);

// 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");
    }
});
```

10.7.3.5. JavaFX controls

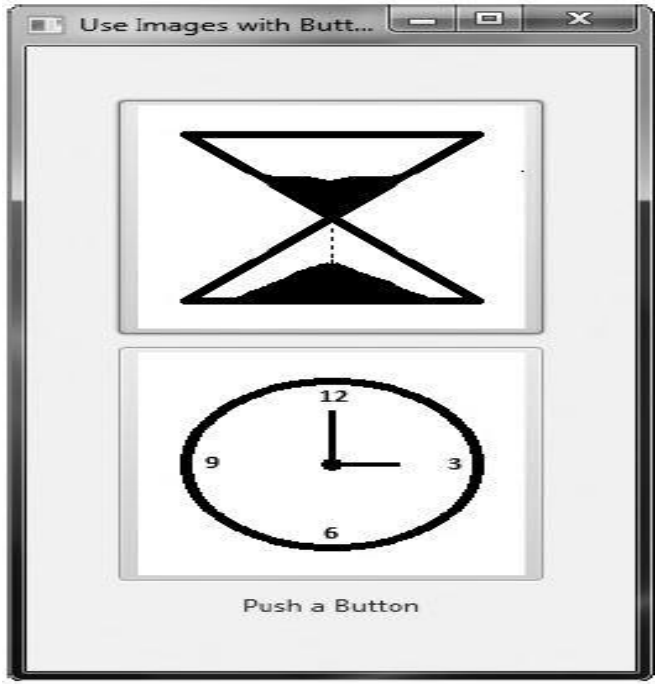
```
        // Handle the action events for the analog clock button.
        btnAnalogClock.setOnAction(new EventHandler<ActionEvent>() {
            public void handle(ActionEvent ae) {
                response.setText("Analog Clock Pressed");
            }
        });

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

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

Sample output



10.7.3.5. JavaFX controls

- A button 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 these modifications make to the previous program, the output will look like this:



10.7.3.5. JavaFX controls

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.
- A toggle button is encapsulated in the **ToggleButton** class.

10.7.3.5. JavaFX controls

- It implements the Toggle interface, which defines functionality common to all types of two-state buttons.
- ToggleButton defines three constructors. This is the one will use:

  ToggleButton(String *str)

- *str* is the text displayed in the button.
- Another constructor allows 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.

10.7.3.5. JavaFX controls

- 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:

10.7.3.5. JavaFX controls

```
// 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);
```

10.7.3.5. JavaFX controls

```
        // 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();
    }
}
```

The resultant output with the button pressed

10.7.3.5. JavaFX controls

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

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

- When the button is pressed, **isSelected()** returns **true.** When the button is released, **isSelected()** returns **false.**
- It is possible to use two or more toggle buttons in a group. In this case, only one button can be in its pressed state at any one time.

10.7.3.5. JavaFX controls

RadioButton

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

  RadioButton(String str)

- *str* is the label for the button.

10.7.3.5. JavaFX controls

- 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.

10.7.3.5. JavaFX controls

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

  final void setToggleGroup(ToggleGroup tg)

- *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.
- 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.

10.7.3.5. JavaFX controls

- First, **setSelected()** is called on the button to be selected. 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.

10.7.3.5. JavaFX controls

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

```
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");
```

10.7.3.5. JavaFX controls

```
// 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.");
    }
});
```

10.7.3.5. JavaFX controls

```
        // 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

10.7.3.5. JavaFX controls

- First, the buttons are created using this sequence:

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

- Next, a ToggleGroup is constructed:

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

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

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

10.7.3.5. JavaFX controls

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

Handling Change Events in a Toggle Group

- When a change takes place, the event handler can easily determine which radio button has been selected and take action accordingly.
- To use this approach, **ChangeListener** must be registered on the toggle group.
- When a change event occurs, then determine which button was selected.

10.7.3.5. JavaFX controls

- To try this approach, remove the action event handlers and the call to **fire()** from the preceding program and substitute the following:

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

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

        // Display the selection.
        response.setText("Transport selected is " + rb.getText());
    }
});
// Select the first button. This will cause a change event
// on the toggle group.
rbTrain.setSelected(true);
```

10.7.3.5. JavaFX controls

- Following import statement is needed to be added which supports the **ChangeListener** interface.

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

- The output from this program is the same as before; each time a selection is made, the **response** label is updated.
- In this case, only one event handler is needed for the enter group, rather than three (one for each button).
- First, a change event listener is registered for the toggle group.
- To listen for change events, **ChangeListener** interface must be implemented.

10.7.3.5. JavaFX controls

- This is done by calling **addListener()** on the object returned by **selectedToggleProperty().**
- The ChangeListener interface defines only one method, called **changed().** It is shown here:

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

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

10.7.3.5. JavaFX controls

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.
- 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.

10.7.3.5. JavaFX controls

```
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);
```

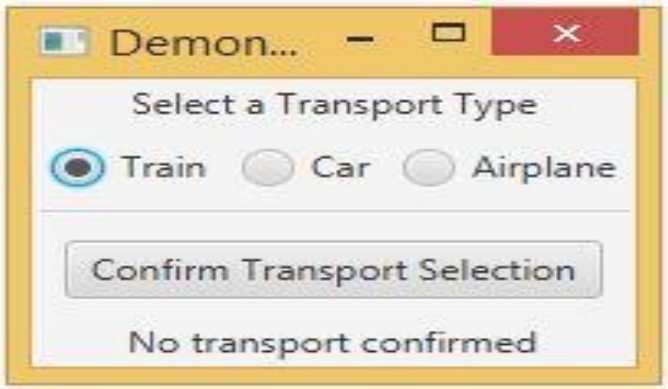
10.7.3.5. JavaFX controls

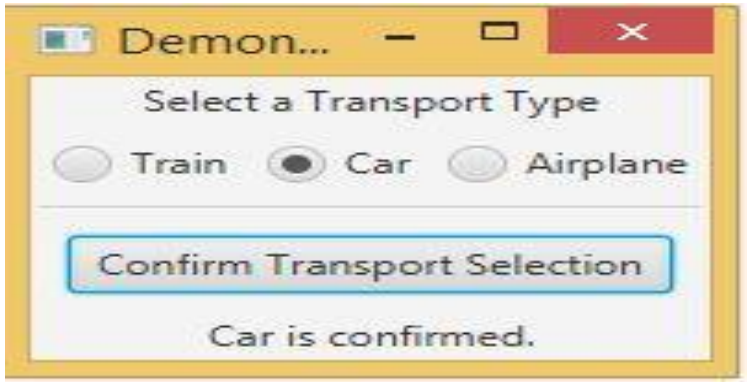
```
// Create two labels.
Label choose = new Label(" Select a Transport Type ");
response = new Label("No transport confirmed");

// 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();
    }
}
```

10.7.3.5. JavaFX controls

- The output from the program is shown here:





10.7.3.5. JavaFX controls

- First, inside the action event handler for the **btnConfirm** button, the selected radio button is obtained by the following line:

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

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

```
final Toggle getSelectedToggle()
```

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

10.7.3.5. JavaFX controls

- A visual separator is used, which is created by this sequence:

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

- The **Separator** class creates a line, which can be either vertical or horizontal. By default, it creates a horizontal line.
- 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.

10.7.3.5. JavaFX controls

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**, and this is the default behavior. The third state is **indeterminate.**
- 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.
- CheckBox defines two constructors. The first is the default constructor. The second lets to specify a string that identifies the box. It is shown here:

  CheckBox(String str)

10.7.3.5. JavaFX controls

- 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.

10.7.3.5. JavaFX controls

```
// 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");
```

10.7.3.5. JavaFX controls

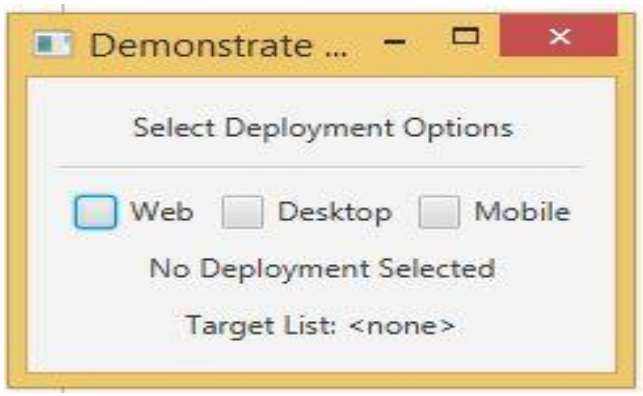
```
// Create a label that will report the state of the
// selected check box.
response = new Label("No Deployment Selected");

// 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();
        }
    });
```

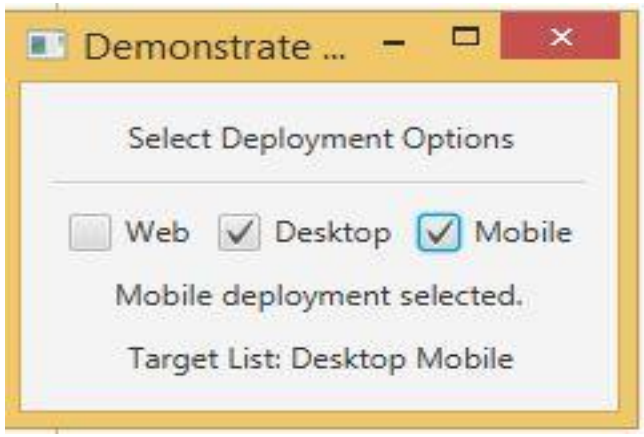
```
cbDesktop.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        if(cbDesktop.isSelected())
            response.setText("Desktop deployment selected.");
        else
            response.setText("Desktop deployment cleared.");

        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();
    }
});
```

10.7.3.5. JavaFX controls

```
        // 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





10.7.3.5. JavaFX controls

- 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 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.

10.7.3.5. JavaFX controls

- When the indeterminate state is enabled, the user can select between checked, unchecked, and indeterminate.
- 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.
- The effect of a three-state check box can be seen by modifying the preceding program.
- First, enable the indeterminate state on the check boxes by calling **setAllowIndeterminate()** on each check box, as shown here:

10.7.3.5. JavaFX controls

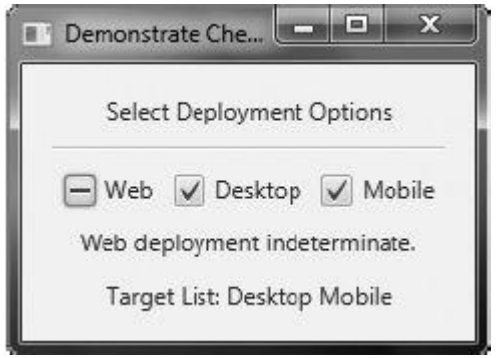
cbWeb.setAllowIndeterminate(true);
cbDesktop.setAllowIndeterminate(true);
cbMobile.setAllowIndeterminate(true);

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

```
cbWeb.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        if(cbWeb.isIndeterminate())
                response.setText("Web deployment indeterminate.");
        else if(cbWeb.isSelected())
                response.setText("Web deployment selected.");
                else
                response.setText("Web deployment cleared.");
                showAll();
    }
});
```

10.7.3.5. JavaFX controls

- 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:



ListView

- List views are controls that display a list of entries from which select one or more.
- Because of the ability to make efficient use of limited screen space, list views are popular alternatives to other types of selection controls.

10.7.3.5. JavaFX controls

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

  class ListView<T>

- **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 to specify the list of entries in the list. It is shown here:

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

- *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.**

10.7.3.5. JavaFX controls

- 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). It is shown here:

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

- **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.

10.7.3.5. JavaFX controls

- Multiple selections are allowed by changing the selection mode. For now, the default single-selection model is used.
- Although ListView provides a default size, sometimes want to set the preferred height and/or width.
- 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, a single call is to set both dimensions at the same time by use of **setPrefSize( ),** shown here:

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

10.7.3.5. JavaFX controls

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

10.7.3.5. JavaFX controls

- 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(),** 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( )
```

10.7.3.5. JavaFX controls

- The following example 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);
```

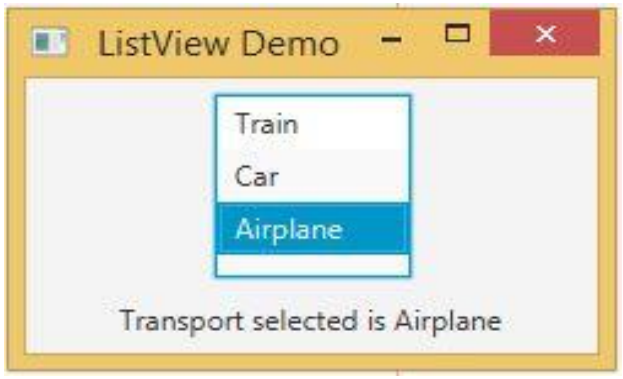
10.7.3.5. JavaFX controls

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

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

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

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

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

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

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

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

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

10.7.3.5. JavaFX controls

- Sample output is shown here:



- First, an **ObservableList** is created by this line:

```
ObservableList<String> transportTypes =

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

10.7.3.5. JavaFX controls

- 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.
- Then the selection model is obtained for **lvTransport:**

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

10.7.3.5. JavaFX controls

ListView Scrollbars

- When the number of items in the ListView exceeds the number that can be displayed within its dimensions, scrollbars are automatically added. For example, if 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:

10.7.3.5. JavaFX controls

Enabling Multiple Selections

- If want to allow more than one item to be selected, 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, can obtain the list of the selections two ways: as a list of selected indices or as a list of selected items.

10.7.3.5. JavaFX controls

- 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 and ObservableList extends **java.util.List.**
- For multiple selections, modify the preceding program as follows.
- First, make **lvTransport final** so it can be accessed within the change event handler. Next, add this line:

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

- It enables multiple-selection mode for **lvTransport**.

10.7.3.5. JavaFX controls

- Finally, replace the change event handler with the one shown here:

```
lvSelModel.selectedItemProperty().addListener(
                                                new ChangeListener<String>() {
            public void changed(ObservableValue<? extends String> changed,
                                    String oldVal, String newVal) {
                    String selItems = "";
                    ObservableList<String> selected =
                                    lvTransport.getSelectionModel().getSelectedItems();
    // Display the selections.
                    for(int i=0; i < selected.size(); i++)
                            selItems += "\n " + selected.get(i);
                    response.setText("All transports selected: " + selItems);
                    }
    });
```

10.7.3.5. JavaFX controls

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

10.7.3.5. JavaFX 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.
- 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>
```

10.7.3.5. JavaFX controls

- **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 to specify the list of entries. It is shown here:

ComboBox(ObservableList<T> list)

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

10.7.3.5. JavaFX controls

- 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.
- The current selection can be obtained by calling **getValue()**, shown here:

```
final T getValue()
```

10.7.3.5. JavaFX controls

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

  final void setValue(T *newVal*)

- *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.

10.7.3.5. JavaFX controls

```
// 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);
    }
```

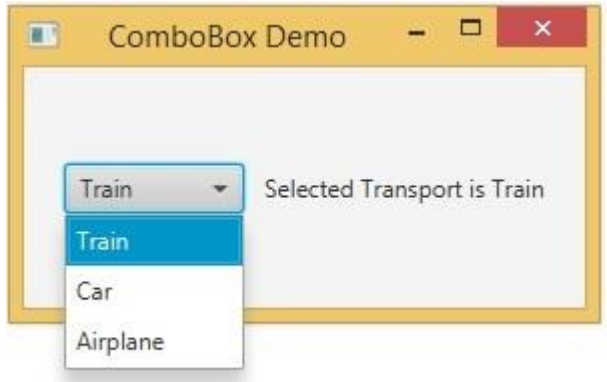
```
// Override the start() method.
public void start(Stage myStage) {

    // 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);
```

10.7.3.5. JavaFX controls

```
// 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



10.7.3.5. JavaFX controls

- 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, the selection can be edit.

10.7.3.5. JavaFX controls

TextField

- TextField 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 to specify the initial contents of the field.
- Although the default size is sometimes adequate, often want to specify its size. This is done by calling **setPrefColumnCount(),** shown here:

final void setPrefColumnCount(int *columns)*

10.7.3.5. JavaFX controls

- The *columns value is used by TextField to determine its size.*
- The text is set in a text field by calling **setText()** and obtain the current text by calling **getText( )**.
- 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)

- *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).
- The following program creates a text field that requests a search string.

10.7.3.5. JavaFX controls

- 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);
    }
```

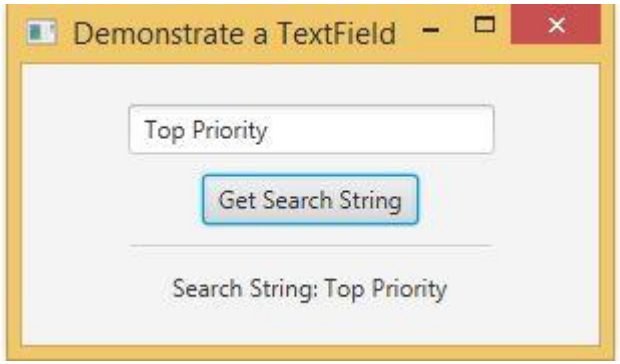
```
// Override the start() method.
public void start(Stage myStage) {

    // 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();
```

10.7.3.5. JavaFX controls

```
        // 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



10.7.3.5. JavaFX controls

ScrollPane

- Sometimes, the contents of a control will exceed the amount of screen space that you want to give to it.
- For an example, want to display a text that is longer than will fit within a small window.
- For that, 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.

10.7.3.5. JavaFX controls

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

  ScrollPane(Node *content)*

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

  final void setContent(Node *content)*

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

10.7.3.5. JavaFX controls

- You can also use **setContent()** to change the content being scrolled by the scroll pane.
- Although a default size is provided, as a general rule, 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, changes what part of the content is visible.

10.7.3.5. JavaFX controls

- 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.
- ScrollPane is able 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)*

10.7.3.5. JavaFX controls

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

  final void setHvalue(double *newHval)*

  final void setVvalue(double *newVval)*

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

10.7.3.5. JavaFX controls

- ScrollPane supports various other options. For example, it is possible to set the minimum and maximum scrollbar positions.
- 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.

10.7.3.5. JavaFX controls

```
// 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);
```

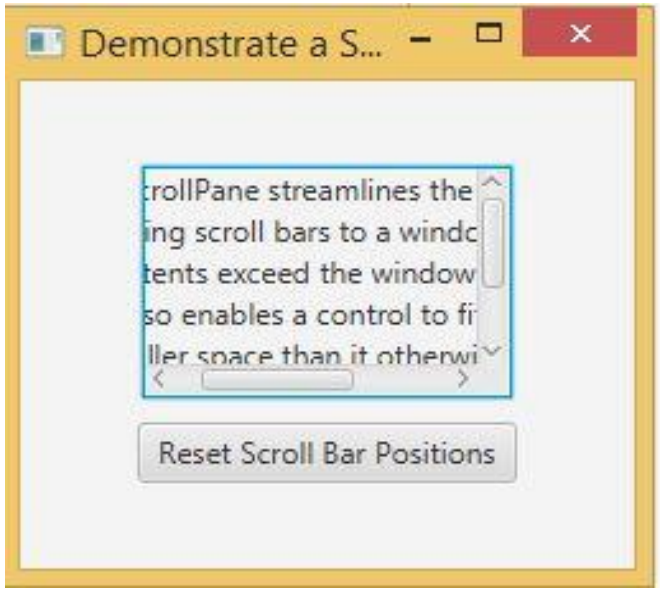
10.7.3.5. JavaFX controls

```
// 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();
  }
}
```

10.7.3.5. JavaFX controls

- Sample output is shown here:



10.7.3.5. JavaFX controls

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. This allows hierarchical data to be presented in a compact, yet expandable form.
- TreeView implements a conceptually simple, tree-based data structure.

10.7.3.5. JavaFX controls

- A tree begins with a single *root node that indicates the start of the tree.*
- Under the root are one or more child nodes, there are two types of child nodes: leaf nodes (also called terminal nodes), which have no children, and branch nodes, which form the root nodes of subtrees.
- A subtree is simply a tree that is part of a larger tree. The sequence of nodes that leads from the root to a specific node is called a ***path***.
- TreeView automatically provides scrollbars when the size of the tree exceeds the dimensions of the view.
- TreeView is a generic class that is defined like this:

class TreeView<T>

10.7.3.5. JavaFX controls

- **T** specifies the type of value held by an item in the tree. This will be of type String.
- TreeView defines two constructors. The one will use:

TreeView(TreeItem<T> rootNode)

- *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.**
- It is important to state that TreeItem does not inherit Node**.** Thus, TreeItems are not general-purpose objects.

10.7.3.5. JavaFX controls

- TreeItem is a generic class, as shown here:

  class TreeItem<T>

- **T** specifies the type of value held by the TreeItem and 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, create the TreeView by passing the root node to its constructor.
- Selection events can handle in the TreeView in a way similar to the way of handling them in a ListView**,** through the use of a change listener.
- To do so, first, obtain the selection model by calling **getSelectionModel()**.

10.7.3.5. JavaFX controls

- 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.
- The value of a TreeItem is obtained by calling **getValue().** 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.

10.7.3.5. JavaFX controls

- 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");
```

10.7.3.5. JavaFX controls

```
// 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();
```

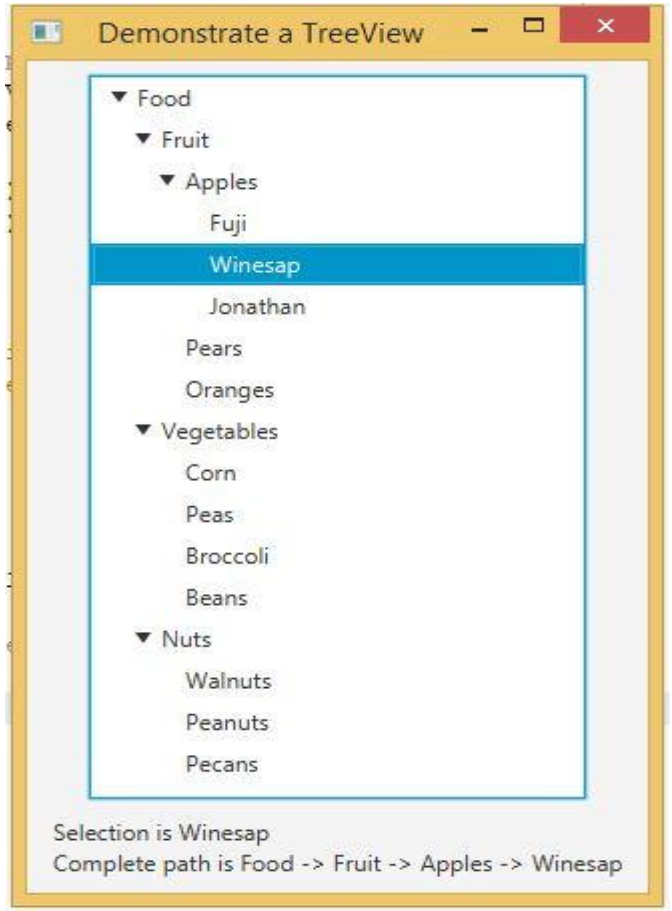
10.7.3.5. JavaFX controls

```
// 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();
    }
}
```

10.7.3.5. JavaFX controls

- Sample output is shown here:



- 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.
- After all of the nodes have been constructed, the root nodes of each subtree are added to the root node of the tree.

10.7.3.5. JavaFX controls

- 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 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();
}
```

10.7.3.5. JavaFX controls

- 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.

10.7.3.6. Effects and transforms

Effects

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

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

10.7.3.6. Effects and transforms

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

  final void setEffect(Effect *effect*)

- *effect* is the effect that will be applied.
- To specify no effect, pass **null**. Thus, to add an effect to a node, first create an instance of that effect and then pass it to setEffect().
- Once this has been done, the effect will be used whenever the node is rendered (as long as the effect is supported by the environment).
- **Glow** produces an effect that gives a node a glowing appearance. To use a glow effect, first create a Glow instance.

10.7.3.6. Effects and transforms

- This is the constructor:

  Glow(double *glowLevel*)

- *glowLevel* specifies the amount of glowing, which must be a value between 0.0 and 1.0.
- After a Glow instance has been created, the glow level can be changed by using **setLevel()**, shown here:

  final void setLevel(double glowLevel)

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

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

10.7.3.6. Effects and transforms

- *radius* specifies the radius of the shadow inside the node. In essence, the radius describes the size of the shadow.
- The color of the shadow is specified by *shadowColor*. The type Color is the JavaFX type **javafx.scene.paint.Color**.
- It defines a large number of constants, such as **Color.GREEN**, **Color.RED**, and **Color.BLUE**, which makes it easy to use.

10.7.3.6. Effects and transforms

Transforms

- Transforms are supported by the abstract **Transform** class, which is packaged in **javafx.scene.transform**.
- Four of its subclasses are Rotate, Scale, Shear, and Translate. And it is possible to perform more than one transform on a node.
- One way to add a transform to a node is to add it to the list of transforms maintained by the node.
- This list is obtained by calling **getTransforms()**, which is defined by Node. It is shown here:

```
final ObservableList<Transform> getTransforms()
```

10.7.3.6. Effects and transforms

- It returns a reference to the list of transforms. To add a transform, simply add it to this list by calling **add()**. The list is clear by calling **clear().**
- To remove a specific element use **remove().**
- In some cases, can specify a transform directly, by setting one of Node's properties. For example, can set the rotation angle of a node, with the pivot point being at the center of the node, by calling **setRotate()**, passing in the desired angle.
- A scale is set by using **setScaleX()** and **setScaleY( )**, and can translate a node by using **setTranslateX()** and **setTranslateY()**.
- Any transforms specified on a node directly will be applied after all transforms in the transforms list.

10.7.3.6. Effects and transforms

- **Rotate** rotates a node around a specified point. It defines several constructors. Here is one example:

  Rotate(double *angle*, double *x*, double *y*)

- *angle* specifies the number of degrees to rotate. The center of rotation, called the ***pivot point***, is specified by *x* and *y*.
- It is also possible to use the default constructor and set these values after a Rotate object has been created.
- This is done by using the **setAngle()**, **setPivotX()**, and **setPivotY()** methods, shown here:

  final void setAngle(double *angle*)

  final void setPivotX(double *x*)

  final void setPivotY(double *y*)

10.7.3.6. Effects and transforms

- *angle* specifies the number of degrees to rotate and the center of rotation is specified by *x* and *y*.
- **Scale** scales a node as specified by a scale factor. **Scale** defines several constructors. The one will use is:

  Scale(double *widthFactor*, double *heightFactor*)

- *widthFactor* specifies the scaling factor applied to the node's width, and *heightFactor* specifies the scaling factor applied to the node's height.
- These factors can be changed after a Scale instance has been created by using **setX()** and **setY()**, shown here:

  final void setX(double *widthFactor*)

  final void setY(double *heightFactor*)

10.7.3.6. Effects and transforms

- **Text** is a class packaged in **javafx.scene.text**.
- It creates a node that consists of text. Because it is a node, the text can be easily manipulated as a unit and various effects and transforms can be applied.

Demonstrating Effects and Transforms

- The following program demonstrates the use of effects and transforms.
- It does so by creating four buttons called Rotate, Scale, Glow, and Shadow.
- Each time one of these buttons is pressed, the corresponding effect or transform is applied to the button.

10.7.3.6. Effects and transforms

```
// Demonstrate rotation, scaling, glowing, and inner shadow.

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.transform.*;
import javafx.scene.effect.*;
import javafx.scene.paint.*;

public class EffectsAndTransformsDemo extends Application {

    double angle = 0.0;
    double glowVal = 0.0;
    boolean shadow = false;
    double scaleFactor = 1.0;

    // Create initial effects and transforms.
    Glow glow = new Glow(0.0);
    InnerShadow innerShadow = new InnerShadow(10.0, Color.RED);
    Rotate rotate = new Rotate();
    Scale scale = new Scale(scaleFactor, scaleFactor);
```

```
    // Create four push buttons.
    Button btnRotate = new Button("Rotate");
    Button btnGlow = new Button("Glow");
    Button btnShadow = new Button("Shadow off");
    Button btnScale = new Button("Scale");

    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("Effects and Transforms Demo");

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

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

10.7.3.6. Effects and transforms

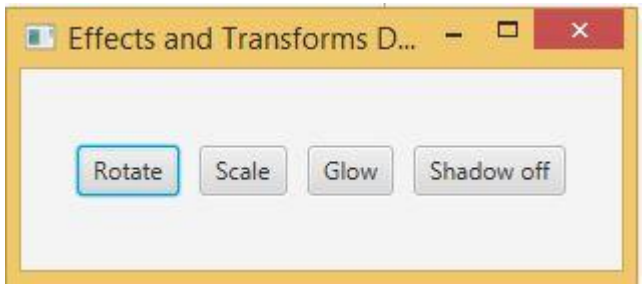
```
    // Create a scene.
    Scene myScene = new Scene(rootNode, 300, 100);

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

    // Set the initial glow effect.
    btnGlow.setEffect(glow);

    // Add rotation to the transform list for the Rotate button.
    btnRotate.getTransforms().add(rotate);

    // Add scaling to the transform list for the Scale button.
    btnScale.getTransforms().add(scale);

    // Handle the action events for the Rotate button.
    btnRotate.setOnAction(new EventHandler<ActionEvent>() {

    public void handle(ActionEvent ae) {
        // Each time button is pressed, it is rotated 30 degrees
        // around its center.
        angle += 30.0;

        rotate.setAngle(angle);
        rotate.setPivotX(btnRotate.getWidth()/2);
        rotate.setPivotY(btnRotate.getHeight()/2);
    }
});
```

```
// Handle the action events for the Scale button.
btnScale.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        // Each time button is pressed, the button's scale is changed.
        scaleFactor += 0.1;
        if(scaleFactor > 1.0) scaleFactor = 0.4;

        scale.setX(scaleFactor);
        scale.setY(scaleFactor);
    }
});

// Handle the action events for the Glow button.
btnGlow.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        // Each time button is pressed, its glow value is changed.
        glowVal += 0.1;
        if(glowVal > 1.0) glowVal = 0.0;

        // Set the new glow value.
        glow.setLevel(glowVal);
    }
});
```

10.7.3.6. Effects and transforms

```
    // Handle the action events for the Shadow button.
    btnShadow.setOnAction(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent ae) {
            // Each time button is pressed, its shadow status is changed.
            shadow = !shadow;
            if(shadow) {
                btnShadow.setEffect(innerShadow);
                btnShadow.setText("Shadow on");
            } else {
                btnShadow.setEffect(null);
                btnShadow.setText("Shadow off");
            }
        }
    });

    // Add the label and buttons to the scene graph.
    rootNode.getChildren().addAll(btnRotate, btnScale, btnGlow, btnShadow);

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

Sample output



10.7.3.6. Effects and transforms

Adding Tooltips

- A tooltip is a short message that is displayed when the mouse hovers over a control.
- To add a tooltip, call the **setTooltip()** method defined by **Control**. (Control is a base class for all controls.)
- The setTooltip() method is shown here:

  final void setTooltip(Tooltip *tip*)

- *tip* is an instance of Tooltip, which specifies the tooltip. Once a tooltip has been set, it is automatically displayed when the mouse hovers over the control.

10.7.3.6. Effects and transforms

- The **Tooltip** class encapsulates a tooltip. The constructor is:

  Tooltip(String *str*)

- *str* specifies the message that will be displayed by the tooltip.
- To see tooltips in action, try adding the following statements to the CheckboxDemo program shown earlier.

```
cbWeb.setTooltip(new Tooltip("Deploy to Web"));
cbDesktop.setTooltip(new Tooltip("Deploy to Desktop"));
cbMobile.setTooltip(new Tooltip("Deploy to Mobile"));
```

- After these additions, the tooltips will be displayed for each check box.

10.7.3.6. Effects and transforms

Disabling a Control

- Any node in the scene graph, including a control, can be disabled under program control.
- To disable a control, use **setDisable()**, defined by Node. It is shown here:

  final void setDisable(boolean *disable*)

- If *disable* is **true**, the control is disabled; otherwise, it is enabled. Thus, using setDisable(), you can disable a control and then enable it later.